Sports programming in practice

(Programowanie sportowe w praktyce)

Bartosz Kostka

Praca magisterska

Promotor: dr hab. Jakub Radoszewski
Instytut Informatyki
Wydział Matematyki, Informatyki i Mechaniki
Uniwersytet Warszawski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

15 lutego 2021

Abstract

The topic of this paper is sports programming, which is a mind sport focusing on solving algorithmic problems. The paper describes selected recent techniques and methods used in this field, along with examples of applications in the form of problems from selected competitions.

Tematem pracy jest programowanie sportowe – sport umysłowy skupiający się na rozwiązywaniu problemów algorytmicznych. Praca opisuje wybrane najnowsze techniki i metody używane w tej dziedzinie, wraz z przykładowymi zastosowaniami w formie zadań z wybranych konkursów.

Contents

Chapter 1

Introduction

Nowadays, developers and engineers in high-tech companies have to solve many difficult and interesting problems. You can think for example, how to find the shortest way from one place to another in Google Maps, or how to suggest potential friends in the enormous graph of relations between users on Facebook. Under the hood, solving these problems resembles a bit of puzzle-solving, like a crossword or a sudoku in a magazine. This problem-solving skill is what companies are looking for and what potential employees have to develop.

As an indirect result of demand for such problem-solving skills, people started specializing in this field and thus the sports programming (also known as competitive programming) was formed. It is a mind sport focusing on solving these algorithmic puzzles. Contestants are given a set of problems (sometimes directly extracted from the real-life) which they need to solve in a limited time. They have to implement the solution in a form of a programming code. Solutions are often checked (judged) automatically. Organizers prepare a set of potential scenarios (commonly known as testcases) on which the contestants solutions are checked – they are checking the behaviour of the program, i.e. if it produces a valid output and fits into other constraints, such as time and memory limits.

Such programming competitions come in many different kinds. The first important competition that a young contestant can participate in is International Olympiad in Informatics for high school students. There is also a famous team competition for university students called International Collegiate Programming Contest. In this competition, teams of three students from the same university compete against each other, using only one computer. Many companies also conduct yearly championships branded with their name, for instance we have Google Code Jam, or Facebook Hacker Cup. There are also special platforms hosting various competitions and companies that were created just for this purpose. Some examples might include Atcoder, Codechef, Codeforces, or Topcoder. Almost all of these platforms also have some kind of a rating system, so contestants can compare themselves with their friends or people from all

over the world. Later in this chapter, we will describe in brief some of the most popular platforms and competitions.

As the sports programming started to grow more popular, the problems also began to become more complex. Some contestants even started to specialize in various fields, such as geometry or text algorithms. They started to discover many interesting algorithms and data structures which have their origin in competitive programming, with some real-life applications. I want to briefly mention two examples. First, on one of the Russian training camps, two students developed a problem about finding all distinct palindromes in a given word. For the solution of this problem, they came up with the data structure that is currently known as the “palindromic tree” or "eertree". This approach not only solved this problem, but people started to use this data structure to solve many different problems, even the ones that were thought that they cannot be solved faster in the past. We will discuss this data structure and various problems that can be solved in a better time complexity with it in chapter 5: Palindromes. The second example comes from China. High school students who want to participate in the International Olympiad in Informatics are asked to write a simple research report about some chosen algorithm or data structure. One of these participants solved a problem that was also causing a lot of problems for programmers for several years. He came up with the data structure called “Segment tree beats” that allows fast operations on integer sequences, such as adding some value over a continuous subsequence or calculating the minimum value over other continuous sequences. We will discuss this data structure in chapter 3: Segment trees rediscovered.

On the other hand, people also started digging up some older algorithms and found their new applications. The wavelet tree and matrix were data structures used in text processing since 2003. People rediscovered this data structure in 2016, when a paper was published about how it can be used in competitive programming. This structure allowed people to solve some of the problems in a clean, elegant manner, which was also really quick and easy to implement. We mention this data structure in chapter 3: Segment trees rediscovered.

The goal of this paper is to spread the awareness of sports programming and also to document several of these new ideas and methods that were discovered in the last decade. One of the problems that the sports programming community has is that only some of these ideas are properly described in academic papers, while most of them can be found only on some blogs and websites spread across the Internet in various languages. This paper attempts to partially fill in this gap.

We also include various problems with solutions to show how these methods are used in real-life problems. These problems come from old competitions and training camps. All the problems have links to online judges, where one can test their solutions.

We should also mention how competitive programming differs from research in algorithms. In research, people solve open-ended questions. In sports programming

contestants solve (in most cases) well-established problems with known solutions (prepared beforehand). Sports programming in some way simulates the research process, with many caveats: you are asked to write a solution to the problem in some programming language (in a limited time), but you do not have to write a formal proof of your solution.

Acknowledgements

I would like to thank my supervisor, Dr. Jakub Radoszewski, for his guidance through each stage of the process of writing this thesis.

I would also like to acknowledge Prof. Krzysztof Loryś and Dr. Rafał Nowak for inspiring my interest in algorithmics and sports programming.

Finally, I want to thank all my friends who helped me write this thesis by sharing their knowledge and experience, and by providing their feedback: Kamil Dębowski, Jarosław Kwiecień, Wojciech Nadara, Mateusz Radecki, Mateusz Rzepecki, Marek Sokołowski, and Marcin Smulewicz.

1.1. Competitions and platforms

In this section, we discuss selected international programming competitions and platforms.

International Olympiad in Informatics (IOI)

`https://ioinformatics.org/`

International Olympiad in Informatics is the most famous programming competition for high-school students. The competition started in 1989 in Bulgaria and is being organized annually since then, each year in a different location. Each participating country can send a representation consisting of up to four high school students, but students still participate individually. Nowadays (as of 2020), 87 countries participate in the IOI.

This competition consists of two sessions (in two different days). During each session, contestants are solving 3 problems in 5 hours. For each problem, contestants can get up to 100 points. Problems are also divided into subtasks with different scores. IOI problems are famous for being non-standard, which includes open-input problems (the contestants are given a set of inputs which they can examine, and their task is to produce the output files during the competition), or communication problems

(contestants have to write two separate programs that communicate with each other using a provided interface).

Country representations for IOI are selected through national competitions. Some of these olympiads are well-known in the community and they often provide training opportunities and materials available for everyone:

- Croatian Open Competition in Informatics (COCI) – series of monthly online competitions available on `https://hsin.hr/coci/`,
- Japanese Olympiad in Informatics (JOI) – mirror contests from the final round of the olympiad and the spring selection camp are open to everyone, `https://www.ioi-jp.org/`,
- Polish Olympiad in Informatics (POI) – after each edition, problems are translated into English and published on `https://szkopul.edu.pl/`,
- USA Computing Olympiad (USACO) – monthly contests in several divisions (varying in difficulty) available on `http://usaco.org/`.

There are also various regional competitions for high school students with the IOI format:

- Asia-Pacific Informatics Olympiad (APIO) – online contest for countries from South Asia and Western Pacific, first held in 2007, `http://apio-olympiad.org/`),
- Baltic Olympiad in Informatics (BOI) – onsite regional contest for Nordic and Baltic countries, first held in 1995,
- Balkan Olympiad in Informatics (also abbreviated to BOI) – onsite regional contest for countries from Balkan region, first held in 1993,
- Central European Olympiad in Informatics (CEOI) – onsite regional contest for Central European countries, first held in 1994, `http://ceoi.inf.elte.hu/`.

Contestants of the IOI are awarded medals similarly to other scientific olympiads: 1/12 participants with the highest score get the gold medal, next 1/6 get silver medal, and next 1/4 get bronze medal. In total, top 50% of all participants are awarded medals.

In chapter 6: Dynamic programming optimizations, we solve two problems from the IOI: *Batch scheduling* from IOI 2012 and *Aliens* from IOI 2016. These problems are the first known appearances of some techniques that were later frequently used in various competitions.

IOI Winners from the last 10 years

2019	Benjamin Qi	United States of America
2018	Benjamin Qi	United States of America
2017	Yuta Takaya	Japan
2016	Ce Jin	China
2015	Jeehak Yoon	Republic of Korea
2014	Ishraq Huda	Australia
2013	Lijie Chen	China
2012	Johnny Ho	United States of America
2011	Gennady Korotkevich	Belarus
2010	Gennady Korotkevich	Belarus

International Collegiate Programming Contest (ICPC)

`https://icpc.baylor.edu/`

International Collegiate Programming Contest is a team competition for university students. Each team consists of three students from the same university.

Teams have to solve about a dozen of problems during a 5-hour window. Problems are graded binary – the solution has to pass all the tests to be considered correct. The members of one team have to share one computer.

ICPC is a multi-staged contest, starting with regional level competitions. The best teams are invited to the World Finals, happening annually. In total, in 2019, almost 60 000 participants from 103 countries participated in over 400 regional competitions. 135 teams were invited to the 2019 World Finals.

Teams are ranked according to the number of problems solved. In case of ties, the total time needed to solve these problems is used as a tie-breaker. The total time is calculated as a sum of times needed to accept each solved problem, increased by a 20 minute penalty for each incorrect submission for these problems. Usually, top 4 teams at the World Finals receive gold medals, next 4 teams get silver medals, and next 4 teams – bronze medals, resulting in total of 12 teams getting awards.

In chapter 6 (Dynamic programming optimizations), we discuss problem *Money for Nothing* from ICPC World Finals 2017, and in chapter 8 (Matroids), we solve problem *Coin Collector* from ICPC regional contest – 2011 Southwestern Europe Regional Contest.

ICPC champions from the last 10 years

2019	Mikhail Ipatov Vladislav Makeev Grigory Reznikov	Moscow State University
2018	Mikhail Ipatov Vladislav Makeev Grigory Reznikov	Moscow State University
2017	Ivan Belonogov Ilya Zban Vladimir Smykalov	St. Petersburg ITMO University
2016	Stanislav Ershov Alexey Gordeev Igor Pyshkin	St. Petersburg State University
2015	Gennady Korotkevich Artem Vasilyev Borys Minaiev	St. Petersburg ITMO University
2014	Dmitry Egorov Pavel Kunyavskiy Egor Suvorov	St. Petersburg State University
2013	Mikhail Kever Gennady Korotkevich Niyaz Nigmatullin	St. Petersburg ITMO University
2012	Eugeny Kapun Mikhail Kever Niyaz Nigmatullin	St. Petersburg ITMO University
2011	Luyi Mo Zejun Wu Jialin Ouyang	Zhejiang University
2010	Bin Jin Zhao Zheng Zhuojie Wu	Shanghai Jiaotong University

AtCoder

`https://atcoder.jp`

AtCoder is a programming platform based in Japan. AtCoder as a company and a platform existed since 2011, but they hosted mostly Japanese contests. They transformed into an international platform in 2016, as they started marketing globally and providing problem statements in English and Japanese. They host three types of official contests:

- AtCoder Grand Contest (AGC) – the most difficult and challenging contests on the platform. These contests don't have a regular schedule, but they are conducted mostly on a monthly basis. The results from these contests are counted towards choosing the finalists of the onsite contest called *AtCoder World Tour Finals*.
- AtCoder Regular Contest (ARC) – regular contests, similar to AGC, but slightly easier. These contests are often sponsored by external companies to serve as a recruiting opportunity.
- AtCoder Beginner Contest (ABC) – these contests contain easy and educational problems, mostly for those who are new to competitive programming. They are conducted more-or-less weekly.

In chapter 2 (Extensions of binary search), we discuss problem *Stamp Rally* from AtCoder Grand Contest 002.

AtCoder World Tour Finals winners

2019	Yuhao Du	China

Codechef

`https://codechef.com`

In 2009, the educative initiative Codechef was founded by Directi. The goal of this initiative was to improve and expand the Indian programming community. In 2010, Codechef launched the program with a goal to get an Indian college team to win the ICPC contest. This program was later extended to Indian IOI participants.

Each month, Codechef hosts three types of competitions:

- Long Challenge – 10 days long individual competition with around 8 problems, where usually one of the problems is focused on optimization (it serves as a kind of a tie-breaker).
- Cookoff – individual competition with 5 questions, lasting 2.5 hours with medium difficulty.
- Lunchtime – a contests targeted to beginners with 4 questions for 3 hours.

Snackdown is an annual competition by Codechef, first organized (as a local contest) in 2010. After a five-year break, Snackdown returned in 2015 as a global competition. It is a team competition (each team consists of two contestants). In 2019, over 27 000 teams (and 40 000 individuals) participated in Snackdown.

In chapter 3 (Segment trees rediscovered), we discuss problem *Chef and Swaps* from Codechef September Challenge 2014.

Snackdown champions in the last 5 years

2019	Gennady Korotkevich	Belarus
	Borys Minaiev	Russia
2017	Alex Danilyuk	Russia
	Oleg Merkurev	Russia
2016	Gennady Korotkevich	Belarus
	Borys Minaiev	Russia
2015	Apia Du	China
	Yinzhan Xu	China

Codeforces

`https://codeforces.com`

Codeforces is a Russian-based platform, launched in 2009. The platform was founded by a group of programmers from Saratov State University led by Mikhail Mirzayanov and it is currently developed at ITMO University in Sankt Petersburg, Russia.

Codeforces holds frequent regular rounds. Contestants are rated using ranking system similar to Elo rating system (known from chess). Rounds are often targeted to people from specific ranges of rating (called divisions).

Codeforces is famous for its frequent competitions, active community, and its platform for developing problems and contests called „Polygon" (`https://polygon.codeforces.com/`).

In chapter 3 (Segment trees rediscovered), we talk about two problems from Codeforces: *The Child and Sequence* from CF round #250, and *Destiny* from CF Round #429.

Facebook Hacker Cup

`https://facebook.com/codingcompetitions/hacker-cup/`

Facebook Hacker Cup is an annual algorithmic competition organized by Facebook, which started in 2011. The competition consists of three online elimination rounds, concluded by the onsite finals in one of the Facebook offices.

A unique feature of the Facebook Hacker Cup is that during each round when a contestant wants to submit their solution, they are given the input data and they have to run the solution to produce the output file in a short time (a couple of minutes) which is then submitted to the platform. Because of this, contestants can use any programming language they wish.

Facebook Hacker Cup winners

2019	Gennady Korotkevich	Belarus
2018	Mikhail Ipatov	Russia
2017	Petr Mitrichev	Russia
2016	Makoto Soejima	Japan
2015	Gennady Korotkevich	Belarus
2014	Gennady Korotkevich	Belarus
2013	Petr Mitrichev	Russia
2012	Roman Andreev	Russia
2011	Petr Mitrichev	Russia

Google Code Jam

`https://g.co/codejam/`

Google Code Jam is an annual competition organized by Google. First edition took place in 2003, with prize pool of 20 000 USD. Up to 2007, Code Jam was conducted on TopCoder platform (see below). After that, Google developed its own platform for coding competitions. Up to 2017, contestants had to download the input file and run the program locally to produce output that should be uploaded back. In 2019, a new version of the platform was introduced, where contestants are submitting the source code, which is run on the platform.

Between 2015 and 2018, a new competition format focused on distributed algorithms was run in parallel to Code Jam, called Distributed Code Jam. Google is also organizing two other algorithmic competitions:

- Google Hash Code (`https://g.co/hashcode/`) – team-based programming competition focused on solving optimization problems,
- Google Kick Start (`https://g.co/kickstart/`) – individual competition with slightly easier problems compared to the regular Code Jam. It consists of several independent rounds (8 in 2019), and it serves mostly as a recruitment event.

In 2019, over 74 000 contestants participated in Google Code Jam.

Some GCJ problems led to further academic research; for an example, see [Dymchenko and Mykhailova, 2015].

Google Code Jam winners in the last 10 years

2019	Gennady Korotkevich	Belarus
2018	Gennady Korotkevich	Belarus
2017	Gennady Korotkevich	Belarus
2016	Gennady Korotkevich	Belarus
2015	Gennady Korotkevich	Belarus
2014	Gennady Korotkevich	Belarus
2013	Ivan Metelsky	Belarus
2012	Jakub Pachocki	Poland
2011	Makoto Soejima	Japan
2010	Egor Kulikov	Russia

Distributed Code Jam winners

2018	Mateusz Radecki	Poland
2017	Andrew He	United States of America
2016	Bruce Merry	South Africa
2015	Bruce Merry	South Africa

Topcoder

`https://topcoder.com/`

Topcoder (known to 2013 as TopCoder) is an American company founded in 2001 by Jack Hughes as a crowdsourcing company, selling community services to business clients [Lakhani et al., 2010].

Topcoder organizes regular algorithmic challenges which are known as "Single Round Matches" (SRMs). Each SRM has three problems in two divisions. Each round concludes in a "challenge phase", where contestants can "hack" other contestants' solutions, i.e. provide a testcase on which the rival's solution will fail (produce wrong output, exceed limits, and so on).

Topcoder also holds Topcoder Open. The competition consists of several elimination rounds, with finals hosted in various cities across the US. Since 2015, Topcoder started organizing regional events accompanying the finals.

Besides sports programming, Topcoder focuses on many different areas. As of 2019, Topcoder Open has 6 tracks: Algorithm, Development, First2Finish (quick chal-

lenge asking competitors to fix bugs or do a small task), Marathon (long optimization problems), UI Design, and QA Competition.

TCO Algorithm Champions in the last 10 years

2019	Gennady Korotkevich	Belarus
2018	Petr Mitrichev	Russia
2017	Yuhao Du	China
2016	Makoto Soejima	Japan
2015	Petr Mitrichev	Russia
2014	Gennady Korotkevich	Belarus
2013	Petr Mitrichev	Russia
2012	Egor Kulikov	Russia
2011	Makoto Soejima	Japan
2010	Makoto Soejima	Japan

Yandex Algorithm

`https://algorithm.contest.yandex.com/`

Yandex is a Russian-based company specializing in Internet-related products and services, including web search, translator, maps, and many others. In 2013, they started organizing an algorithmic competition called "Yandex Algorithm". The contest consists of a qualification round, three elimination rounds and onsite finals.

Besides algorithms, they launched two other tracks in this competition: optimization track and machine learning track.

YA winners

2018	Gennady Korotkevich	Belarus
2017	Gennady Korotkevich	Belarus
2016	Egor Kulikov	Russia
2015	Gennady Korotkevich	Belarus
2014	Gennady Korotkevich	Belarus
2013	Gennady Korotkevich	Belarus

Chapter 2

Extensions of binary search

Binary search is one of the oldest known algorithms. It owes its fame to the fact that the idea of sorting objects in order to decrease the amount of time needed to search for given elements is natural and intuitive and it was known even in the ancient times [Knuth, 1997], while the first implementation of this algorithm comes from 1962. [Bottenbruch, 1962]

In this chapter, we will discuss various extensions and applications of this very famous algorithm.

For clarity, we will always consider the binary search algorithm on intervals with inclusive left end and exclusive right end, i.e. $[P, Q)$. So our implementation of the solution to the typical problem with a guessing game, where one of the players chooses a number (let us say between 1 and 100) and the second one has to guess this number by asking questions "Is the chosen number greater than or equal to x?", will look as follows:

```
// in our example P = 1 and Q = 101
Function guessTheNumber(P, Q):
    while Q − P > 1
        mid ← ⌊½(P + Q)⌋;
        // ask about the number in the middle
        if isGreaterOrEqual(mid)
            Q ← mid;
        else
            P ← mid;
    return P
```

2.1. Parallel binary search

Imagine that we want to answer Z queries about the index of the largest value less than or equal to some X_i (for $i = 1, 2, \ldots, Z$) in some sorted 0-indexed array A. Each query can be answered using binary search.

Let us consider the following array A:

A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7
1	3	5	7	9	9	13	15

with queries: $X = [8, 11, 4, 5]$. We can use binary search for each query sequentially.

query	$X_1 = 8$	$X_2 = 11$	$X_3 = 4$	$X_4 = 5$
step 1	answer in $[0, 8)$ check A_4 $X_1 < A_4 = 9$	answer in $[0, 8)$ check A_4 $X_2 \geq A_4 = 9$	answer in $[0, 8)$ check A_4 $X_3 < A_4 = 9$	answer in $[0, 8)$ check A_4 $X_4 < A_4 = 9$
step 2	answer in $[0, 4)$ check A_2 $X_1 \geq A_2 = 5$	answer in $[4, 8)$ check A_6 $X_2 < A_6 = 13$	answer in $[0, 4)$ check A_2 $X_3 < A_2 = 5$	answer in $[0, 4)$ check A_2 $X_4 \geq A_2 = 5$
step 3	answer in $[2, 4)$ check A_3 $X_1 \geq A_3 = 7$	answer in $[4, 6)$ check A_5 $X_2 \geq A_5 = 9$	answer in $[0, 2)$ check A_1 $X_3 \geq A_1 = 3$	answer in $[2, 4)$ check A_3 $X_4 < A_3 = 7$
step 4	answer in $[3, 4)$ $index = 3$	answer in $[5, 6)$ $index = 5$	answer in $[1, 2)$ $index = 1$	answer in $[2, 3)$ $index = 2$

We processed this table by columns (queries), but notice that in each row we often repeat access to certain values of our array. This does not make any difference in our small example problem (as we can access all elements in $O(1)$), but in more complex problems this might be essential to solve such problems efficiently (as we will show in a moment). To limit access to the values, we can process the table by rows (steps). Moreover, note that we can arbitrarily choose the order in which we answer

questions in a single row. Let us look at the pseudocode implementing this approach.

```
N ← len(A);
for i ← 0 to N − 1
    P[i] ← 0;
    Q[i] ← N;
for step ← 1 to log N
    // important_values will be a map from value to indices
    // of queries asking for this value
    important_values ← map();
    for i ← 0 to N − 1
        mid[i] ← ⌊½(P[i] + Q[i])⌋;
        important_values[mid[i]].append(i);
    for (value, queries) in important_values in some order
        for query in queries
            if value > X_query
                P[query] ← mid[query];
            else
                Q[query] ← mid[query];
```

We will show now how this approach can be useful in some real-life problems.

Problem Meteors

18th Polish Olympiad in Informatics.
Limits: 35s, 64MB.
`https://kostka.dev/sp/met`

Byteotian Interstellar Union (BIU) has recently discovered a new planet in a nearby galaxy. The planet is unsuitable for colonisation due to strange meteor showers, which on the other hand make it an exceptionally interesting object of study.

The member states of BIU have already placed space stations close to the planet's orbit. The stations' goal is to take samples of the rocks flying by. The BIU Commission has partitioned the orbit into m sectors, numbered from 1 to m, where the sectors 1 and m are adjacent. In each sector there is a single space station, belonging to one of the n member states.

Each state has declared a number of meteor samples it intends to gather before the mission ends. Your task is to determine, for each state, when it can stop taking samples, based on the meter shower predictions for the years to come.

Input

The first line of the standard input gives two integers, n and m ($1 \leq n, m \leq 300\,000$), separated by a single space, that denote, respectively, the number of BIU member states and the number of sectors the orbit has been partitioned into.

In the second line there are m integers o_i ($1 \leq o_i \leq n$), separated by single spaces, that denote the states owning stations in successive sectors.

In the third line there are n integers p_i ($1 \leq p_i \leq 10^9$), separated by single spaces, that denote the numbers of meteor samples that the successive states intend to gather.

In the fourth line there is a single integer k ($1 \leq k \leq 300\,000$) that denotes the number of meteor showers predictions. The following k lines specify the (predicted) meteor showers chronologically. The i-th of these lines holds three integers l_i, r_i, a_i (separated by single spaces), which denote that a meteor shower is expected in sectors $l_i, l_{i+1}, \ldots, r_i$ (if $l_i \leq r_i$) or sectors $l_i, l_{i+1}, \ldots, m, 1, \ldots, r_i$ (if $l_i > r_i$), which should provide each station in those sectors with a_i meteor samples ($1 \leq a_i \leq 10^9$).

In tests worth at least 20% of the points it additionally holds that $n, m, k \leq 1000$.

Output

Your program should print n lines on the standard output. The i-th of them should contain a single integer w_i, denoting the number of shower after which the stations belonging to the i-th state are expected to gather at least p_i samples, or the word `NIE` (Polish for *no*) if that state is not expected to gather enough samples in the foreseeable future.

Example

For the input data:

```
3 5
1 3 2 1 3
10 5 7
3
4 2 4
1 3 1
3 5 2
```

the correct result is:

```
3
NIE
1
```

Solution

First let us focus of finding the number of showers satisfying one particular state. A naive way to check this is to simulate all the showers sequentially and count the number of meteor samples in owned sectors. This solution works in $O(mk)$ and can be sped up to $O((m+k)\log m)$ if we use the fact that the samples appear only in continuous segments. We can use a standard segment tree to simulate the meteor rains (add value to a segment) and check how many samples we have in each sector owned by a state.

Instead of asking what is the minimal number of showers w_i for each state, we will check if w showers are enough, which can be done using binary search. Now instead of sequentially trying to find the answer for each state, we can do it in parallel, as mentioned above: in every step we iterate over sorted candidates for w_i.

Let us consider the time complexity of this solution. Our binary search uses $\log k$ phases. In each phase, we have to simulate the process as described above (which we can do in $O((m+k)\log m)$. In total, this solution works in $O((m+k)\log m \log k)$ time.

Problem Stamp Rally

AtCoder Grand Contest 002.
Limits: 2s, 256MB.
`https://kostka.dev/sp/sta`

We have an undirected graph with N vertices and M edges. The vertices are numbered 1 through N, and the edges are numbered 1 through M. Edge i connects vertices a_i and b_i. The graph is connected.

On this graph, Q pairs of brothers are participating in an activity called Stamp Rally. The Stamp Rally for the i-th pair will be as follows:

- One brother starts from vertex x_i, and the other starts from vertex y_i.
- The two explore the graph along the edges to visit z_i vertices in total, including the starting vertices. Here, a vertex is counted only once, even if it is visited multiple times, or visited by both brothers.
- The score is defined as the largest index of the edges traversed by either of them. Their objective is to minimize this value.

Find the minimum possible score for each pair.

Constraints

- $3 \leq N \leq 10^5$
- $N - 1 \leq M \leq 10^5$
- $1 \leq a_i < b_i \leq N$
- The given graph is connected.
- $1 \leq Q \leq 10^5$
- $1 \leq x_j < y_j \leq N$
- $3 \leq z_j \leq N$

Input

The input is given in the following format:

N M
a_1 b_1
a_2 b_2
$\vdots$
a_M b_M
Q
x_1 y_1 z_1
x_2 y_2 z_2
$\vdots$
x_Q y_Q z_Q

Output

Print Q lines. The i-th line should contain the minimum possible score for the i-th pair of brothers.

Example

For the input data:

```
5 6
2 3
4 5
1 2
1 3
1 4
1 5
6
2 4 3
2 4 4
2 4 5
1 3 3
1 3 4
1 3 5
```

the correct result is:

```
1
2
3
1
5
5
```

Solution

We will find a way to answer all the queries simultaneously using parallel binary search.

If we want to answer just one query, we can simply ask if score s is enough to visit z_i vertices. To do so, we can look at our original graph, considering only edges with labels less than or equal to s, and determine connected components. The easiest way to do it is to keep disjoint sets with vertices of components (aka union-find). If number of vertices in all of components that brothers can visit (we will have at most two components, depending if they end up in the same component) is at least z_i, then score is at most s. Therefore we can use binary search to minimize the score.

If we binary search independently for each query, we end up with a solution working in $O(QM \log M \lg^* N)$ time. We can speed it up by parallelizing our binary search to end up with $O((Q+M) \log M \lg^* N)$ time complexity.

2.2. Fractional cascading

Fractional cascading was first introduced in 1986 in [Chazelle and Guibas, 1986a] and [Chazelle and Guibas, 1986b] as a technique used in computational geometry.

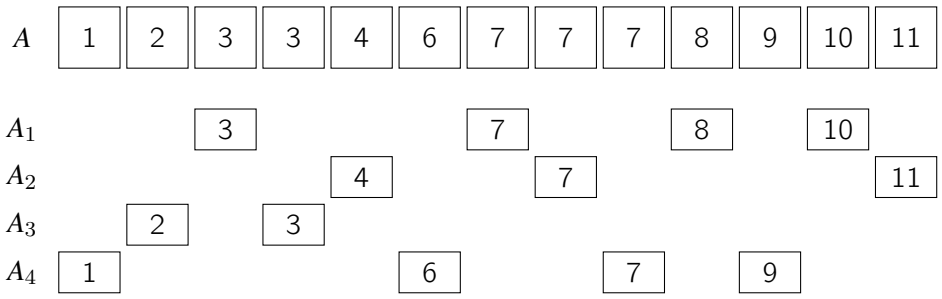
To show how this method works, let us consider the following problem. We are given k sorted lists of integers (catalogs) of lengths $n_1, n_2, \ldots, n_k$ and we want to answer the following queries: find the smallest number greater than or equal to some x in each of these lists.

For example, let us consider four lists:

- $A_1 = [3, 7, 8, 10]$,
- $A_2 = [4, 7, 11]$,
- $A_3 = [2, 3]$,
- $A_4 = [1, 6, 7, 9]$.

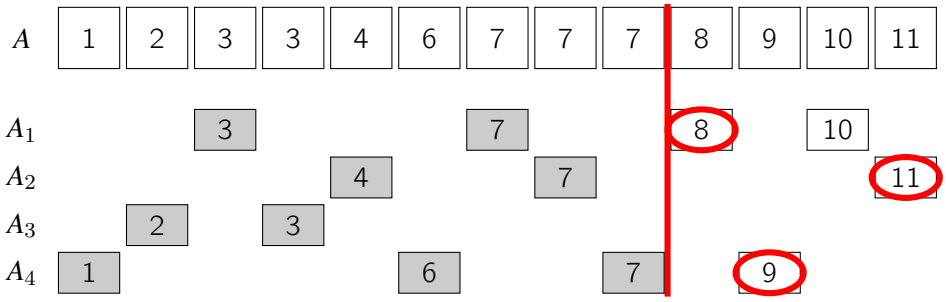
For $x = 8$, we should return that 8, 11, 9 are answers in A_1, A_2, and A_4, respectively, and for A_3 there is no such integer.

The first solution is to simply use binary search on each of the input lists. Then the time complexity will be (in the worst case when all the lists have similar lengths): $O(k \log \frac{N}{k})$ where $N = \sum_{i=1}^{k} n_i$. In search for a better solution, we can think of merging all the lists into one large list and perform only one binary search on this new list. In our example, we will consider the following list denoted by A.



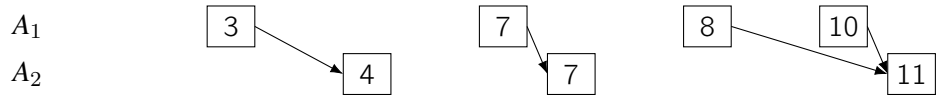
Note that this list can be constructed in $O(N + N \log k)$ time by using a method similar to the one used in merging two halves in the merge sort algorithm.

Now we still need to get answers for all the lists. We binary search over A trying to find an integer greater than or equal to x. This takes $O(\log N)$. Now we notice that we need to find the first element from each list equal or to the right of this element. It is easy to notice that the index of the answer for each list is how many elements from this list are to the left plus one. To answer quickly, we can construct an additional list with all the indices of the answers for each element in our new list. For instance, in our case for $x = 8$, we can store $[2, 2, \perp, 3]$.

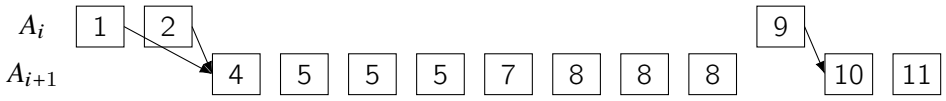


This allows us to answer queries in total $O(\log N + k)$ time, but using additional $O(Nk)$ memory, as we need to store the answer for each element in A.

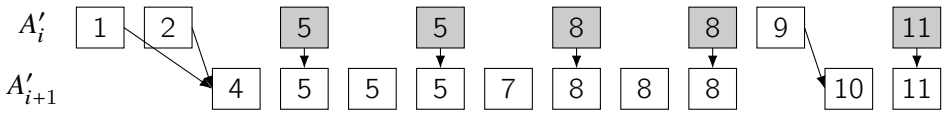
Let us try to improve this idea. Instead of keeping all the answers for all the elements, let us just add pointers where we can deduct answer from the answer of the previous list. For example, if we know that 8 is the answer in A_1, then answer in A_2 has to be 11, as this is the smallest element greater than 8.



Unfortunately, in some cases these pointers will not help us that much, especially when we have long segments with elements only from one catalog, for example:



It turns out that can we merge these two ideas and come up with a decent solution. Let us create a new set of lists $A'_1, A'_2, \ldots, A'_k$, where $A'_k = A_k$, and for every $i = k-1, k-2, \ldots, 1$: A'_i will be equal to the union of A_i and every *other* element of A'_{i+1} (in gray on the picture below). This way our pointers will always be useful, as they will be pointing either to the element itself, or to the proper successor in the next catalog.

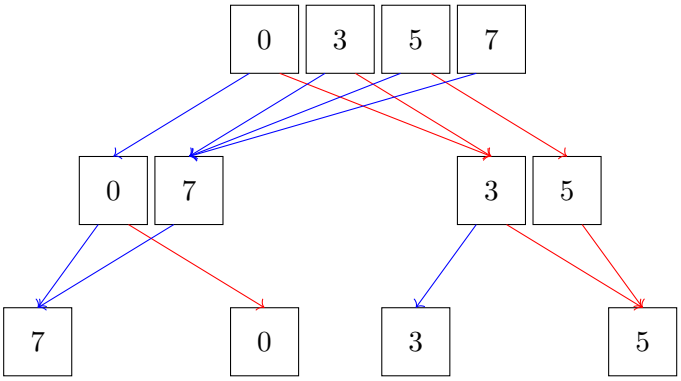


Now, we can use binary search only on the first catalog (A'_1) and then use our pointers to check the answers in other arrays. The answer for each following catalog will be approximately an element pointed by our pointer, i.e. either the element being pointed to, or its predecessor (in the example above, if we look for $x = 7$, the answer for A_i is 8, while the answer for A_{i+1} is equal to 7). Well, almost. We might end up

with elements not being a member of our original list. To fix this, while we build A'_i we can add pointers to the next element which originally comes from A_i.

We just have to estimate how many new elements we will create. In general, we can see that $|A'_1| = |A_1| + \frac{1}{2}|A_2| + \frac{1}{4}|A_3| + \ldots$, and in total: $|A'_1| + |A'_2| + \ldots = \sum_{i=1}^{k} |A_i|(1 + \frac{1}{2} + \frac{1}{4} + \ldots) \leq 2N$, so we will have at most N new elements! In the end, we end up with time complexity for each query is $O(\log N + k)$, with additional $O(N)$ memory.

We often use this technique in segment trees. Imagine that in each node in a segment tree we store a sorted list of elements and every node contains a list which was constructed by merging two lists belonging to its children. Here, if we want to do binary search on these lists, normally we perform binary search on each list separately. We can speed it up by storing two pointers for each element. One pointer will point to the corresponding answer (smallest index greater than or equal to the current element) in the left child, while the second will point to the answer in the right child. This way, we just need to perform one binary search in the root and then use these pointers for finding answers in every other node that we need to check.



These pointers can be easily computed while we merge our lists without any additional burden. In general, when we need to touch around $O(\log N)$ nodes in our tree, this speeds up our queries from $O((\log N)^2)$ time to simply $O(\log N)$ time, but we pay with extra $O(N)$ memory.

Problem Subtree Minimum Query

Educational Codeforces Round 33.
Limits: 6s, 512MB.
`https://kostka.dev/sp/smq`

You are given a rooted tree consisting of n vertices. Each vertex has a number written on it; number a_i is written on vertex i.

Let's denote $d(i, j)$ as the distance between vertices i and j in the tree (that is,

the number of edges in the shortest path from i to j). Also let's call the k-blocked subtree of vertex x the set of vertices y such that both these conditions are met:

- x is an ancestor of y (every vertex is an ancestor of itself);
- $d(x, y) \le k$.

