Faster Algorithms for the Longest Common Increasing Subsequence Problem

Szybsze algorytmy dla problemu
Najdłuższego Wspólnego Podciągu Rosnącego

Anadi Agrawal

Praca licencjacka

Promotor: dr Paweł Gawrychowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

17 czerwca 2021

Abstract

The Longest Common Increasing Subsequence (LCIS) is a natural variant of the classical Longest Common Subsequence (LCS), in which we additionally require the common subsequence to be strictly increasing. For both problems, we know that there is no strongly subquadratic algorithm unless SETH fails (which is considered unlikely). However, for the latter problem, we have known since 1980 that a slightly subquadratic $\mathcal{O}(n^2/\log^2 n)$ algorithm exists. For the former problem, the same approach does not seem to work, though, and only in 2020, Duraj was able to break the $\mathcal{O}(n^2)$ barrier by designing an $\mathcal{O}(n^2(\log\log n)^2/\log^{1/6} n)$ time algorithm, which was later improved to $\mathcal{O}(n^2\log\log n/\sqrt{\log n})$ by Agrawal and Gawrychowski [ISAAC 2020]. In this thesis, we describe the latter algorithm, and further improve it to work in $\mathcal{O}(n^2(\log\log n)^2/\log^{2/3} n)$ time by combining the ideas from both solutions. We also survey the existing work and include some experimental results.

Problem najdłuższego wspólnego podciągu rosnącego (NWPR) jest jednym z wariantów klasycznego problemu najdłuższego wspólnego podciągu (NWP). W tym wariancie dodatkowo wymagamy, aby szukany wspólny podciąg był ściśle rosnący. Dla obu problemów wiadomo, że nie istnieje silnie podkwadratowy algorytm, chyba że SETH nie jest prawdą (co jest uważane za mało prawdopodobne). Jednakże, dla problemu NWP wiadomo, że istnieje nieco podkwadratowy algorytm o złożoności $\mathcal{O}(n^2/\log^2 n)$. Dla NWPR, podobne podejście wydawało się nie działać i dopiero w 2020, Duraj był w stanie uzyskać algorytm w złożoności $\mathcal{O}(n^2(\log\log n)^2/\log^{1/6} n)$, które zostało poprawione do $\mathcal{O}(n^2\log\log n/\sqrt{\log n})$ przez Agrawala i Gawrychowskiego [ISAAC 2020]. W tej pracy, przedstawiony jest ten drugi algorytm, jak i jego ulepszenie do złożoności $\mathcal{O}(n^2(\log\log n)^2/\log^{2/3} n)$ poprzez połączenie pomysłów z obu rozwiązań wspomnianych wyżej. Dodatkowo, przedstawiany jest obecny stan wiedzy dla problemu NWPR, wraz z pewnymi eksperymentalnymi wynikami.

Contents

Chapter 1

Introduction

In the classical Longest Common Subsequence (LCS) problem, we aim to find the length of the longest subsequence, common to two strings given in the input. The textbook algorithm for this problem works in $\mathcal{O}(n^2)$, but we also have asymptotically faster algorithms working in $\mathcal{O}\left(n^2/\log^2 n\right)$ [10] for constant alphabets and $\mathcal{O}\left(n^2 \log\log n/\log^2 n\right)$ [8] for general alphabets. It is unlikely that a strongly subquadratic algorithm exists [1], and even achieving $\mathcal{O}\left(n^2/\log^{7+\varepsilon} n\right)$ [2] would lead to some unexpected consequences.

The problem we consider in this thesis is a variation of the LCS problem, and we define it as follows:

Problem: Longest Common Increasing Subsequence (LCIS)

Input: integer sequences $A[1..n]$ and $B[1..n]$

Output: largest ℓ such that there exist indices $i_1 < \ldots < i_\ell$ and $j_1 < \ldots < j_\ell$ with the property that (i) $A[i_k] = B[j_k]$, for every $k = 1, \ldots, \ell$, and (ii) $A[i_1] < \ldots < A[i_\ell]$.

The straightforward dynamic programming from LCS [12] would yield an $\mathcal{O}(n^3)$ solution for LCIS. However, by exploiting the properties of LCIS, this approach can be optimised to $\mathcal{O}\left(n^2\right)$ [13] (even in linear space [11]). Similarly, as in LCS, the existence of a strongly subquadratic algorithm would refute SETH [7]. The SETH is the conjecture that there is no algorithm solving k-SAT for all $k \in \mathbb{N}$ in $\mathcal{O}(2^{\delta n})$ for any $\delta < 1$.

The $\mathcal{O}(n^2)$ barrier was unbeaten for this problem for a long time. The usual "Four Russians" technique does not seem to be directly applicable here, as in the dynamic programming, it was hard to memorise information using $o(n^2)$ words. However, recently this barrier was broken by Duraj [6], who presented the $\mathcal{O}\left(n^2(\log\log n)^2/\log^{1/6} n\right)$ solution exploiting the combinatorial properties of LCIS. It was later improved by

Agrawal and Gawrychowski to $\mathcal{O}\left(n^2 \log\log n/\sqrt{\log n}\right)$ [3] with an algorithm based on tabulation.