You are given m queries to the tree. i-th query is represented by two numbers x_i and k_i, and the answer to this query is the minimum value of a_j among such vertices j such that j belongs to k_i-blocked subtree of x_i.

Write a program that would process these queries quickly!

Input

The first line contains two integers n and r ($1 \le r \le n \le 100\,000$) – the number of vertices in the tree and the index of the root, respectively.

The second line contains n integers $a_1, a_2, \ldots, a_n$ ($1 \le a_i \le 10^9$) – the numbers written on the vertices.

Then $n - 1$ lines follow, each containing two integers x and y ($1 \le x, y \le n$) and representing an edge between vertices x and y. It is guaranteed that these edges form a tree.

Next line contains one integer m ($1 \le m \le 10^6$) – the number of queries to process. *Note that the queries are given in a modified way.*

Then m lines follow, i-th line containing two numbers p_i and q_i, which can be used to restore i-th query ($1 \le p_i, q_i \le n$).

i-th query can be restored as follows. Let $last$ be the answer to the previous query (or 0 if $i = 1$). Then $x_i = ((p_i + last) \bmod n) + 1$, and $k_i = (q_i + last) \bmod n$.

Output

Print m integers. The i-th of them has to be equal to the answer to the i-th query.

Example

For the input data:

```
5 2
1 3 2 3 5
2 3
5 1
3 4
4 1
2
1 2
2 3
```

the correct result is:

```
2
5
```

Solution

First, we use DFS to calculate depth of each vertex and the size of its subtree. Then we build a segment tree on these values (depths and sizes) in the order generated by an Eulerian path, in which each node will store the list of numbers written on vertices of some contiguous subsequence from our path. We can sort these lists by depth of the corresponding vertices (we sort in linear time by merging sorted lists). As we will be always looking for the minimum value restricted by some depth, we can just store prefix minimums.

Now, for each query, we just need to find all the nodes related to the vertices in our subtree (it will be a contiguous subsequence) and then perform binary search on each of the nodes to find the deepest vertex that we can take and get the corresponding minimum (of the prefix up to this vertex). To perform these binary searches, we can use fractional cascading, so we can answer each query in $O(\log n)$ time, so the total time complexity of this solution ends up being $O(n + q \log n)$.

Dynamic Fractional Cascading

Fractional Cascading can also be used on dynamic queries, i.e. for lists (catalogs) that can change in time (we can add/delete/modify elements). This version of the problem is a little bit more complex, as we need to both update the augmented catalogs and the pointers between them, but it is still possible to perform in the same complexity as the static version. For more details, see [Mehlhorn and Näher, 1990].

Chapter 3

Segment trees rediscovered

3.1. Segment tree beats

Segment tree is a really powerful data structure that should be in the repertoire of every sports programmer. The very basic problem that is solvable by this structure can be defined as follows. We are given a sequence of integers $(a_1, a_2, \ldots, a_n)$. On this sequence we define two operations:

1. $update(i, j, p)$ – update all values between indices i and j (inclusive) using some parameter p,
2. $query(i, j)$ – return the aggregated value for a continuous subsequence between indices i and j.

The most common $update$ variants are:

1. + variant: for each value $a_i, a_{i+1}, \ldots, a_j$, add p,
2. $\max$ variant: for each value $a_i, a_{i+1}, \ldots, a_j$, change value a_k to $\max(a_k, p)$. In other words: if the value a_k was less than p, change it to p.

Similarly, we define two $query$ variants:

1. + variant: return the sum $a_i + a_{i+1} + \ldots + a_j$,
2. $\max$ variant: return $\max(a_i, a_{i+1}, \ldots, a_j)$.

Therefore, we have four different variants of the whole problem: $(+, +)$, $(+, \max)$, $(\max, \max)$, and $(\max, +)$, where the first operation in the pair is the variant of the updates, while second one is the variant of the queries. Three of these problems (except for $(\max, +)$) had an easy solution using segment trees with the time complexity

$O(n + m \log n)$, where m is the number of operations (updates and queries). The last variant was causing a lot of problems, until in 2016, a solution was proposed by Ryui Ji, in form of a research report required from the candidates of the Chinese national team for the International Olympiad in Informatics[1]. Below we will describe this approach.

We assume here that the reader has some basic knowledge of the general segment tree with lazy propagation.

Let us once again define the problem. We want to have a structure over the integer sequence $(a_1, a_2, \ldots, a_n)$ that allows us to do the following operations:

1. $update(i, j, p)$, for every k between indices i and j, change a_k to $\max(a_k, p)$,
2. $query(i, j)$, return the sum $a_i + a_{i+1} + \ldots + a_j$.

Let us recollect that each node in the segment tree contains some aggregated values about the segment it covers. In our node, we will keep the following values:

- min - the minimum value in the segment,
- $count\_min$ - how many such minimum values we have in this segment,
- $second\_min$ - second minimum value in this segment (or infinity, if all elements are equal, and the second minimum does not exist),
- sum - the sum of all elements in the segment,
- $lazy$ - a variable used for lazy propagation – if we want to go deeper into this subtree, we have to first update (maximize) the children of the current node with the value $lazy$ (if it is not empty).

Please note that all these values (except the last one) can be modified by lazy tags in their ancestors.

You can check that from all these values given for two children, we can easily compute values for the parent.

How we can $update$ these values? We have three possible cases:

- $p \leq min$. In this case, nothing will be changed, as even the smallest value in the segment is greater than the parameter p.
- $min < p < second\_min$. In this case only values equal to min will change to p. We have to update min to p and change sum to $sum' + (p - min') \cdot count\_min$, where sum' and min' represent values before the update. We also should update the value $lazy$ to p, if it was not set, or was smaller than p.

[1] https://codeforces.com/blog/entry/57319

- $p \geq second\_min$. This is the only case that we cannot solve in this node, and we have to call the $update$ recursively in both of our children. After that, we update the values in our node in $O(1)$ time.

If the values are updated properly, sum over all $O(\log n)$ nodes covering the segment in the $query$ will be the answer for this query. Please note that sometimes we also have to propagate $lazy$ values further down, like in the typical segment tree.

Now, the correctness of all the operations on this data structure should not be hard to see, but what about the time complexity? To calculate this complexity, we would like to count how many times we have to call $update$ recursively, i.e. be in the third case mentioned above ($p \geq second\_min$). To estimate the complexity, we will introduce a potential function for the nodes in the tree. Let us say that the potential $P(a)$ of the node a is the number of distinct integers in the segment covered by this node (without considering lazy updates). In the beginning, the total potential $\sum_{a \in nodes} P(a) = O(n \log n)$, as in the worst case every element is different and is covered by $O(\log n)$ nodes. Now, every update can increase the potential by at most $O(\log n)$, as we can add a new value in at most $O(\log n)$ nodes.

Finally, let us notice that every time we fall into the third case and call our $update$ recursively, we will change at least two values in this node (min and $second\_min$) to p (as $p \geq second\_min$), therefore the number of distinct integers in this node (and its subtree) will decrease. We can see now that we can decrease our potential at most $O((n+q)\log n)$ times. Therefore the overall amortized time complexity is indeed $O((n+q)\log n)$, which was a pretty surprising result.

Problem The Child and Sequence

Codeforces Round #250.
Limits: 4s, 256MB.
`https://kostka.dev/sp/chi`

At the children's day, the child came to Picks's house and messed his house up. Picks was angry at him. A lot of important things were lost, in particular the favorite sequence of Picks.

Fortunately, Picks remembers how to repair the sequence. Initially he should create an integer array $a[1], a[2], \ldots, a[n]$. Then he should perform a sequence of m operations. An operation can be one of the following:

1. Print operation l, r. Picks should write down the value of $\sum_{i=l}^{r} a[i]$.
2. Modulo operation l, r, x. Picks should perform assignment $a[i] = a[i] \mod x$ for each i ($l \leq i \leq r$).

3. Set operation k, x. Picks should set the value of $a[k]$ to x (in other words perform an assignment $a[k] = x$).

Can you help Picks to perform the whole sequence of operations?

Input

The first line of input contains two integers: n, m $(1 \le n, m \le 10^5)$. The second line contains n integers, separated by spaces: $a[1], a[2], \ldots, a[n]$ $(1 \le a[i] \le 10^9)$ – initial values of array elements.

Each of the next m lines begins with a number $type$ $(type \in \{1, 2, 3\})$.

- If $type = 1$, there will be two integers more in the line: l, r $(1 \le l \le r \le n)$. This means that we consider the first operation (print) with values l and r.
- If $type = 2$, there will be three integers more in the line: l, r, x $(1 \le l \le r \le n; 1 \le x \le 10^9)$, which correspond to the second operation.
- If $type = 3$, there will be two integers more in the line: k, x $(1 \le k \le n; 1 \le x \le 10^9)$, which correspond the third operation.

Output

For each operation 1, please print a line containing the answer. Notice that the answer may exceed the 32-bit integer.

Examples

For the input data:	the correct result is:
5 5 1 2 3 4 5 2 3 5 4 3 3 5 1 2 5 2 1 3 3 1 1 3	8 5

Note:

- At first, $a = \{1, 2, 3, 4, 5\}$.
- After operation 1, $a = \{1, 2, 3, 0, 1\}$.

- After operation 2, $a = \{1, 2, 5, 0, 1\}$.
- At operation 3, $2 + 5 + 0 + 1 = 8$.
- After operation 4, $a = \{1, 2, 2, 0, 1\}$.
- At operation 5, $1 + 2 + 2 = 5$.

And for the input data:	the correct result is:
10 10	49
6 9 6 7 6 1 10 10 9 5	15
1 3 9	23
2 7 10 9	1
2 5 10 8	9
1 4 7	
3 3 7	
2 7 9 9	
1 2 4	
1 6 6	
1 5 9	
3 1 10	

Solution

We will show how we can use the general segment tree to solve this problem. We can see that the first and the last operation are "standard", so let us focus on the modulo operations. The key observation is a fact that if we use the operation and some value $a[i]$ will change, it will be at least halved, i.e. the new $a[i]$ is at most $\frac{a[i]}{2}$. Because of that, we can change value of $a[i]$ at most $\log_2 a[i]$ times. Let us now introduce the potential function for this problem: $P(i)$ for value $a[i]$ will be exactly $\log_2 a[i]$. Therefore, every time we have to update some value, its potential will decrease by at least one. The total potential at the beginning is, of course, $O(n \log A)$, where A denotes the maximum value of a_i.

Let us not forget about the third operation, that might increase the potential. Fortunately, if we set the value of some element to x, we increase its potential by at most $\log_2 x$.

Therefore, we will keep a traditional segment tree and store sum and max in the nodes. If we try to apply modulo operation on some node, we will check if the maximum value is smaller than the parameter of the operation. If that is the case, then we will change the value, and we know that we will do this not very often.

The total time complexity will be $\mathcal{O}(n \log n \log A)$.

Moreover, we can modify the last operation (assign value) to also work on intervals, i.e. we can set $a[k] = x$ for $l \leq k \leq r$. We will leave solving this extended problem for the reader as an exercise.

3.2. Historic information for sequences

In this section, we will take a look at another problem that is quite difficult to solve by the typical segment tree, but we will show another technique that will help us to solve it simply and efficiently.

Problem The Resistor

Tsinghua Training 2015.
Limits: 2s, 128MB.
`https://kostka.dev/sp/res`

Note: The statement of this problem was slightly simplified.

Dr. Picks designed a complex resistor. The resistor consists of n independent water tanks numbered from 1 to n. Each water tank is cylindrical and has a value at the top and at the bottom which allows the water to flow at the rate of $1\frac{m^3}{s}$ per second. The upper valve of each water tank is connected to the faucet, which can supply water indefinitely, and the lower valve is not connected to anything, allowing water to flow out. In the beginning, in ith water tank, there is a_i cubic meters of water.

Dr. Picks will then need to debug this complex resistor. He will perform the following five operations:

1. Open upper valves of all the water tanks between l and r (inclusive) and let the water flow in for x seconds.
2. Open lower valves of all the water tanks between l and r (inclusive) and let the water flow out for x seconds.
3. Set the amount of water of all the water tanks between l and r to exactly $x\frac{m^3}{s}$.
4. Measure the amount of water in some chosen water tank k.
5. Measure what was the maximum amount of water in some chosen water tank k (as the water-soaked places will leave obvious water stains).

Now, he performed all these operations several times, but he lost his notes. Can you help him determine all the water levels he measured?

Input

The first line of the standard input contains two integers n and m $(1 \leq n, m \leq 500\,000)$ – the number of water tanks and the number of operations that Dr. Picks performed. The next line contains n integers $a_1, a_2, \ldots, a_n$ $(0 \leq a_i \leq 10^9)$ – the initial amount of water in the water tanks. Last m lines contain a description of operations, in one of the following forms:

- $1\ l_i\ r_i\ x_i$ $(1 \leq l_i \leq r_i \leq n, 0 \leq x_i \leq 10^9)$ – this operation indicates opening of all the upper valves of water tanks between l_i and r_i for x_i seconds,
- $2\ l_i\ r_i\ x_i$ $(1 \leq l_i \leq r_i \leq n, 0 \leq x_i \leq 10^9)$ – this operation indicates opening of all the lower valves of water tanks between l_i and r_i for x_i seconds,
- $3\ l_i\ r_i\ x_i$ $(1 \leq l_i \leq r_i \leq n, 0 \leq x_i \leq 10^9)$ – this operation indicates that we set the amount of water in all the water tanks between l_i and r_i to exactly $x \frac{m^3}{s}$.
- $4\ y_i$ $(1 \leq y_i \leq n)$ – this operation indicates measuring the amount of water in y_ith water tank,
- $5\ y_i$ $(1 \leq y_i \leq n)$ – this operation indicates measuring the maximum amount of water that was in y_ith water tank so far.

Output

For each measuring operation, output the measured amount in a separate line.

Example

For the input data:	the correct result is:
5 6 1 2 3 4 5 2 1 3 2 4 1 1 1 4 1 5 3 3 1 5 4 4 2	0 3 4

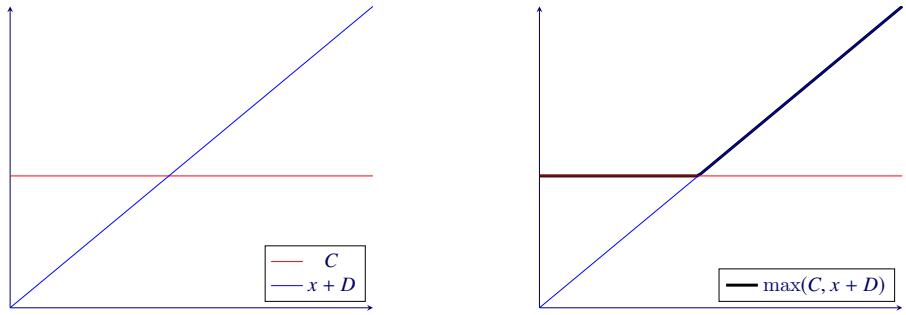
Solution

We have the following operations that change our initial sequence:

1. change a_i to $a_i + x$ for $l \leq i \leq r$,
2. change a_i to $\max(0, a_i - x)$ for $l \leq i \leq r$,
3. change a_i to x for $l \leq i \leq r$.

Moreover, after every operation, we have to keep track of the maximum value for every a_i (we are calling this information the *historic information*). We will say that after every operation, we will update some auxiliary value $m_i = \max(m_i, a_i)$. Then, we have queries about a_i and m_i.

Let us observe that every operation changing the sequence a_i is of the form "change a_i to $\max(C, a_i + D)$", where C and D are some constants. We will represent these operations as functions $f_{C,D}$. For example, the first operation can be represented as $f_{0,x}$, while the third one $f_{x,-\infty}$. Let us think for a moment how this general function $f_{C,D}$ looks like:



These functions have an interesting property that if we apply one such function to another, we still get a function of the same kind. Let us take: $f = f_{C_1,D_1}$ and $g = f_{C_2,D_2}$, then:

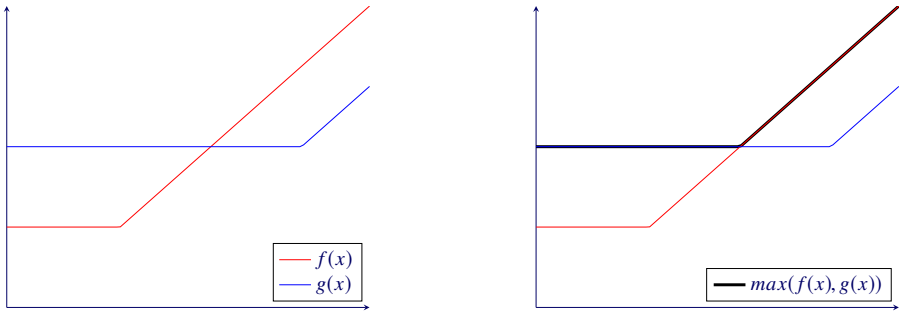
$$g(f(x)) = \max(C_2, \max(C_1, x + D_1) + D_2) = \max(\max(C_2, C_1 + D_2), x + D_1 + D_2).$$

If we take $C' = \max(C_2, C_1 + D_2)$ and $D' = D_1 + D_2$, we have two constants, so we can say that $g(f(x)) = \max(C', x + D')$. Therefore, if we forget about the fifth query (print the maximum), we can simply store two constants C and D for each node in the segment tree to perform all the operations.

What about the last query, the historic information that we have to keep? We cannot do it simply using lazy propagation, as we would have to store many updates, as the values might change several times before we will have to push down a lazy tag.

In this particular problem, we are lucky enough, because it turns out that if we apply maximum to these functions, then we still have a function of the form $f_{C,D}$:

$$\max(f(x), g(x)) = \max(\max(C_1, x + D_1), \max(C_2, x + D_2)) =$$
$$= \max(C_1, C_2, x + D_1, x + D_2) = \max(\max(C_1, C_2), x + \max(D_1, D_2))$$



Therefore, we can also keep a similar function that will represent the maximum value for each possible argument. Let us say that we will keep four values in each node C, D, and their equivalents for historical values C_{max} and D_{max}. Function $f_{C_{max},D_{max}}$ will represent the historical values. If we want to push historical values A_{max} and B_{max} to the node with values (C, D, C_{max}, D_{max}), then we have:

$$f_{C_{max},D_{max}} = \max\left(f_{C_{max},D_{max}}, f_{A_{max},B_{max}}(f_{CD})\right)$$

Meaning that either the historical value does not change (is greater), or we fold the propagated historical value function with the current value. We have shown that both folding and maximum still results in the same type of function.

To conclude, we just need to use a segment tree with lazy propagation. In each node we store four values symbolizing two functions, one for actual value and one for the historical values. For each operation changing the sequence, we lazily update the functions, using the formulas above. When we need to print an answer to some query, we traverse to the leaf, and we apply the corresponding function in this node to the value given at the beginning. Please note that the value itself doesn't matter at all, in particular, we can also answer the following queries: what would be the value of this element if the value at the beginning was different.

3.3. Wavelet trees

A wavelet tree is a relatively new data structure that was introduced by Grossi, Gupta, and Vitter in 2003 [Grossi et al., 2003] to represent compressed suffix arrays. Since then, this data structure has found many applications, from string processing to geometry [Navarro, 2014]. Its applications in sports programming were also mentioned

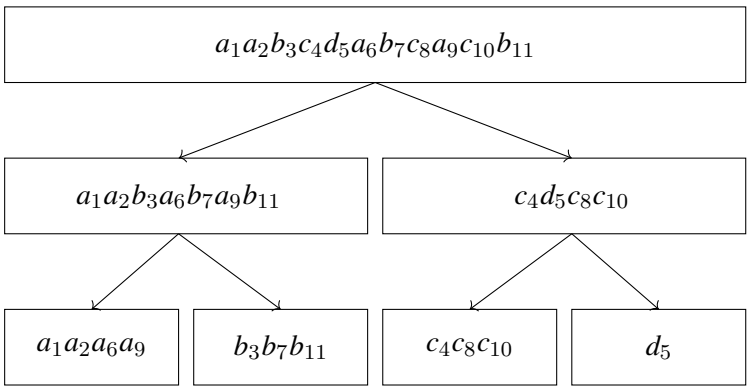
in [Castro et al., 2016]. In this section, we will first introduce the data structure and then quickly move to its applications in real problems.

Wavelet tree

Let us consider the following word: $aabcdabcacb$. For clarification, we will add the position of each letter as its index, so our word will be represented as

$$a_1a_2b_3c_4d_5a_6b_7c_8a_9c_{10}b_{11}.$$

Now we will build a segment tree, but with a slight twist. Each inner node will divide letters into two groups: one group less than or equal to some chosen letter in the alphabetic order, and the second one: all letters greater than this chosen letter. It is really important that we still keep the relative order in these groups. Therefore our tree will look as follows:



Please note that we will have exactly $|\Sigma|$ leaves, where Σ is the alphabet. Almost always we can compress the alphabet first (i.e. map all letters to numbers from 1 to $|\Sigma|$), so we will have at most $p = \min(|\Sigma|, n)$ different elements, where n is the length of the sequence. Then, the height of such a tree will be at most $\log p$, if we always choose a letter "in the middle" as the pivot.

Now, we will try to answer the following queries: given integers l, r, and k, what is the k-th smallest (alphabetically) letter from all letters between indices l and r. In other words, if we will take all letters between l and r and sort them, which letter will be in the k-th place in the sorted sequence (numbered from 1).

To do so, let us change our representation a little bit. Instead of keeping words, we will just remember if each letter is going to the left child (and represent it as 1), or to the right child (0). Now the whole tree can be represented in the following array:

a_1	a_2	b_3	c_4	d_5	a_6	b_7	c_8	a_9	c_{10}	b_{11}
1	1	1	0	0	1	1	0	1	0	1

a_1	a_2	b_3	a_6	b_7	a_9	b_{11}	c_4	d_5	c_8	c_{10}
1	1	0	1	0	1	0	1	0	1	1

a_1	a_2	a_6	a_9	b_3	b_7	b_{11}	c_4	c_8	c_{10}	d_5

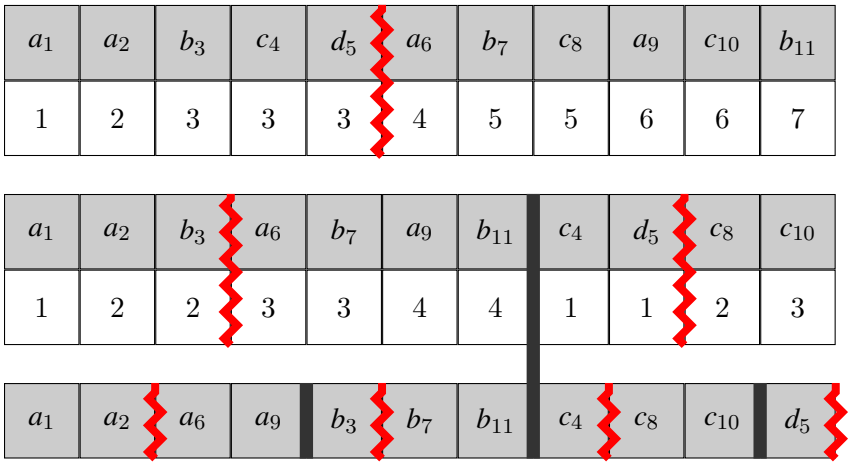
Black vertical lines show the division between words. Please note that we do not need them, and we also do not need the letters in grey cells, but they are shown just for clarity.

Now let us go back to our query. At each level, we have to decide if we want to go to the left child, or to the right child. To do so, we have to know how many letters from the given segment fall into each of the children. It is pretty easy, given our representation above, because we just need to count all the ones in this segment. Moreover, we can precalculate the prefix sums in each level to check the number of ones in any given segment in $O(1)$ time. Let the number of ones be c_1. Then we have two options:

- $c_1 \leq k$. Then we know that we have to look for k-th smallest letter in the left child (in the same segment),

- $c_1 > k$. Then we are looking for the $c1 - k$-th smallest letter in the right child (still in the same segment).

Now the only problem left is how to determine the segment in each child. Once again, we will use the prefix sums: we just need to know how many zeroes or ones already appeared in our word so far and move to this index. In the following picture, the red line symbolizes some possible segment border. We can just look at the prefix sum that immediately precedes it to figure out where exactly will it be in the left child, and subtract it from the number of all letters up to this point, to have the position of this border in the right child.



So let us try to recollect everything and write the code to answer our queries:

Function $Query(l, r, k)$**:**
  return $QueryOnLevel(0, l, r, k)$

Function $QueryOnLevel(level, l, r, k)$**:**
  if *node is a leaf*
    return *corresponding letter*
  $c1 \leftarrow prefix\_sums[level][r] - prefix\_sums[level][l-1]$;
  if $c1 \leq k$
    return $QueryOnLevel(level + 1, prefix\_sums[level][l], prefix\_sums[level][r], k)$
  else
    $new\_l \leftarrow (l - prefix\_sums[level][l]) + prefix\_sums[level][n]$;
    `// We are padding here all letters from the left child to move to the right side of the table.`
    $new\_r \leftarrow (r - prefix\_sums[level][r]) + prefix\_sums[level][n]$;
    return $QueryOnLevel(level + 1, new\_l, new\_r, k - c1)$

Now let us also notice that some updates might be pretty easy. For example swapping two neighbouring letters in our initial word should be relatively easy. We just need to swap these letters, and change at most two values in our prefix sums on each level.

If we want to swap any two letters, we can do that as well, but we need to have some additional data structure, for example a segment tree. Then we can add/remove values from it and quickly get answer how many zeroes/ones are in some queried segment. Then the time complexity of queries and updates rises to $O(\log n \cdot \log |\Sigma|)$.

Now let us move to some real problems from algorithmic competitions.

Problem Destiny

Codeforces Round #429.
Limits: 2.5s, 512MB.
`https://kostka.dev/sp/des`

Once, Leha found in the left pocket an array consisting of n integers, and in the right pocket q queries of the form (l, r, k). If there are queries, then they must be answered. Answer for the query is minimal x such that x occurs in the interval $[l, r]$ strictly more than $\frac{r-l+1}{k}$ times or -1 if there is no such number. Help Leha with such a difficult task.

Input

First line of input data contains two integers n and q $(1 \le n, q \le 3 \cdot 10^5)$ – number of elements in the array and number of queries respectively.

Next line contains n integers $a_1, a_2, \ldots, a_n$ $(1 \le a_i \le n)$ – Leha's array.

Each of next q lines contains three integers l, r and k $(1 \le l \le r \le n, 2 \le k \le 5)$ – description of the queries.

Output

Output answer for each query in the new line.

Examples

For the input data:

```
4 2
1 1 2 2
1 3 2
1 4 2
```

the correct result is:

```
1
-1
```

Whereas for the input data:

```
5 3
1 2 1 3 2
2 5 3
1 2 3
5 5 2
```

the correct result is:

```
2
1
2
```

Solution

Of course we will start by building the wavelet tree on the sequence given in the input. To answer a query, we will start from the root, mark it and move down recursively. In each step, we will take all marked nodes at a given level and look at their children. If the child has at least $\frac{r-l+1}{k}$ numbers from the given interval, we mark it (as it might contain our answer). Please note that on each level we will mark at most k nodes. Therefore we can answer each query in $O(k \log n)$ time.

Problem Chef and Swaps

Codechef September Challenge 2014.
Limits: 1s, 64MB.
`https://kostka.dev/sp/swa`

This time, Chef has given you an array A containing N elements.

He has also asked you to answer M of his questions. Each question sounds like: "How many inversions will the array A contain, if we swap the elements at the i-th and the j-th positions?".

An inversion is such a pair of integers (i, j) that $i < j$ and $A_i > A_j$.

Input

The first line contains two integers N and M $(1 \leq N, M \leq 2 \cdot 10^5)$ - the number of integers in the array A and the number of questions respectively.

The second line contains N space-separated integers - $A_1, A_2, \ldots, A_N$, respectively $(1 \leq A_i \leq 10^9)$.

Each of next M lines describes a question by two integers i and j $(1 \leq i, j \leq N)$ - the 1-based indices of the numbers we'd like to swap in this question.

Mind that we don't actually swap the elements, we only answer "what if" questions, so the array doesn't change after the question.

Output

Output M lines. Output the answer to the i-th question of the i-th line.

Examples

For the input data:	the correct result is:
6 3 1 4 3 3 2 5 1 1 1 3 2 5	5 6 0

Note:

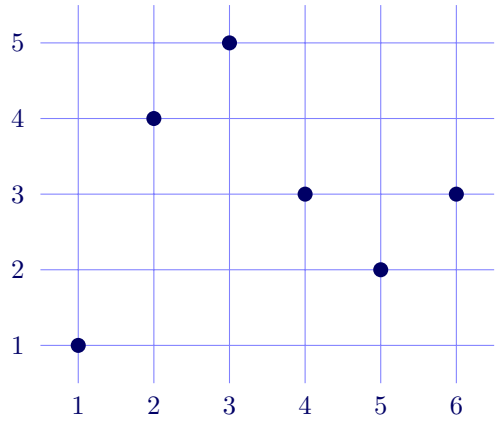
Inversions for the first case: $(2, 3), (2, 4), (2, 5), (3, 5), (4, 5)$.

Inversions for the second case: $(1, 3), (1, 5), (2, 3), (2, 4), (2, 5), (4, 5)$.

In the third case the array is $[1, 2, 3, 3, 4, 5]$ and there are no inversions.

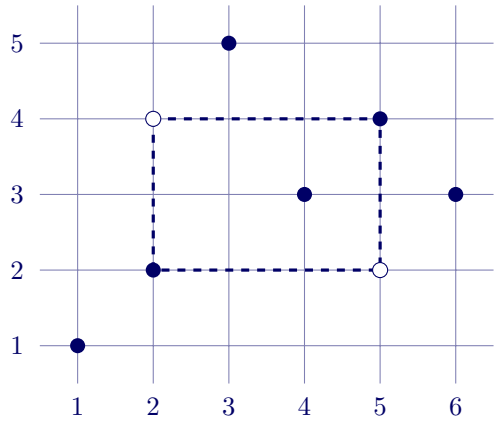
Solution

This is a really cool problem with inversions. First, let us represent the array from the input in the geometric manner. The following points represent elements of the array $[1, 4, 5, 3, 2, 3]$, where y axis represents values and x axis represents indices:



Now every inversion is a pair of two different points, where the left one is higher than the right one. We can count inversions in the original table pretty easy in $O(n \log n)$ time, using the count tree or merge sort algorithm. Please note that we can renumber the values, as there will be at most n different values and only the relative order matters.

Now let us think what will happen if we swap a pair of points. Let us say we will swap the second element with the fifth one. Then our array will look as follows:



Please note that for every point outside of the rectangle in the picture above, the number of inversions that this element generates stays the same (either we still have inversion with our element, or we exchange this inversion for the inversion with the swapped element).

That is not the case for all points inside the rectangle. If we had the inversion with our swapped elements, now we lose both of them. Otherwise, if we didn't have the inversions, now we have two new. Therefore we just need to find the number of points in some rectangle on the plane. The difference in inversions will be exactly twice the number of points inside this rectangle.

This query can be done pretty easy offline, using the inclusion-exclusion principle. We can create events in the corners of every rectangle and then offline sweep all elements and count the number of points inside each rectangle starting at the origin. From these rectangles, we can recover all the results.

But we can use the wavelet tree to realize these queries online. First, let us create a Wavelet tree on the array from the input. Now we want to count elements between values a and b, between indices l and r. We will do this in a recursive manner. We have three possible cases.

- Node in the wavelet tree has no values between a and b, then the answer is 0.
- Node has all values between a and b, then the answer is the number of elements in this node.
- Node has some values between a and b. Then we have to recursively call our function in both children, mapping our intervals properly in them.

One might check that we just need at most $O(\log |\Sigma|)$ recursive calls (the argument is similar as in the regular interval tree), so the total time complexity is exactly $O(\log |\Sigma|)$.

Wavelet matrix

In [Claude et al., 2015], another representation of the wavelet tree was introduced, called the *wavelet matrix*.
The idea is pretty simple, instead of keeping overall structure that each child is aligned with its parent, in each level we put all left children to the left and all right children to the right. This bit-vector can be further compressed, with a slightly worse performance. The whole data structure to be constructed directly (and on-the-fly) over larger alphabets.

Chapter 4

Tree decompositions

In this chapter, we will focus on two techniques that will allow us to efficiently solve various problems on trees.

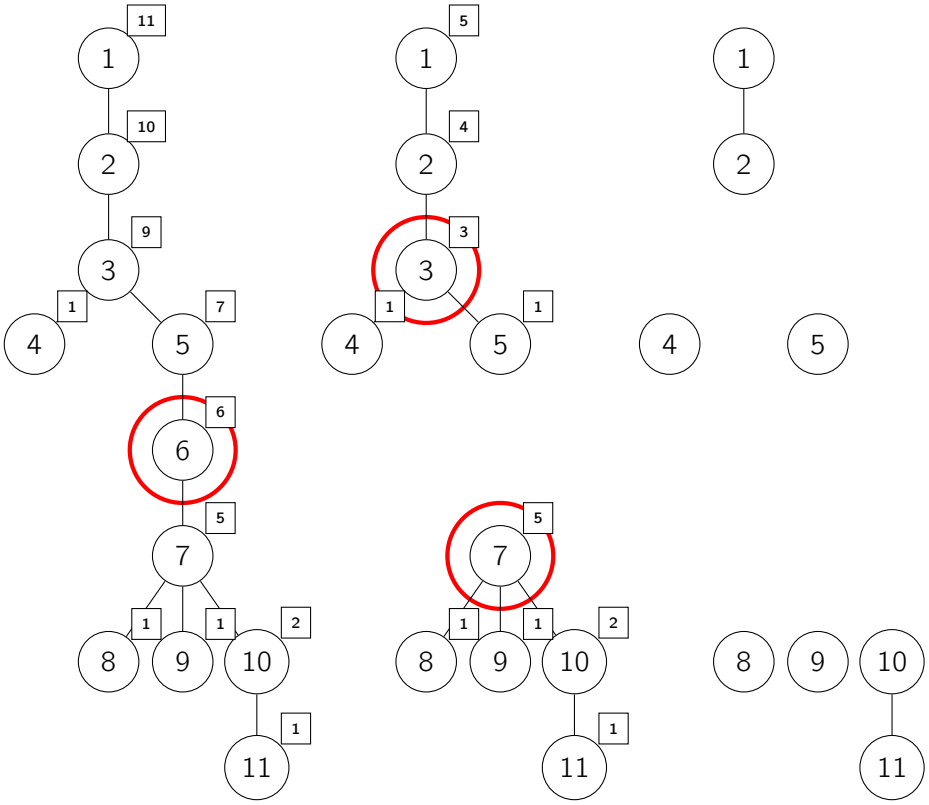
4.1. Centroid decomposition

First, we will explain how we can use the divide and conquer technique on trees. Normally, when we use this technique, we split a problem into smaller subproblems. In the case of trees, we would like to divide the tree into a set of subtrees. Ideally, we would like to have two subtrees of size $\frac{N}{2}$, where N denotes the number of nodes in the original tree, but quite often it is almost impossible to find such a division (as an example, consider a star, i.e. one vertex connected to every other vertex).

To help with this problem, we will use the notion of a *centroid* (hence the name *centroid decomposition*). A centroid is a vertex in our tree which we can remove and get subtrees of sizes at most $\frac{N}{2}$. Note that there can be at most one subtree of exactly that size.

In each tree we can find a centroid (you can prove that in a given tree there can be at most two centroids, and in the case of two centroids they have to be connected by an edge) and we can do it quite easily. We can start from an arbitrary node in the tree and calculate the sizes of each subtree originating from the children of this node. If every subtree has size at most $\frac{N}{2}$, then we are done, we have found the centroid. Otherwise, there is exactly one subtree with more than $\frac{N}{2}$ vertices. We can move to this subtree by changing the node to the corresponding neighbour and continue the process. We can implement it by running DFS two times. With the first one, we calculate the sizes of the subtrees with the root in the arbitrary node. Then, in the second DFS, we check the sizes of the subtrees and move towards the centroid. Note that we can update the sizes of subtrees in constant time (or even simpler, we do not have to consider a new subtree that is formed "above" our moving root).