This thesis is based on the author's work published in ISAAC 2020, although it improves the upper bound given in that paper to $\mathcal{O}(n^2(\log\log n)^2/\log^{2/3} n)$. In the beginning, an overview of the progress made for LCIS is presented. Further, the algorithm from ISAAC and its improvement is shown. In the end, a subset of these algorithms was implemented and compared on selected tests.

Chapter 2

An Overview of the Solutions

2.1 Preliminaries

We work with sequences of integers. For sequence A, $A[i]$ denotes the value of the i-th element of the sequence, while $A[1..i]$ denotes the prefix of length i. $|A|$ is the length of that sequence.

The sequences we are given in the input are denoted by A and B. The length of each of these sequences is equal to n.

Definition 2.1.1 (Matching pair)**.** A pair of indices (i, j) is called a matching pair if $A[i] = B[j]$. Further, it is called a v-pair if $A[i] = B[j] = v$.

The $LCIS(i, j)$ is defined as LCIS of sequences $A[1..i], B[1..j]$. Further, $LCIS{\rightarrow}(i, j)$ denotes the longest strictly increasing subsequence of $A[1..i]$ and $B[1..j]$ which includes both $A[i]$ and $B[j]$ (so in particular, $A[i] = B[j]$).

2.2 LCIS for Constant Alphabet

Restricting the alphabet to a constant size simplifies the LCIS problem. In particular, this problem can be solved in linear time.

Define $\mathsf{last}[i][c]$ as the largest index $j < i$ such that $A[i] = c$. If such j does not exist then $\mathsf{last}[i][c] = 0$. Similarly $\mathsf{next}[i][c]$ is defined as the smallest index $j > i$ such that $B[j] = c$. If such j does not exist then $\mathsf{next}[i][c] = \infty$.

Consider $\mathsf{dp}[i][r]$ defined as the smallest j, such that $LCIS^{\rightarrow}(i, j) = r$. If such j does not exist, then $\mathsf{dp}[i][r] = \infty$. Additionally, we set $\mathsf{dp}[0][0] = 0$. Such dp can be easily calculated in $\mathcal{O}(1)$ using the following transition:

$$\mathsf{dp}[i][r] = \min_{c<A[i]} \{\, \mathsf{next}[\mathsf{dp}[\mathsf{last}[i][c]][r-1]][A[i]] \,\}$$

The result is the largest r such that there exist i fulfilling $\mathsf{dp}[i][r] < \infty$. Because the size of the alphabet bounds r, the above solution works in $\mathcal{O}(n+m)$. Details of how to calculate dp can be found in the following pseudocode:

Algorithm 1 Calculate dp for constant alphabet

```
1:  procedure CalculateDP
2:      ▷ S denotes the size of the alphabet
3:      for c = 1..S do
4:          next[n][c] = ∞
5:          last[1][c] = 0
6:      for i = 2..n do
7:          for c = 1..S do
8:              last[i][c] ← last[i − 1][c]
9:          last[i][A[i − 1]] ← i − 1
10:     for i = n − 1..1 do
11:         for c = 1..S do
12:             next[i][c] ← next[i − 1][c]
13:         next[i][A[i + 1]] ← i + 1
14:
15:     dp[0][0] ← 0
16:     for i = 1..n do
17:         for r = 1..n do
18:             dp[i][r] ← ∞
19:             for c = 1..A[i] − 1 do
20:                 dp[i][r] ← min{ dp[i][r], next[dp[last[i][c]][r − 1]][A[i]] }
```

2.3 $\mathcal{O}(n^2)$ Time and Space Algorithm for LCIS

The solution proposed by Yang, Huang and Chao [13] is based on dynamic programming. In particular, the algorithm keeps $\mathsf{dp}^i_j[k]$ equal to the smallest possible ending number of the increasing subsequence common to $A[1..i]$ and $B[1..j]$ whose length is equal to k.

The key observation is that dp^i_{j-1} and dp^{i-1}_j differs from dp^i_j on at most one entry. Exploiting this property, the authors were able to obtain $\mathcal{O}(n^2)$ solution. As the following solution achieves better bounds, we skip the details of this one.

2.4 $\mathcal{O}(n^2)$ Time and $\mathcal{O}(n)$ Space Algorithm for LCIS

The solution for LCIS achieving these bounds was first proposed by Sakai [11]. However, the solution presented there can be simplified. Therefore, the presented

solution is a simpler version of the algorithm presented in that work.

Let $\mathsf{dp}[i][j]$ denote the maximal length of the increasing subsequence common to $A[1..i]$ and $B[1..j]$ ending with number $B[j]$. Then, the result is equal to $\max_i\{\,\mathsf{dp}[n][i]\,\}$.

An entry in dp can be calculated as follows:

$$\mathsf{dp}[i][j] = \begin{cases} \mathsf{dp}[i-1][j], & \text{if } A[i] \neq B[j], \\ \max\limits_{k<j \wedge B[k]<B[j]} \{\,\mathsf{dp}[i-1][k]+1\,\}, & \text{otherwise.} \end{cases}$$

By inspecting the above transition, we see that if we calculate dp in row-major order, then it suffices to keep only its last row. Thus, the remaining part is how to calculate dp quickly. The idea is to synchronise all calculations for a single row.

For fixed row i, calculate the entries in increasing order of j. Keep an additional variable called best, initialised with 0. Then for fixed j we encounter three cases:

1. Set $\mathsf{dp}[i][j] := \mathsf{dp}[i-1][j]$ and $\mathsf{best} := \max\{\,\mathsf{best}, \mathsf{dp}[i-1][j]+1\,\}$ for $B[j] < A[i]$
2. Set $\mathsf{dp}[i][j] = \mathsf{best}$ for $B[j] = A[i]$
3. Set $\mathsf{dp}[i][j] = \mathsf{dp}[i-1][j]$ for $B[j] > A[i]$

Because we analyze entries in order of increasing values j, in the second case the value of the variable best is equal to $\max_{k<j \wedge B[k]<B[j]}\{\,\mathsf{dp}[i-1][k]+1\,\}$. Details on the exact implementation using a single array of length n can be found beneath.

Algorithm 2 Calculate linear space dp in $\mathcal{O}(n^2)$

```
1:  procedure CalculateDP
2:      for j = 1..n do
3:          dp[j] ← 0
4:      for i = 1..n do
5:          best ← 0
6:          for j = 1..n do
7:              if A[i] = B[j] then
8:                  dp[j] ← best
9:              if A[i] < B[j] then
10:                 best ← max{ best, dp[i][j] + 1}
```

2.5 The First Subquadratic Algorithm

The first one to break the $\mathcal{O}(n^2)$ was Duraj [6]. Here, we outline the main Lemma (without proof) from Duraj's work rather than the algorithm itself. Later, we exploit it in the $\mathcal{O}(n^2/\log^{2/3-\varepsilon})$ algorithm.

Definition 2.5.1 (Significant pairs)**.** Let (x, y) be a v-pair. We call (x', y') a valid pair if it is a v-pair and $(x', y') \neq (x, y), x' \leq x, y' \leq y$. We call (x, y) a significant pair if $LCIS^{\rightarrow}(x, y) > LCIS^{\rightarrow}(x', y')$ for any valid pair (x', y').

Lemma 2.5.2. *For any two sequences A, B, such that $|A|, |B| \leq n$, the number of significant pairs can be bounded by $\mathcal{O}(n^2/\log^{1/3} n)$.*

Chapter 3

Why is it Hard to Beat $\mathcal{O}(n^2)$

As the upper bounds for the $LCIS$ in the worst case were nowhere close to the linear time, the natural question raised if it is possible to get closer to it. Duraj, Künnemann and Polak proved that a strongly subquadratic algorithm, i.e. an $\mathcal{O}(n^{2-\varepsilon})$ algorithm, would refute SETH [7]. In this thesis, the main ideas from their publication are outlined. Details, including proofs, can be found in the original work.

3.1 Hypothesis

Hypothesis 3.1.1 (Strong Exponential Time Hypothesis (SETH))**.** Consider any $\delta \in (0, 1)$. There is no algorithm solving k-SAT for all $k \in \mathbb{N}$ in $\mathcal{O}(2^{\delta n})$.

SETH implies the following conjecture:

Hypothesis 3.1.2 (Orthogonal Vectors (OV) conjecture)**.** Let $A, B \subseteq \{0, 1\}^d$ where $d = \omega(\log n)$, $|A| = |B| = n$. Two vectors a, b are called orthogonal if $\sum_{i=1}^{d} a[i] \cdot b[i] = 0$. There is no $\mathcal{O}(n^{2-\varepsilon})$ algorithm for any $\varepsilon > 0$ determining if there exist $a \in A$ and $b \in B$ such that a, b are orthogonal.

3.2 Reduction to OV Conjecture

Definition 3.2.1 (Inflation)**.** For a sequence $A = \langle a_1, a_1, \ldots, a_n \rangle$ its inflation is defined as follows:

$$\mathsf{inflate}(A) = \langle 2a_0 - 1, 2a_0, 2a_1 - 1, 2a_1, \ldots, 2a_n - 1, 2a_n \rangle$$

Lemma 3.2.2. *For any two sequences A, B, $LCIS(\mathsf{inflate}(A), \mathsf{inflate}(B)) = 2 \cdot LCIS(A, B)$.*

3.2.1 Separator Sequences

Definition 3.2.3 (Separator sequences)**.** Sequences $A = \alpha_0\alpha_1 \dots \alpha_{n-1}$ and $B = \beta_0\beta_1 \dots \beta_{n-1}$ are called separator sequences if for each $i, j \in \{0, 1, \dots, n-1\}$ $LCIS(\alpha_0\alpha_1 \dots \alpha_i, \beta_0\beta_1 \dots \beta_j) = i+j+C$ for a fixed constant C. Here, $\alpha_0, \alpha_1, \dots, \alpha_{n-1}$, $\beta_0, \beta_1, \dots, \beta_{n-1}$ might denote a block of letters.