In the picture below, we are given an example. The root of this tree is 1. Inside small rectangles on the right hand size of each vertex, we can see the size of the subtree rooted at each vertex. We start at the root and we check if all of our subtrees have size at most $\frac{N}{2}$. If that is not the case, we enter the subtree violating this condition, until we reach the centroid. In our example, we reach 6. Then, we remove the centroid from the tree, and then recursively we can find the centroids in each of the remaining components. We find 3 and 7 and we are left with just single nodes and edges.



Using this decomposition, we are finally ready to use divide and conquer on trees. Let us say that we want to calculate the number of paths fulfilling some properties. We can find the centroid and consider all the paths going through it. Then we can remove our centroid from the tree and we are left with a forest, where every component has at most half the number of vertices from the original tree. We can continue the process in each of the components (finding the centroid, removing it, and continuing solving subproblems). If we can compute the answer for one instance of a problem in linear time, then it is easy to conclude that the total runtime will be $O(N \log N)$, as we will have $\log N$ layers of problems (as we divide the size by at least two, each time we go into subproblems), and the total number of vertices in each layer is at most N.

This technique might seem rather trivial, but it is quite powerful and can be used in many difficult problems.

Problem Race

International Olympiad in Informatics 2011, first day.
Limits: 3 s, 256 MB.
`https://kostka.dev/sp/rac`

In conjunction with the IOI, Pattaya City will host a race: the International Olympiad in Racing (IOR) 2011. As the host, we have to find the best possible course for the race.

In the Pattaya-Chonburi metropolitan area, there are N cities connected by a network of $N-1$ highways. Each highway is bidirectional, connects two different cities, and has an integer length in kilometers. Furthermore, there is exactly one possible path connecting any pair of cities. That is, there is exactly one way to travel from one city to another city by a sequence of highways without visiting any city twice.

The IOR has specific regulations that require the course to be a path whose total length is exactly K kilometers, starting and ending in different cities. Obviously, no highway (and therefore also no city) may be used twice on the course to prevent collisions. To minimize traffic disruption, the course must contain as few highways as possible.

Implementation

Write a procedure `best_path(N,K,H,L)` that takes the following parameters:

- N - the number of cities. The cities are numbered 0 through $N-1$.
- K – the required distance for the race course.
- H – a two-dimensional array representing highways. For $0 \leq i < N-1$, highway i connects the cities $H[i][0]$ and $H[i][1]$.
- L – a one-dimensional array representing the lengths of the highways. For $0 \leq i < N-1$, the length of highway i is $L[i]$.

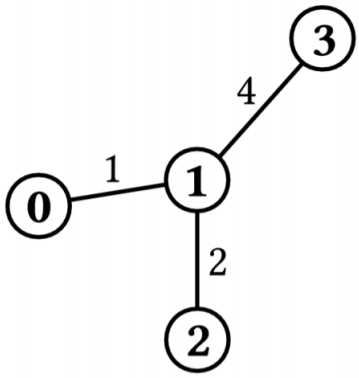
You may assume that all values in the array H are between 0 and $N-1$, inclusive, and that the highways described by this array connect all cities as described above. You may also assume that all values in the array L are integers between 0 and $1\,000\,000$, inclusive. Your procedure must return the minimum number of highways on a valid race course of length exactly K. If there is no such course, your procedure must return -1.

Examples

Example 1.

Consider the case shown in the figure below, where $N = 4$, $K = 3$,

$$H = \begin{bmatrix} 0 & 1 \\ 1 & 2 \\ 1 & 3 \end{bmatrix}, L = \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix}$$

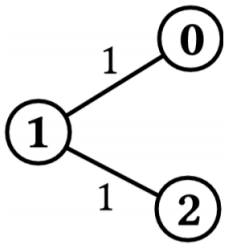


The course can start in city 0, go to city 1, and terminate in city 2. Its length will be exactly $1\ \mathrm{km} + 2\ \mathrm{km} = 3\ \mathrm{km}$, and it consists of two highways. This is the best possible course; therefore `best_path(N,K,H,L)` must return 2.

Example 2.

Consider the case shown in the figure on the right, where $N = 3$, $K = 3$,

$$H = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}, L = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

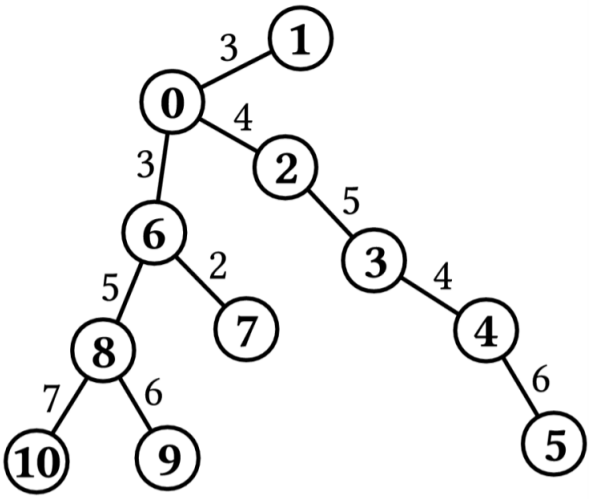


There is no valid course. In this case, `best_path(N,K,H,L)` must return -1.

Example 3.

Consider the case shown in Figure 3, where $N = 11$, $K = 12$,

$$H = \begin{bmatrix} 0 & 1 \\ 0 & 2 \\ 2 & 3 \\ 3 & 4 \\ 4 & 5 \\ 0 & 6 \\ 6 & 7 \\ 6 & 8 \\ 8 & 9 \\ 8 & 10 \end{bmatrix} L = \begin{bmatrix} 3 \\ 4 \\ 5 \\ 4 \\ 6 \\ 3 \\ 2 \\ 5 \\ 6 \\ 7 \end{bmatrix}$$



One possible course consists of 3 highways: from city 6 via city 0 and city 2 to city 3. Another course starts in city 10 and goes via city 8 to city 6. Both of these courses have length exactly $12\ \mathrm{km}$, as required. The second one is optimal, as there is no valid course with a single highway. Hence, `best_path(N,K,H,L)` must return 2.

Subtasks

Subtask 1 (9 points)

- $1 \leq N \leq 100$
- $1 \leq K \leq 100$
- The network of highways forms the simplest possible line: For $0 \leq i < N - 1$, highway i connects cities i and $i + 1$.

Subtask 2 (12 points)

- $1 \leq N \leq 1\,000$
- $1 \leq K \leq 1\,000\,000$

Subtask 3 (22 points)

- $1 \leq N \leq 200\,000$
- $1 \leq K \leq 100$

Subtask 4 (57 points)

- $1 \leq N \leq 200\,000$
- $1 \leq K \leq 1\,000\,000$

Solution

We are given a weighted tree and we are asked to find a simple path of length K, minimizing the number of edges on this path.

We will use centroid decomposition to solve this problem. First, we need to think about how to solve the problem if we know that the path goes through some vertex v. We will run some graph traversal algorithm (such as DFS or BFS) to compute the distance from v to every other vertex and depth of vertices when we root the tree in v. Now, we just need to check if there exist two vertices a and b such that $dist(a) + dist(b) = K$ and then later we will try to minimize $depth(a) + depth(b)$, the number of edges on this path.

When we have a list of all the distances and depths, we can simply check for each i if $K - dist(i)$ can be found in the list of distances. Minimizing the number of edges is

pretty straightforward as we can always choose the minimum depth of corresponding vertices (if there are many vertices with the same distance from v). The only tricky part is to make sure that we will not take two vertices from the same subtree, as the path has to go through v. To do this, we can either compute all values for a subtree, check them with values from the preceding subtrees, and then add values from the currently considered subtree; or keep the identifiers of subtrees for each calculated distance. Note that we just need to keep information about two different subtrees for each distance.

This technique allows us to solve this problem in $O(N \log N + K)$ time.

4.2. Heavy-light decomposition

The second technique will allow us to quickly perform queries on a tree. The key idea is to split the tree into several non-intersecting paths, in a way that we can reach each vertex using at most $\log N$ of such paths.

We will once again start by calculating the size of all the subtrees in a given tree (let denote them as $subtree\_size$), rooting our tree in some arbitrary vertex. We will call an edge from parent a to child b *heavy*, if

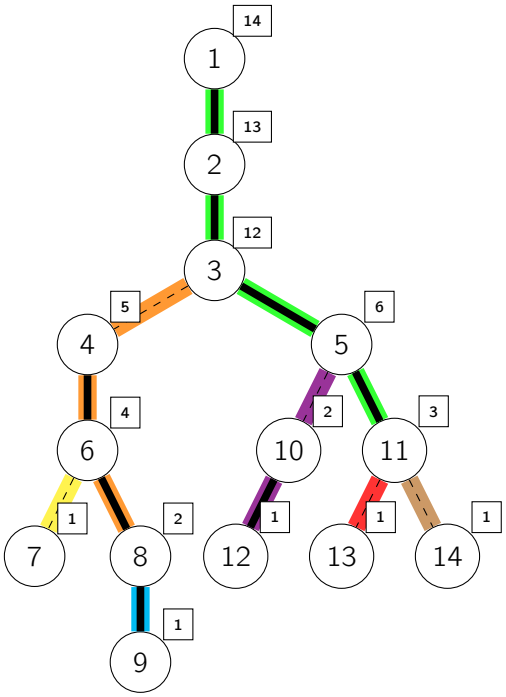
$$\frac{1}{2} subtree\_size(a) \leq subtree\_size(b).$$

All other edges will be called *light*.

Similarly as we argued earlier that only one subtree can violate centroid conditions, we can see that from each node there can be at most one heavy edge going down (otherwise the sum over all subtree sizes would be larger than the size of the whole subtree rooted in some node).

Now we are ready to define paths. Consider a vertex that does not have any heavy edges going down from it. Now start going up until you reach a light edge (which will end the path) or until you reach the root of the tree. This way every path will consist of one light edge at the top (except for the paths ending at the root) and some number of heavy edges. We will call these paths *heavy*.

Below we can see the division into heavy paths. Heavy edges are in bold, light edges are thin and dashed. Different colours denote different paths in our division.



It is easy to see that the paths will not intersect with each other. We can also notice that every time we traverse through a light edge, the size of the subtree decreases at least by half, ergo every vertex in our tree can be reached by using at most $\log N$ paths.

Heavy-light decomposition on vertices

In some problems, it is easier to think about the division into sets of vertices, not edges. In this case, we can remove light edges and we are left with a similar division, but every vertex belongs only to one set.

We would like to answer some queries on our tree. Let us say that our tree was weighted and we would like to answer queries about the maximum value on given paths from a to b.

First, we will divide our path into two parts, splitting it in the lowest common ancestor of a and b (note that finding LCA can be a part of answering the query). Now every part can be divided further into at most $\log N$ heavy paths. Then for every heavy path, we can store a segment tree (or more commonly we keep one segment tree, where every path has its own segment) with the maximum over intervals, so we can query an interval that belongs to the path. Note that every path can be queried in $O(\log N)$, so the total time complexity of one query is $O(\log^2 N)$. We can speed it up slightly by noticing that for every heavy path (except two: one at the top and the bottom of each part), we will ask about just the prefix of some heavy path, not the whole path. Moreover, we can say that for every heavy path except one we will ask about the prefix of some heavy path. Therefore we can often precompute answers for

each prefix and end up with $O(\log N)$ time complexity, as we can get answers for all heavy paths in $O(1)$, except for at most one heavy path (the one at the very top).

Problem Synchronization

Japanese Olympiad in Informatics Open Contest 2013.
Limits: 8 s, 256 MB.
`https://kostka.dev/sp/syn`

The JOI Co., Ltd. has N servers in total around the world. Each server contains an important piece of information. Different servers contain different pieces of information. The JOI Co., Ltd. is now building digital lines between the servers so that the pieces of information will be shared with the servers. When a line is built between two servers, pieces of information can be exchanged between them. It is possible to exchange pieces of information from one server to another server which is reachable through the lines which are already built.

Each server has a high-performance synchronization system. When two servers can exchange pieces of information each other and they contain different pieces of information, they automatically synchronize the pieces of information. After a synchronization between the server A and the server B, both of the servers A and B will contain all the pieces of information which are contained in at least one of the servers A and B before the synchronization.

In order to reduce the cost, only $N-1$ lines can be built. After the $N-1$ lines are built, there will be a unique route to exchange pieces of information from one server to another server without passing through the same server more than once.

In the beginning (at time 0), no lines are built. Sometimes, lines are built in a harsh condition (e.g. in a desert, in the bottom of a sea). Some of the lines become unavailable at some point. Once a line becomes unavailable, it is not possible to use it until it is rebuilt.

It is known that, at time j $(1 \leq j \leq M)$, the state of exactly one line is changed.

We need to know the number of different pieces of information contained in some of the servers at time $M+1$.

Write a program which, given information of the lines to be built and the state of the lines, determines the number of different pieces of information contained in some of the servers.

Input

Read the following data from the standard input.

- The first line of input contains three space separated integers N, M, Q. This means the number of the servers is N, a list of M changes of the state of the lines is given, and we need to know the number of different pieces of information contained in Q servers.
- The i-th line $(1 \leq i \leq N-1)$ of the following $N-1$ lines contains space separated integers X_i, Y_i. This means the line i, when it is built, connects the server X_i and the server Y_i.
- The j-th line $(1 \leq j \leq M)$ of the following M lines contains an integer D_j. This means the state of the line D_j is changed at time j. Namely, if the line D_j is unavailable just before time j, this line is built at time j. If the line D_j is available just before time j, this line becomes unavailable at time j. When the state is changed at time j, all the synchronization processes will be finished before time $j+1$.
- The k $(1 \leq k \leq Q)$ of the following Q lines contains an integer C_k. This means we need to know the number of different pieces of information contained in the server C_k in the end.

Output

Write Q lines to the standard output. The k-th line $(1 \leq k \leq Q)$ should contain an integer, the number of different pieces of information contained in the server C_k in the end.

Constraints

All input data satisfy the following conditions.

- $2 \leq N \leq 100\,000$.
- $1 \leq M \leq 200\,000$.
- $1 \leq Q \leq N$.
- $1 \leq X_i \leq N, 1 \leq Y_i \leq N, (for 1 \leq i \leq N-1)$.
- $1 \leq D_j \leq N-1$ $(1 \leq j \leq M)$.
- $1 \leq C_k \leq N$ $(1 \leq k \leq Q)$.
- The values of C_k are distinct.
- If all of the lines are built, there will be a route from each server to every other server through the lines.

Subtasks

Subtask 1 [30 points]: $Q = 1$ is satisfied.

Subtask 2 [30 points]: $X_i = i$, $Y_i = i + 1$ $(1 \leq i \leq N - 1)$ are satisfied.

Subtask 3 [40 points]: There are no additional constraints.

Example

For the input data:

```
5 6 3
1 2
1 3
2 4
2 5
1
2
1
4
4
3
1
4
5
```

the correct result is:

```
3
5
4
```

Note:

In the beginning, we assume the server i $(1 \leq i \leq 5)$ contains the piece i of information.

- At time 1, the line 1 is built and the servers 1, 2 become connected. Then, both of the servers 1, 2 contain the pieces 1, 2 of information.
- At time 2, the line 2 is built, and the servers 1, 3 become connected. Including the line 1, the servers 1, 2, 3 are connected. The servers 1, 2, 3 contain the pieces 1, 2, 3 of information.
- At time 3, the line 1 becomes unavailable because it was available just before this moment.
- At time 4, the line 4 is built and the servers 2, 5 become connected. Both of the servers 2, 5 contain the pieces 1, 2, 3, 5 of information. Note that the servers 1, 2 cannot exchange pieces of information each other because the line 1 became unavailable.

- At time 5, the line 4 becomes unavailable.
- At time 6, the line 3 is built and the servers 2, 4 become connected. Then, both of the servers 2, 4 contain the pieces 1, 2, 3, 4, 5 of information.

As explained above, in the end, the servers 1, 4, 5 have 3, 5, 4 different pieces of information, respectively.

Solution

We are given a tree and each vertex contains some parts of data. We will activate and deactivate edges, and each time some vertices are connected, they are syncing all data with each other. Our task is to find how many parts of data each vertex contains in the end.

Let us root the tree. First, let us notice that every connected component shares the same parts of data. In particular, we can store all the data in the topmost node of this component (we will call this vertex a *representative* of this component). Now when we activate a new edge, we merge two components (top and bottom one), so we just need to find the representative of the top component and share all the data from the representative from the bottom component. To find the top representative we can store the number of active connections between the root and every other node in a segment tree spanned on the flatten tree (generated from the Euler tour). Then we can find the representative of a top component in a similar fashion as finding the LCA – we can apply binary lifting, which we can do in $O(\log^2 N)$ time.

Now we want to make sure that we will not share duplicate information when we re-activate some edge. To do so, we can keep for each edge the amount of data that was shared last time we activated this edge. Then we can simply update the amount of data stored in the top representative by adding data stored in the bottom representative decreased by the amount of data shared in the last sync.

Finally, the answer for each vertex is simply the amount of data stored in its representative.

Now we can notice that we can update the number of connections and find the representative even faster if we decide to use the heavy-light decomposition on our tree. The total time complexity of this solution is $O(N \log N)$.

Chapter 5

Palindromes

Let us recollect that *palindromes* are words (or sequences) that read the same forwards as backwards, like `kayak`, or $(1, 2, 2, 2, 2, 1)$. These objects are mostly purely theoretical, as they do not have any real life applications. We just like symmetrical things. But it turns out that palindromes are pretty regular and have many interesting properties that make them intriguing objects to include in programming problems.

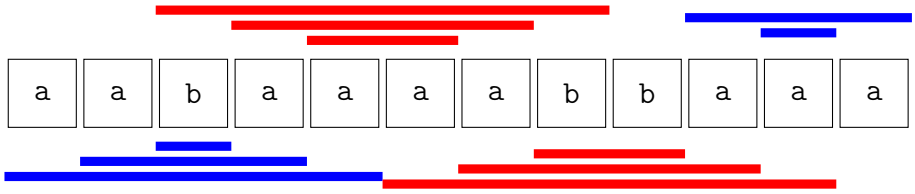
It turns out that the nature also likes palindromes. In genetics, palindromes have a slightly different meaning (for clarity, to distinguish these objects, we will call them „genetic palindromes"). Deoxyribonucleic acid (DNA) is formed out of two strings of nucleotides which are always paired in the same way: adenine (`A`) with thymine (`T`), and cytosine (`C`) with guanine (`G`). A sequence is called a genetic palindrome, when it is equal to its reverse, but nucleotides are replaced with their complements. For example, `TGCA` is a genetic palindrome, because its complement is equal to `ACGT`, which is `TGCA` in reverse. In 2004, it was discovered that many bases of the Y-chromosome have genetic palindromes. If one side is damaged, a palindrome structure allows the chromosome to repair itself by bending over to match with the non-damaged side with itself [Bellott et al., 2014].

In this chapter, we will discuss two results: the first one by Manacher from 1975, who introduced an algorithm to find all maximal palindromic substrings of a string in linear time, while the second one (by Rubinchik and Shur, from 2015, 40 years after Manacher's) will introduce the eertree – a data structure that will allow us to store and process all distinct palindromes in a given string.

5.1. Manacher's algorithm

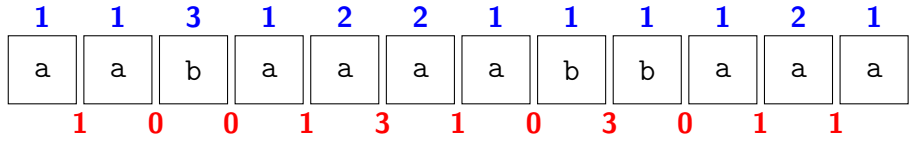
We will discuss the algorithm first introduced in [Manacher, 1975], and then extended in [Apostolico et al., 1993]. For a given string S (of length N) we will find a number of palindromic substrings of S, i.e. several pairs (i, j), such that $S_i S_{i+1} \dots S_j$

is a palindrome. First of all, we will consider two classes of palindromes: palindromes of odd length (which we will call "odd palindromes") and palindromes of even length ("even palindromes"). On the picture below, we highlighted some odd (blue) and even (red) palindromes.



Note that the structure of palindromes is extremely organized. If $S_l S_{l+1} \dots S_r$ is a palindrome, then $S_{l+1} \dots S_{r-1}$ has to be a palindrome as well (if such a word exists). We can also notice that each palindrome has a center. For odd palindromes this is a letter in the middle of the palindrome. For even palindromes, we will consider that the center is between the two letters in the middle. So now, to find all the palindromic substrings, we can start from every possible center (every letter or each space between letters for odd and even palindromes respectively), and check how far we can go in each direction till we find a mismatch (i.e. the letter on the left side will be different from the letter on the right side) or reach the end of the string. The number of steps we can make in this naive algorithm will be equal to the number of palindromes with this center. We will call this number a *radius* originating from this center.

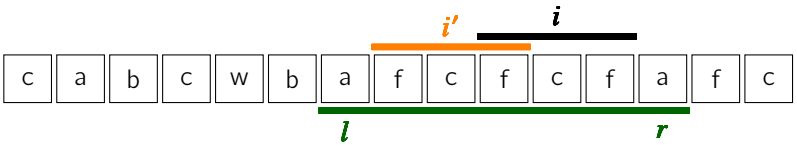
Below we calculated the radii for our example string (odd in blue, and even in red). Note that for odd palindromes, the radius is always at least 1, as a single letter itself is always a palindrome.



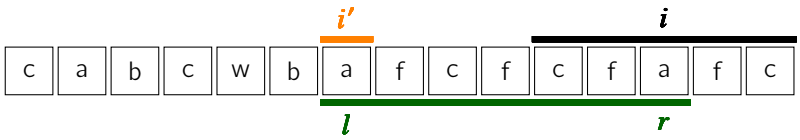
Now if we calculate these radii, we can easily compute the number of all palindromic substrings, it will be simply a sum of all these radii. So how can we compute these radii?

We will focus only on odd palindromes, as the algorithm will be almost the same both for odd and even palindromes. The idea that we will use is similar to the idea from the KMP algorithm. We will compute the radii from left to right, but we will always keep a palindrome that ends on the rightmost position among those we have found so far (we will call this palindrome the *rightmost palindrome*). We will denote the bounds of this palindrome by $[l, r]$. Let us say we have calculated all the radii up to i, and we now want to calculate $odd\_radius[i]$. There are two cases we need to consider:

- $i \leq r$. Here we want to use some data we gathered before. Note that we are inside the rightmost palindrome, ergo we can use some information from the left side of this palindrome. We can find the mirrored value i' on the left side and check the value $odd\_radius[i']$.



In many cases, we can assign $odd\_radius[i] \leftarrow odd\_radius[i']$, as we already checked where is the first mismatch. The only problem is when we hit the border of the rightmost palindrome, e.g. in the following case:



Note that previously we used the information from inside the palindrome. Beyond it, on the right side, we are not sure about the structure of the word, so the radius originating from i can be much larger than $odd\_radius[i']$. What we can do is to assign $odd\_radius[i] \leftarrow \min(odd\_radius[i'], r - i + 1)$ and then try to increase the radius naively.

- $i > r$. In this case, we use our naive algorithm, i.e. we try to expand the radius until we find a mismatch or we hit the bounds of the word.

What is the complexity of this algorithm? We need to consider how many steps our naive algorithm makes. We can run it for every position (N times), but every time we use it and we do not have a mismatch, the right border of the rightmost palindrome also moves to the right. We can do it at most N times, ergo the total amortized time complexity of this algorithm is simply $O(N)$.

Note that in both cases we might need to update the rightmost palindrome.

```
// We set up the bounds of the rightmost palindrome.
(l, r) ← (0, −1);
for i ← 0 to N − 1
    if i > r
        odd_radius[i] ← 1;
    else
        i' ← l + r − i;
        odd_radius[i] ← min(odd_radius[i'], r − i + 1);
    // We check the bounds and if letters match.
    while i − odd_radius[i] ≥ 0 and i + odd_radius[i] < N and
      S_{i−odd_radius[i]} == S_{i+odd_radius[i]}
        odd_radius[i] ← odd_radius[i] + 1;
    // Update the bounds of the rightmost palindrome.
    if r < i + odd_radius[i]
        (l, r) ← (i − odd_radius[i] + 1, i + odd_radius[i] − 1);
```

5.2. Eertree

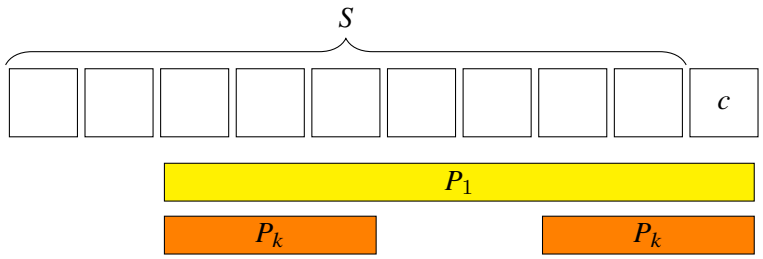
In this section, we will describe a relatively new data structure for storing and processing all palindromes in a given word. *Eertree* (also known as a *palindromic tree*) was first introduced during the summer training camp in Petrozavodsk and later described in [Rubinchik and Shur, 2015, Rubinchik and Shur, 2018].

The problem we will solve in this section is how to count all the distinct palindromes in a given word. For example, in the string `eertree` we have 7 distinct palindromes: `e`, `t`, `r`, `ee`, `rtr`, `ertre`, `eertree`.

We will start with a small, but a very important observation.

Observation 1. *A word of length n can have at most n distinct non-empty palindromes.*

Proof. We will prove this fact by induction. For an empty word, the statement above is of course true. Now let us say that we want to add a new character c to the word. All new palindromes are within the suffixes of the word Sc; let us denote the palindromic suffixes of Sc by $P_1, P_2, \ldots, P_i$, in the order of decreasing length, i.e. $|P_1| > |P_2| > \ldots > |P_i|$. Note that at least one such suffix exists, as the single letter is also a palindrome. We claim that the only palindrome that can be new (did not appear earlier in the word) is P_1. For all other, shorter, palindromes (P_k for $k > 1$) we claim that they appeared earlier in our word, as they are proper suffixes of P_1, but as P_1 is a palindrome, then they are also prefixes of P_1, so they appear in S.



□

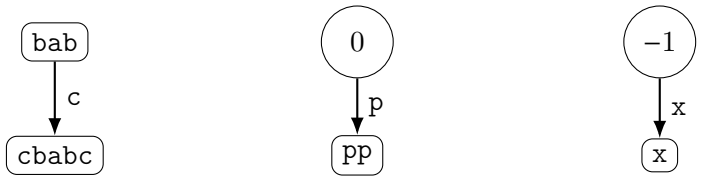
Therefore, we have proven that if we want to store all palindromes, a linear structure should be enough, as we will have at most n palindromes for a word of length n. Moreover, we have shown that if we process this word from left to right, every new letter will add at most one new palindrome.

Now let us introduce the data structure. The eertree will consist of:

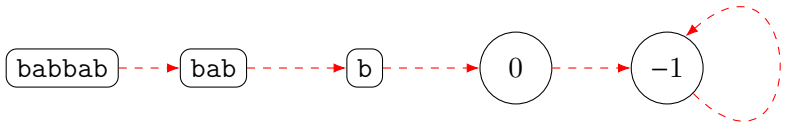
- Nodes that will represent all distinct non-empty palindromes and two special artificial palindromes:
  - an empty palindrome (denoted by 0),
  - a palindrome of length -1 (denoted by -1).

  We will discuss these special nodes in more detail below.

- Directed edges (A, B, c) from palindrome A to palindrome B labeled by a letter c, in a way that $B = cAc$, i.e. B can be obtained from A by adding a letter c on the left and on the right of the palindrome B. Node 0 has edges to all palindromes of length 2 (as we add letters to the empty palindrome), while the node -1 has edges to all palindromes of length 1 (as we start with palindrome of length -1, we skip one letter).

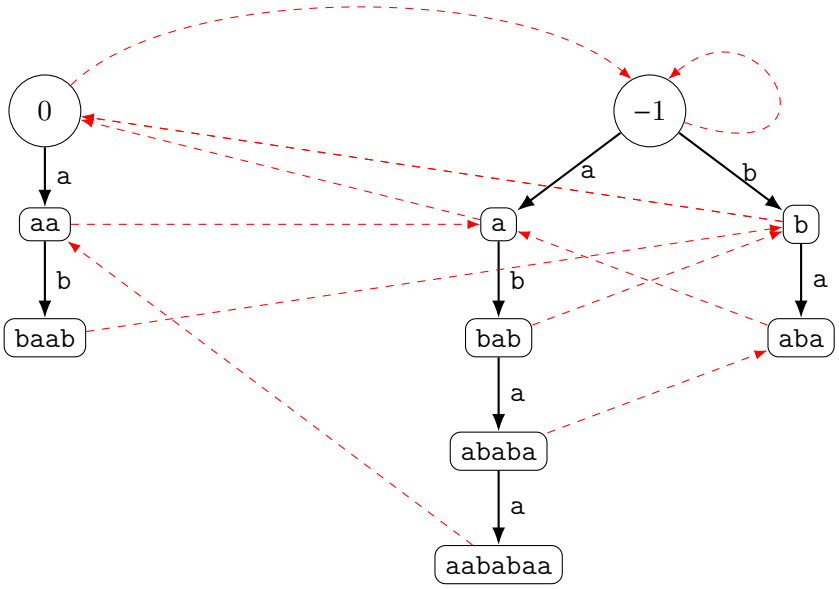


- Suffix links between nodes, in a way that the suffix link goes from palindrome P to the longest palindrome that is a proper suffix of P. Please note that this suffix is exactly P_2 from the observation above.



We assume here that all palindromes of length 1 have a suffix link to node 0, and both artificial palindromes (0 and -1) have suffix links to -1.

Below we present the full eertree for word `aababaab`. Please note that the eertree is, in fact, two trees – one for the even palindromes (on the left hand side in the picture below), and one for odd palindromes (on the right hand side). Suffix links are not limited to nodes within one tree.



Constructing the eertree

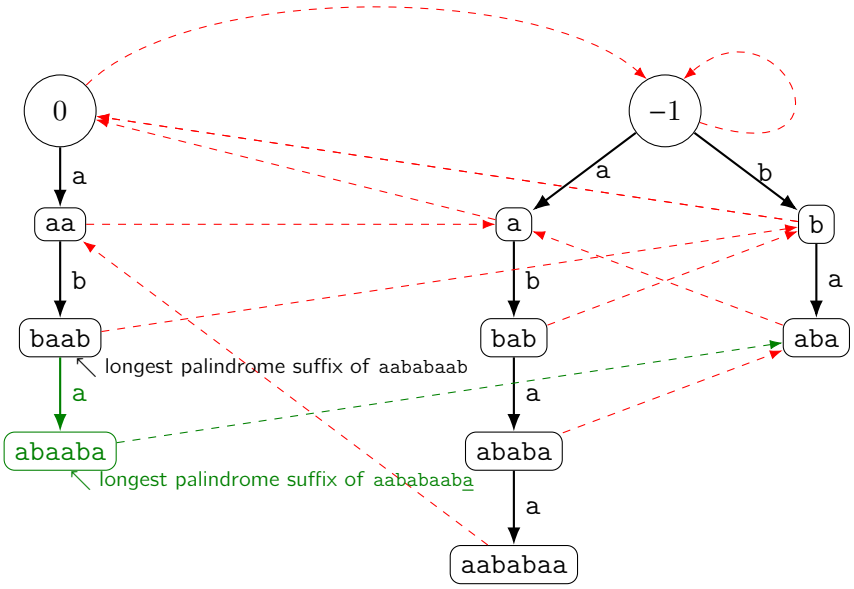
We will build the eertree incrementally. We will parse the letters in a given word S from left to right and update the eertree for prefixes of the word S. We will also keep a pointer pointing at the longest palindrome which is a suffix of a currently processed prefix.

Initialization is simple: we create two artificial nodes -1 and 0 with their respective suffix links. The pointer initially points at 0.

Now imagine that we already constructed eertree for the prefix `aababaab` (as in the previous picture). We want to add a letter `a` to expand the prefix to `aababaaba`.

Now notice that our pointer of the longest suffix palindrome is pointing at baab. Moreover, if we walk through our eertree via suffix links, we will visit every suffix palindrome in the decreasing order of their length, ending at node -1. Let us denote these suffix palindromes by $(Q_1, Q_2, \ldots, Q_i)$, in our example we have the following suffix palindromes: (baab, b, 0, -1). Now based on the proof of our initial observation, we know that the only new palindrome that can be added to our eertree is one of the palindromic suffixes which can be expanded with the new letter (in our case a) added to both sides of Q_k, i.e. aQ_ka. Therefore, we can check every palindrome $Q_1, Q_2, \ldots, Q_i$ and check if the letter before this palindrome is equal to the letter we are currently processing. If that is true, then we check if we already have an edge with that letter. If that is the case, we just move our pointer there. Otherwise, we create a new node with the edge labeled with a letter a. In our example, we are quite lucky, as baab can be already expanded to abaaba, therefore we just create the new node.

Now the only thing left is to find the suffix link for this newly created node. Note that we can do this in the very same way as before – after finding Q_k, we continue our travel through suffix links to find Q'_k, where $|Q_k| > |Q'_k|$ and Q'_k can be expanded with letter a. This node already exists in our eertree (as we cannot add new two palindromes with one letter), and this will be a new destination of our suffix link. In our example, we just move to the next palindrome using the suffix link from baab, which is b, and b can be expanded with letter a, and the node aba already exists (as we mentioned), therefore we just create the suffix link from abaaba pointing to aba. Our pointer of the longest palindromic suffix now points at aQ_ka (the node that we possibly just created). In the picture below, we show exactly how we have to modify the tree after processing the new letter in our word.



Now, let us try to figure out the complexity of constructing this data structure. First, we know that we will have at most n distinct palindromes (where n is the length

of the word that we are constructing the eertree for), therefore we will have at most $n+2$ nodes in our tree (2 additional nodes are for the special "empty" palindromes). As we mentioned, we can just keep the edges with labels, length of the palindrome, and a suffix link, so we keep $O(1)$ information in one node, giving total memory complexity of $O(n)$.

Now when it comes to the time complexity, we will focus on the amortized time. To do so, let us introduce a potential function equal to the sum of lengths of two words:

1. longest palindromic suffix of the already considered prefix (the node where the pointer is pointing out),
2. palindrome corresponding to the node that the suffix link of that palindrome is pointing at.

Every time we process a new letter, we add 4 to our potential (as we add 2 letters to both the longest palindromic suffix and its suffix link). That is also the only time when we increase our potential. Every time we use suffix links to find the palindrome that we can expand, we decrease our potential by some positive value. The minimum value of the potential is -1 (in the worst case when both words are the special empty palindromes), therefore we will use suffix links at most $4n + 1$ times, which yields to the total amortized time complexity of $O(n)$.

Note that we assumed here that the size of our alphabet (Σ) is constant. If that is not the case, we can still use our data structure with $O(n)$ memory and $O(n \log |\Sigma|)$ time, using some kind of a dictionary with logarithmic queries.

Problem Palindromes

Asia-Pacific Olympiad in Informatics 2014.
Limits: 1 s, 128 MB.
`https://kostka.dev/sp/pal`

You are given a string of lower-case Latin letters. Let us define substring's "occurrence value" as the number of the substring occurrences in the string multiplied by the length of the substring. For a given string find the largest occurrence value of a palindromic substring.

Input

The only line of input contains a non-empty string s of lower-case Latin letters (a-z).

Output

Output one integer – the largest occurrence value of a palindromic substring.

Examples

For the input data:	the correct result is:
`abacaba`	7
For the input data:	the correct result is:
`www`	4

Note: In the first sample there are seven palindromic substrings:
$\{a, b, c, aba, aca, bacab, abacaba\}$.