We construct such sequences for $n = 2^k$ such that $|A_k| = |B_k| = \mathcal{O}(k2^k)$, and $C = 2^k$. Denote $A_k = \alpha_k^0\alpha_k^1 \dots \alpha_k^{2^k-1}$, $B_k = \beta_k^0\beta_k^1 \dots \beta_k^{2^k-1}$. Additionally, by s_k denote the largest value in sequences A_k, B_k. The construction is as follows:

1. $A_0 = B_0 = \langle 1 \rangle$
2. $\alpha_k^{2i} = \mathsf{inflate}(\alpha_{k-1}^i) \circ \langle 2s_{k-1} + 2 \rangle$, $\alpha_k^{2i+1} = \langle 2s_{k-1} + 1, 2s_{k-1} + 3 \rangle$ for $k > 0$
3. $\beta_k^{2i} = \mathsf{inflate}(\beta_{k-1}^i) \circ \langle 2s_{k-1} + 1 \rangle$, $\beta_k^{2i+1} = \langle 2s_{k-1} + 2, 2s_{k-1} + 3 \rangle$ for $k > 0$

It can be proven that A_k, B_k are separator sequences. We also define $\hat{A}_k, \hat{B}_k$ as reversed and multiplied by -1 version of respectively A_k, B_k. It is straightforward to observe, that we can decompose $\hat{A}_k = \hat{\alpha}_k^0\hat{\alpha}_k^1 \dots \hat{\alpha}_k^{2^k-1}$, $\hat{B}_k = \hat{\beta}_k^0\hat{\beta}_k^1 \dots \hat{\beta}_k^{2^k-1}$ such that for any $i, j \in \{0, 1, \dots, 2^k - 1\}$, $LCIS(\hat{\alpha}_k^i \dots \hat{\alpha}_k^{2^k-1}, \hat{\beta}_k^j \dots \hat{\beta}_k^{2^k-1}) = 2 \cdot (2^k - 1) - i - j + 2^k$.

3.2.2 Sequence for a Vector

Let $A = \{a_1, a_2, \dots, a_n\}$, $B = \{b_1, b_2, \dots, b_n\}$ denote two sets defined as in OV conjecture. We define sequences u_i, v_i in a following way:

$$\langle u_i[2p-1], u_i[2p] \rangle = \begin{cases} \langle 2p-1, 2p \rangle, & \text{if } a_i[p] = 0, \\ \langle 2p-1, 2p-1 \rangle, & \text{otherwise} \end{cases}$$

$$\langle v_i[2p-1], v_i[2p] \rangle = \begin{cases} \langle 2p, 2p-1 \rangle, & \text{if } b_i[p] = 0, \\ \langle 2p, 2p \rangle, & \text{otherwise} \end{cases}$$

Lemma 3.2.4. $LCIS(u_i, v_j) = d - \sum_{k=1}^{d} a_i[k] \cdot b_j[k]$

3.2.3 Final Construction

Lemma 3.2.5. *Let $u_0, u_1, \dots, u_{n-1}, v_0, v_1, \dots, v_{n-1}$ be integer sequences of length at most δ. Then, there exist sequences X, Y of length $\mathcal{O}(\delta \cdot n \log n + \sum_i(|u_i| + |v_i|))$, constructible in linear time, such that $LCIS(X, Y) = \max_{i,j}\{LCIS(u_i, v_j)\} + C$ for a constant C depending only on n and l.*

The idea behind this construction is to use separator sequences defined in Definition 3.2.3. We interlace slightly modified sequences $A_k, \hat{A}_k$ with sequences

$u_0, \ldots, u_{n-1}$, and $B_k, \hat{B}_k$ with sequences $v_0, \ldots, v_{n-1}$. For $k \sim \log n$, we achieve desired bounds.

Assume there is a $\mathcal{O}(n^{2-\varepsilon})$ time algorithm for the LCIS problem. Take two sets of vectors, A and B, as defined in OV conjecture. From Lemma 3.2.4 for these vectors, we can construct sequences $u_1, \ldots, u_n$, $v_1, \ldots, v_n$ of the length bounded by $2d$, such that for every $a \in A, b \in B$, the LCIS of respective sequences is equal to d if and only if a and b are orthogonal.

From Lemma 3.2.5 we construct two sequences X, Y such that their LCIS is equal to $C + \max_{i,j}\{LCIS(u_i, v_j)\}$. We can find the LCIS in $\mathcal{O}(n^{2-\varepsilon})$ what gives us $\mathcal{O}((nd \log n)^{2-\varepsilon}) = \mathcal{O}(n^{2-\varepsilon}\mathsf{poly}(d)\mathsf{polylog}(n))$ time algorithm to solve OV problem. This algorithm refutes Hypothesis 3.1.2 and consequently Hypothesis 3.1.1.

Chapter 4

Main Solution

The idea of this algorithm is to combine two solutions which are fast only in some special scenarios. We start with presenting each of these solutions, and in the final section we show how to combine them to obtain a fast algorithm in general case.