- a has 4 occurrences in the given string, its occurrence value is $4 \cdot 1 = 4$,
- b has 2 occurrences in the given string, its occurrence value is $2 \cdot 1 = 2$,
- c has 1 occurrence in the given string, its occurrence value is $1 \cdot 1 = 1$,
- aba has 2 occurrences in the given string, its occurrence value is $2 \cdot 3 = 6$,
- aca has 1 occurrence in the given string, its occurrence value is $1 \cdot 3 = 3$,
- $bacab$ has 1 occurrence in the given string, its occurrence value is $1 \cdot 5 = 5$,
- $abacaba$ has 1 occurrence in the given string, its occurrence value is $1 \cdot 7 = 7$.

So, the largest occurrence value of palindromic substrings is 7.

Subtasks

1. (8 points) $|s| \leq 100$,
2. (15 points) $|s| \leq 1000$,
3. (24 points) $|s| \leq 10\,000$,
4. (26 points) $|s| \leq 100\,000$,
5. (27 points) $|s| \leq 300\,000$.

Solution

We will build an eertree to find all the palindromes that occur in a given word. We need to find the number of occurrences of each palindrome in s. During the construction of the tree, we will keep for each node how many times this palindrome occurred as the longest suffix of some prefix of s, let us denote that value by $counter(word)$. After that, we can conclude that the number of occurrences of palindrome p is the sum of $counter(q)$ from all the palindromes q that can be reached from p via the suffix links. It turns out that the suffix links form a tree (with the root in -1), so we can use a dynamic programming on that tree to calculate these values.

Then, we can iterate over all palindromes and multiply the number of occurrences by their lengths and find the maximum product.

The total time and space complexity is $O(|s|)$.

In the next problem, we will show an interesting property based on lengths of palindromes, and how we can use it.

Problem Even palindromes

2nd Polish Olympiad in Informatics.
Limits: 0.1 s, 32 MB.
`https://kostka.dev/sp/eve`

Please note that the limits were increased in comparison to the original problem.

The factorization of a word into even (of even length) palindromes is its division into separable words, each of which is an even palindrome. For example, the word `bbaabbaabbbaaaaaaaaaaaabbbaa` can be divided into two even palindromes

`bbaabb|aabbbaaaaaaaaaaaabbbaa`,

or into eight even palindromes:

`bb|aa|bb|aa|bb|baaaaaaaaaaab|bb|aa`.

The first factorization contains the smallest possible number of palindromes, the second is the factorization into the maximum possible number of even palindromes. Note that a word may have many different distributions into even palindromes, or none.

Write a program that for a given word examines whether it can be broken down into even palindromes. If not, it writes only one word `NO`, and if it can be decomposed, it writes its factorizations into a minimum and a maximum number of even palindromes.

Input

The standard input contains one word S consisting of at least 1 and at most 200 000 small letters of the English alphabet.

Output

If the word can be divided into even palindromes, output two lines. In the first line output a sequence of words separated by spaces – the factorization of a given word into the minimum number of even palindromes. In the second line, output the factorization of a given word into the maximum number of even palindromes.

Otherwise, output a single line containing a single word `NO`.

Examples

For the input data:

```
bbaabbaabbbaaaaaaaaaaaabbbaa
```

the correct result is:

```
bbaabb aabbbaaaaaaaaaaaabbbaa
bb aa bb aa bb baaaaaaaaaaaab bb aa
```

For the input data:

```
abcde
```

the correct result is:

```
NO
```

For the input data:

```
abaaba
```

the correct result is:

```
abaaba
abaaba
```

Solution

This solution is heavily inspired by an article from [Diks et al., 2018].

Let us first focus on finding the factorization into a maximum number of even palindromes. We will use a greedy approach: we will find the shortest prefix of the word that is an even palindrome, cut it from the original word and then continue the process. We claim that this approach will find a correct factorization into a maximum number of even palindromes (if such a factorization exists) or determine that we cannot divide our word into even palindromes.

Theorem 1. *The greedy algorithm stated above works in the case of finding the optimal factorization into a maximum number of even palindromes.*

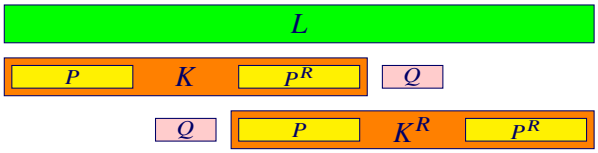
Proof. We will prove this by induction over the length of the word. Of course, for a word of length 2 (or even 0), the thesis is satisfied. Now let us assume that the greedy algorithm finds the optimal solution for all words of length at most n. Now let us choose a word S of length $n+2$ that can be divided into even palindromes. Let us take an optimal factorization into the maximum number of such palindromes $A = (A_1, A_2, \ldots)$ and the factorization found by our algorithm $B = (B_1, B_2, \ldots)$. If $A_1 = B_1$, then by induction, we know that the shorter word can be optimally divided by our greedy algorithm. So let us assume that $A_1 \neq B_1$. Then $|A_1| > |B_1|$, so B_1 is a proper prefix of A_1. Now we will use the following lemma:

Lemma 1. *If K and L are even palindromes and K is a proper prefix of L, then L can be divided into at least two even shorter palindromes.*

Proof. We will consider two cases. If $|K| \leq \frac{1}{2}|L|$, then we have the following situation:

As L is a palindrome, then K is a border of L. If we cut K from the beginning and the end, an even palindrome M (maybe empty) will remain, so we can divide L into (K, M, K).

When $|K| > \frac{1}{2}|L|$, let us denote the overlapping segment by P. P has to be an even palindrome (as $P = P^R$ in the middle). Then if we cut P from the beginning and from the end of K, then we are once again left with another even palindrome – Q.



Then we can divide L into (P, Q, P, Q, P), which concludes our lemma. □

Thus we have shown that A is not an optimal factorization into the maximum number of even palindromes. □

Now how we can implement this algorithm in linear time? We will compute for each suffix what is the shortest even palindrome which is a prefix of this suffix, i.e. we will compute the following array

$$skip(i) = \min_{k>0} S[i \ldots i+k] \text{ is an even palindrome.}$$

Then we can use this array to cut shortest palindromes greedily. To compute $skip$, we will use the $radius$ array from Manacher's algorithm (compare with the previous section). Let us say that index i *influences* index j if

$$i - radius[i] + 1 \leq j \leq i.$$

We can notice that for given j, $skip[j]$ is determined by the closest $i \geq j$, where i influences j. Then

$$skip[j] = 2 \cdot (i + 1 - j).$$

We will process the word from right to left, keeping positions that influence the currently processed index on a stack. We can easily update the stack, keeping the closest index that influences the current position, based on the formula above. Finally, based on the index that influences the current position, we can calculate the $skip$ array in linear time.

Now, let us go back to the second part of the problem, i.e. finding a factorization into a minimum number of even palindromes. It turns out that the greedy approach does not work in this case. Indeed, let us consider the word `aaaaaabbaa`. A greedy approach will divide this word into `aaaaaa|bb|aa` (three even palindromes), but it can be divided into just two even palindromes: `aaaa|aabbaa`. We need to find another method.

From now on, we will assume that we are just looking for a factorization into a minimum number of palindromes, without consideration for the parity of their length. The general solution which we will describe can be easily modified to consider only even palindromes.

To solve this problem, we will use dynamic programming. Let $dp(i)$ denote the minimum number of palindromes that we can divide the prefix $S[1..i]$ into, and $dp(0) = 0$. Then we can say that:

$$dp(i) = \min_{p \text{ – palindrome ending at } i} (dp(i - |p|) + 1)$$

Then the minimum number of palindromes will be kept in $dp(|S|)$. During our calculations, we can store how we can get this factorization. We already know how

we can iterate over all palindromes being suffixes of some prefix. We can go through all the suffix links to check every palindrome fitting this condition. Unfortunately, this solution still works in $O(|S|^2)$ time, as every prefix can have a linear number of palindromes which are its suffixes.

We can speed up this solution by finding a better way to iterate over these suffix palindromes.

Observation 2. *Let $Q_1, Q_2, \ldots, Q_k$ be the sequence of all palindromes which are suffixes of some word S in the order of decreasing length. Then the sequence $(Q_1, Q_2, \ldots, Q_k)$ can be divided into $\log |S|$ groups in a way that the lengths of palindromes in every group form an arithmetic progression.*

Proof. Let us construct such a division. For the first group, we will take Q_1 (we will call this element a leader of the first group) and every Q_i with length at least half of the length of the leader, i.e. $|Q_i| \geq \frac{1}{2}|Q_1|$. Then the rest of the palindromes that will remain can be divided recursively, and we will have at most $\log |S|$ groups, as every time we half the length of the leader in the group.

Now we just need to prove that every group indeed contains palindromes which lengths form an arithmetic progression. Let us take two palindromes Q_j and Q_i from one group, where Q_i is a leader. Q_j is a suffix of Q_i, and as Q_i is a palindrome, then Q_j is also its prefix, hence Q_j is a border of Q_i. Therefore Q_i has a period $|Q_i| - |Q_j|$.

Moreover, as $|Q_i| - |Q_j| \leq \frac{1}{2}|Q_j|$ for every Q_j in the group, using periodicity lemma, we can conclude that Q_i has a period of length d:

$$d = \gcd_{Q_j \neq Q_i} \left\{ |Q_i| - |Q_j| \right\}.$$

As d divides $|Q_i| - |Q_j|$, $|Q_j|$ is an element of the sequence where the initial term is equal to $|Q_i|$ and the common difference is equal to $-d$. We just need to reason why there are no "holes" in our arithmetic progression, that is, we want to determine that every suffix of length $z = |Q_i| - td$ for $\frac{1}{2}|Q_i| \leq z \leq |Q_i|$ is indeed a palindrome. This can be proven with an observation that every border of a palindrome also has to be a palindrome. □

We will construct this division for all nodes in our palindromic tree. In every node we will additionally store:

- the difference of the arithmetic progression starting at this node,
- the number of palindromes that follow p in this progression,
- pointer to the next progression (with a different difference).

All this information can be computed during the creation of the suffix link – we can calculate the difference between the length of these palindromes and check this difference with the difference in the node where the suffix link points to.

Note that this division into progressions does not have to be the same division as described in the proof of our observation, but it is the division into maximal (not extendable) progressions, therefore it will have $O(\log |S|)$ progressions.

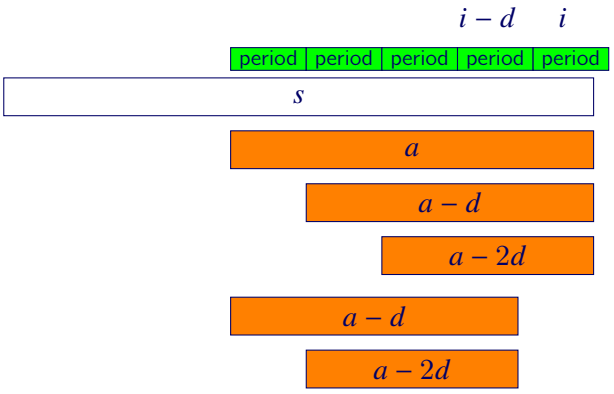
Now, let us denote the maximum length of a palindrome belonging to the arithmetic progression defined by a palindrome a by $rep(a)$. Below we will also often identify the palindrome with its length, depending on the context. Then we can define the following array: $dp_{arithm}(i, rep(a))$ – the minimum number of palindromes in the division of prefix of length i into palindromes, where the division ends by a palindrome represented by $rep(a)$. Note that this array has only $O(n \log n)$ values, based on the previous observation.

Now let us assume that we calculated all values $dp_{arithm}(j, *)$ and $dp(j)$ for $j < i$ and now we want to calculate $dp_{arithm}(i, *)$ and $dp(i)$.

Let us recall the formula from before:

$$dp(i) = \min_{p \text{ – palindrome ending at } i} (dp(i - |p|) + 1)$$

Now instead of iterating over all the suffix palindromes (ending at i), we will iterate only over all the arithmetic progressions of these suffix palindromes.



Note that we cannot have a suffix palindrome ending at $i - d$ of length a, where d is the difference of the arithmetic progression containing palindrome a. If that was the case, it would mean that we can have a palindrome of length $a + d$ ending at i, but we already established that a is the longest suffix palindrome from this progression.

Finally, we can conclude that we do not have to iterate over all the palindromes from every arithmetic progression, we just need to check two values for each progression:

$$dp(i) = \min_{\text{arithmetic progression } (a,d)} (dp_{arithm}(i - d, rep(a - d)), dp(i - (a - k \cdot d)) + 1).$$

In a similar fashion we can update $dp_{arithm}(i, *)$.

This leads us to an $O(n \log n)$ solution with $O(n \log n)$ memory. We can shave off this memory down to $O(n)$, but it was not required in this problem.

Chapter 6

Dynamic programming optimizations

Dynamic programming is a problem-solving method used widely in many problems. In this section, we will talk about various general and problem-specific optimizations and tricks that allow us to speed up the dynamic programming solutions.

6.1. Sum over subsets (SOS)

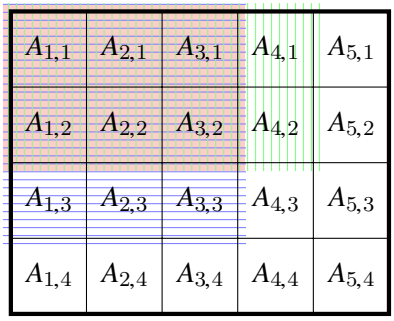
We will start with a slightly different problem. We want to calculate the prefix sums on some $2D$ array A, i.e.:

$$pref[i][j] = \sum_{a=1}^{i} \sum_{b=1}^{j} A_{a,b}$$

For simplicity, we are assuming here that the array is indexed from 1.

There are two easy methods to solve this problem. The first uses the inclusion-exclusion principle with dynamic programming. If we want to calculate $pref[i][j]$, to $A_{i,j}$ we just need to add $pref[i][j-1]$ and $pref[i-1][j]$ and subtract $pref[i-1][j-1]$ which we counted twice:

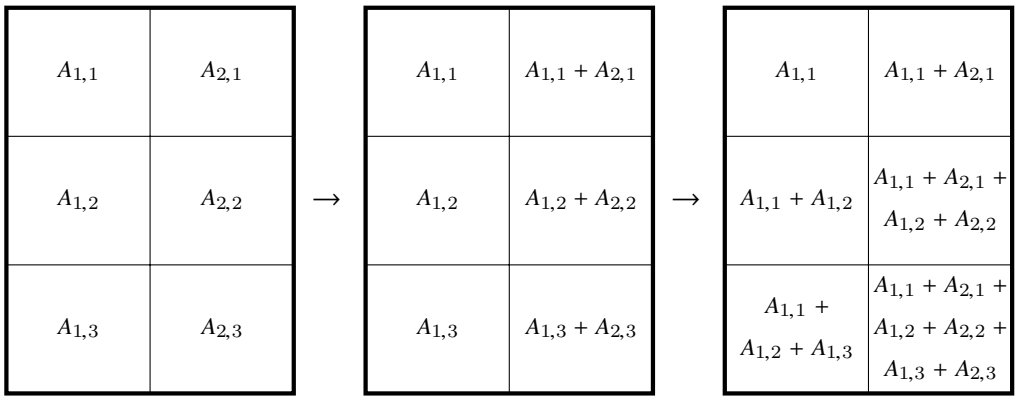
$$pref[i][j] = A_{i,j} + \text{pref[i][j-1]} + \text{pref[i-1][j]} - \text{pref[i-1][j-1]}$$



Therefore we can calculate all the prefix sums in $O(nm)$ time. There is another method that solves the same problem. We can calculate the prefix sums in rows (or columns) and then switch dimensions and calculate the prefix sums again:

```
// calculate the prefix sums for rows
for i ← 1 to n
    for j ← 1 to m
        a[i][j] += a[i][j − 1];
// calculate the prefix sums for columns
for j ← 1 to m
    for i ← 1 to n
        a[i][j] += a[i − 1][j];
```

Below we can see how this algorithm works on a simple array of size 3×2.



We will use the approach with prefix sums to solve a similar problem on bitmasks. Imagine we have 2^n bitmasks and each bitmask b has some corresponding value $A(b)$.

We would like to compute the following function for each mask b:

$$F(b) = \sum_{a \subseteq b} A(a).$$

The brute-force solution works in $O(4^n)$ time: for each mask b, we iterate over all other masks a, checking if $a \subseteq b$, and if that is the case, we add $A(a)$ to $F(b)$.

Iterating over submasks

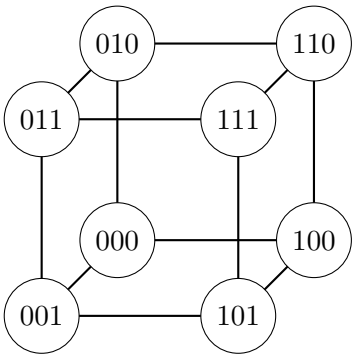
There is a method that allows us to iterate only over submasks of a given mask b.

```
s ← b;
do
    foo(s);
    s ← (s − 1) & b;              // & denotes bitwise-AND here
while s ≠ 0;
```

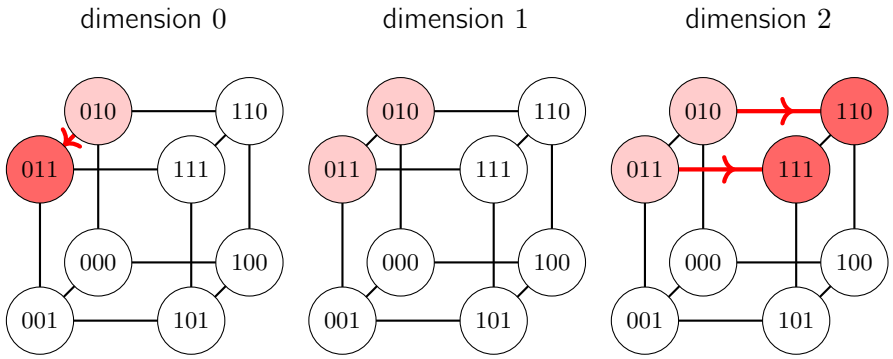
In every step of this loop, we decrease s by one, ergo we change the rightmost 1 to 0 followed by ones, for example $\ldots 10000 \rightarrow \ldots 01111$. Then we remove all bits that are not set in b by ANDing it with b. We can see that $(s-1)\&b$ will be the next submask in the decreasing order of submasks of b.

In many cases, we need to iterate over all submasks for every bitmask. What is the time complexity of this approach? To answer this question we have to determine how many submasks of all bitmasks there are. We can easily show that for every bit, the bit either has to be in mask and submask, only in mask, or neither in mask nor submask. Therefore, as we have n bits, the number of all submasks is 3^n.

Now, these bitmasks will correspond to an n-dimensional array of size $2\times 2\times \ldots \times 2$. We can imagine this array as an n-dimensional hypercube where each vertex corresponds to some bitmask.

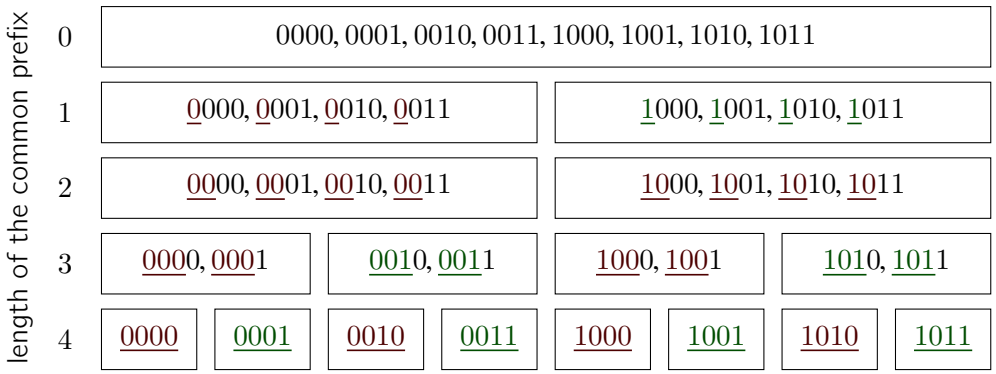


We can use the very same algorithm that we used for two-dimensional prefix sums, but extended to n dimensions. We iterate over dimensions and when we consider the i-th dimension, we add the values from all vertices with 0 in the i-th position to their counterparts with a bit set to 1 in that dimension. Below we can see how we will propagate the value from 010 to all bitmasks that contain 010 as a submask. Note that we consider dimensions from right to left.




```
// for each dimension
for i ← 0 to n − 1
    // iterate over all masks
    for mask ← 0 to 2^n − 1
        // if the i-th bit in mask is equal to 0
        if mask & 2^i == 0
            // then add value to its counterpart
            A[mask | 2^i] += A[mask];          // | denotes bitwise OR
```

Please note that this method can be also interpreted differently. We are trying to divide the submasks of every bitmask into some non-intersecting sets, depending on the prefix. In the picture below, whenever our initial mask has 1 in i-th place, we can divide all the bitmask from the current group into two subsets: one containing all the masks with 0 in i-th place and one containing all the mask with 1 in i-th place. In this way, we will end up with sets with only one bitmask. Now, our solution is basically a dynamic programming on the tree created by this division. For example, let us take a look at the division of some bitmask: 1011.



This solution works in $O(n2^n)$ time and $O(2^n)$ memory and it is really easy to implement (just a few lines, as we can see in the pseudocode).

Problem Shifts

Google Kick Start 2019, round G.
Limits: 40s, 1GB.
`https://kostka.dev/sp/shi`

Aninda and Boon-Nam are security guards at a small art museum. Their job consists of N shifts. During each shift, at least one of the two guards must work.

The two guards have different preferences for each shift. For the i-th shift, Aninda will gain A_i happiness points if he works, while Boon-Nam will gain B_i happiness points if she works.

The two guards will be happy if both of them receive at least H happiness points. How many different assignments of shifts are there where the guards will be happy?

Two assignments are considered different if there is a shift where Aninda works in one assignment but not in the other, or there is a shift where Boon-Nam works in one assignment but not in the other.

Input

The first line of the input gives the number of test cases, T ($1 \leq T \leq 100$). T test cases follow. Each test case begins with a line containing the two integers N and H ($1 \leq N \leq 20, 0 \leq H \leq 10^9$), the number of shifts and the minimum happiness points required, respectively. The second line contains N integers. The i-th of these integers is A_i ($0 \leq A_i \leq 10^9$), the amount of happiness points Aninda gets if he works during the i-th shift. The third line contains N integers. The i-th of these integers is B_i ($0 \leq B_i \leq 10^9$), the amount of happiness points Boon-Nam gets if she works during the i-th shift.

Output

For each test case, output one line containing `Case #x: y`, where x is the test case number (starting from 1) and y is the number of different assignments of shifts where the guards will be happy.

Example

For the input data:

```
2
2 3
1 2
3 3
2 5
2 2
10 30
```

the correct result is:

```
Case #1: 3
Case #2: 0
```

Note: In Sample Case #1, there are $N = 2$ shifts and $H = 3$. There are three possible ways for both Aninda and Boon-Nam to be happy:

- Only Aninda works on the first shift, while both Aninda and Boon-Nam work on the second shift.
- Aninda and Boon-Nam work on the first shift, while only Aninda works on the second shift.
- Both security guards work on both shifts.

In Sample Case #2, there are $N = 2$ shifts and $H = 5$. It is impossible for both Aninda and Boon-Nam to be happy, so the answer is 0.

Solution

Let us iterate over all bitmasks for both guards and calculate how many happiness points they receive for this mask. For guard $X \in \{A, B\}$, let us denote that by $hapinessX(mask)$. We can pick one guard, say A, and for every $maskA$ with $hapinessA(maskA) \geq H$, we just need to know how many $maskB$s, such that

$$maskB \mid maskA = 2^N - 1,$$

have $hapinessB(maskB) \geq H$. Let us denote that by $count(maskA)$.

This can be calculated using Sum over Subsets (SOS) optimization. Let $good(maskB) = 1$ if $hapinessB(maskB) \geq H$, otherwise $good(maskB) = 0$. Then

$$count(maskA) = \sum_{maskA' \subseteq maskB} good(maskB).$$

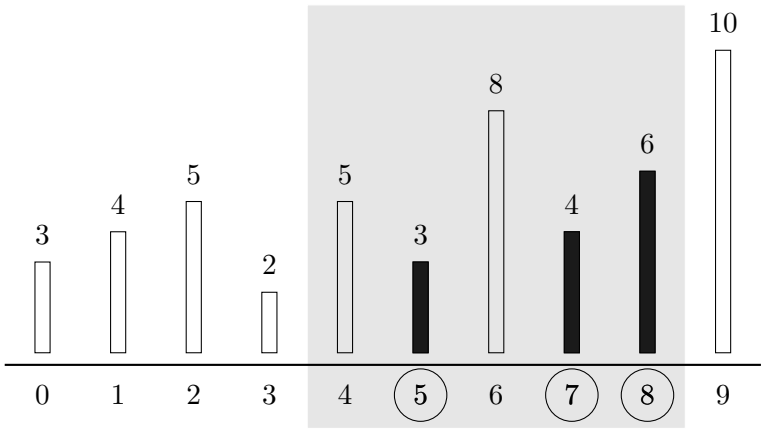
where $maskA'$ denotes the mask obtained from $maskA$ by flipping all its bits.

The total time and memory complexity is $O(N2^N)$.

6.2. Monotone queue optimization

We will start by reminding a classical problem. We are given a sequence $(a_0, a_1, a_2, \ldots, a_{n-1})$ and we want to calculate the minimum for each subsequence of k consecutive elements, i.e. we want to calculate the sequence $(m_0, m_1, \ldots, m_{n-k-1})$, where $m_i = \min_{i \le j < i+k} a_j$.

This problem is easily solvable in $O(n)$ time using a monotone queue. We will maintain a deque of indices of suffix minimums in order of increasing values in the proper range (last k values). Below we can find an example for a window of size $k = 5$ after considering the element with index 8. The elements that will be in the deque are highlighted.




```
// in D we will keep the indices of increasing elements
D ← deque();
for i ← 0 to n − 1
    // pop at most one element out of bounds from the front
    while not D.empty() and i − D.front() ≥ k
        D.pop_front();
    // pop elements larger than a_i from the back
    while not D.empty() and a_{D.back()} ≥ a_i
        D.pop_back();
    // insert i at the back
    D.push_back(i);
    // produce the result
    if i − k + 1 ≥ 0
        m_{i−k+1} ← a_{D.front()};
```

How we can use this algorithm in dynamic programming? Quite often our dynamic programming formula looks like this:

$$dp(i) = \min_{k(i) <= j < i} (dp(j) + cost(i))$$

where $k(i)$ is some **increasing**, and $cost(i)$ is some other function depending on i. We can generalize it even more, by replacing $dp(j)$ with some other function $f(j)$ depending on j (for example $f(j) = dp(j) + j^2$), so we can have the following formula:

$$dp(i) = \min_{k(i)<=j<i}(f(j) + cost(i))$$

The typical solution is as follows:

```
dp[0] ← 0;
for i ← 1 to n
    dp[i] ← INF;
    for j ← k(i) to i − 1
        dp[i] ← min(dp[i], f(j) + cost(i));
```

We may notice that the transitions are similar to the problem above. We are looking for some minimum value for values of function f on some consecutive elements and we can optimize the typical dynamic programming.

We will use a similar algorithm as above, but we will maintain the deque D of indices such that $D_j < D_{j+1}$ and $cost(D_j) \leq cost(D_{j+1})$.

```
for i ← 0 to n
    // pop elements out of the bounds from the front
    while not D.empty() and D.front() < k(i)
        D.pop_front();
    // calculate result using the front of the deque
    if not D.empty()
        dp[i] ← cost(i) + f(D.front());
    // pop elements from the back with larger values
    while not D.empty() and f(D.back()) > f(i)
        D.pop_back();
    // insert i at the back
    D.push_back(i);
```

This algorithm works in $O(n)$ time (each index can be inserted and removed at most once from the deque).

Let us try to use this observation in some problems.

Problem Watching Fireworks is Fun

Codeforces Round #219.
Limits: 4s, 256MB.
https://kostka.dev/sp/fir

A festival will be held in a town's main street. There are n sections in the main street. The sections are numbered 1 through n from left to right. The distance between each adjacent sections is 1.

In the festival m fireworks will be launched. The i-th $(1 \le i \le m)$ launching is on time t_i at section a_i. If you are at section x $(1 \le x \le n)$ at the time of i-th launching, you'll gain happiness value $b_i - |a_i - x|$ (note that the happiness value might be a negative value).

You can move up to d length units in a unit time interval, but it's prohibited to go out of the main street. Also you can be in an arbitrary section at initial time moment (time equals to 1), and want to maximize the sum of happiness that can be gained from watching fireworks. Find the maximum total happiness.

Note that two or more fireworks can be launched at the same time.

Input

The first line contains three integers n, m, d $(1 \le n \le 150\,000; 1 \le m \le 300; 1 \le d \le n)$.

Each of the next m lines contains integers a_i, b_i, t_i $(1 \le a_i \le n; 1 \le b_i \le 10^9; 1 \le t_i \le 10^9)$. The i-th line contains description of the i-th launching.

It is guaranteed that the condition $t_i \le t_{i+1}$ $(1 \le i < m)$ will be satisfied.

Output

Print a single integer – the maximum sum of happiness that you can gain from watching all the fireworks.

Examples

For the input data:

```
50 3 1
49 1 1
26 1 4
6 1 10
```

the correct result is:

```
-31
```

For the input data:

```
10 2 1
1 1000 4
9 1000 4
```

the correct result is:

```
1992
```

Solution

Of course we will try to come up with a dynamic programming solution. Let $dp(i, j)$ denote the maximum sum of happiness you can gain from watching first i fireworks and standing in section j in the moment of i-th launch. Then we have the following transitions:

$$dp(i, j) = \max_{-td \le k \le td} (dp(i-1, j+k) + b_i - |a_i - j|)$$

where t is the time after the previous launch, i.e. $t = t_i - t_{i-1}$.

We have $O(nm)$ states, each transition can be done in $O(n)$ time, therefore the total time complexity is $O(n^2 m)$. The space complexity is $O(nm)$, which is also too much for the constrains in this problem.

We can speed it up, by using our monotone queue optimization. First let us note that the second part in the formula $(b_i - |a_i - j|)$ does not depend on k, therefore we need to focus on maximizing the first part. We can use here a segment tree (which will result in $O(nm \log n)$ time), but we can also use the monotone queue, as the interval $[j - td, j + td]$ is independent for all the fireworks. This allows us to speed up the solution to $O(nm)$.

We should also use the rolling array technique (keeping only previous row for the dp array), to reduce the space complexity to $O(n)$.

This problem can be also solved in $O(m \log m)$ time if we consider the fireworks as functions and their slopes.

Problem Eggs

7th Polish Olympiad in Informatics.
Limits: 1s, 32MB.
`https://kostka.dev/sp/egg`

It is known that an egg dropped from a sufficiently high height breaks. Formerly one floor was enough, but genetically modified chickens lay eggs that are unbreakable even after being dropped from $100\,000\,000$ floors. Egg strength tests are carried out using skyscrapers.

A special egg strength scale has been developed: the egg has a strength of k floors if dropped from k-th floor does not crash, but dropped from $k + 1$, it crashes. In case when we use the skyscraper with n floors we assume that the egg breaks when

we drop it from $(n+1)$-th floor. We also assume that every egg dropped from floor number 0 never crashes.

The head of the laboratory decided to introduce savings in the research process. They limited the number of eggs that can be broken during an experiment to determine egg strength of a given species. In addition, the number of egg drops should be minimized. This means that having a certain number of eggs of a given species and a skyscraper with n floors, you should determine in as few attempts as possible what is the strength of eggs of a given species.

Communication

Your task is to write the following function:

- `void perform_experiment(int` n`, int` m`)` – this function will be called at the beginning of each new experiment (there might be more than one experiment in one test case), meaning that we are supposed to find the strength of eggs of some species, using a skyscraper with n floors $(1 \leq n \leq 100\,000\,000)$ and using m eggs $(1 \leq m \leq 1000)$.

Your function can call the following functions:

- `bool ask_query(int` x`)` – this function is used to ask a question, returns whether the egg dropped from a certain floor x withstands or breaks down;
- `void answer(int` k`)` – this function should be called only once, when you determined the strength of this egg species k. After using this function your function should finish (but do not finish the whole program!).

Note: Do not assume that the judge actually sets some egg strength before starting a given experiment. The judge can choose the strength during the experiment in a way to match all previously given answers and to force your program to ask as many questions as possible. You should strive to get the number of questions asked by your program in the worst case as small as possible.

Solution

Let us focus on how to calculate the necessary number of drops. Let us use the dynamic programming approach. Let $dp(n,m)$ denote the minimum number of drops needed to determine the strength of the eggs for a skyscraper with n floors, using m eggs. The first observation is as follows: if we know already that the egg crashes

from floor l and does not crash from floor r, we can think about it as a problem for a skyscraper with $r - l$ floors.

Therefore we can come up with the following formulas:

$$\begin{cases} dp(0, m) = 0 \\ dp(n, m) = 1 + \min_{1 \le j \le n-1} \max(dp(j, m-1), dp(n-j, m)) \text{ for } n > 0 \end{cases}$$

The second equation checks all possible j where we can drop an egg and then checks two possibilities: either the egg crashed (and we have to check j floors with $m - 1$ eggs) or it didn't crash (and we have to check $n - j$ floors with m eggs).

Using these transitions, we can implement a solution working in $O(n^2 m)$ time. We need to resolve one detail: how to find the optimal drops, but we will leave this as an exercise for the reader.

Let us use try to optimize this solution. In the second equation, let us denote $cost_1(j) = dp(j, m-1)$ and $cost_2(j) = dp(n-j, m)$. $cost_1$ is a non-increasing function, while $cost_2$ is a non-decreasing function, therefore $\max(cost_1, cost_2)$ is bitonic and we can find an optimum $opt(i)$ of function $\max(cost_1, cost_2)$ as a root of a function $cost_1 - cost_2$ using binary search. This results in $O(nk \log n)$ solution.

To speed it up even further, let us notice that if we move from n to $n+1$, $cost_1(j)$ stays the same, while $cost_2(j)$ does not decrease, hence $opt(n+1) \ge opt(n)$. Therefore, when we want to calculate $opt(n+1)$, we can start from $opt(n)$ and increase it gradually until $cost_1$ becomes smaller than $cost_2$. This results in $O(nk)$ time solution.

This particular problem can be solved even faster, by changing the dimensions of the dynamic programming. Let $dp_2(k, m)$ denote the number of consecutive floors we can check using k drops and m eggs. Then:

$$\begin{cases} dp_2(k, 0) = dp_2(0, m) = 0 \\ dp_2(k, m) = dp_2(k-1, m) + dp_2(k-1, m-1) + 1 \text{ for } k, m > 1 \end{cases}$$

We may notice that the second formula resembles the formula for calculating the Newton binomial symbol:

$$\binom{k}{m} = \binom{k-1}{m} + \binom{k-1}{m-1}$$

And that is the right clue, as we can prove that:

$$dp_2(k, m) = \sum_{i=1}^{m} \binom{k}{i}$$

Total time complexity of this solution is $O(mk)$, where k is the number of drops we need to perform in the given experiment, i.e. the smallest integer for which $dp_2(k, m) \ge$

n. How we can estimate k? If $m = 1$, then in the worst case we need exactly $k = n$ drops. If $m = 2$, we have:

$$dp_2(k, 2) = \binom{k}{1} + \binom{k}{2} = k + \frac{k(k-1)}{2} > \frac{k^2}{2}$$

So we need at most $k < \sqrt{2n} < 15\,000$ drops in the worst case. Larger m give even better estimates.

Logarithmic ideas

Please note that in the problems above, other optimizations were possible. In the first problem, we could use the segment tree to get $O(nm \log n)$ complexity. In the second problem, we mentioned the binary search solution, with an additional $\log$-factor. In these problems, logarithmic optimizations were too slow, as we were able to reduce complexity by the linear order, but in other problems, using an additional $\log$ from adding binary/ternary search or a segment tree might be the only way to go.

6.3. Convex hull optimization

In this section, we will discuss the convex hull trick, which was first described in [Brucker, 1995], but the first well-known occurrence in sports programming was the problem "Batch Scheduling" from IOI 2002 (which we will describe below).

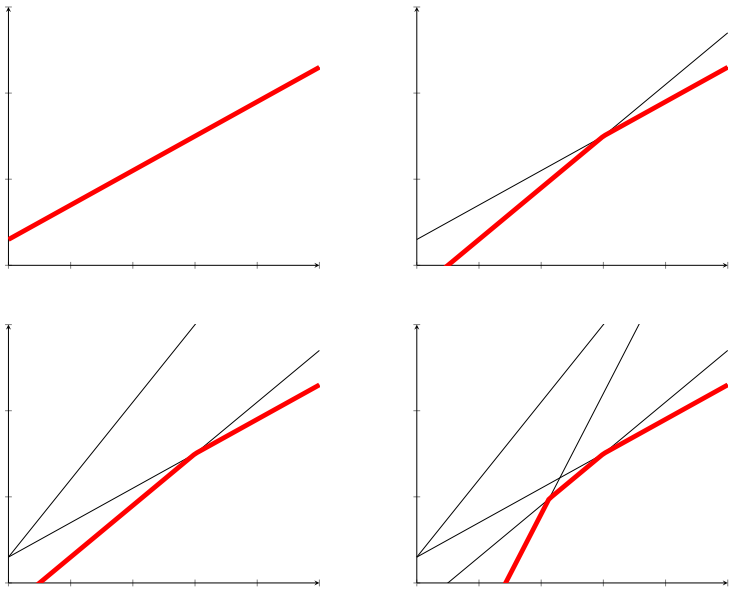
We will focus on the dynamic programming solutions with the following form:

$$dp(i) = \min_{j<i}(f(i) \cdot cost(j) + dp(j))$$

where $cost$ is a monotone function. The naive solution works in $O(n^2)$, but we can speed it up to $O(n \log n)$ or even to $O(n)$ in some cases. The key observation is that the function that we are trying to minimize resembles a line in $\mathbb{R}^2$. In particular, the whole problem can be reduced to simply two operations:

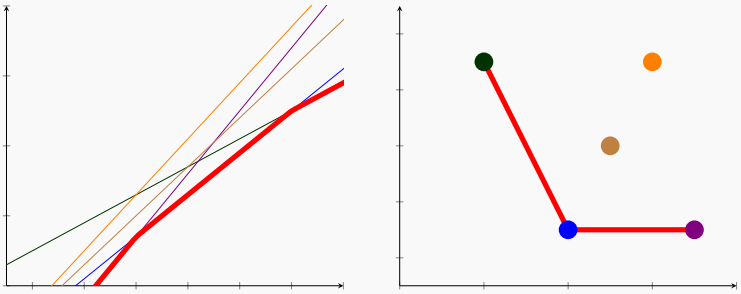
- inserting a linear function $mx+c$ to a set of linear functions (in the formula above $m = cost(j)$ and $c = dp(j)$),
- finding the minimum value from all the functions in the set for some argument x (in the formula above $x = f(i)$).

We will try to maintain a lower convex hull of the linear functions. In the picture below, we show how the convex hull (denoted by the red line) changes as we add more functions. We may notice that every line corresponds to at most one segment in the convex hull.



Dual problem

Instead of keeping linear functions $y = mx + c$, we can keep only points (m, c). Therefore we will try to minimize the dot product of points (m, c) with $(x, 1)$, which will be the same as minimizing the value of the function. Then we need to keep the convex hull on points, instead of lines.



First, we will try to solve an even simpler problem, where $f(i)$ is also a monotone function (here we do not care if it has the same monotonicity as $cost$, but notice that if both of these functions are, let us say, increasing, then the problem is not interesting).

Problem Batch scheduling

International Olympiad in Informatics 2002, second day.
Limits: 1s, 64MB.
`https://kostka.dev/sp/bat`

There is a sequence of N jobs to be processed on one machine. The jobs are numbered from 1 to N, so that the sequence is $1, 2, \ldots, N$. The sequence of jobs must

be partitioned into one or more batches, where each batch consists of consecutive jobs in the sequence. The processing starts at time 0. The batches are handled one by one starting from the first batch as follows. If a batch b contains jobs with smaller numbers than batch c, then batch b is handled before batch c. The jobs in a batch are processed successively on the machine. Immediately after all the jobs in a batch are processed, the machine outputs the results of all the jobs in that batch. The output time of a job j is the time when the batch containing j finishes.

A setup time S is needed to set up the machine for each batch. For each job i, we know its cost factor F_i and the time T_i required to process it. If a batch contains the jobs $x, x+1, \ldots, x+k$, and starts at time t, then the output time of every job in that batch is $t + S + (T_x + T_{x+1} + \ldots + T_{x+k})$. Note that the machine outputs the results of all jobs in a batch at the same time. If the output time of job i is O_i, its cost is $O_i \times F_i$. For example, assume that there are 5 jobs, the setup time $S = 1$, $(T_1, T_2, T_3, T_4, T_5) = (1, 3, 4, 2, 1)$, and $(F_1, F_2, F_3, F_4, F_5) = (3, 2, 3, 3, 4)$. If the jobs are partitioned into three batches $\{1, 2\}, \{3\}, \{4, 5\}$, then the output times are $(O_1, O_2, O_3, O_4, O_5) = (5, 5, 10, 14, 14)$ and the costs of the jobs are $(15, 10, 30, 42, 56)$, respectively. The total cost for a partitioning is the sum of the costs of all jobs. The total cost for the example partitioning above is 153.

You are to write a program which, given the batch setup time and a sequence of jobs with their processing times and cost factors, computes the minimum possible total cost.

Input

Your program reads from standard input. The first line contains the number of jobs N, $1 \leq N \leq 10\,000$. The second line contains the batch setup time S which is an integer, $0 \leq S \leq 50$. The following N lines contain information about the jobs $1, 2, \ldots, N$ in that order as follows. First on each of these lines is an integer T_i, $1 \leq T_i \leq 100$, the processing time of the job. Following that, there is an integer F_i, $1 \leq F_i \leq 100$, the cost factor of the job.

Output

Your program writes to standard output. The output contains one line, which contains one integer: the minimum possible total cost.

Examples

For the input data:	the correct result is:
2 50 100 100 100 100	45000

For the input data:	the correct result is:
5 1 1 3 3 2 4 3 2 3 1 4	153

Solution

Of course we will use dynamic programming to solve this problem. For simplicity of calculating the cost, we will construct batches from right to left. Let $dp(i)$ denote the minimum total cost of partitioning jobs $(i, i+1, \ldots, N)$ into batches. Then:

$$dp(i) = \min_{j>i} \left(dp(j) + f(i) \cdot cost_i(j) \right).$$

where:

- $f(i) = F_i + F_{i+1} + \ldots + F_N$,
- $cost_i(j) = T_i + T_{i+1} + \ldots + T_{j-1}$.

Let us notice that both f and $cost_i$ are monotone.

We want to minimize $dp(j) + f(i) \cdot cost_i(j)$ with regard of j. When will we choose some j_1 over j_2 (where $j_1 > j_2$)?

$$f(i) \cdot cost_i(j_1) + dp(j_1) < f(i) \cdot cost_i(j_2) + dp(j_2)$$

$$f(i) < \frac{dp(j_2) - dp(j_1)}{cost_i(j_1) - cost_i(j_2)}$$

Let us denote the expression on the right by $cmp(j_1, j_2)$, i.e.: $cmp(j_1, j_2) = (dp(j_2) - dp(j_1))/(cost_i(j_1) - cost_i(j_2))$. Note that the denominator does not depend on i anymore. Then we will say that j_1 is better than j_2 if $cmp(j_1, j_2) > f(i)$.

Observation 1. *For $i < j_3 < j_2 < j_1$, if $cmp(j_1, j_2) > cmp(j_2, j_3)$, then j_2 will never be chosen as the optimum.*

Proof. We have two cases, depending on the relation between $f(i)$ and $cmp(j_1, j_2)$:

- $f(i) \geq cmp(j_1, j_2)$, then $f(i) > cmp(j_2, j_3)$, and $f(i) < cmp(j_3, j_2)$, and we can choose j_3 over j_2,
- $f(i) < cmp(j_1, j_2)$, then we simply should choose j_1 over j_2. □

The first observation shows that we can only maintain a set of candidates $(j_l, j_{l+1}, \ldots, j_r)$, where $cmp(j_l, j_{l+1}) \leq cmp(j_{l+1}, j_{l+2}) \leq \ldots \leq cmp(j_{r-1}, j_r)$.

Going back to our interpretation with functions and convex hull, this means that we will keep only the lower convex hull. Moreover, these functions will be given in the order of decreasing slopes, therefore we can just keep this hull using a stack. Then we can find the optimum for each i using binary search, which will result in $O(n \log n)$ solution. But we can use one more observation to speed it up even more.

Observation 2. *If $cmp(j_1, j_2) < f(i)$ for $i < j_2 < j_1$, then we do not have to consider j_1 in further steps $(i - 1, i - 2, \ldots)$.*

Proof. This is trivial, as $f(i)$ is a decreasing function. □

The second observation is telling us how we can choose the optimal j for i: we will remove elements from the front of the set of candidates mentioned above, until $cmp(j_l, j_{l+1}) \geq f(i)$. As you can notice, we only need to append elements to the back of this set, and pop elements from the front and back, therefore we can use a deque again. In this simplified case, this algorithm is just a variation on the monotone queue

optimization, where the element that is monotone is the slope of the function.

```
D ← deque();
for i ← n − 1 to −1
    // pop elements out of the bounds from the front
    while D.size() ≥ 2 and cmp(D.front(), next(D.front())) < f(i)
        D.pop_front();
    // calculate result using the front of the deque
    if not D.empty()
        dp[i] ← dp[D.front()] + f(i) · cost(D.front());
    // pop elements no longer needed
    while D.size() ≥ 2 and cmp(prev(D.back()), D.back()) > cmp(D.back(),
     i)
        D.pop_back();
    // insert i at the back
    D.push_back(i);
```

This solution works in $O(n)$ time and $O(n)$ memory.

To summarize, we managed to optimize $O(n^2)$ solution to $O(n)$ by using two observations. The first one allowed us to store the candidates for the optimum as a lower convex hull of linear functions on the stack, which resulted in $O(n \log n)$ solution. Sometimes (when f is also a monotone function) we can do one more step and reduce this problem to keeping the monotone queue with proper candidates, which results in $O(n)$ time solution.

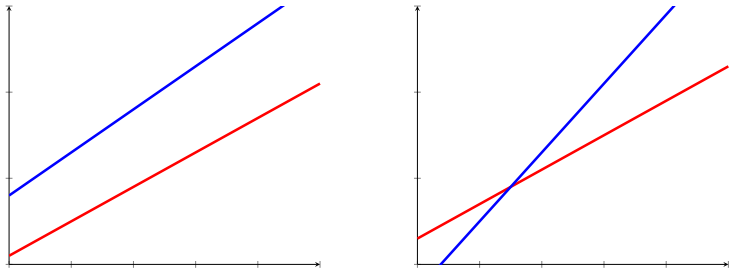
6.3.1. Li Chao tree

Now let us focus on the online version of this problem. We need to have a data structure that will allow us to keep a set of linear functions and to find the minimum value of these functions for some given argument. We only assume that every two of these functions intersect at most once.

Li Chao introduced the following data structure, which was first presented during his lecture at Zhejiang Provincial Selection in 2013 and is now commonly known as "Li Chao tree". The main idea is to just keep a segment tree on these linear functions. We will keep a segment tree, where each node in the segment tree covers some interval and store one function. We will make sure that for every point p covered by a tree, we can find a leaf covering the interval containing this point, so we can find a function that minimize all the values in this segment by traversing all the functions from the leaf to the root and checking values of these functions in p.

If we have only one function (red on the picture), we will just keep this function over the whole interval. Now imagine we want to add a new function (blue on the

picture) to our set of functions. What can happen?



We have two possible cases:

1. either one of the functions will majorize the second (will be always greater or lesser), as shown on the left hand side of the picture above,
2. or they will cross somewhere within this interval, as shown on the right hand side.

In the first case, we can just modify the function in the node (if needed). In the second case, we need to find in which child the intersection point (the point where the minimum of these function changes) occurs and continue our recursive process there. For the second child, we can just choose the function that minimizes the segment. We can do that by checking the values of these functions in one of the bounds of the corresponding interval (let us say the left bound) and in the middle of that interval.

Function *InsertFunction(Node node, Function f)*:

```
mid ← ½(node.left + node.right);
// below we are checking which function is lower on the left
   bound of the interval, and in the middle of the interval
lowerLeft ← f(node.left) < node.f(node.left);
lowerMid ← f(mid) < node.f(mid);
if lowerMid
    node.f = f;
if node is a leaf
    return
if lowerLeft = lowerMid
    InsertFunction(node.leftChild, f);
else
    InsertFunction(node.rightChild, f);
```

We can also do it dynamically: we can use binary search to find the intersection point and split the interval into two (uneven) intervals. Then, each node in our segment tree will cover some interval, but each node will still store exactly one function.

Now if we want to check the minimum for some value, we just walk over the path to the leaf of this value and take the minimum value from all the functions found on this path.

Function *GetMinimumValue(Node* $node$*,* x*)*:
    if *node is a leaf*
        return $node.f(x)$
    $mid \leftarrow \frac{1}{2}(node.left + node.right)$;
    if $x < mid$
        return $\min(node.f(x),$ *GetMinimumValue(*$node.leftChild,\ x$*))*
    else
        return $\min(node.f(x),$ *GetMinimumValue(*$node.rightChild,\ x$*))*

Please note that we can also adapt this approach for other functions that can cross in at most one point or just line segments.

Removing elements from the persistent data structures

One may ask if we can delete some functions from the set kept in the Li Chao tree. There is a handy method that allows us to delete elements from any persistent data structure by adding just an additional $\log$ factor in the time complexity. We assume here that we do not have to do it online, so we know in advance when each element will be removed.
We are keeping a segment tree over the time T. In each node we will keep a set of elements that correspond to this time (are inserted in the data structure in this period). Then for each element that should be added and eventually later removed from that data structure, we just need to add this element in $O(\log T)$ nodes. Then if we want to know something about the data structure at subsequent moments of time, we run the DFS algorithm in this segment tree. We keep a copy of this persistent data structure. When we enter the node, we add the elements in this node, but when we leave the node, we need to remove these elements. Here we are using the persistency – for example we can keep all the updates on the stack, so we can remove all the updates we did in this node.

6.4. Knuth optimization

Gilbert and Moore in [Gilbert and Moore, 1959] tried to solve the problem of finding the optimal binary tree. In this section, we will describe a simplified version of this problem. We are given values $a_1 < a_2 < \ldots < a_n$ with corresponding probabilities of being chosen $p_1, p_2, \ldots, p_n$ and we are asked to organize them into a binary tree in a way that the sum of the probabilities multiplied by the depths of the values in the tree is minimum. (The root has depth 1.) One can notice that we can simplify the

problem and forget about the values a_i and just focus on the probabilities. The only thing that we have to remember is that we cannot reorder these probabilities.

We will start by proposing the natural dynamic programming solution. Let $dp(l, r)$ will be the cost of the optimal tree that may be built over the values from l to r. Then:

$$\begin{cases} dp(i,i) = p_i \\ dp(l,r) = \min_{l \le i \le r}(dp(l,i-1) + (p_l + p_{l+1} + \ldots + p_r) + dp(i+1,r)) \end{cases}$$

In the second equation, we are looking for the optimal value a_i to take as a root of the subtree for all the values between l and r and we are paying the cost of putting all the nodes in the next level and the cost of two subtrees (one from l to $i-1$ and the second one from $i+1$ to r). Please note that all the sums can be precalculated (and we will denote $cost(l, r) = (p_l + p_{l+1} + \ldots + p_r)$). The total complexity of this dynamic programming solution (proposed by Gilbert and Moore) is $O(n^3)$ time and $O(n^2)$ memory.

Let us try to notice something about the function $cost$. We will say that the function $cost$ for $a \le b \le c \le d$:

- is *monotone* if
$$cost(b,c) \le cost(a,d),$$
- satisfies the *quadrangle inequality (QI)* if
$$cost(a,c) + cost(b,d) \le cost(a,d) + cost(b,c).$$

The first property is also known in the literature as *monotonicity on the lattice of intervals (MLI)*.

In our case, it is easy to notice that our cost function is monotone (if we broaden our interval, we have to pay more). To prove that our function satisfies QI, we just need to see that

$$cost(a,c) + cost(b,d) = (pref_c - pref_{a-1}) + (pref_d - pref_{b-1})$$
$$= (pref_d - pref_{a-1}) + (pref_c - pref_{b-1}) = cost(b,c) + cost(a,d)$$

where $pref_k = p_1 + p_2 + \ldots + p_k$.

Yao in [Yao, 1980] proved that these two properties of the cost function are sufficient to speed up any dynamic programming using it from $O(n^3)$ to $O(n^2)$ time. We need two observations.

Observation 1. *For the following dynamic programming function:*

$$\begin{cases} dp(i,i) = 0 \\ dp(l,r) = \min_{l \le k \le r} dp(l,k-1) + dp(k+1,r) + cost(l,r), \end{cases}$$

if $cost$ function is monotone and satisfies QI, then dp function also satisfies QI.

We skip the proof here, as it involves only case analysis. For more details, see Yao's original paper.

Observation 2. *For the dynamic programming function from the first observation, let us denote the optimum value of k for $dp(l,r)$ mentioned above as $opt(l,r)$ (if there are several such optimums, then we choose the latest one). Then $opt(l,r-1) \leq opt(l,r) \leq opt(l+1,r)$.*

In our case with the optimal binary tree problem, that means that extending the tree by adding a new value a_r to the already optimal tree for values $(a_l, a_{l+1}, \ldots, a_{r-1})$ will not move the root of the tree to the left (and symmetrically).

Proof. We will prove that $opt(l,r-1) \leq opt(l,r)$, as the second inequality is symmetric. We just need to show that for $l \leq k_1 \leq k_2 \leq r$:

$$\begin{gathered} dp(l,k_2-1)+dp(k_2+1,r-1)+cost(l,r-1) \\ \leq dp(l,k_1-1)+dp(k_1+1,r-1)+cost(l,r-1) \\ \Downarrow \\ dp(l,k_2-1)+dp(k_2+1,r)+cost(l,r) \leq dp(l,k_1-1)+dp(k_1+1,r)+cost(l,r) \end{gathered}$$

As dp satisfies QI, we can say that:

$$dp(k_1+1,r-1)+dp(k_2+1,r) \leq dp(k_2+1,r-1)+dp(k_1+1,r)$$

Then we can add $cost(l,r-1)+cost(l,r)+dp(l,k_1-1)+dp(l,k_2-1)$ to both sides and we get:

$$\begin{aligned} &(dp(l,k_1-1)+dp(k_1+1,r-1)+cost(l,r-1))+(dp(l,k_2-1)+dp(k_2+1,r)+cost(l,r)) \\ &\leq (dp(l,k_1-1)+dp(k_1+1,r)+cost(l,r))+(dp(l,k_2-1)+dp(k_2+1,r-1)+cost(l,r-1)), \end{aligned}$$

which yields the implication above. □

We have shown that for $dp(l,r)$ we need to look for the optimum k only in range $[opt(l,r-1), opt(l+1,r)]$. The algorithm then works as follows: we can calculate the values of our dynamic programming algorithm in the order of increasing range $d = r - l$, but with our observation, the running time for a fixed d will be equal to $(opt(d+1,2)-opt(d,1)+1)+(opt(d+2,3)-opt(d+1,2)+1)+\ldots+(opt(n,n-d+1)-opt(n-1,n-d)+1) = opt(n,n-d+1)-opt(d,1)+n$, so only $O(n)$ time. The overall

running time is $O(n^2)$.

```
precalculate cost function;
for len ← 0 to n − 1
    for a ← 1 to n − len
        b ← a + len;
        for i ← opt(a, b − 1) to opt(a + 1, b)
            costAtI = dp(l, i − 1) + dp(i + 1, r) + cost(l, r);
            if costAtI ≤ dp(a, b)
                dp(a, b) ← costAtI;
                opt(a, b) ← i;
return dp(1, n)
```

Knuth in [Knuth, 1971] first proposed the solution for the problem of finding the optimal binary search tree working in $O(n^2)$ time, but his approach was problem-specific. Yao, on the other hand, gave us simple conditions for the cost function which are applicable in many problems, see [Yao, 1982] and [Bar-Noy and Ladner, 2004].

Problem Drilling

Algorithmic Engagements 2009.
Limits: 2s, 64MB.
`https://kostka.dev/sp/dri`

Byteman is the person in charge of a team that is looking for crude oil reservoirs. He has made two boreholes: he found crude oil in point A and found out that there is no crude oil in point B. It is known that the oil reservoir occupies a connected fragment of segment $\overline{AB}$ with one end at point A. Now Byteman has to check, how far, along the segment connecting points A and B, does the oil reservoir reach. It is not that simple, however, because in some locations one can drill faster than in other locations. Moreover, Byteman's team is rather small, so they can drill in at most one location at a time. Byteman's boss would like him to predetermine when he will be able to identify the boundary of the oil reservoir.

Byteman has asked you for help. He has divided the segment connecting points A and B into $n+1$ segments of equal length. If we assume that point A has coordinate 0, and point B coordinate $n+1$, then there are points with coordinates $1, 2, 3, \ldots, n$ between them. It is enough to find the farthest from A of these points in which some crude oil occurs. Byteman has informed you about the amounts of time necessary for making boreholes in these points – they are equal to $t_1, t_2, \ldots, t_n$ respectively. You should create such a plan of drilling, that the time necessary to identify the oil reservoir's boundary is shortest possible, assuming the worst-case scenario.

Input

The first line of the standard input contains a single positive integer n ($1 \leq n \leq 2000$). The second line contains n positive integers $t_1, t_2, \ldots, t_n$ separated by single spaces ($1 \leq t_i \leq 10^6$).

Output

Your program should write a single integer to the standard output - the smallest amount of time that Byteman has to spend (assuming the worst-case scenario) drilling in search of oil, to be sure that he will identify the reservoir's boundary.

Example

For the input data:	the correct result is:
`4` `8 24 12 6`	`42`

Solution

This solution is based on [Idziaszek, 2015].

First, let us assume that all the times are equal. In this case, in the worst scenario, we have to drill $\lfloor \log n \rfloor + 1$ times (using the binary search).

The first idea is to construct a dynamic programming solution that will work in $O(n^3)$ time. Let $dp(l, r)$ denote the optimal time of finding the reservoir's boundary within the bounds l and r (both inclusively), assuming that $l - 1$ contains oil, while $r + 1$ does not. Then:

$$dp(l, r) = \min_{l \leq i \leq r} (t_i + \max(dp(l, i - 1), dp(i + 1, r))) \qquad (*)$$

We are looking here for a place to drill (i), and then we will call our function recursively – we will spend at most $\max(dp(l, i-1), dp(i+1, r))$ time, depending on the answer in i.

To speed up this algorithm, we will need several observations. First, let us note that the monotonicity condition for dp is fulfilled. If we already have result $dp(l, r)$ for some interval $[l, r]$, then expanding this interval in any direction will never cause

result to decrease. We know that with the increasing i, the value $dp(a, i-1)$ does not decrease, and the value $dp(i+1, b)$ does not increase. Therefore we can find an index $opt(l, r) \in (l-1, r)$, so that the following conditions are fulfilled:

- $dp(l, i-1) \leq dp(i+1, r)$ for $i \leq opt(l, r)$,
- $dp(l, i-1) > dp(i+1, r)$ for $i > opt(l, r)$.

Because of that, we can rewrite the dynamic programming formula (*):

$$dp(l, r) = \min(\min_{l \leq i \leq opt(l,r)}(t_i + dp(i+1, r)), \min_{opt(l,r) < i \leq r}(t_i + dp(l, i-1))) \qquad (**)$$

Now, we can also notice that opt is also monotonic, i.e.:

$$opt(l, r-1) \leq opt(l, r) \leq opt(l+1, r)$$

Indeed, if we take any $i \leq opt(l, r-1)$ and use the fact that dp is monotonic, ergo expanding the segment will not decrease the cost, then $dp(l, i-1) \leq dp(i+1, r-1) \leq dp(i+1, r)$, which proves that $i \leq opt(l, r)$. The right hand side of the inequality above is symmetric.

Now the only thing left is how to calculate these minimums in (**). We will calculate dp function in the order of the increasing lengths of the segments. We will use the monotone queue here. We will keep n pairs of deques, let us denote them by $A[i]$ and $B[i]$ for $i = 1, 2, \ldots, n$. In $A[l]$ we will store minimums for the right minimum (in order of increasing indices) for segments starting in l, while in $B[r]$ we will store minimums for the left minimum (in order of decreasing indices) for segments starting in r. As we calculate $dp(l, r)$ after calculating $dp(l-1, r)$ and $dp(l, r-1)$, all updates in these deques can be done in amortized $O(1)$ time, which leads to the total time and

memory complexity of $O(n^2)$.

```
for len ← 0 to n − 1
    for l ← 1 to n − len
        r ← l + len;
        for i ← opt(l, r − 1) to opt(l + 1, r)
            if dp(l, i − 1) ≤ dp(i + 1, r)
                opt(l, r) ← i;
        while not A[l].empty() and A[l].back().second ≥ t[r] + dp(l, r − 1)
            A[l].pop_back();
        A[l].push(r, t[r] + dp(l, r − 1));
        while not (A[l].empty()) and A[l].front() ≤ opt(l, r)
            A[l].pop_front();
        while not B[r].empty() and B[r].back().second ≥ t[l] + dp(l + 1, r)
            B[r].pop_back();
        B[r].push(l, t[l] + dp(l + 1, r));
        while not (B[r].empty()) and B[r].front() > opt(l, r)
            B[r].pop_front();
        dp(l, r) = min(A[l].front(), B[r].front());
return dp(1, n)
```

Note that this solution is merely based on Yao's method, but we did not used this method explicitly.

6.5. Divide and conquer optimization

We will start this section with a very classical problem from the ICPC World Finals.

Problem Money for Nothing

ICPC World Finals 2017.
Limits: 4s, 1GB.
`https://kostka.dev/sp/mon`

In this problem, you will be solving one of the most profound challenges of humans across the world since the beginning of time – how to make lots of money.

You are a middleman in the widget market. Your job is to buy widgets from widget producer companies and sell them to widget consumer companies. Each widget consumer company has an open request for one widget per day, until some end date, and a price at which it is willing to buy the widgets. On the other hand, each widget

producer company has a start date at which it can start delivering widgets and a price at which it will deliver each widget.

Due to fair competition laws, you can sign a contract with only one producer company and only one consumer company. You will buy widgets from the producer company, one per day, starting on the day it can start delivering, and ending on the date specified by the consumer company. On each of those days you earn the difference between the producer's selling price and the consumer's buying price.

Your goal is to choose the consumer company and the producer company that will maximize your profits.

Input

The first line of input contains two integers m and n $(1 \leq m, n \leq 500\,000)$ denoting the number of producer and consumer companies in the market, respectively. It is followed by m lines, the i-th of which contains two integers p_i and d_i $(1 \leq p_i, d_i \leq 10^9)$, the price (in dollars) at which the i-th producer sells one widget and the day on which the first widget will be available from this company. Then follow n lines, the j-th of which contains two integers q_j and e_j $(1 \leq q_j, e_j \leq 10^9)$, the price (in dollars) at which the j-th consumer is willing to buy widgets and the day immediately after the day on which the last widget has to be delivered to this company.

Output

Display the maximum total number of dollars you can earn. If there is no way to sign contracts that gives you any profit, display 0.

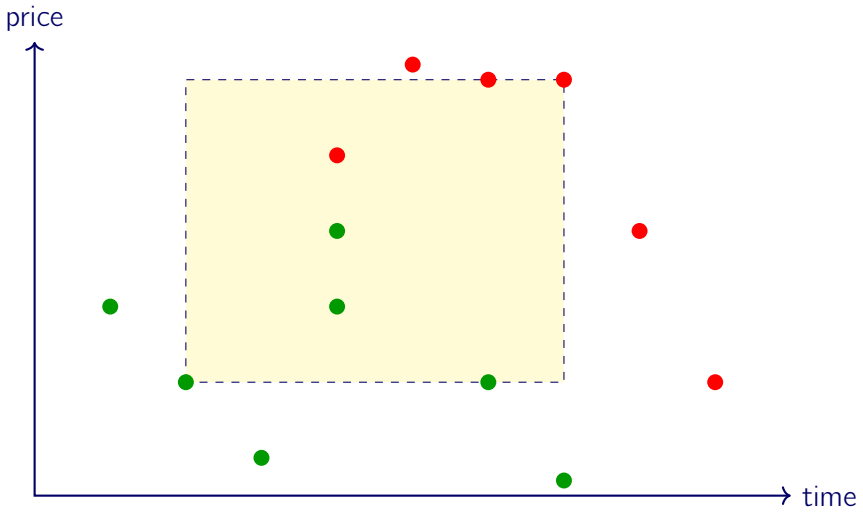
Examples

For the input data:	the correct result is:
2 2 1 3 2 1 3 5 7 2	5

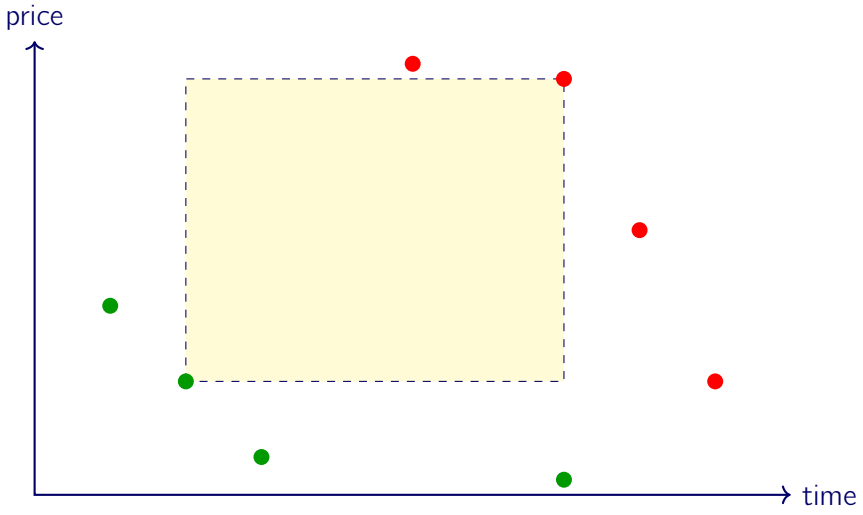
And for the input data:	the correct result is:
1 2 10 10 9 11 11 9	0

Solution

If we think about producers and consumers as 2D points, we are looking for the maximum area of some rectangle, where the bottom-left point is some producer (the green points on the picture below), and the top-right point of this rectangle is some consumer (the red points).

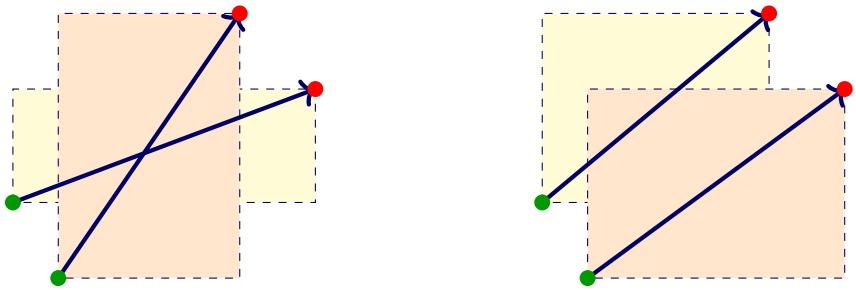


The first observation is quite easy: if for a producer p we can find any producer that is both cheaper and can start producing faster than p, then we can forget about p. A similar thing can be said about consumers. All remaining red and green points will form two chains.



Now for some chosen producer company, let us try to find the best consumer. We can iterate over all consumers in $O(n)$ time and choose the rectangle with the maximum

area. Now, we can observe that if we match all consumers with their respective best producers, and draw arrows between them, then the arrows will never cross. We can formally prove it by contradiction, i.e. we have the situation as shown on the left hand side of the picture below:



If we swap these pairs, as shown on the right hand side, we can see that we will get rectangles with an area not smaller than of the ones on the left. One can check that formally by writing proper inequalities.

Therefore, we can find the optimal consumer c for some producer p, and then we can use divide and conquer to split this problem into two: for all producers to the left of p find their optimal consumers, knowing that either they are equal to c or they are somewhere to the left of c. Similarly for the right side.

This results in $O((n+m)\log(n+m))$ solution.

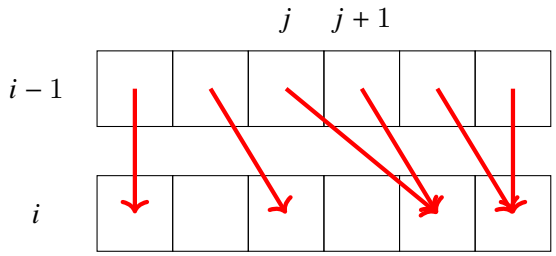
We will use the approach described in the problem above to optimize another form of dynamic programming:

$$dp(i,j) = \min_{k<j} (dp(i-1,k) + cost(j,k))$$

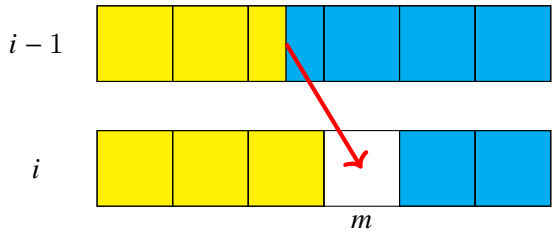
for $i = 1, 2, \ldots, N$ and $j = 1, 2, \ldots, M$ and where $cost$ is some function computable in $O(1)$.

In the general case, we can solve it in $O(NM^2)$ time. Let us once again denote the smallest optimal k for $dp(i,j)$ as $opt(i,j)$. If the condition $opt(i,j) \leq opt(i,j+1)$ is fulfilled, then we can speed it up to $O(NM \log M)$ time.

What does this condition actually mean? If we are calculating the dp function for layer i, let us think where the optimums are in comparison to the previous row $i-1$ Our condition means that if we will draw arrows from every element $opt(i-1,j)$ to the smallest element $opt(i,*)$ greater than or equal to $opt(i-1,j)$, these arrows will never cross themselves.



Because of that, we can use the divide and conquer approach, as seen in the problem with consumers and producers. When we calculate the next row, we can find the optimum for the middle index m in $O(M)$ time. Now if we find this optimum, let us note that all the elements to the left of m can have their optimum at the same place or to the left of the optimum for m, i.e. $opt(i,l) \leq opt(i,m)$ for $l < m$. We can make a symmetric statement for the elements to the right of m. Moreover, we can continue using the divide and conquer method in both parts that we have now.



Therefore for calculating the layer i, we can use the following recursive function:

Function *calculateDp*(i, l, r, opt_l, opt_r)**:**

```
opt ← opt_l;
mid ← (1/2)(l + r);
// find the optimum for mid in linear time
for k ← opt_l + 1 to opt_r
    if dp(i − 1, k) + cost(mid, opt) < dp(i − 1, opt) + cost(mid, opt)
        opt ← k;
dp(i, mid) ← dp(i − 1, opt) + cost(mid, opt);
// and solve two subproblems for the left and right side
calculateDp(i, l, mid − 1, opt_l, opt);
calculateDp(i, mid + 1, R, opt, opt_r);
```

which works in $O(M \log M)$. This allows us to reduce the time complexity to $O(NM \log M)$. The biggest problem in using this approach is to prove that opt fulfills the monotonicity condition.