4.1 Notation

Let σ be the sequence consisting of all distinct integers present in A and B, arranged in the increasing order, and $\mathsf{cnt}(v)$ be the total number of occurrences of $\sigma[v]$ in A and B. Without loss of the generality we can assume that $\sigma[v] = v$, and write v instead of $\sigma[v]$.

Throughout the paper, $\log x$ denotes $\log_2 x$.

4.2 First Solution (Few Distinct Elements)

In this section we describe an algorithm for finding LCIS in $\mathcal{O}(|\sigma| \cdot n^2 / \log n)$ time.

Let $\mathsf{dp}_v[i][j]$ denote the largest possible length of a sequence C such that:

1. C is an increasing common subsequence of $A[1..i]$ and $B[1..j]$,
2. C consists of elements not larger than v.

Then, our goal is to compute $\mathsf{dp}_{|\sigma|}[n][n]$.

All $|\sigma| \cdot n^2$ entries in dp can be calculated in $\mathcal{O}(1)$ time each using the following recurrence:

$$\mathsf{dp}_{v+1}[i][j] = \begin{cases} \mathsf{dp}_v[i-1][j-1] + 1, & \text{if } A[i] = B[j] = v+1, \\ \max\{\mathsf{dp}_v[i][j], \mathsf{dp}_{v+1}[i-1][j], \mathsf{dp}_{v+1}[i][j-1]\}, & \text{otherwise.} \end{cases}$$

In order to decrease the time we will speed up calculating dp_{v+1} from dp_v. Because calculating dp_{v+1} only requires the knowledge of dp_v, we will only keep the current dp_v and update all of its entries to obtain dp_{v+1}.

Lemma 4.2.1. $0 \leq dp_v[i][j] - dp_v[i][j-1] \leq 1$ *and* $0 \leq dp_v[i][j] - dp_v[i-1][j] \leq 1$.

Proof. A subsequence of $B[1..(j-1)]$ is still a subsequence of $B[1..j]$, so $\mathsf{dp}_v[i][j-1] \leq \mathsf{dp}_v[i][j]$. Consider a sequence C corresponding to $\mathsf{dp}_v[i][j]$, and let C' be C without the last element. Because C is a subsequence of $B[1..j]$, C' is a subsequence of $B[1..(j-1)]$. So, C' is an increasing subsequence of $A[1..i]$ and $B[1..(j-1)]$, hence $|C'| \leq \mathsf{dp}_v[i][j-1]$. As $|C| = |C'| + 1$, we conclude that $\mathsf{dp}_v[i][j] \leq \mathsf{dp}_v[i][j-1] + 1$. The second part of the Lemma follows by a symmetrical reasoning. □

Instead of maintaining dp_v, we keep another table $\mathsf{dp}'_v[i][j] = \mathsf{dp}_v[i][j] - \mathsf{dp}_v[i][j-1]$ (where $\mathsf{dp}_v[i][j] = 0$ for $j < 1$). Due to Lemma 4.2.1, each entry of dp'_v is either 0 or 1. This allows us to store each row of dp'_v by partitioning it into $\mathcal{O}(n/b)$ blocks of length b, with every block represented by a bitmask of size b saved in a single machine word, where $b = \alpha \log n$ for some constant α to be fixed later. By definition, $\mathsf{dp}_v[i][j] = \sum_{k=1}^{j} \mathsf{dp}'_v[i][k]$. In addition to dp'_v, we store the value of $\mathsf{dp}_v[i][j]$ for every block boundary, so $\mathcal{O}(n^2/b)$ values overall. This will allow us later to recover any $\mathsf{dp}_v[i][j]$ in constant time by retrieving the value at the appropriate block boundary and adding the number of 1s in a prefix of some bitmask. We preprocess such prefix sums for every possible bitmask in $\mathcal{O}(2^b \cdot b)$ time and space. To implement updates efficiently we also need the following lemma.

Lemma 4.2.2. $0 \leq dp_{v+1}[i][j] - dp_v[i][j] \leq 1$

Proof. Because allowing using more elements cannot decrease the length, $\mathsf{dp}_v[i][j] \leq \mathsf{dp}_{v+1}[i][j]$. Let C be a sequence corresponding to $\mathsf{dp}_{v+1}[i][j]$, and let C' be C without the last element. Because C is strictly increasing, the elements of C' are not larger than v, so $|C'| \leq \mathsf{dp}_v[i][j]$. Then, using $|C'| + 1 = |C|$ we obtain that $\mathsf{dp}_{v+1}[i][j] - 1 \leq \mathsf{dp}_v[i][j]$. □