For further reference, see [Galil and Park, 1992].

Problem Guardians of the Lunatics

IOI 2014 Practice Contest 2 @ Hackerrank.
Limits: 7 s, 512 MB.
`https://kostka.dev/sp/gua`

You are in charge of assigning guards to a prison where the craziest criminals are sent. The L cells form a single row and are numbered from 1 to L. Cell i houses exactly one lunatic whose *craziness level* is C_i.

Each lunatic should have one guard watching over them. Ideally, you should have one guard watching over each lunatic. However, due to budget constraints, you only have G guards to assign. You have to assign which lunatics each guard should watch over in order to minimize the total risk of having someone escape.

Of course, you should assign each guard to a set of adjacent cells. The *risk level* R_i that the lunatic in cell i can escape is given by the product of their craziness level C_i and the number of lunatics the guard assigned to them is watching over. Getting the sum of the R_i's from $i = 1$ to $i = L$ will give us the total amount of risk, R, that a lunatic might escape.

Given L lunatics and G guards, what is the minimum possible value of R?

Input

The first line of input contains two space-separated positive integers: L and G ($1 \leq L \leq 8000$, $1 \leq G \leq 800$), the number of lunatics, and the number of guards respectively.

The next L lines describe the craziness level of each of the lunatics. The i-th of these L lines describes the craziness level C_i ($1 \leq C_i \leq 10^9$) of the lunatic in cell block i.

Output

Output a line containing the minimum possible value of R.

Example

For the input data:	the correct result is:
6 3 11 11 11 24 26 100	299

Note: The first guard should be assigned to watch over the first three lunatics, each having a craziness level of 11. The second guard should be assigned to watch over the next two lunatics, having craziness levels of 24 and 26. The third guard should be assigned to the craziest lunatic, the one having a craziness level of 100.

The first three lunatics each have a risk level of 33, the product of 11 (their craziness level), and 3 (the number of lunatics their guard is watching over). The next three lunatics have a risk level of 48, 52, and 100. Adding these up, the total risk level is 299.

Solution

Let $dp(n, m)$ denote the minimum total cost of partitioning the first m lunatics between n guards. Then we can write the following dynamic programming formula:

$$\begin{cases} dp(1, m) = cost(1, m) \\ dp(n, m) = \min_{0 \le k \le m} (dp(n-1, k) + cost(k+1, m)) \text{ for } n > 1 \end{cases}$$

We are here determining what will be the group watched by the last guard. Note that $cost$ can be calculated in $O(1)$ time after $O(L)$ preprocessing by calculating the prefix sums, as $cost(i, j) = (C_i + C_{i+1} + \ldots + C_j) \cdot (j - i + 1)$.

Now we can speed it up by using the observation that $opt(n, m)$ (optimum k for the minimum in the transition above) is monotone for a fixed n. That means that we can use the divide and conquer optimization that will result in $O(LG \log G)$ time solution with $O(LG)$ memory. Memory can be reduced to $O(L)$, by using the rolling array method. To prove the monotonicity, use the fact that $cost$ satisfies quadrangle inequality (compare with the section about the Knuth's optimization).

As $cost$ in this particular problem fulfills QI, we can check if we can use the Knuth's optimization. It turns out that we can and we can end up with the solution with $O(L^2)$ time complexity.

6.6. Lagrange optimization

The last optimization we will demonstrate originated in [Aggarwal et al., 1994] and uses the method of Lagrange multipliers. For more details about this method, see [de la Fuente, 2000] and [Bertsekas, 2014].

We will show how the Lagrange optimization works in the form of a solution to the following problem.

Problem Low-cost airlines

Algorithmic Engagements 2012.
Limits: 7s, 256MB.
`https://kostka.dev/sp/low`

Byteasar goes on a long-awaited vacation, which he is going to spend basking in the sun on the golden sands of the beaches of the Bytocki Sea. Taking into account his biorhythm, the weather forecast and the cultural attractions of Bytocia, Byteasar determined a recreation factor for each of the vacation days, which means how much fun Byteasar will have on a given day. Each of the coefficients is an integer; perhaps negative - it means that Byteasar would rather be at home and weeding his garden that day.

Fortunately, Byteasar does not have to spend his entire vacation at the seaside. His favorite low-cost airlines have prepared a promotion thanks to which Byteasar can buy k air tickets at an extremely attractive price (each ticket is for a trip to the Bytocki Sea and back).

Help Byteasar plan his vacation in such a way that the sum of the recreational factors for the days he will spend at the seaside is as high as possible, assuming that during his vacation he can fly to the seaside at most k times. For simplicity, we assume that planes run at night.

Input

The first line of input contains two integers n and k $(1 \leq k \leq n \leq 1\,000\,000)$. In the second line there are n numbers r_i $(-10^9 \leq r_i \leq 10^9)$ which describe the recreation coefficients of successive days of Byteasar's leave.

Output

The only line of output should contain one integer, which represents the sum of the recreation coefficients in the optimal vacation plan.

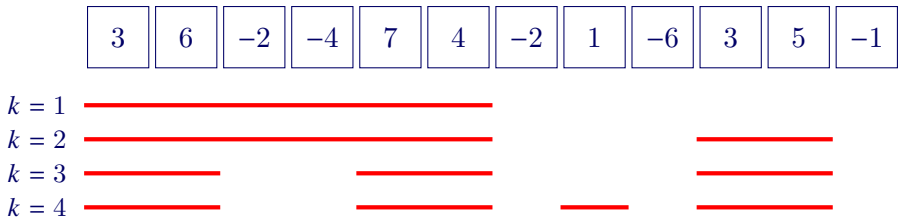
Example

For the input data:	the correct result is:
5 2 7 -3 4 -9 5	13

Solution

When $k = 1$ it is a somewhat standard problem and we can solve it in many ways, the most famous one is probably Kadane's algorithm (fun note: Kadane designed his linear solution within a minute at a seminar at Carnegie Mellon University) [Bentley, 1984].

Now, let us go back to the original problem and consider the following input array: $[3, 6, -2, -4, 7, 4, -2, 1, -6, 3, 5, -2]$. When $k = 1$, we can easily notice that we should take $[3, 6, -2, -4, 7, 4]$ to get the best result (14), but when we have $k = 2$, we can add $[3, 5]$ as another interval and get 22. Now if we continue, for $k = 3$, the optimal answer is to take $[3, 6], [7, 4], [3, 5]$. If $k = 4$ we can add last 1. Notice that we will not increase our answer any more if we add more intervals, as we already used all positive numbers.



We can make an interesting observation from this example. When we increase k, in every step we will either add an interval that is completely outside of any intervals already chosen or we will split an already chosen interval by choosing the interval with the minimum sum and removing it. Proving this observation we will leave as an exercise for the reader.

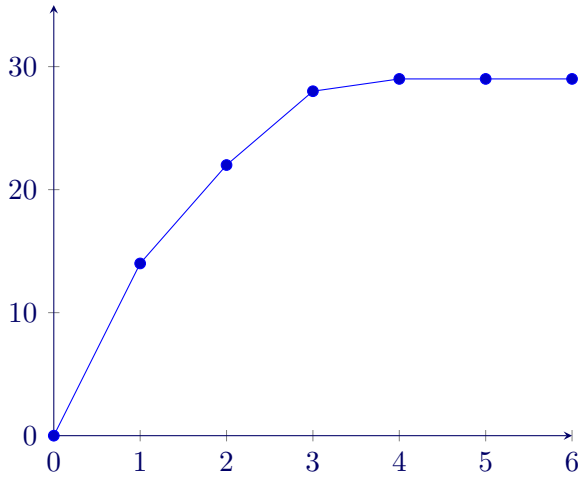
Now we can make some further observations. It is pretty easy to notice that if $ans(k)$ is the answer for k intervals, then $ans(k) \leq ans(k + 1)$, i.e. if we have more

intervals, then we can improve the result. Moreover, when k increases, then the difference between the results cannot increase, i.e.

$$ans(k) - ans(k-1) \geq ans(k+1) - ans(k).$$

This is a direct consequence of our first observation, as if we could add/remove an interval in step $k+1$ that is better for our solution than in step k, then we should swap these intervals and get a better solution in step k. Note that the inequality above means that the function ans is *concave*.

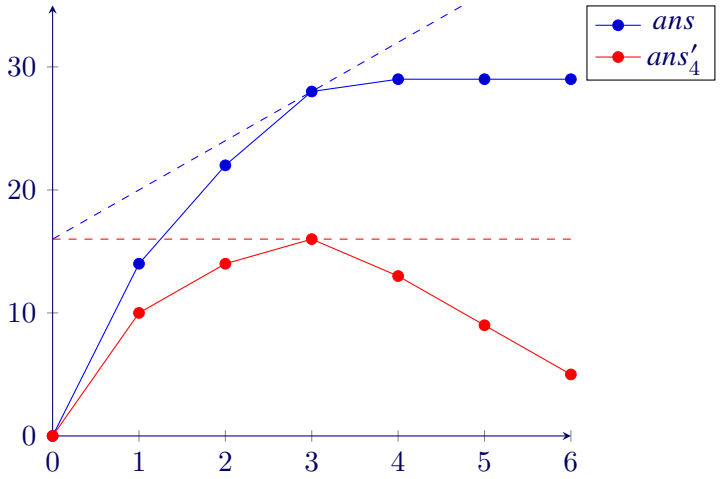
In our example, the function ans has the following shape.



Now let us go back and try to figure out the solution when we do not have a limit for k, i.e. we can choose as many intervals as we want. We can clearly choose all the positive integers and leave the negative behind. This solution works in $O(n)$.

Let us think about a different algorithm. We will add a penalty for creating a new interval, let us say C. This penalty is added so that our solution will be discouraged from creating new intervals all the time. For a fixed penalty, we use a dynamic programming algorithm to compute the answer in linear time. Let the result of the algorithm be $f(k)$. Of course when $C = 0$ this algorithm will find the solution with unlimited k, but when $C = \infty$ we will find the solution with $k = 0$. Now, what will happen in between?

Let us use a new function called ans'_C which will be equal to the cost of the solution for different values of k, but decreased by our penalty, i.e. $ans'_C(k) = ans(k) - C \cdot k$. One can check that ans'_C is also a concave function.



Note that when we compute $f(4)$, we will find some maximum value of $ans'_4(k)$, which in our case is for $k = 3$. Also notice that from $ans'_4(3)$ we can easily deduct $ans(3)$.

Can we find the optimal value C that will show the solution for a given k? The answer is positive. As our function ans is concave, we can take a tangent to the function for a given k, and then the slope of this function will determine the optimal C. (Check out the dashed lines in the plot above.) To find this C, we use binary search, as k is monotone with respect to C. In each step of the search, for a given value of C we compute k by finding the optimal solution using the dynamic programming approach.

Keep in mind that the line corresponding to the value C can be the tangent for many values of k.

This allows us to solve the problem in $O(n \cdot \log V)$, where $V = \sum_i \max(0, r_i)$, which is the maximum possible value of our result.

To use this trick in the general case, we need to prove that the function that we are trying to optimize in concave/convex (to be more precise, we need to prove the concave Monge property, see [Aggarwal et al., 1994] for more details). In most cases it is quite difficult to prove (even in this task, we skipped the formal proof), so during the competition it is not recommended to try to prove it formally. Sometimes you should trust your instinct or write a brute-force solution that can check this property on random testcases.

6.7. Summary

Below, we will shortly summarize all the optimizations we discussed in this chapter in a handy cheat sheet.

1. **Sum over subsets (SOS)**

- Transitions: $dp(mask) = \sum_{submask \subset mask} dp(submask)$, $\sum$ can be replaced by other operations.
- Complexity improvement: $O(4^n) \rightarrow O(n2^n)$.

2. **Monotone queue optimization**

- Transitions: $dp(i) = \min_{p(i) \leq j < i}(f(j)+cost(i))$, where $f(j)$ depends on $dp(j)$.
- Conditions: $p(i)$ is an increasing function.
- Complexity improvement: $O(n^2) \rightarrow O(n)$.

3. **Convex hull optimization**

- Transitions: $dp(i) = \min_{j<i}(f(i) \cdot cost(j) + dp(j))$.
- Conditions: $cost$ is a monotone function.
- Complexity improvement: $O(n^2) \rightarrow O(n \log n)$.
- If f is also monotone, then we can improve the time to $O(n)$.

4. **Knuth optimization**

- Transitions: $dp(l, r) = \min_{l \leq i \leq r}(dp(l, i) + dp(i, r) + cost(l, r))$.
- Conditions:
  - for opt: $opt(l, r-1) \leq opt(l, r) \leq opt(l+1, r)$,
  - or for $cost$: for all $a \leq b \leq c \leq d$: monotonicity (i.e. $cost(b, c) \leq cost(a, d)$) and quadrangle inequality (i.e. $cost(a, c) + cost(b, d) \leq cost(a, d) + cost(b, c)$).
- Complexity improvement: $O(n^3) \rightarrow O(n^2)$.

5. **Divide and conquer optimization**

- Transitions: $dp(i, j) = \min_{k<j}(dp(i-1, k) + cost(j, k))$.
- Condition: $opt(i, j) \leq opt(i, j+1)$.
- Complexity improvement: $O(nm^2) \rightarrow O(nm \log m)$.

6. **Lagrange optimization**

- Condition: Function that we are trying to optimize is concave/convex.
- Complexity improvement: $O(nk) \rightarrow O(n \log V)$.

As an exercise, we will also introduce a problem that focuses on applying these optimizations.

Problem Aliens

International Olympiad in Informatics 2016, second day.
Limits: 2s, 2GB.
`https://kostka.dev/sp/ali`

Our satellite has just discovered an alien civilization on a remote planet. We have already obtained a low-resolution photo of a square area of the planet. The photo shows many signs of intelligent life. Our experts have identified points of interest in the photo. The points are numbered from 0 to $n-1$. We now want to take high-resolution photos that contain all of those n points.

Internally, the satellite has divided the area of the low-resolution photo into an m by m grid of unit square cells. Both rows and columns of the grid are consecutively numbered from 0 to $m-1$ (from the top and left, respectively). We use (s,t) to denote the cell in row s and column t. The point number i is located in the cell (r_i, c_i) . Each cell may contain an arbitrary number of these points.

Our satellite is on a stable orbit that passes directly over the *main* diagonal of the grid. The main diagonal is the line segment that connects the top left and the bottom right corner of the grid. The satellite can take a high-resolution photo of any area that satisfies the following constraints:

- the shape of the area is a square,
- two opposite corners of the square both lie on the main diagonal of the grid,
- each cell of the grid is either completely inside or completely outside the photographed area.

The satellite is able to take at most k high-resolution photos. Once the satellite is done taking photos, it will transmit the high-resolution photo of each photographed cell to our home base (regardless of whether that cell contains some points of interest). The data for each photographed cell will only be transmitted once, even if the cell was photographed several times. Thus, we have to choose at most k square areas that will be photographed, assuring that:

- each cell containing at least one point of interest is photographed at least once, and
- the number of cells that are photographed at least once is minimized.

Your task is to find the smallest possible total number of photographed cells.

Implementation details

You should implement the following function (method):

`int64 take_photos(int n, int m, int k, int[] r, int[] c)`

- n: the number of points of interest,
- m: the number of rows (and also columns) in the grid,
- k: the maximum number of photos the satellite can take,
- r and c: two arrays of length n describing the coordinates of the grid cells that contain points of interest. For $0 \leq i \leq n-1$, the i-th point of interest is located in the cell $(r[i], c[i])$.

The function should return the smallest possible total number of cells that are photographed at least once (given that the photos must cover all points of interest).

Examples

Example 1.

`take_photos(5, 7, 2, [0, 4, 4, 4, 4], [3, 4, 6, 5, 6])`

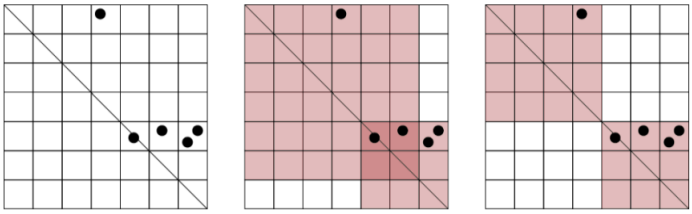
In this example we have a 7×7 grid with 5 points of interest. The points of interest are located in four different cells: $(0, 3)$, $(4, 4)$, $(4, 5)$, and $(4, 6)$. You may take at most 2 high-resolution photos.

One way to capture all five points of interest is to make two photos: a photo of the 6×6 square containing the cells $(0, 0)$ and $(5, 5)$, and a photo of the 3×3 square containing the cells $(4, 4)$ and $(6, 6)$. If the satellite takes these two photos, it will transmit the data about 41 cells. This amount is not optimal.

The optimal solution uses one photo to capture the 4×4 square containing cells $(0, 0)$ and $(3, 3)$ and another photo to capture the 3×3 square containing cells $(4, 4)$ and $(6, 6)$. This results in only 25 photographed cells, which is optimal, so `take_photos` should return 25.

Note that it is sufficient to photograph the cell $(4, 6)$ once, even though it contains two points of interest.

This example is shown in the figures below. The leftmost figure shows the grid that corresponds to this example. The middle figure shows the suboptimal solution in which cells were photographed. The rightmost figure shows the optimal solution.

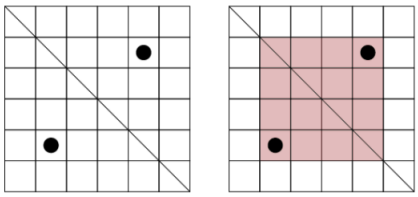


Example 2.

```
take_photos(2, 6, 2, [1, 4], [4, 1])
```

Here we have 2 points of interest located symmetrically: in the cells $(1, 4)$ and $(4, 1)$. Any valid photo that contains one of them contains the other one as well.

Therefore, it is sufficient to use a single photo. The figures below show this example and its optimal solution. In this solution the satellite captures a single photo of 16 cells.



Subtasks

For all subtasks, $1 \le k \le n$.

1. (4 points) $1 \le n \le 50$, $1 \le m \le 100$, $k = n$,
2. (12 points) $1 \le n \le 500$, $1 \le m \le 1000$, for all i such that $0 \le i \le n - 1$, $r_i = c_i$,
3. (9 points) $1 \le n \le 500$, $1 \le m \le 1000$,
4. (16 points) $1 \le n \le 4000$, $1 \le m \le 1\,000\,000$,
5. (19 points) $1 \le n \le 50\,000$, $1 \le k \le 100$, $1 \le m \le 1\,000\,000$,
6. (40 points) $1 \le n \le 100\,000$, $1 \le m \le 1\,000\,000$.

Sample grader

The sample grader reads the input in the following format:

- line 1: integers n, m and k,

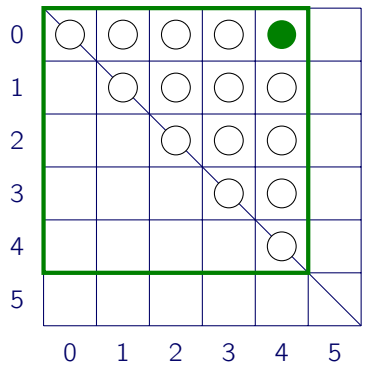
- line $2 + i$ $(0 \leq i \leq n - 1)$: integers r_i and c_i.

Solution

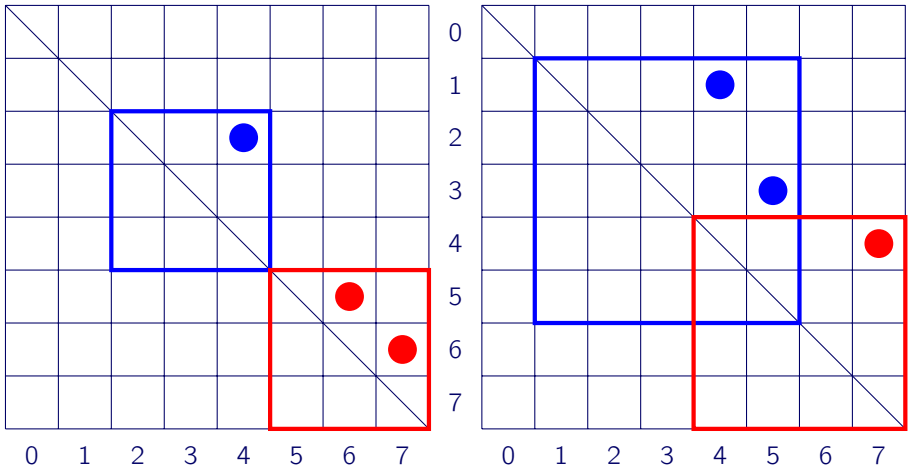
We will briefly describe the dynamic programming solution and ways to optimize it using the optimizations from this chapter.

First, let us observe that we just need to think about points of interests above diagonal, as we can mirror the picture below it (as pictures are squares, so they will be covered by them anyway).

Moreover, we can notice that some of the points are redundant. We will say that point a dominates b, if $r[b] \geq r[a]$ and $c[b] \leq c[a]$; then b is not relevant, as all squares covering a will also cover b. In the picture below, all the white points are dominated by the green point. We can remove redundant points by sorting the values by r and then checking proper values using a stack.



When we remove all irrelevant points, we are left with points with increasing r and c. Now we can notice that if a photo covers points i and j, then it covers also all the points between i and j and the area of this photo is equal to $(c[j] - r[i] + 1)^2$. Note that some photos may overlap, so we need to calculate the area of these overlaps. Overlaps only depend on the last point from the previous photo and the area is equal to $\max(0, c[i-1] - r[i] + 1)^2$ (you can find both cases on the pictures below).



So now we can come up with the cost function for covering points between i and j:

$$cost(i, j) = (c[j] - r[i] + 1)^2 - \max(0, c[i-1] - r[i] + 1)^2$$

And we can finally describe the dynamic programming transitions. Let $dp(i, j)$ denote the cost of covering prefix of j points with i photos. Then:

$$dp(i, j) = \min(dp(i-1, j), \min_{0 \le p < j} dp(i-1, p) + cost(p+1, j))$$

With these transitions, we can solve this problem in $O(n^2k)$ time with $O(nk)$ space, which was enough to solve subtask 3.

To solve subtask 4, we can prove that in the dynamic programming above $opt(i, j-1) \le opt(i, j) \le opt(i+1, j)$, where $opt(i, j)$ is the optimal value of p in formula above. We can use the Knuth optimization the to reduce the complexity to $O(n^2)$.

For subtask 5, we have two options. We can use the fact that $opt(i, j-1) \le opt(i, j)$ and all values $dp(*, j)$ can be calculated from all $dp(*, j-1)$ and we can apply the divide and conquer optimization to come up with an $O(kn \log n)$ solution. We can also use the convex hull optimization, after expanding $dp(i, j)$ into linear function terms. This optimization gives us an $O(kn)$ solution.

Finally, to solve the final subtask, we need to use the last optimization. When we consider $dp(i, k)$ as a function on k, we can prove that dp is convex. Then we can use the Lagrange optimization and end up with $O(n \log(n+m))$.

Chapter 7

Polynomials

In this section we will discuss polynomials. First, we will discuss the general problem of multiplication of two polynomials. Then, we will talk about various representations of polynomials and focus on evaluation and interpolation of polynomials. Finally, we will present Fast Fourier transform (FFT). After that, we will take a look at other bitwise convolutions of polynomials.

Polynomials and their representations

We will consider polynomials only in one variable x in the following form $A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$ (a_i are *coefficients* of this polynomial). This representation is called a *coefficient representation*. We will pretty often represent these polynomials just as a vector of coefficients, i.e.: $(a_0, a_1, a_2, \ldots, a_n)$.

We want to find a fast way to multiply two polynomials $A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1} + a_n x^n$ and $B(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_{m-1} x^{m-1} + b_m x^m$. To compute $C(x) = A(x) \cdot B(x)$, we need to calculate:

$$\begin{aligned} C(x) = (a_n \cdot b_m)x^{n+m} + (a_{n-1} \cdot b_m + a_n \cdot b_{m-1})x^{n+m-1} + \\ +(a_{n-2} \cdot b_m + a_{n-1} \cdot b_{m-1} + a_n \cdot b_{m-2})x^{n+m-2} + \\ + \ldots + (a_0 \cdot b_1 + a_1 \cdot b_0)x + a_0 \cdot b_0 \end{aligned}$$

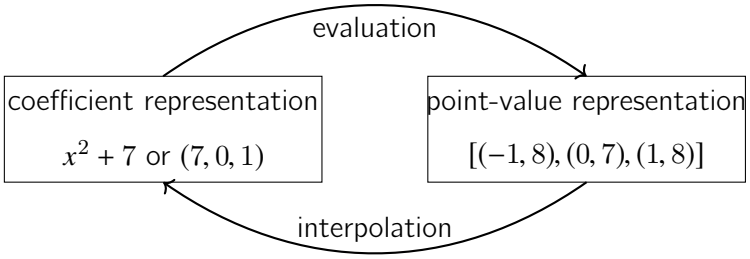
In particular, we can see that the straightforward method of multiplying two polynomials works in $O(nm)$ time.

Let us try to find something faster. A polynomial in variable x can also be represented as a plot of function $A(x)$. Moreover, every $n+1$ distinct points (pairs $(x_i, A(x_i))$ for $i = 0, 1, \ldots, n$ and distinct x_i) uniquely determine the polynomial of degree at most n. Let us prove this fact right now. Assume that there are two polynomials A and B of degree at most n that fit these points, i.e. $A(x_i) = B(x_i)$ for $i = 0, 1, \ldots, n$. Then let us consider polynomial $A(x) - B(x)$. It has at least $n + 1$ roots, therefore it has to be the

zero polynomial (because it cannot have degree more than n) and $A(x) = B(x)$. This representation ($n+1$ pairs of (point, value)) is called a *point-value representation*.

When we consider this representation, multiplying (and adding) is pretty easy. For multiplying, we just need to multiply values in the same points $(A \cdot B)(x_i) = A(x_i) \cdot B(x_i)$, but we need to remember that we need more points (exactly $n + m + 1$, where n and m are degrees of A and B respectively), as the degree of the resulting polynomial is greater.

So if we want to multiply two polynomials faster, we have to switch from the coefficient representations to the point-value representations, multiply them and then go back. The first operation is called *evaluation*, while the second one is called *interpolation*.



Below, we will describe how we can perform these operations.

7.1. Evaluation

For evaluation, we will use the *Horner's method* (also known as Horner's scheme), which comes from 1819 [Horner, 1819]. This method uses this simple observation:

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n \\ &= a_0 + x\Big(a_1 + x\Big(a_2 + x\big(a_3 + \cdots + x(a_{n-1} + x\, a_n)\cdots\big)\Big)\Big) \end{aligned}$$

which corresponds to the following iterative code:

Function *evaluate*($A = [a_0, a_1, \ldots, a_n]$, x)**:**
    $res \leftarrow 0$;
    for $i \leftarrow n$ **down to** *0*
        $res \leftarrow x \cdot res + a_i$;
    return res

Unfortunately, in a general case, we need to run this function n times for n distinct points, therefore this evaluation is still slow ($O(n^2)$ time).

7.2. Interpolation

For interpolation, there are two famous methods to find the coefficient representation of a polynomial. First we will introduce *Lagrange polynomial*. The idea to interpolate the polynomial A is as follows. Let us find polynomials L_i, such that:

$$\begin{cases} L_i(x_i) = 1 \\ L_i(x_j) = 0 \quad \text{for } i \neq j \end{cases}$$

For simplicity, let us denote that $y_i = A(x_i)$ for $i = 0, 1, \ldots, n$, where n is the degree of the polynomial A. Then our interpolating polynomial will take the following form:

$$L(x) = L_0 y_0 + L_1 y_1 + \ldots + L_n y_n$$

It turns out that polynomials L_i are pretty easy to define, for example:

$$L_0 = \frac{(x - x_1)(x - x_2)\ldots(x - x_n)}{(x_0 - x_1)(x_0 - x_2)\ldots(x_0 - x_n)}$$

The numerator guarantees that for every $x_j \neq x_0$, the whole expression will be calculated to zero, while the denominator guarantees that the expression will be equal to one in case of x_0 (we are removing all the unnecessary expressions from the numerator). Please note that the polynomial in the numerator is of degree at most $n-1$, while the denominator is just some constant (there are no variables in it).

Newton's interpolation shows another approach. Let us use induction. If we want to interpolate using only one given value (x_0, y_0), it is pretty easy:

$$N_0(x) = y_0$$

If we want to add another value (i.e. (x_1, y_1)), we still want to keep that $N_1(x_0) = y_0$, therefore we will add something that will evaluate to zero in x_0, like $(x - x_0)$:

$$N_1(x) = y_0 + c_1(x - x_0)$$

We can continue this approach, until we end up with the following polynomial:

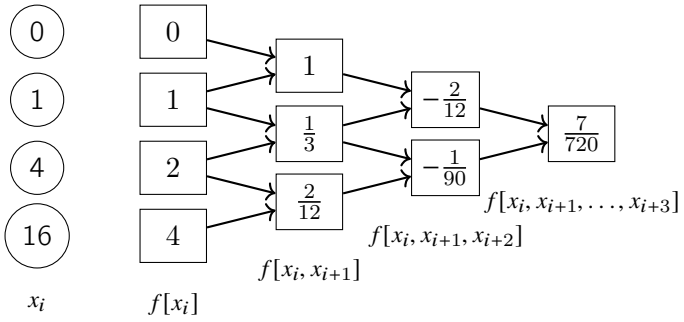
$$N_n(x) = y_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \ldots + c_n(x - x_0)(x - x_1)\ldots(x - x_{n-1})$$

Now we just need to find the constants c_i. There is a handy method to calculate these constants using *difference quotients*. They are defined recursively as follows:

$$\begin{cases} f[x_i] = y_i \\ f[x_i, x_{i+1}, \ldots, x_j] = \frac{f[x_{i+1}, x_{i+2}, \ldots, x_j] - f[x_i, x_{i+1}, \ldots, x_{j-1}]}{x_j - x_i} \end{cases}$$

You may notice that in particular for two elements, the difference quotient is equal to the difference of values divided by the difference of the corresponding points, hence the name. This term is often used in calculus.

It turns out that c_i is equal exactly to $f[x_0, x_1, \ldots, x_i]$. We will now show how to calculate these quotients, by trying to interpolate function that is not a polynomial: $f(x) = \sqrt{x}$ by a polynomial of degree 3. This is just a small experiment, but this approach is widely used when we need to find some approximation of a function that is not a polynomial. We need to get four values of this function, so let us take $f(0) = 0, f(1) = 1, f(4) = 2$ and $f(16) = 4$. Now we will create the following diagram:



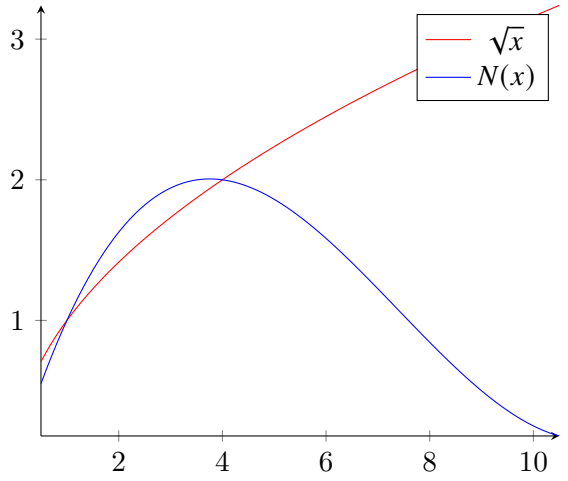
You may notice that each of the values in the boxes from the second level is a difference between previous boxes (pointed with arrows) divided by the difference between corresponding pairs of x_i (in circles). Now we can take first box in each column and create the Newton's polynomial:

$$N(x) = 0 + 1(x - 0) - \frac{2}{12}(x - 0)(x - 1) + \frac{7}{720}(x - 0)(x - 1)(x - 4)$$

or simpler:

$$N(x) = \frac{217}{180}x - \frac{31}{144}x^2 + \frac{7}{720}x^3$$

We can also see how our polynomial is quite close to the original function.



Unfortunately, these two methods still work in $O(n^2)$ time.

Problem Stack machine programmer

Centeral European Regional Contest 2011.
Limits: 1s, 128 MB.
`https://kostka.dev/sp/smp`

Many ciphers can be computed much faster using various machines and automata. In this problem, we will focus on one particular type of machines called stack machine. Its name comes from the fact that the machine operates with the well-known data structure – stack. The later-stored values are on the top, older values at the bottom. Machine instructions typically manipulate the top of the stack only.

Our stack machine is relatively simple: It works with integer numbers only, it has no storage beside the stack (no registers etc.) and no special input or output devices. The set of instructions is as follows:

- `NUM X`, where `X` is a non-negative integer number, $0 \leq \texttt{X} \leq 10^9$. The NUM instruction stores the number `X` on top of the stack. It is the only parametrized instruction.
- `POP`: removes the top number from the stack.
- `INV`: changes the sign of the top-most number. $(42 \rightarrow -42)$
- `DUP`: duplicates the top-most number on the stack.
- `SWP`: swaps (exchanges) the position of two top-most numbers.
- `ADD`: adds two numbers on the top of the stack.
- `SUB`: subtracts the top-most number from the *second one* (the one below).

- `MUL`: multiplies two numbers on the top of the stack.
- `DIV`: integer division of two numbers on the top. The top-most number becomes divisor, the one below dividend. The quotient will be stored as the result.
- `MOD`: modulo operation. The operands are the same as for the division but the remainder is stored as the result.

All binary operations consider the top-most number to be the *right* operand, the second number the *left* one.

All of them remove both operands from the stack and place the result on top in place of the original numbers. If there are not enough numbers on the stack for an instruction (one or two), the execution of such an instruction will result into a program failure. A failure also occurs if a divisor becomes zero (for `DIV` or `MOD`) or if the result of any operation should be more than 10^9 in absolute value. This means that the machine only operates with numbers between $-1\,000\,000\,000$ and $1\,000\,000\,000$, inclusive. To avoid ambiguities while working with negative divisors and remainders: If some operand of a division operation is negative, the absolute value of the result should always be computed with absolute values of operands, and the sign is determined as follows: The quotient is negative if (and only if) exactly one of the operands is negative. The remainder has the same sign as the dividend. Thus, 13 div -4 = -3, -13 mod 4 = -1, -13 mod -4 = -1, etc. If a failure occurs for any reason, the machine stops the execution of the current program and no other instructions are evaluated in that program run.

The trouble with such machines is that someone has to write programs for them. Just imagine, how easy it would be if we could write a program that would be able to write other programs. In this contest problem, we will (for a while) ignore the fact that such a "universal program" is not possible. And also another fact that most of us would lose our jobs if it existed.

Your task is to write a program that will automatically generate programs for the stack machine defined below.

Input

The input contains several test cases. Each test case starts with an integer number N ($1 \le N \le 5$), specifying the number of inputs your program will process. The next N lines contain two integer numbers each, V_i and R_i. V_i ($0 \le V_i \le 10$) is the input value and R_i ($0 \le R_i \le 20$) is the required output for that input value. All input values will be distinct. Each test case is followed by one empty line. The input is terminated by a line containing one zero in place of the number of inputs.

Output

For each test case, generate any program that produces the correct output values for all of the inputs. It means, if the program is executed with the stack initially containing only the input value V_i, after its successful execution, the stack must contain one single value R_i. Your program must strictly satisfy all conditions described in the specification of the stack machine, including the precise formatting, amount of white space, maximal program length, limit on numbers, stack size, and so on. Of course, the program must not generate a failure. Print one empty line after each program, including the last one.

Examples

For the input data:	the correct result is:
3	DUP
1 3	MUL
2 6	NUM 2
3 11	ADD
	END
1	
1 1	END
2	NUM 3
2 4	MOD
10 1	DUP
	MUL
0	END

Solution

The solution is pretty straightforward: we need to find a polynomial that matches with given pairs of points-values using chosen interpolation method and then carefully construct a stack machine that will evaluate this polynomial.