We now describe how to obtain the table storing the values of dp'_{v+1} by modifying the table storing the values of dp'_v. To this end, we use the recursion for $\mathsf{dp}_{v+1}[i][j]$ and process the rows one-by-one. We start by copying the corresponding i-th row of dp'_v, and then update the entries going from left to right. In the j-th step, we would like to have correctly determined the values of $\mathsf{dp}'_{v+1}[i][1], \mathsf{dp}'_{v+1}[i][2], \ldots, \mathsf{dp}'_{v+1}[i][j]$ that together encode the values of $\mathsf{dp}_{v+1}[i][1], \mathsf{dp}_{v+1}[i][2], \ldots, \mathsf{dp}_{v+1}[i][j]$. However, during this process we are no longer guaranteed that $\mathsf{dp}_{v+1}[i][j] \leq \mathsf{dp}_{v+1}[i][j+1]$, To overcome this issue, we immediately propagate each value to the right: after increasing $\mathsf{dp}_{v+1}[i][j]$ (by one due to Lemma 4.2.2) we also increase every $\mathsf{dp}_{v+1}[i][k]$ equal to the original value of $\mathsf{dp}_{v+1}[i][j]$, for all $k > j$. This translates into setting $\mathsf{dp}'_{v+1}[i][j]$ to 1

and setting $\mathsf{dp}'_{v+1}[i][k]$ to 0, for the smallest $k > j$ such that $\mathsf{dp}'_{v+1}[i][k] = 1$, if such exists. To implement this efficiently, we maintain k while considering $j = 1, 2, \ldots, n$ in $\mathcal{O}(n)$ overall time. The details of this procedure are shown in Algorithm 3.

Algorithm 3 Calculate the i-th row of dp'_{v+1}

```
1:  procedure CalculateRow(v, i)
2:      ptr ← 1
3:      cur_value ← 0
4:      prv_value ← 0
5:      prv_phase ← 0
6:      for j = 1..n do
7:          dp'_{v+1}[i][j] = dp'_v[i][j]
8:      for j = 1..n do
9:          if ptr ≤ j then ptr ← j + 1
10:         while ptr ≤ n and dp'_{v+1}[i][ptr] = 0 do
11:             ptr ← ptr + 1
12:         cur_value ← cur_value + dp'_{v+1}[i][j]
13:         ▷ cur_value = Σ_{j'=1}^{j} dp'_{v+1}[i][j'] = max{dp_v[i][j], dp_{v+1}[i][j − 1]}
14:         ▷ prv_phase = dp_v[i − 1][j − 1]
15:         if A[i] = B[j] = v + 1 and cur_value = prv_phase then
16:             dp'_{v+1}[i][j] ← 1
17:             cur_value ← cur_value + 1
18:             if ptr ≤ n then dp'_{v+1}[i][ptr] ← 0
19:         prv_phase ← prv_phase + dp'_v[i − 1][j]
20:         prv_value ← prv_value + dp'_{v+1}[i − 1][j]
21:         ▷ prv_value = dp_{v+1}[i − 1][j]
22:         if cur_value < prv_value then
23:             cur_value ← prv_value
24:             dp'_{v+1}[i][j] ← 1
25:             if ptr ≤ n then dp'_{v+1}[i][ptr] ← 0
```

We speed up Algorithm 3 by a factor of b by considering whole blocks of dp'_{v+1} instead of single entries. Consider a single block of dp'_{v+1} consisting of the values of $\mathsf{dp}'_{v+1}[i][j], \mathsf{dp}'_{v+1}[i][j+1], \ldots, \mathsf{dp}'_{v+1}[i][j+b-1]$, and assume that they have been already partially updated by propagating the maximum. To calculate their correct values we need the following information:

1. $\mathsf{dp}'_v[i-1][j], \mathsf{dp}'_v[i-1][j+1], \ldots, \mathsf{dp}'_v[i-1][j+b-1]$,
2. $\mathsf{dp}'_{v+1}[i-1][j], \mathsf{dp}'_{v+1}[i-1][j+1], \ldots, \mathsf{dp}'_{v+1}[i-1][j+b-1]$,
3. $\mathsf{dp}'_{v+1}[i][j], \mathsf{dp}'_{v+1}[i][j+1], \ldots, \mathsf{dp}'_{v+1}[i][j+b-1]$,
4. $\mathsf{dp}_v[i-1][j-1]$,

5. $\mathsf{dp}_{v+1}[i-1][j-1]$,

6. $\mathsf{dp}_{v+1}[i][j-1]$,

7. for which indices $j, j+1, \ldots, j+b-1$ we have $A[i] = B[j] = v+1$.

In fact, we can rewrite the procedure so that instead of the values $\mathsf{dp}_v[i-1][j-1]$, $\mathsf{dp}_{v+1}[i-1][j-1]$, $\mathsf{dp}_{v+1}[i][j-1]$ only the differences $\mathsf{dp}_{v+1}[i-1][j-1] - \mathsf{dp}_v[i-1][j-1]$ and $\mathsf{dp}_{v+1}[i][j-1] - \mathsf{dp}_{v+1}[i-1][j-1]$ are needed. By Lemma 4.2.1 and Lemma 4.2.2, both differences belong to $\{0, 1\}$, so the whole information required for calculating the correct values consists of $4b+2$ bits. Blocks dp' are already stored in separate machine words, and we can prepare, for every v, an array with the j-th entry set to 1 when $B[j] = v$, partitioned into n/b blocks of length b, where each block is saved in a single machine word, in $\mathcal{O}(|\sigma| \cdot n)$ time. This allows us to gather all the required information in constant time and use a precomputed table of size $\mathcal{O}(2^{4b+2})$ that stores a single machine word encoding the correct values in a block for every possible combination. Additionally, the table stores the number of 1s to the right of the block that should be changed to 0. The table can be prepared in $\mathcal{O}(2^{4b+2} \cdot b)$ time by a straightforward modification of Algorithm 3. Now we can update a whole block in constant time by retrieving the precomputed answer, but then we still might need to remove some 1s on its right. Instead of removing them one-by-one we work block-by-block. In more detail, we maintain a pointer to the nearest block that might contain a 1. Let the number of 1s there be ℓ and the number of 1s that still need to be removed be s. As long as $s > 0$, we remove $\min\{\ell, s\}$ leftmost 1s from the current block in constant time using a precomputed table of size $\mathcal{O}(2^b \cdot b)$, decrease s by $\min\{\ell, s\}$, and move to the next block. This amortises to constant time per block over the row.

We set $b = \frac{\log n}{5}$ as to make the required preprocessing $o(n)$. Then, the overall complexity of the algorithm becomes $\mathcal{O}(|\sigma| \cdot n^2 / \log n)$.

4.3 Second Solution (Rare Elements)

In this section we describe an algorithm for solving LCIS in $\mathcal{O}(\sum_{v=1}^{|\sigma|} (\mathsf{cnt}(v))^2 (1 + \log^2(n/\mathsf{cnt}(v))))$ time.

For every matching pair (x, y), we will compute $LCIS^{\rightarrow}(x, y)$, called the result for (x, y). The algorithm proceeds in phases corresponding to the elements of σ, and in the v-th step computes the results for all v-pairs. During this computation we maintain, for every $r = 1, 2, \ldots, n$, a structure $D(r)$ that allows us to quickly determine, given any (x, y), if there exists an already processed matching pair (x', y') with result r such that $x' < x$ and $y' < y$. Each $D(r)$ is implemented using the following lemma.

Lemma 4.3.1. *We can maintain a set of points $S \subseteq [n] \times [n]$ under inserting a batch of $u \leq n$ points in amortisied $\mathcal{O}(u(1 + \log \frac{n}{u}))$ time and answering a batch of $q \leq n$ queries of the form "given (x, y), is there $(x', y') \in S$ such that $x' < x$ and $y' < y$" in $\mathcal{O}(q(1 + \log \frac{n}{q}))$ time.*

Proof. We first describe a slower solution that achieves the claimed bounds only for $q = 1$, and then extend it to larger values of q. For the latter, we could have also used balanced search trees with dynamic finger property, such as the level linked (2,4)-trees [9]. However, this results in a somewhat complicated solution, and we opt for a self-contained description. We also note that the related question of implementing basic operations on two sets of size n and m, where $m \leq n$, in time $\mathcal{O}(m \log(n/m))$ goes back to the work of Brown and Tarjan [5].

We observe that if the current S contains two distinct points (x_i, y_i) and (x_j, y_j) with $x_i \leq x_j$ and $y_i \leq y_j$ then there is no need to keep (x_j, y_j). Thus, we keep in S only points that are not dominated. Let $(x_1, y_1), \ldots, (x_k, y_k)$ be these points arranged in the increasing order of x coordinates (observe that we cannot have two non-dominated points with the same x coordinate). So, $x_1 < x_2 < \ldots < x_k$, where $k \leq n$, and because the points are not dominated also $y_1 > y_2 > \ldots y_k$. We store the x coordinates in a BST. This clearly allows us to answer a single query (x, y) in $\mathcal{O}(\log n)$ time by locating the predecessor of x. To insert a point (x, y), we first check that it is not dominated by locating the predecessor of x. Then, we might need to remove some of the subsequent x coordinates that correspond to points that are dominated by (x, y). This can be efficiently implemented by maintaining a doubly-linked list of all points, and linking each x coordinate with its corresponding point. Insertion takes $\mathcal{O}(\log n)$ time plus another $\mathcal{O}(\log n)$ for every removed point, so $\mathcal{O}(\log n)$ amortised time, and a query concerning (x, y) reduces to finding the predecessor of x among the x_is in $\mathcal{O}(\log n)$ time.

We first explain how to process a batch of q queries. We first sort them in $\mathcal{O}(q(1 + \log(n/q)))$ time using radix sort with base b. We use a BST that allows split and merge in $\mathcal{O}(\log s)$ time, where s is the number of stored elements, for example AVL trees. Additionally, we store the size of the subtree in every node. Then we have the following easy proposition.