7.3. Fast Fourier transform

Up to now, we have not managed to come up with a method how to multiply two polynomials in the coefficient representation in $o(n^2)$ time. Please note that we have

shown that if we will find a faster way to change between the coefficient representation and the point-value representation, we can multiply in $O(n)$ time. Please also note that we can choose any points to calculate point-value representation, but we did not use this fact before.

First of all, let us think how to compute values of some polynomial $A(x)$ for $x = 1$ and $x = -1$ in parallel.

$$A(1) = a_0 + a_1 + a_2 + a_3 + \ldots + a_n$$
$$A(-1) = a_0 - a_1 + a_2 - a_3 + \ldots \pm a_n$$

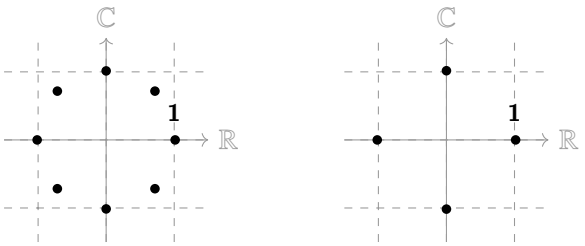
Therefore we can group coefficients with odd and even indexes and sum them separately. Moreover, a similar trick works for any x:

$$A(x) = (a_0 + a_2x^2 + a_4x^4 + \ldots) + x(a_1 + a_3x^2 + a_5x^4 + \ldots)$$
$$A(-x) = (a_0 + a_2x^2 + a_4x^4 + \ldots) + (-x)(a_1 + a_3x^2 + a_5x^4 + \ldots)$$

In this case we can just calculate two polynomials of size $\frac{n}{2}$ in x^2.

Unfortunately, we cannot continue our method, because $\mathbb{R}$ is limited.

To our rescue come complex numbers. First let us assume that the degree of polynomial n is a power of two (we can pad extra zeroes, if that is not the case). Now consider n solutions of the equation $x^n = 1$. The following picture shows these solutions for $n = 8$ and $n = 4$ respectively.



In general, if we take $\omega_n = \cos\left(\frac{2\pi}{n}\right) + i\sin\left(\frac{2\pi}{n}\right)$, then all such solutions are of the form $\omega_n^k = \cos\left(k\frac{2\pi}{n}\right) + i\sin\left(k\frac{2\pi}{n}\right)$ for $k = 0, 1, 2, \ldots, n-1$.

Moreover, if we take all the solutions for $n = 8$, i.e.

$$\{\omega_8^0, \omega_8^1, \omega_8^2, \omega_8^3, \omega_8^4, \omega_8^5, \omega_8^6, \omega_8^7\}$$

and consider their squares, we get respectively:

$$\{\omega_8^0, \omega_8^2, \omega_8^4, \omega_8^6, \omega_8^0, \omega_8^2, \omega_8^4, \omega_8^6\}$$

therefore, we have only half of values (one may notice that they correspond to the solutions for $n = 4$, i.e. $\{\omega_4^0, \omega_4^1, \omega_4^2, \omega_4^3\}$, shown on the right side of the figure above). Now we can show how we can use our trick.

We will calculate values in points ω_n^k for $k = 0, 1, 2, \ldots, n-1$. To do so, let us notice that:

$$\begin{aligned} A(\omega_n^i) &= \left(a_0 + a_2\omega_n^{2i} + a_4\omega_n^{4i} + \ldots\right) + \omega_n^i\left(a_1 + a_3\omega_n^{2i} + a_5\omega_n^{4i} + \ldots\right) = \\ &= \left(a_0 + a_2\omega_{n/2}^{i} + a_4\omega_{n/2}^{2i} + \ldots\right) + \omega_n^i\left(a_1 + a_3\omega_{n/2}^{i} + a_5\omega_{n/2}^{2i} + \ldots\right) \end{aligned}$$

Therefore, to calculate the values of n polynomials of size n, we just need to calculate values of n polynomials of size $\frac{n}{2}$ ($\frac{n}{2}$ on odd coefficients and $\frac{n}{2}$ on even coefficients, only on squares of roots) and then add their values properly. Once again, for $n = 8$, we have (below, we will use vector notation for polynomials):

$$\begin{cases} (a_0, a_1, a_2, a_3, \ldots, a_7)(\omega_8^0) = (a_0, a_2, a_4, a_6)(\omega_4^0) + \omega_8^0 \cdot (a_1, a_3, a_5, a_7)(\omega_4^0) \\ (a_0, a_1, a_2, a_3, \ldots, a_7)(\omega_8^1) = (a_0, a_2, a_4, a_6)(\omega_4^1) + \omega_8^1 \cdot (a_1, a_3, a_5, a_7)(\omega_4^1) \\ \ldots \\ (a_0, a_1, a_2, a_3, \ldots, a_7)(\omega_8^6) = (a_0, a_2, a_4, a_6)(\omega_4^2) + \omega_8^6 \cdot (a_1, a_3, a_5, a_7)(\omega_4^2) \\ (a_0, a_1, a_2, a_3, \ldots, a_7)(\omega_8^7) = (a_0, a_2, a_4, a_6)(\omega_4^3) + \omega_8^7 \cdot (a_1, a_3, a_5, a_7)(\omega_4^3) \end{cases}$$

Moreover, we can continue this process recursively to obtain the time complexity $O(n \log n)$. Please note that we are simplifying the following code, using the fact that $\omega_n^{i+\frac{n}{2}} = -\omega_n^i$.

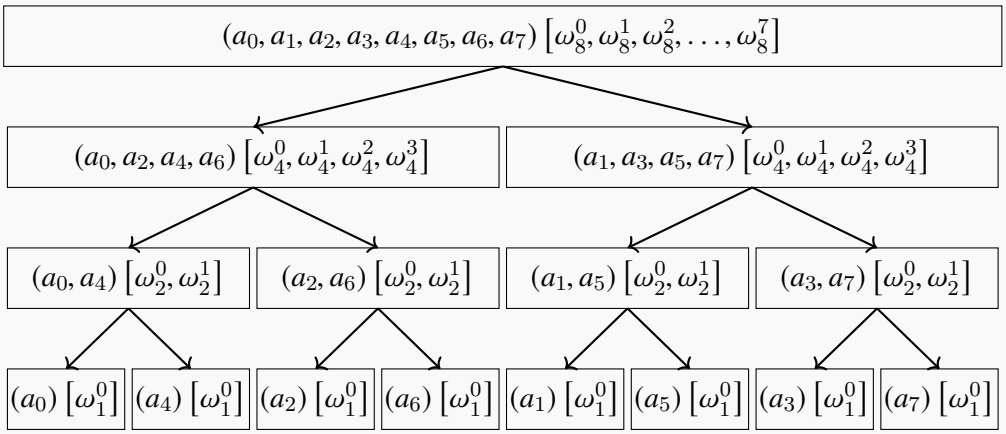
Function *FFT(*$A = [a_0, a_1, \ldots, a_{n-1}]$*)*:

```
if n = 1
    return a_0
res_even ← FFT([a_0, a_2, a_4, ...]);
res_odd ← FFT([a_1, a_3, a_5, ...]);
res ← array of size n;
ω ← cos(2π/n) + i sin(2π/n);
for k ← 0 to n/2
    // below we define two auxiliary values which we will use
       in further calculations
    left ← res_even_k;
    right ← ω^i · res_odd_k;
    res_k ← left + right;
    res_{k+n/2} ← left − right;
return res
```

The FFT function returns a vector of n elements, where i-th element corresponds to the value $A(\omega_n^i)$ for $i = 0, 1, \ldots, n-1$.

Iterative FFT

To speed up the FFT algorithm, we can completely remove the recursion in the following way. Let us consider the recursion tree for $n = 8$. Each node contains a vector of coefficients (in round brackets) and the roots where we calculate values of our polynomial (in square brackets):



Now instead of going top-down, we will try a bottom-up approach. We will go from the bottom-most layer all the way to the top. In each layer, we need to calculate values in corresponding roots, therefore in each step, we will try to combine two values from the children to get the new values.
In the leaves we should store values of (a_i) in 1. Now, for each node above the leaves, we would like to use values from its children to calculate values needed for the node itself and we can do it, cf. the last loop in the algorithm above, the names $left$ and $right$ were not accidental).
For example, if we want to know the values of (a_1, a_3, a_5, a_7) in ω_4^0, ω_4^1, ω_4^2, and ω_4^3 (which are required for the right node in the second row), we can compute these values easily from the previously computed values. For example:

$$(a_1, a_3, a_5, a_7)\left(\omega_4^1\right) = (a_1, a_5)\left(\omega_2^1\right) + \omega_4^1 \cdot (a_3, a_7)\left(\omega_2^1\right)$$
$$(a_1, a_3, a_5, a_7)\left(\omega_4^3\right) = (a_1, a_5)\left(\omega_2^1\right) - \omega_4^1 \cdot (a_3, a_7)\left(\omega_2^1\right)$$

Now, we just need to find the proper order for the leaves. This is pretty easy if we look at their binary representations:

$$(000, 100, 010, 110, 001, 101, 011, 111),$$

and reverse them:

$$(000, 001, 010, 011, 100, 101, 110, 111).$$

We can see that their reversed binary representations correspond to the sequence $(0, 1, 2, 3, \ldots, n-1)$, therefore we can easily shuffle leaves to the correct order.

So now our algorithm will work as follows:

Function *Iterative-FFT*($A = [a_0, a_1, \ldots, a_{n-1}]$)**:**
    shuffle A according to the order described above;
    for $level \leftarrow 1$ **to** $\log n$
        for $i \leftarrow 0$ **to** $2^{level-1}$
            for $j \leftarrow i$ **to** n **by** 2^{level}
                $left \leftarrow A[j]$;
                $right \leftarrow \omega^i \cdot A[j + 2^{level-1}]$;
                $A[j] \leftarrow left + right$;
                $A[j + 2^{level-1}] \leftarrow left - right$;
    return A

So now we can change from the coefficient representation to the point-value representation efficiently. But we still need to come back to the coefficient representation. How to do this?

Let us notice that the evaluation in FFT might be considered as applying the following transformation matrix T on the vector of coefficients:

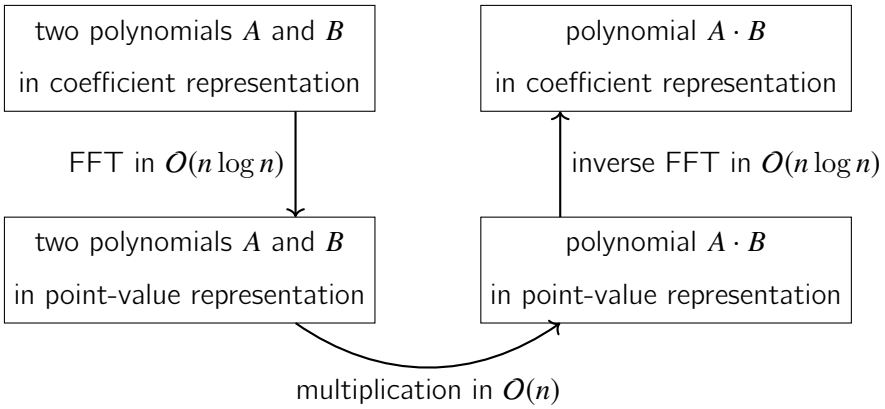
$$\underbrace{\begin{bmatrix} \omega_n^0 & \omega_n^0 & \omega_n^0 & \ldots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \omega_n^2 & \ldots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \omega_n^4 & \ldots & \omega_n^{2n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \omega_n^{2n-2} & \ldots & \omega_n^{(n-1)n-(n-1)} \end{bmatrix}}_{T} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} A(\omega_n^0) \\ A(\omega_n^1) \\ A(\omega_n^2) \\ \vdots \\ A(\omega_n^{n-1}) \end{pmatrix}$$

Therefore, if we want to compute the interpolation from these points, we can simply find an inverse matrix to this one. Fortunately, this matrix has some special property: every row is a geometric progression. Such matrices are known in literature as **Vandermonde matrices**. A square Vandermonde matrix is invertible if and only if all ratios of these progressions are distinct and that is the case in our matrix. Moreover, an explicit formula for the inverse is known [Turner, 1966, Macon and Spitzbart, 1958]. In our case the inverse is really simple:

$$T^{-1} = \left(-\frac{1}{n}\right) T$$

Therefore, we can use FFT for interpolation as well, changing only ω to $-\omega$ and dividing the results by n.

To conclude the whole algorithm, let us look at the following diagram:



FFT for two polynomials at the same time

You might notice that in FFT, we need to calculate the transform for two polynomials A and B. There is a pretty nice trick that allows us to do it simultaneously, using imaginary numbers once again. Consider the following polynomial: $X(x) = A(x) + iB(x)$. Now if we want to calculate values of A and B in ω_n^i, we just need to know values of $X(\omega_n^i)$. In particular, let us notice that $\overline{X(\overline{x})} = A(x) - iB(x)$. Therefore:

$$A(\omega_n^i) = \frac{1}{2}\left(X(\omega_n^i) + \overline{X(\omega_n^{n-i})}\right)$$
$$B(\omega_n^i) = \frac{1}{2i}\left(X(\omega_n^i) - \overline{X(\omega_n^{n-i})}\right)$$

Moreover, it can also work for the inverse-FFT: we just need to calculate inverse-FFT for $Y = A + iB$.

Problem Polynomial

25th Polish Olympiad in Informatics, third stage, second day.
Limits: 15s, 128MB.
`https://kostka.dev/sp/pol`

Byteasar was misbehaving in mathematics class, and for punishment he is to evaluate a very long polynomial W with n integer coefficients

$$W(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-2} x^{n-2} + a_{n-1} x^{n-1}$$

at points $q^1, q^2, \ldots, q^n$. To help the teacher verify his results quickly, Byteasar should first provide the remainder modulo m of the sum of these values, and then provide the remainders modulo m of all the successive values.

Not only does he misbehave frequently, Byteasar is also rather lazy, so he asked you (surprisingly politely) for help in this task right before heading out to a party. To his credit, Byteasar is quite bright, only too lazy to follow up his ideas. Saying his goodbyes, he shared his hunch that the following properties could drastically reduce the amount of calculations necessary to get the results: n is a power of two, and the remainder of q^n modulo m is 1 (i.e., $q^n \bmod m = 1$).

Input

In the first line of the standard input, there are three integers n, m and q ($n \geq 1$, n is a power of two, $2 \leq m \leq 10^9$, $1 \leq q < m$, $q^n \bmod m = 1$), separated by single spaces.

In the second line, there is a sequence of n integers, separated by single spaces, specifying successive coefficients of the polynomial in this order: $a_0, a_1, \ldots, a_{n-1}$ ($0 \leq a_i \leq 10^9$).

Output

A single integer should be printed to the first line of the standard output – the remainder modulo m of the sum of values of the polynomial W at the points $q^1, q^2, q^3, \ldots, q^n$. In the second line the remainders modulo m of $W(q^1)$, $W(q^2)$, $W(q^3)$, $\ldots$, $W(q^n)$ should be printed, separated by single spaces.

Examples

For the input data:	the correct result is:
`4 13 5` `3 2 2 1`	`12` `6 2 9 8`

Explanation for the example: The polynomial is $W(x) = 3+2x+2x^2+x^3$, so its values at successive points are $W(5) = 188$, $W(5^2) = 16\,928$, $W(5^3) = 1\,984\,628$, $W(5^4) = 244\,923\,128$. The number in the first output line is the remainder modulo 13 of $188 + 16\,928 + 1\,984\,628 + 244\,923\,128 = 246\,924\,872$, which is 12. The second line provides the remainders modulo 13 of aforementioned summands.

Grading

If the sum's remainder is correct but one of the successive values is not, your program will be awarded up to 40% of the test's score, provided that all the n numbers in the second line are in the range from 0 to $m - 1$.

Subtask	Property	Score
1	$n \leq 2^{10}$	17
2	$n \leq 2^{15}$	9
3	$n \leq 2^{20}$	74

Solution

This problem is a slight variation on FFT called Number Theoretic Transform (NTT). We just need to notice that instead of calculating each value in $O(n)$ time, we can divide our problem into 2 smaller subproblems:

$$(a_0, a_1, a_2, \ldots, a_{n-1})(x) = (a_0, a_2, \ldots, a_{n-2})(x^2) + x \cdot (a_1, a_3, \ldots, a_{n-1})(x^2)$$

Using the fact that $q^n = 1$, we can calculate all values in a similar way as in FFT in $O(n \log n)$ time.

Please note that NTT can in several cases replace FFT, as it has many advantages. We use here just integers (in modular arithmetic), therefore we won't have precision errors. Note that we should pick a generator to use as q. In general calculations, we can use two (or more) different prime numbers and then later extract the result using Chinese Remainder Theorem. Unfortunately, NTT is pretty slow compared to FFT, because of the necessity to use the modulo function, but uses less memory [Tommila, 2003]. If you need precision or are restricted by memory, you can consider using NTT instead of FFT. For more details, check out [Feng and Li, 2017].

7.4. Applications of the polynomial multiplication

In this section, we will describe several applications of FFT in competitive programming.

First, we will discuss how we can use FFT to multiply large integers. Let us note that integers can be represented as a polynomial, where its digits are the coefficients. For example 4243 can be represented in base $b = 10$ as:

$$3 \cdot b^0 + 4 \cdot b^1 + 2 \cdot b^2 + 4 \cdot b^3$$

We can also choose a different base (such as 100, or $1\,000\,000$), but we need to be careful with numerical precision.

Then we can multiply two integers as their polynomials in $O(n \log n)$ time, where n is the maximal number of digits of these numbers in the chosen base.

Now we will introduce another problem allowing us to use FFT from a different angle.

Problem Biotechnology Laboratory

The 2017 ACM-ICPC Brazil Finals.
Limits: ?s, ?MB.
`https://kostka.dev/sp/bio`

A weighted string is defined over an alphabet Σ and a function f that assigns a weight to each character of the alphabet. Thus we can define the weight of a string s as the sum of the weights of all characters in s.

Several problems of bioinformatics can be formalized as problems in weighted string. An example is protein mass spectrometry, a technique that allows you to identify proteins quite efficiently. We can represent each amino acid with a distinct character and a protein is represented by the character string relative to its amino acids.

One of the applications of protein mass spectrometry is database searching. For this, the protein chain is divided into strings, the mass of each strand is determined, and the mass list is compared to a protein database. One of the challenges for this technique is dealing with very large strings, which can have several possible substrings. The number of substrings selected is critical for good results.

On his first day of internship at a renowned biotechnology lab, Carlos was tasked with determining, for an s chain, the amount of distinct weights found when evaluating the weights of all nonempty consecutive strings of s.

Carlos could not think of an efficient solution to this task, but fortunately he knows the ideal group to assist him.

Assuming that s consists of lowercase letters and each letter has a different weight between 1 and 26: letter `a` has weight 1, letter `b` has weight 2, and so on. Show that your team can help Carlos impress his supervisor within the first week with a solution that can easily handle the largest strings in existence.

Input

Only one line, containing the string s formed by lowercase letters, the length of which $|s|$ satisfies $1 \leq |s| \leq 10^5$.

Output

Your program should produce a single line with an integer representing the number of distinct weights of the nonempty consecutive substrings of s.

Examples

For the input data:	the correct result is:
`abbab`	8
Whereas for the input data:	the correct result is:
`adbbabdcdbcbacdabbaccdac`	56

Solution

Let us calculate prefix sums for value for each prefix of s. Now the problem can be reduced to finding all distinct values of $pref_j - pref_i$, where $i < j$.

Now let us introduce a polynomial P, in which coefficients are defined as follows:

$$p_i = \begin{cases} 1 \text{ if } \exists\ pref_i \\ 0 \text{ otherwise} \end{cases}$$

Now our problem looks pretty similar to multiplication of polynomials, but with subtraction, rather than addition. To do so, we can just pad this polynomial, for example by adding $pref_n$, then we will always be in positive values.

So let us define the second polynomial P', such that:

$$p'_i = \begin{cases} 1 \text{ if } \exists\ pref_n - pref_i \\ 0 \text{ otherwise} \end{cases}$$

So now we just need to calculate $P \cdot P'$ using FFT, and then count all non-zero coefficients.

Another problem that we can solve using FFT is the string matching problem discussed and solved in [Clifford and Clifford, 2007]. We are given a text $t = t_0 t_1 \dots t_{n-1}$ and a pattern $p = p_0 p_1 \dots p_{m-1}$. In the standard approach, we are saying that the pattern occurs in the text at location i if $p_j = t_{i+j}$ for $j = 0, 1, \dots, m$. For example, the pattern `ana` occurs in `banana` at positions 1 and 3. There are many linear algorithms

that solve this problem, but we will introduce a solution involving FFT that we will later use for more general problems.

Here we will assume that the characters correspond to integers from 1 to $|\Sigma|$.

When we want to check if the pattern occurs at some position i, we can calculate the following expression:

$$d_i = \sum_{j=0}^{m-1} (p_j - t_{i+j})^2 .$$

If all the characters are in the right place, then $d_i = 0$. Note that for binary strings this expression is equal to the Hamming distance (concept introduced in [Hamming, 1950]), i.e. the number of mismatches between these two strings.

To calculate these values, we will expand the expression above:

$$d_i = \sum_{j=0}^{m-1} (p_j - t_{i+j})^2 = \sum_{j=0}^{m-1} p_j^2 - 2 \sum_{j=0}^{m-1} p_j \cdot t_{i+j} + \sum_{j=0}^{m-1} t_{i+j}^2$$

The first term can be easily calculated for all i in $O(m)$, the second one is just the correlation (convolution, where one of the polynomials is reversed), so it can be calculated in $O(n \log n)$ time using FFT, while the last one can be calculated in $O(n)$.

Moreover, we can expand this idea by adding wildcards, i.e. characters that match with every other character. We often denote wildcards with an asterisk (`*`). For example, for the text `ba*ana*`, the pattern `*a` will occur at positions 0, 2, and 4.

In this problem, we will use similar approach. We will say that the wildcards will correspond to 0 and calculate the following distance:

$$d'_i = \sum_{j=0}^{m-1} p_j t_{i+j} (p_j - t_{i+j})^2 .$$

If either p_j or t_{i+j} is the wildcard, then the second character does not matter, therefore it will not contribute to the distance.

We can also calculate these distances quickly as:

$$d'_i = \sum_{j=0}^{m-1} p_j t_{i+j} (p_j - t_{i+j})^2 = \sum_{j=0}^{m-1} p_j^3 t_{i+j} - 2 \sum_{j=0}^{m-1} p_j^2 \cdot t_{i+j}^2 + \sum_{j=0}^{m-1} p_j t_{i+j}^3$$

And every single term can be calculated using FFT.

Another approach with trigonometry

Below we will show another approach to the pattern matching problem, using a little bit different idea.

We will assume here that the letters in the text and the pattern correspond to integers from 0 to $|\Sigma|-1$. To solve the problem, we will create two polynomials A and B with the following coefficients:

$$a_k = \cos\left(\frac{2\pi t_k}{|\Sigma|}\right) + i \sin\left(\frac{2\pi t_k}{|\Sigma|}\right)$$

$$b_k = \cos\left(\frac{2\pi p_{m-k-1}}{|\Sigma|}\right) - i \sin\left(\frac{2\pi p_{m-k-1}}{|\Sigma|}\right)$$

Please note that we are explicitly reversing the pattern here.

Then we will calculate $C = A \cdot B$, and then the coefficients, seemingly in a magic way, will tell us if the pattern occurs on the given position. Let us look closely at this expression. In particular, we will look at the $m+k-1$-th coefficient of C for $k = 1, \ldots, n$:

$$c_{m+k-1} = \sum_{j=0}^{m-1} a_{k+j} \cdot b_{m-j-1} = \sum_{j=0}^{m-1} \left(\cos\left(\frac{2\pi t_{k+j}}{|\Sigma|}\right) + i \sin\left(\frac{2\pi t_{k+j}}{|\Sigma|}\right)\right) \cdot \left(\cos\left(\frac{2\pi p_j}{|\Sigma|}\right) - i \sin\left(\frac{2\pi p_j}{|\Sigma|}\right)\right)$$

If there is a match of the pattern, then $t_{k+j} = p_j$ for all $j = 0, \ldots, m$, so the values of the trigonometric functions are equal and we have:

$$c_{m+k-1} = \sum_{j=0}^{m-1} a_{k+j} \cdot b_{m-j-1} = \sum_{j=0}^{m-1} \left(\cos\left(\frac{2\pi p_j}{|\Sigma|}\right)^2 + \sin\left(\frac{2\pi p_j}{|\Sigma|}\right)^2\right) = \sum_{j=0}^{m-1} 1 = m$$

Therefore, if the pattern occurs at the i-th position in a given text, then $c_{m+k-1} = m$. Moreover, the opposite is also true. If at least one of the characters is different, then at least one of the products is not equal to 1, then $c_{m+k-1} \neq m$.

Problem Fuzzy Search

Codeforces Round #296.
Limits: 3s, 256MB.
`https://kostka.dev/sp/fuz`

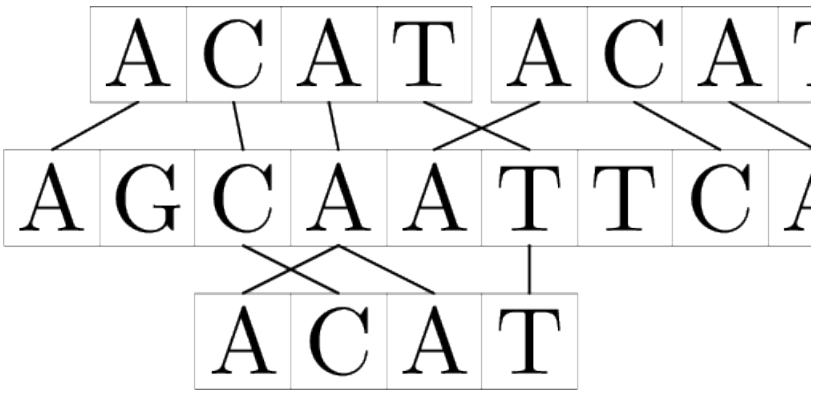
Leonid works for a small and promising start-up that works on decoding the human

genome. His duties include solving complex problems of finding certain patterns in long strings consisting of letters textttA, `T`, `G`, and `C`.

Let's consider the following scenario. There is a fragment of a human DNA chain, recorded as a string S. To analyze the fragment, you need to find all occurrences of string T in a string S. However, the matter is complicated by the fact that the original chain fragment could contain minor mutations, which, however, complicate the task of finding a fragment. Leonid proposed the following approach to solve this problem.

Let's write down integer $k \geq 0$ – the error threshold. We will say that string T occurs in string S on position i $(1 \leq i \leq |S| - |T| + 1)$, if after putting string T along with this position, each character of string T corresponds to the some character of the same value in string S at the distance of at most k. More formally, for any j $(1 \leq j \leq |T|)$ there must exist such p $(1 \leq p \leq |S|)$, that $|(i + j - 1) - p| \leq k$ and $S[p] = T[j]$.

For example, corresponding to the given definition, string `ACAT` occurs in string `AGCAATTCAT` in positions 2, 3 and 6.



Note that at $k = 0$ the given definition transforms to a simple definition of the occurrence of a string in a string.

Help Leonid by calculating in how many positions the given string T occurs in the given string S with the given error threshold.

Input

The first line contains three integers $|S|$, $|T|$, k $(1 \leq |T| \leq |S| \leq 200\,000, 0 \leq k \leq 200\,000)$ – the lengths of strings S and T and the error threshold.

The second line contains string S.

The third line contains string T.

Both strings consist only of uppercase letters `A`, `T`, `G`, and `C`.

Output

Print a single number – the number of occurrences of T in S with the error threshold k by the given definition.

Example

For the input data:	the correct result is:

`10 4 1`
`AGCAATTCAT`
`ACAT`
	`3`

Solution

We are looking for a way to reduce this text problem to polynomial multiplication.

We will create bitmasks for each character (out of four) both for the text and the pattern signifying if the character occurs in the given position. For example, for `AGCAATTCAT` and the character `T`, we will have vector (0000011001). Now if we calculate the correlation (convolution where one of the vectors is reversed) between the corresponding text and pattern vectors for each character, we will know how many characters are in the right places. Therefore we can check if the sum over all characters is equal to $|T|$. If that is the case, then we have a match.

We still need to add the error threshold k. To do so, let us modify the text vectors, by expanding all occurrences k to the left and k to the right, for instance our vector above for text `AGCAATTCAT` and the character T will become (00001111011) for $k = 1$. We can calculate these vectors simply in $O(|S|)$ using prefix sums and use these modified vectors in the algorithm described above.

7.5. Bitwise convolutions

Let us introduce another transformation that helps us compute some convolutions of two polynomials.

Previously, we tried to calculate polynomial with coefficients c_k, such that:

$$c_k = \sum_{i+j=k} a_i b_j$$

Now we will try to compute the polynomial with bitwise `xor` on indexes, instead of sum, i.e.:

$$c_k = \sum_{i \oplus j = k} a_i b_j$$

Such expression is called a *xor-convolution* of polynomials A and B. We will denote this operation by $\oplus$, i.e. $C = A \oplus B$ will mean that C is the xor-convolution of A and B.

The algorithm will be really similar to FFT. We will once again try to find the transformation matrix T and then use it to calculate the convolution $C = A \oplus B$, i.e. we will:

- transform vectors A and B, by multiplying them by the transformation matrix T,
- multiply values of TA and TB linearly, i.e. calculate vector TC, in which $TC[i] = TA[i] \cdot TB[i]$,
- transform TC back to C form, by calculating $T^{-1}(TC)$.

How to find T? Let us first think how to find T for polynomials of size 2, i.e. some matrix

$$T_2 = \begin{bmatrix} t_{0,0} & t_{0,1} \\ t_{1,0} & t_{1,1} \end{bmatrix}$$

The following two equations must be satisfied:

$$\begin{cases} (a_0, a_1) \oplus (b_0, b_1) = (a_0 b_0 + a_1 b_1, a_0 b_1 + a_1 b_0) \\ T_2(a_0, a_1) \odot T_2(b_0, b_1) = T_2(a_0 b_0 + a_1 b_1, a_0 b_1 + a_1 b_0) \end{cases}$$

and T_2 has to be invertible. Please note that $\oplus$ denotes xor-convolution, while $\odot$ denotes multiplication "by coordinates", i.e. we multiply two elements with the same index in these vectors.

We can do some calculations and come up with the following solution (please note that it is not unique):

$$T_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Moreover, from this small solution we can deduce the general solution – the following matrix defined recursively:

$$\begin{cases} T_1 = [1] \\ T_{i+1} = \begin{bmatrix} T_i & T_i \\ T_i & -T_i \end{bmatrix} \text{ for } i \geq 1 \end{cases}$$

This transform matrix can be proven correct using the induction method. This matrix is called a *Walsh-Hadamard matrix* and can be calculated implicitly in $O(n^2)$ time, but we can use the recursive form to speed it up to $O(n \log n)$ time, as we will show now.

Let us notice that we would like to take a vector $A = (a_0, a_1, \ldots, a_n)$ and multiply it by $T_{\log n}$ (once again, let us recollect that we can assume that n is a power of two), so if we denote $s = \log n$, we want to calculate:

$$AT_s = A \begin{bmatrix} T_{s-1} & T_{s-1} \\ T_{s-1} & -T_{s-1} \end{bmatrix}$$

So let us divide A into two parts (equal in length) $\begin{bmatrix} A_{left} & A_{right} \end{bmatrix}$, then we have:

$$AT_s = \begin{bmatrix} A_{left} & A_{right} \end{bmatrix} \begin{bmatrix} T_{s-1} & T_{s-1} \\ T_{s-1} & -T_{s-1} \end{bmatrix} =$$
$$= \begin{bmatrix} A_{left}T_{s-1} + A_{right}T_{s-1} & A_{left}T_{s-1} - A_{right}T_{s-1} \end{bmatrix}$$

So we just need to calculate $A_{left}T_{s-1}$ and $A_{right}T_{s-1}$ (separately), and then using one single loop calculate the whole resulting vector.

The whole algorithm can be implemented in a couple of lines:

Function *XOR_convolution*($A = [a_0, a_1, \ldots, a_{n-1}]$)**:**
- **if** $n = 1$
  - **return** a_0
- $A\_left \leftarrow A = [a_0, a_1, \ldots, a_{n/2-1})$;
- $A\_right \leftarrow A = [a_{n/2}, a_{n/2+1}, \ldots, a_{n-1})$;
- $res\_left \leftarrow XOR\_convolution(A\_left)$;
- $res\_right \leftarrow XOR\_convolution(A\_right)$;
- **return** $[res\_left + res\_right, res\_left - res\_right]$

We can also really easily change this algorithm to use the iterative approach:

Function *Iterative-XORConv*($A = [a_0, a_1, \ldots, a_{n-1}]$)**:**
- **for** $level \leftarrow 1$ **to** $\log n$
  - **for** $i \leftarrow 0$ **to** n **by** $2^{level+1}$
    - **for** $j \leftarrow 0$ **to** 2^{level}
      - $left \leftarrow A[i + j]$;
      - $right \leftarrow A[i + j + 2^{level}]$;
      - $A[i + j] \leftarrow left + right$;
      - $A[i + j + 2^{level}] \leftarrow left - right$;
- **return** A

We also need to find the inverse of T, i.e. T^{-1}, and we are quite lucky, as

$$T_k^{-1} = \frac{1}{\sqrt{2}^k} T_k.$$

Therefore, we can use exactly same algorithm to go back. Note that we can divide by the proper power of $\sqrt{2}$ at the end.

AND- and OR-convolution

A similar approach works for other bitwise convolutions, such as AND- and OR-convolution. Below, we will just mention their transformation matrices, but you can deduce them yourself.
Please note that in these cases, the inverse matrix is different from the transformation matrix, so we need to write two separate functions for these convolutions.

AND-convolution

$$\begin{cases} T_1 = 1 \\ T_{i+1} = \begin{bmatrix} 0 & T_i \\ T_i & T_i \end{bmatrix} \text{ for } i \geq 1 \end{cases} \qquad \begin{cases} T_1^{-1} = 1 \\ T_{i+1}^{-1} = c \begin{bmatrix} -T_i^{-1} & T_i^{-1} \\ T_i^{-1} & 0 \end{bmatrix} \text{ for } i \geq 1 \end{cases}$$

OR-convolution

$$\begin{cases} T_1 = 1 \\ T_{i+1} = \begin{bmatrix} T_i & T_i \\ T_i & 0 \end{bmatrix} \text{ for } i \geq 1 \end{cases} \qquad \begin{cases} T_1^{-1} = 1 \\ T_{i+1}^{-1} = c \begin{bmatrix} 0 & T_i^{-1} \\ T_i^{-1} & -T_i^{-1} \end{bmatrix} \text{ for } i \geq 1 \end{cases}$$

Problem And to Zero

Moscow Pre-Finals Workshop 2018, Radewoosh Contest.
Limits: 2s, 256MB.
`https://kostka.dev/sp/and`

You are given an array of length n, which consists of positive natural numbers. In one move you can choose an ordered pair of its elements and replace first of them with bitwise AND of these two elements. What is the minimum possible number of moves needed to obtain a 0 in the array? Also, how many shortest sequences of moves are

there? Two sequences are different if and only if for some k the k-th move is different in the two sequences. Two moves are different if and only if ordered pairs of indices of elements are different. As the second number can be huge, output it modulo $10^9 + 7$.

Input

The first line of the input contains one integer n ($1 \le n \le 10^6$) – the length of the array. The second line contains n integers x_i ($1 \le x_i < 2^{20}$) – the array.

Output

If there is no way to make some zero appear in the array then output -1. If it's possible, output two integers, where first will be equal to minimal number of moves needed to obtain a 0 in the array, and second will be equal to number of shortest sequences of moves. Output second number taken modulo $10^9 + 7$.

Examples

For the input data:	the correct result is:
5 8 3 12 7 15	1 6
Whereas for the input data:	the correct result is:
3 3 5 6	2 12
And for the input data:	the correct result is:
2 3 5	-1

Solution

In this problem, we are looking for the subset with the smallest number of elements in which AND of all elements is equal to 0. Let us denote the size of such the subset by s. Then our answer will be equal to $(s - 1, ways \cdot (s - 1)!)$ (the number of and operations (or moves) we need to perform and the number of $ways$ to choose this subset multiplied by some factorial (as the order of moves does not matter).

First, it is easy to check if there is any way to make some zero appear in the array. If the bitwise-and of all the numbers in the array is not equal to zero, then we should output -1, as there is at least one bit that cannot be changed to zero. We can also notice that the minimal number of moves will be at most 20, as with every move we should set at least one bit to zero in our candidate for zero.

Now, let us create an array B_0 in which the i-th element will keep how many elements in A are equal to i.

We will iterate over the number of operations we have to perform (k). If we want to know what numbers we can get after performing k moves (and the number of ways to obtain these numbers), we just need to calculate B_k as AND-convolution of B_{k-1} and B_0 for $k > 0$. Now, $B_k[i]$ keeps the number of ways to get the i-th element after k moves. When we first find that $B_k[0]$ is different than zero, then we know that the output should be k and $B_k[0]$ multiplied by $k!$ (as the order of moves doesn't matter, as we mentioned earlier).

This solution works in $O(n \log n \cdot \log x)$, where x denotes the maximum element in the array. We can speed it up to $O(n \log n \cdot \log \log x)$ by using binary search over the result (number of moves). Moreover, we can remove the $\log n$ factor from the complexity by using some combinatorics, but it was not necessary in this problem.

Please also note that we can find a test that the number of shortest sequences of moves can be equal to $10^9 + 7$. Because of that, we cannot calculate our convolutions modulo 10^9+7, but we should use, for example, two different prime numbers to perform operations.

Chapter 8

Matroids

In this section, we will start by introducing some examples of matroids. Then we will define them formally and show some observations and lemmas in the matroid theory. After that, we will introduce optimization problems in matroids and show a simple greedy algorithm that solves these problems. Finally, we will focus on the matroid intersection problem and show a polynomial algorithm computing the maximal common independent set in the intersection of any two matroids.

Examples of matroids

If you think you have not seen matroids in your life, you are probably very wrong. Matroids are quite common, both in mathematics and in computer science, especially in linear algebra and graph theory. Below, we will introduce some popular matroids.

In matroids we would like to focus on the concept called *independent sets*. In linear algebra, we have an easy candidate, because there is a notion called linear independency of a set of vectors. Let us recall that we call a set of vectors *linearly dependent* if we can find at least one of the vectors in this set that can be defined as a linear combination of the others, i.e. we can find vector v_i, such that:

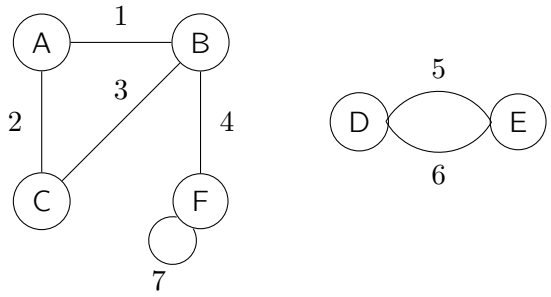
$$v_i = \sum_{\substack{j=1,2,\ldots,n \\ i \neq j}} a_j v_j$$

First, let us consider the following set of vectors $V = \{v_1, v_2, \ldots, v_7\}$:

$$v_1 = (1, 0, 0, 0)$$
$$v_2 = (0, 1, 0, 0)$$
$$v_3 = (1, 1, 0, 0)$$
$$v_4 = (0, 0, 1, 0)$$
$$v_5 = (0, 0, 0, 1)$$
$$v_6 = (0, 0, 0, 2)$$
$$v_7 = (0, 0, 0, 0)$$

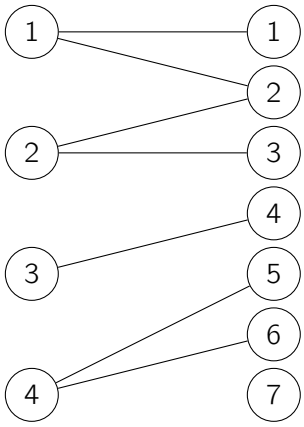
In this example, vectors $\{v_1, v_3, v_4, v_5\}$ are linearly independent, while $\{v_1, v_2, v_3\}$ or $\{v_5, v_6\}$ are linearly dependent.

Now let us consider the following graph:



We would like to identify the vectors by the corresponding edges in the graph (v_i corresponds to the edge with label i). We want to find something similar to linear independency in graphs. Please choose any linearly independent subset of V and check that the corresponding edges in this graph form a forest (acyclic graph or equivalently – a collection of trees). Moreover, you can choose any forest spanned by edges of E and check that it will correspond to a linearly independent subset in V. Now, we hope you realize that a set of linearly independent vectors and a set of edges that forms a forest, in this case, are describing the same object.

We would like to introduce one more representation of the same matroid. Let us consider the matching problem in the following bipartite graph.



The graph was created in the following way. The vertices on the left side represent dimensions, and the vertices on the right side correspond to vectors v_i. Once again, you can check that any valid matching corresponds to some set of independent vectors from V. Therefore, we can guess that all three matroids are isomorphic.

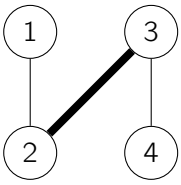
Matroids formally

Now, when we have seen some matroids, let us define them formally. The word 'matroid' was first introduced in [Whitney, 1935], where the author investigated matroids as a set of independent rows in matrices (hence the name). A *matroid* is a tuple $(X, \mathcal{I})$, where X is a finite set of some objects (we will call this set a "*ground set*") and $\mathcal{I}$ describes *independent subsets* of X such that:

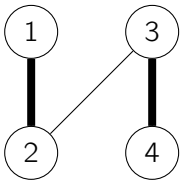
1. $\varnothing \in \mathcal{I}$,
2. if $A \subseteq B$ and $B \in \mathcal{I}$, then $A \in \mathcal{I}$,
3. for $A, B \in \mathcal{I}$, if $|A| < |B|$, then $\exists b \in B \setminus A$, such that $A \cup b \in \mathcal{I}$.

The last property is known as the *exchange property*.

In our previous examples, X was a set of vectors and $\mathcal{I}$ was a set of subsets of X that are linearly independent. We call this matroid a *linear matroid* or *matrix matroid*. X was also a set of edges in some graph, and $\mathcal{I}$ was a set of subsets of these edges that formed a forest in this graph. This matroid is called a *graph matroid*. Finally, the matroid related to the problem is a *transversal matroid*. Please note that here the ground set is the set of vertices from one side, and the independent sets are sets of those vertices that can be chosen in some matching. Matching on edges, unfortunately, is not a matroid. Consider the following graph with two valid matchings A and B:



A (in bold)



B (in bold)

As $|A| < |B|$, we should be able to find an edge in $B \setminus A$ and add it to A to still have a valid matching (using the exchange property) and here we cannot do this.

We will show that the graphic matroid indeed fulfills all matroid properties. First two properties are quite trivial. An empty set of edges is indeed a forest. Similarly, when we remove any number of edges from a forest, we cannot create any new cycles, therefore the second property is also fulfilled. Finally, to prove the exchange property, we need the following observation: if $|A| < |B|$, then forest B has fewer connected components than A. Therefore, from the Dirichlet's pigeonhole principle, there exists a component in B that contains vertices of at least two components of A. Therefore we can choose any two components and then choose any path connecting those components in B and then choose any edge from this path and add it to A.

As an exercise, you can check that both the linear matroid and the transversal matroid are indeed matroids.

Problem Red-black trees

Algorithmic Engagements 2018, the grand final, practice session.
Limits: 1s, 64MB.
`https://kostka.dev/sp/rbt`

You probably know exactly what red–black trees are. Bytek would like to have such a tree, but he is not sure what this term means, so he drew an undirected graph with n vertices and m edges. After that he colored each edge in red or black. He calls a subgraph a *red-black tree*, if it is a tree spanning on all n vertices (a connected acyclic subgraph that contains all vertices).

Let us say that each black edge has weight 1 and the red ones have weight 2. The weight of a tree is a sum of weights of all edges of this tree. For a given graph, find the number of different possible weights of red-black trees.

Input

The first line of the standard input contains two integers n and m ($1 \le n \le 100\,000$, $n - 1 \le m \le 300\,000$), denoting the number of vertices and edges in the graph, respectively. The next m lines describe the edges in the graph. The i-th of these lines

contains three integers a_i, b_i, c_i $(1 \leq a_i, b_i \leq n, a_i \neq b_i, c_i \in \{1, 2\})$ meaning that i-th edge connects vertices a_i and b_i and is black if $c_i = 1$, or red otherwise.

The graph is connected and may contain multiple edges.

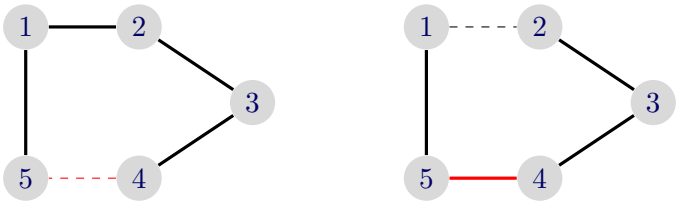
Output

Output one integer in a single line – the number of possible weights of red-black trees.

Example

For the input data:	the correct result is:
5 5 1 2 1 2 3 1 3 4 1 1 5 1 4 5 2	2

Note: Possible weights are 4 and 5, as shown below. The edges that are not selected are marked with a dashed line.



Solution

The problem is to find all possible weights of a spanning tree in a undirected graph, knowing that the weights of the edges are 1 and 2.

We can find the trees that maximize and minimize the weight in $O(m \log m)$ time using for example Kruskal's algorithm. Let us call these trees T_{MAX} and T_{MIN} respectively.

Now we need to notice that we can find trees with all weights between $w(T_{MIN})$ and $w(T_{MAX})$. To prove that, we will use the fact that forests in this graph form a matroid. Let us consider the following algorithm:

$F = T_{MIN}$
while $F \neq T_{MAX}$
  remove some edge $e \in F$, that $e \notin T_{MAX}$,
  add some edge $f \in T_{MAX}$ to F, in a way that F remains a forest

Please note that removing an edge from F will still result in a forest. Moreover, $|F| < |T_{MAX}|$, therefore we can use the exchange property to prove that we can always find some edge f that we can add to F.

Moreover, in each step of this algorithm we exchange only one edge, therefore F is always a tree (it has to remain acyclic). Furthermore, the weight of the tree can change by at most one, therefore we will find trees of all weights between $w(T_{MIN})$ and $w(T_{MAX})$.

To conclude the problem, we just need to find weights of T_{MIN} and T_{MAX} and the answer is $w(T_{MAX}) - w(T_{MIN}) + 1$. Overall, the solution's time complexity depends on the algorithm used to find the minimal and maximal spanning tree and for example, for Kruskal's algorithm works in $O(m \log m)$ time.

Basis of the matroid

We can conclude from the exchange property that every maximal independent set has the same size. We call each such a maximal set the *basis* of this matroid. For graph matroids, it is a simple observation, as every connected component of size k has exactly $k-1$ edges that do not form a cycle in this component. In the linear matroid, the size of the basis is the *dimension* of space described by these vectors, or the *rank*, if we consider these vectors as rows of some matrix. The term rank is also used for matroids, as the rank of the matroid describes the size of the basis of this matroid.

8.1. Optimization problems on matroids

So why do we even focus on matroids? A very interesting fact that we will discuss now is that for optimization problems on matroids, a simple greedy method gives optimal results.

For matroid $(X, \mathcal{I})$, let us introduce a weight function $w : X \to \mathbb{R}^+$. Now, the weight of a subset will be the sum of weights of its elements. Our task will be to find an independent set in I that maximizes this sum.

We will now introduce a simple, greedy algorithm that solves this problem.

$OPT = \varnothing$
for $x \in X$ *sorted non-ascending by* w
  if $(OPT \cup \{x\}) \in \mathcal{I}$
    $OPT \leftarrow OPT \cup \{x\}$
return OPT

The algorithm above finds the optimal solution for any matroid $(X, \mathcal{I})$. Please note that here we solved the maximizing problem. When we want to solve the minimizing problem, we should iterate over all elements of the ground set in the non-decreasing order.

In particular, there is a well-known problem that uses this algorithm. When we take a graph matroid for a connected graph, you can notice that finding the independent set of maximal weight corresponds to finding the maximum spanning tree. In the Kruskal's algorithm [Kruskal, 1956], we greedily add an edge with the maximum weight that was not considered before, if it does not create a cycle with already chosen edges. Here checking if a set of edges does not contain cycle is pretty easy (we can use a disjoint-set structure, aka union-find) and can be done in $O(\alpha(n))$ with the same complexity for adding an edge to the set.

Please note that, in general, checking if a set is independent might be problematic. For example, checking if a set of vectors is independent is more difficult.

Let us prove that the Kruskal's algorithm indeed find the maximum spanning tree. Of course we always keep a forest (we are explicitly checking if we do not form any cycles), therefore let us focus on the maximality.

We want to prove that any set chosen by Kruskal's algorithm belongs to some maximal spanning tree and we will prove it by induction. The base property (for an empty set) is trivially fulfilled. Otherwise, let us assume that we have a subset of edges F already chosen by the algorithm and the maximal spanning tree T containing F. Now if we want to add a new edge e to T, we have two cases:

- $e \in T$, then we can add e to F and the property holds,
- $e \notin T$, then $\{e\} \cup T$ contains a cycle C. Please note that C has some edge c that does not belong to F, because $\{e\} \cup F$ does not form a cycle. As c belongs to T, it was not considered by the algorithm before and has the weight at most as large as e. Then let us consider $T \cup \{e\} \setminus \{f\}$. It is a spanning tree. Moreover this tree has weight not smaller than T, therefore it is a maximal spanning tree that contains $F \cup \{e\}$ and the property still holds.

Therefore, by induction, we proved that with each step of algorithm we keep the maximal spanning forest.

A proof for a general matroid is similar, but more abstract and we will skip it. You can find it for example in [Cormen et al., 2009].

Problem Binary robots

Polish Olympiad in Informatics training camp 2011.
Limits: 2s, 64MB.
`https://kostka.dev/sp/bin`

Binary robots are the latest craze. A binary robot is always designed so that it can do two kinds of job (for example, sewing and prying, or food and philosophizing), but it cannot do both at the same time. Sometimes, fortunately, rarely, the robot, due to a hardware failure, is only capable of one job.

Byteasar runs a binary robots rental company. He rents n robots, each of which has specific capabilities and can be rented at a fixed price. Byteasar can choose from m offers for renting robots, each of them related to a different type of job. The hired robot can only deal with one of the jobs it can perform. Each offer is designed for at most one robot. Of course, Byteasar does not have to rent all robots or accept all offers. Write a program that will calculate how much Byteasar can earn.

Input

First line of the standard input contains three integers: n, m and q ($1 \leq n, m \leq 1\,000\,000, 0 \leq q \leq 2n$) denoting the number of robots, the number of jobs to be performed (that is, the number of offers) and the total number of robotic abilities. Robots are numbered from 1 to n, and jobs from 1 to m.

In the second line there is a sequence of n integers $w_1, w_2, \ldots, w_n$ ($1 \leq w_i \leq 10^9$), where w_i denotes the price for renting the robot i.

The next q lines contain two integers a_i, b_i, ($1 \leq a_i \leq n, 1 \leq b_i \leq m$) indicating that the robot a_i can do the job b_i. No pair (a_i, b_i) repeats. For each $x = 1, 2, \ldots, n$, one or two pairs (x, y) will appear on the input.

Output

Your program should output one integer to the output - Byteasar's maximum profit.

Example

For the input data:	the correct result is:
3 2 4 3 1 4 1 1 2 1 2 2 3 2	7

Solution

We have a bipartite graph, with the robots on one side and the jobs on the other. We have to find a matching that will maximize profit. There is one important restriction: the degree of the vertices representing robots is at most two.

We know that this matching problem (finding a set of vertices that can be matched) is indeed a matroid and we can solve it easily in $O(nm)$, using alternating paths, but we can do this even faster.

Let's consider the following graph: now the jobs will be the vertices and the robots will be represented by edges connecting jobs that the given robot can perform. Now in this graph we are looking for a set of edges such that every connected component has at most one cycle (we call such a graph a *pseudoforest*). So now, we are looking for a pseudoforest that maximizes the total weight of edges.

Pseudoforests in the graph also form a matroid, therefore a greedy approach works too (we can add edges in the non-decreasing order). We just need to have an easy way to check if every connected component has at most one cycle. We can do this with a slightly modified disjoint-set structure (union-find).

The overall time complexity is $O(n \log n + n\alpha(m))$.

8.2. Matroid intersection

Unfortunately, an intersection of two matroids is rarely a matroid, but we still can find the largest common independent set in two matroids over the same ground set. The approach that we will introduce in this section was first described in [Lawler, 1975] and [Edmonds, 1979].

First let us introduce a new matroid – the *partition matroid* – which should be

quite intuitive. Let us consider k boxes with different candies (candies will be our ground set in this example). For each box, we set some limit, how many candies can we take from this box. Now we will call a set of candies independent if for every box, we did not exceed the limit of chosen candies in this box.

Now let us consider a bipartite graph $((V_1, V_2), E)$. Now we will define a partition matroid on E in a way that we cannot choose more than one edge connected to every vertex in V_1. In a similar fashion, we can define another partition matroid with limits on vertices in V_2. Then, the problem of finding the largest common independent set in the intersection of these two matroids is exactly the problem of finding the largest matching in this graph, which unfortunately is not a matroid.

But we can still find a maximum matching in a bipartite graph in polynomial time! Moreover, we will show in this section a polynomial time algorithm that can find the largest common independent set for the intersection of any matroids.

To do so, let us solve the following problem.

Problem Coin Collector

2011 Southwestern Europe Regional Contest.
Limits: 2s, 128MB.
`https://kostka.dev/sp/coi`

As a member of the Association of Coin Minters (ACM), you are fascinated by all kinds of coins and one of your hobbies involves collecting national currency from different countries. Your friend, also an avid coin collector, has her heart set on some of your precious coins and has proposed to play a game that will allow the winner to acquire the loser's collection (or part thereof).

She begins by preparing two envelopes, each of them enclosing two coins that come from different countries. Then she asks you to choose one of the two envelopes. You can see their contents before making your choice, and also decline the offer and take neither. This process is repeated for a total of r times. As the game progresses, you are also allowed to change your mind about your previous picks if you think you can do better. Eventually, your friend examines the envelopes in your final selection, and from among them, she picks a few envelopes herself. If her selection is non-empty and includes an even number of coins from every country (possibly zero), she wins and you must hand over your entire coin collection to her, which would make years of painstaking effort go to waste and force you to start afresh. But if you win, you get to keep the coins from all the envelopes you picked.

Despite the risks involved, the prospect of enlarging your collection is so appealing that you decide to take the challenge. You'd better make sure you win as many coins as possible.

Input

The first line of the standard input contains one integer t, the number of testcases. The first line of each test case is the number r of rounds ($1 \leq r \leq 300$); a line with $r = 0$ indicates the end of the input. The next r lines contain four non-negative integers $0 \leq a, b, c, d < 10\,000$, meaning that your friend puts coins from countries labeled a and b inside one of the envelopes, and c and d inside the other one ($a \neq b$, $c \neq d$).

Output

Print a line per test case containing the largest number of coins you are guaranteed.

Example

For the input data:

```
2
4
0 1 0 5
5 1 0 5
1 2 0 1
1 5 2 0
6
1 4 1 4
2 4 2 4
0 3 0 3
0 4 0 4
4 3 4 3
1 3 1 3
```

the correct result is:

```
6
8
```

Solution

The simplified problem is as follows:

- There are r pairs of envelopes and we are allowed to pick at most one of the envelopes from each pair.
- At the end, there cannot be any subset of chosen envelopes that contains an even number of coins from each country.

Let us consider a graph, where vertices represent the countries, and edges are envelopes (each envelope contains exactly two coins).

Now, think about matroids. The first condition is, of course, a partition matroid - we are allowed to choose at most one edge from each box containing two edges. The second condition is also a matroid: if there exists a subset of edges in which every vertex has an even degree, that means there exists a cycle in this graph. Therefore we are looking for an acyclic graph, or, in other words, a graphic matroid.

So now our task is to find the largest subset of the intersection of a partition matroid and a graphic matroid. How we should approach this problem? Below we will try to show some intuition, how this problem should be solved. We will skip unnecessary proofs.

Let us try greedily. We will start with an empty set of envelopes/edges OPT and consider each pair of edges (u, v) and try to expand this set, keeping an invariant that OPT fulfills both conditions. If the first element from this pair (u) does not form a cycle with already chosen edges, add it to our set ($OPT = OPT \cup \{u\}$). If that is not the case, that means that we have found a cycle C (we keep an invariant that the previous set did not contain a cycle). Let us break this cycle by removing some edge $e \in C$. That means that there was an edge e' that was in pair with e, and we can check if we can add edge e' to our set. Therefore we need to check if $OPT \cup \{u\} \setminus \{e\} \cup \{e'\}$ does not contain any cycles. If that is the case, we have found a proper set, that has a larger size than the previous one. Otherwise, we can continue our search, find a cycle C_1 in $OPT \cup \{u\} \setminus \{e\} \cup \{e'\}$ and so on...

We have shown that the matching problem is an instance of the matroid intersection problem; we will borrow an idea of an augmenting path, but in a slightly different fashion. If we consider a directed bipartite graph G_{OPT}, this time on edges, where on the left side we will consider edges belonging to OPT, and on the other side edges not belonging to the chosen set, we can have the following edges in G_{OPT} for $a \in OPT$ and $b \in E \setminus OPT$:

- (a, b) if (a, b) are in a pair (of envelopes),
- (b, a) if $OPT \setminus \{a\} \cup \{b\}$ does not contain cycles.

Therefore, for a pair (u, v), we are looking for an augmenting path

$$(b_1 \in \{u, v\}, a_1, b_2, a_2, \ldots, b_k)$$

in G_{OPT} that will allow us to add all edges $b_1, b_2, \ldots, b_k$ and remove edges $a_1, a_2, \ldots a_{k-1}$. So we need to find a path that goes from any edge from the pair to any edge with out-degree 0.

Please note that not every path is good. We might find a path that involves some edges that were already removed (the graph G_{OPT} is changing while we are

constructing OPT and we might use the same vertex in G_{OPT} several times, but we cannot remove the same edge more than once), but that means that there will be a shortcut in this path. If we find the shortest path fulfilling our condition, that means that there will not be any shortcuts.

It can be proven that OPT is optimal if we cannot find any augmenting path and thus this algorithm is correct.

To finalize, what we have to do is to consider pairs one by one, construct a new graph for already chosen subset OPT and check if we can augment this set by finding an augmenting path in the constructed graph G_{OPT}. This solution can be implemented to work in $O(r^3)$ time.

Now let us explain the general algorithm, but first we need to introduce some formal notation. We consider two matroids over the same ground set: $M_1 = (X, \mathcal{I}_1)$ and $M_2 = (X, \mathcal{I}_2)$. By the intersection of these two matroids we mean $M_1 \cap M_2 = (X, \mathcal{I}_1 \cap \mathcal{I}_2)$.

We will start from the empty set and in each step, we will try to increase the common independent set. Let us define an *exchange graph* D_{OPT} for already chosen independent set $OPT \in \mathcal{I}_1 \cap \mathcal{I}_2$:

- D_{OPT} will be a directed bipartite graph with two sets of vertices OPT and $X \setminus OPT$,
- the edges are defined in the following way: for $a \in OPT$ and $b \in X \setminus OPT$:
  - $(a, b) \in E$ if $OPT \setminus \{a\} \cup \{b\} \in \mathcal{I}_1$,
  - $(b, a) \in E$ if $OPT \setminus \{a\} \cup \{b\} \in \mathcal{I}_2$.

That means that we have the edge (a, b), if exchanging a for b, will give an independent set in $\mathcal{I}_1$, and (b, a) if exchanging a for b will result in an independent set in $\mathcal{I}_2$.

We hope to find an augmenting path in this directed graph from $X \setminus OPT$, i.e. sets $A \subset OPT$ and $B \subset X \setminus OPT$, such that $|A| + 1 = |B|$ and $OPT \setminus A \cup B \in \mathcal{I}_1 \cap \mathcal{I}_2$. Let us define two more subsets in $X \setminus OPT$:

- $SOURCES = \{b \in X \setminus OPT : OPT \cup \{b\} \in \mathcal{I}_1\}$,
- $SINKS = \{b \in X \setminus OPT : OPT \cup \{b\} \in \mathcal{I}_2\}$.

Now it turns out that if we will find the shortest path from $SOURCES$ to $SINKS$, we can augment OPT by exchanging objects on this path and we will have larger OPT still fulfilling all conditions.

Therefore, our algorithm will work as follows:

$OPT = \emptyset$
(optionally) greedily add to OPT any $b \in X \setminus OPT$, such that
$OPT \cup \{b\} \in \mathcal{I}_1 \cap \mathcal{I}_2$
while *true*
  build the graph D_{OPT} as described above
  calculate $SOURCES$ and $SINKS$ as defined above
  $augmentingPath \leftarrow$ shortest path from $SOURCES$ to $SINKS$
  if $augmentingPath$ *does not exist*
    return OPT
  else
    augment OPT with $augmentingPath$

We will skip the complete proof that the algorithm above indeed finds the largest independent set in the intersection of two matroids because it is complicated and technical, but you can find it for example in [Edmonds, 2003] or [Welsh, 2010].

Please note that the time complexity is indeed polynomial, but we still need to consider the complexity of adding a new object into an optimal set, and testing if the set is still independent.

While implementing this algorithm, we strongly recommend to write it as generally as possible. In particular, we can have a template that can be used with any two matroids that can provide two methods: one to check if after adding an element x to the already considered set, the set will remain independent, and one to actually add this element to the set.

Problem Pick Your Own Nim

2019 Petrozavodsk Winter Camp, Yandex Cup.
Limits: 2s, 512MB.
`https://kostka.dev/sp/nim`

Alice and Bob love playing the Nim game. In the game of Nim, there are several heaps of stones. On each turn, the player selects any heap and takes some positive number of stones from it. The player who takes the last stone wins the game. They played it so many times that they learned how to determine the winner at a first glance: if there are $a_1, a_2, \ldots, a_n$ stones in the heaps, the first player wins if and only if the bitwise xor $a_1 \oplus a_2 \oplus \ldots \oplus a_n$ is nonzero.

They heard that in some online games players pick their characters before the game, adding a strategic layer. Why not do it with Nim?

They came up with the following version. Alice and Bob each start with several boxes with heaps of stones. In the first phase, they pick exactly one heap from each

box. In the second phase Alice chooses some nonempty subset of those heaps, and the regular Nim game starts on chosen heaps with Bob to move first.

Bob already knows which heaps Alice picked. Help him to perform his picks so that he wins the game no matter which heaps Alice chooses during the second phase.

Input

On the first line, there is a single integer n $(0 \leq n \leq 60)$ the number of heaps picked by Alice.

If $n > 0$ on the next line there are n integers: the sizes of those heaps. Otherwise, this line is omitted.

On the next line, there is a single number m $(1 \leq m \leq 60)$, the number of Bob's boxes.

Each of the next m lines contains the description of a box. Each description starts with a number k_i $(1 \leq k_i \leq 5000)$, the number of heaps in the box. Then k_i numbers follow, denoting the sizes of those heaps.

The size of each heap is between 1 and $2^{60} - 1$, inclusive. The total number of heaps in Bob's boxes does not exceed 5000.

Output

If Bob cannot win (that is, no matter what he picks, Alice can make such a choice that the resulting Nim position is losing), print `-1`. Otherwise, print m integers: the sizes of the heaps Bob should pick from his boxes in the same order in which the boxes are given in the input.

Examples

For the input data:

```
2
1 2
2
2 1 2
3 1 2 3
```

the correct result is:

```
-1
```

whereas for the input data:

```
1
5
2
3 1 2 3
4 4 5 6 7
```

the correct result is:

```
1
6
```

Solution

This problem can be reduced to the matroid intersection problem. We just need to find proper matroids. One is pretty easy: it is just a partition matroid (we have to choose exactly one heap from each box). The second one is a linear matroid. If we consider the heaps as vectors on bits, then if Alice can find a dependent set on these vectors, then she can choose them and then bitwise xor will be 0 and Bob will lose.

Intersection of three matroids

We saw that we can easily solve optimization problems on a simple matroid and we have a polynomial time algorithm for the intersection of two matroids. Unfortunately, we rarely can go any further. Finding the maximum independent set in the intersection of three matroids is NP-hard.
We will now show this fact by a reduction from the Hamiltonian path problem (finding a path that goes through every vertex in a graph) on a directed graph. So now, let us take an instance of this problem, i.e. graph $G = (V, E)$ and consider the following three matroids:

- M_1 will be a graph matroid over E (we do not care about direction here),
- M_2 will be a partition matroid that will guarantee that every vertex has in-degree at most 1,
- M_3 will be a partition matroid that will guarantee that every vertex has out-degree at most 1.

Now it is easy to check that if we can find an independent set in $M_1 \cap M_2 \cap M_3$ of size $|V| - 1$, that means that we can find a Hamiltonian path in G.
Therefore, in general, finding the maximum independent set in the intersection of more than two matroids is NP-hard.

Bibliography

[Aggarwal et al., 1994] Aggarwal, A., Schieber, B., and Tokuyama, T. (1994). Finding a minimum-weight k-link path in graphs with the concave Monge property and applications. *Discrete & Computational Geometry*, 12(3):263–280.

[Apostolico et al., 1993] Apostolico, A., Breslauer, D., and Galil, Z. (1993). Parallel detection of all palindromes in a string. *Theoretical Computer Science*, 141(1&2):163–173.

[Bar-Noy and Ladner, 2004] Bar-Noy, A. and Ladner, R. E. (2004). Efficient algorithms for optimal stream merging for media-on-demand. *SIAM Journal on Computing*, 33(5):1011–1034.

[Bellott et al., 2014] Bellott, D. W., Hughes, J. F., Skaletsky, H., Brown, L. G., Pyntikova, T., Cho, T.-J., Koutseva, N., Zaghlul, S., Graves, T., Rock, S., et al. (2014). Mammalian Y chromosomes retain widely expressed dosage-sensitive regulators. *Nature*, 508(7497):494–499.

[Bentley, 1984] Bentley, J. (1984). Programming pearls: algorithm design techniques. *Communications of the ACM*, 27(9):865–873.

[Bertsekas, 2014] Bertsekas, D. P. (2014). *Constrained optimization and Lagrange multiplier methods*. Academic press.

[Bottenbruch, 1962] Bottenbruch, H. (1962). Structure and use of ALGOL 60. *Journal of the ACM*, 9(2):161–221.

[Brucker, 1995] Brucker, P. (1995). Efficient algorithms for some path partitioning problems. *Discrete Applied Mathematics*, 62(1-3):77–85.

[Castro et al., 2016] Castro, R., Lehmann, N., Pérez, J., and Subercaseaux, B. (2016). Wavelet trees for competitive programming. *Olympiads in Informatics*, 10:19–37.

[Chazelle and Guibas, 1986a] Chazelle, B. and Guibas, L. J. (1986a). Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1-4):133–162.

[Chazelle and Guibas, 1986b] Chazelle, B. and Guibas, L. J. (1986b). Fractional cascading: II. applications. *Algorithmica*, 1(1-4):163–191.

[Claude et al., 2015] Claude, F., Navarro, G., and Ordóñez Pereira, A. (2015). The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32.

[Clifford and Clifford, 2007] Clifford, P. and Clifford, R. (2007). Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53–54.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

[de la Fuente, 2000] de la Fuente, A. (2000). *Mathematical methods and models for economists*. Cambridge University Press.

[Diks et al., 2018] Diks, K., Idziaszek, T., Lacki, J., and Radoszewski, J. (2018). *Przygody Bajtazara. 25 lat Olimpiady Informatycznej*. PWN.

[Dymchenko and Mykhailova, 2015] Dymchenko, S. and Mykhailova, M. (2015). Declaratively solving tricky Google Code Jam problems with Prolog-based ECLiPSe CLP system. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 2122–2124.

[Edmonds, 1979] Edmonds, J. (1979). Matroid intersection. In *Annals of Discrete Mathematics*, volume 4, pages 39–49. Elsevier.

[Edmonds, 2003] Edmonds, J. (2003). Submodular functions, matroids, and certain polyhedra. In *Combinatorial Optimization—Eureka, You Shrink!*, pages 11–26. Springer.

[Feng and Li, 2017] Feng, X. and Li, S. (2017). Design of an area-effcient million-bit integer multiplier using double modulus NTT. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(9):2658–2662.

[Galil and Park, 1992] Galil, Z. and Park, K. (1992). Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92(1):49–76.

[Gilbert and Moore, 1959] Gilbert, E. N. and Moore, E. F. (1959). Variable-length binary encodings. *Bell System Technical Journal*, 38(4):933–967.

[Grossi et al., 2003] Grossi, R., Gupta, A., and Vitter, J. S. (2003). High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850. Society for Industrial and Applied Mathematics.

[Hamming, 1950] Hamming, R. W. (1950). Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160.

[Horner, 1819] Horner, W. G. (1819). XXI. a new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*, pages 308 – 335.

[Idziaszek, 2015] Idziaszek, T. (2015). Informatyczny kącik olimpijski: Wiercenia. *Delta, czerwiec 2015*.

[Knuth, 1971] Knuth, D. (1971). Optimum binary search trees. *Acta Informatica*, 1(1):14–25.

[Knuth, 1997] Knuth, D. E. (1997). *The art of computer programming*, volume 3. Pearson Education.

[Kruskal, 1956] Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50.

[Lakhani et al., 2010] Lakhani, K. R., Garvin, D. A., and Lonstein, E. (2010). Topcoder (a): Developing software through crowdsourcing. *Harvard Business School General Management Unit Case 610-032*.

[Lawler, 1975] Lawler, E. L. (1975). Matroid intersection algorithms. *Mathematical programming*, 9(1):31–56.

[Macon and Spitzbart, 1958] Macon, N. and Spitzbart, A. (1958). Inverses of Vandermonde matrices. *The American Mathematical Monthly*, 65(2):95–100.

[Manacher, 1975] Manacher, G. (1975). A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM*, 22(3):346–351.

[Mehlhorn and Näher, 1990] Mehlhorn, K. and Näher, S. (1990). Dynamic fractional cascading. *Algorithmica*, 5(1-4):215–241.

[Navarro, 2014] Navarro, G. (2014). Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20.

[Rubinchik and Shur, 2015] Rubinchik, M. and Shur, A. M. (2015). EERTREE: an efficient data structure for processing palindromes in strings. In *International Workshop on Combinatorial Algorithms*, pages 321–333. Springer.

[Rubinchik and Shur, 2018] Rubinchik, M. and Shur, A. M. (2018). EERTREE: an efficient data structure for processing palindromes in strings. *European Journal of Combinatorics*, 68:249–265.

[Tommila, 2003] Tommila, M. (2003). A C++ high performance arbitrary precision arithmetic package.

[Turner, 1966] Turner, L. R. (1966). Inverse of the Vandermonde matrix with applications. *Nasa Technical Note*.

[Welsh, 2010] Welsh, D. J. (2010). *Matroid theory*. Courier Corporation.

[Whitney, 1935] Whitney, H. (1935). On the abstract properties of linear dependence. *American Journal of Mathematics*, 57(3):509–533.

[Yao, 1980] Yao, F. F. (1980). Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, pages 429–435.

[Yao, 1982] Yao, F. F. (1982). Speed-up in dynamic programming. *SIAM Journal on Algebraic Discrete Methods*, 3(4):532–540.