Proposition 4.3.2. *We can split BST into at most b smaller BSTs containing $\Theta(s/b)$ elements each in $\mathcal{O}(b(1 + \log \frac{s}{b}))$ time.*

Proof. As long as there is a BST of size at least $2s/b$ we split it into two BSTs of (roughly) equal sizes. Assuming for simplicity that both s and b are powers of 2, this takes $\mathcal{O}(\sum_{i=0}^{\log b - 1} 2^i \log(s/2^i))$ overall time, which can be bounded by calculating $\int_1^b \log(s/x)dx = \mathcal{O}(b(1 + \log(s/b)))$. □

Then, we split the BST into at most q smaller BSTs containing $\Theta(s/q)$ elements each, where s is the number of stored elements, using Proposition 4.3.2. Because

queries are sorted, we can determine for each of them the relevant BST by a linear scan, and then query the relevant BST in $\mathcal{O}(1 + \log(s/q))$ time, so $\mathcal{O}(q(1 + \log \frac{n}{q}))$ overall.

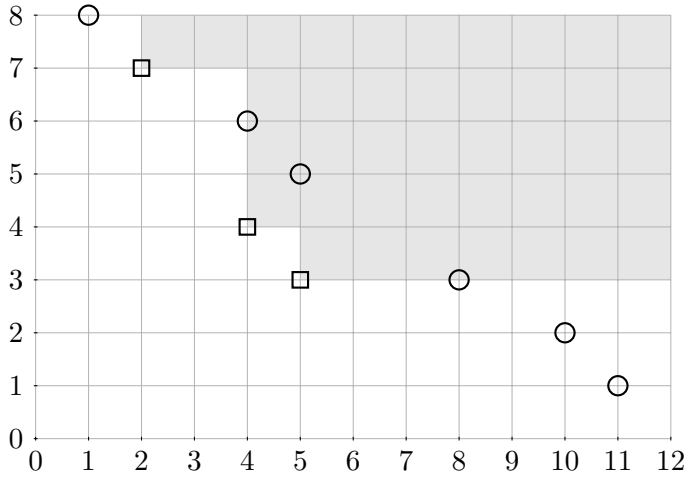


Figure 4.1: Processing a batch of 3 insertions, with the already existing and new points denoted by circles and squares, respectively.

We now explain how to process a batch of u insertions. We start with determining which of the new points are dominated by the already stored points in $\mathcal{O}(u(1 + \log(s/u))$ time using the above method. This also allows us to determine, for each new point (x, y), the range of already stored points $(x_i, y_i), (x_{i+1}, y_{i+1}), \ldots, (x_j, y_j)$ that should be removed from the structure because of inserting (x, y). See Figure 4.1. This takes additional $\mathcal{O}(\ell)$ time by traversing the doubly-linked list, where ℓ is the number of points to be removed. As in a query, we split the BST into at most u smaller BSTs containing $\Theta(s/u)$ elements each, and merge a sorted list of new points with the list of smaller BSTs in $\mathcal{O}(u(1 + \log(s/u)))$ time. Then, each range of the points that should be removed is either fully contained in a single smaller BSTs, or consists of a prefix of a smaller BST, then a range of full smaller BSTs, and finally a suffix of a smaller BSTs. By splitting a smaller BST in $\mathcal{O}(\log(s/q))$ time and assigning a single credit to every stored element, we can hence implement all deletions in $\mathcal{O}(u(1 + \log(s/u)))$ time. Finally, we insert each new point into the appropriate smaller BST. This might take more than $\mathcal{O}(\log(s/u))$ time per element if there are more than s/u insertions to the same smaller BST. In such case, we build an AVL tree containing all these $\ell \geq s/u$ new points in $\mathcal{O}(\ell)$ time, and then insert the $\Theta(s/q)$ already existing points there in $\mathcal{O}(s/q \log \ell) = \mathcal{O}(\ell \log(s/q))$ time, and discard the smaller BST. Finally, we merge the BSTs into pairs, quadruples, and so on. By the calculation from the proof of Proposition 4.3.2 this also takes $\mathcal{O}(u(1 + \log(u/b)))$ time. □

Lemma 4.3.1 is already enough to binary search for the result of (x, y) in $\mathcal{O}(\log^2 n)$ time due to the following property.

Lemma 4.3.3. *Consider any r and an already processed matching pair (x', y') with result r. Then either $r = 1$ or there exists an already processed matching pair (x'', y'') with result $r - 1$ such that $x'' < x'$ and $y'' < y'$.*

Proof. Assume that $r \geq 2$ and consider a sequence C which realises the result for (x', y'). Then $C[1..|C| - 1]$ is an increasing subsequence of both $A[1..(x' - 1)]$ and $B[1..(y' - 1)]$. Let $A[x'']$ and $B[y'']$ be its last elements in A and B, respectively. Then $x'' < x'$, $y'' < y'$, and $A[x''] = B[y'']$, so (x'', y'') is a matching pair, and because C is strictly increasing this matching pair must have been already processed. □

However, our goal is to spend $\mathcal{O}(1 + \log^2(n/\mathsf{cnt}(v)))$ time per every (x, y). We exploit the following property.

Lemma 4.3.4. *Consider two v-pairs (x, y_1) and (x, y_2), where $y_1 < y_2$. The result for (x, y_2) is at least as large as for (x, y_1).*

Proof. Consider a sequence C which realises $LCIS^{\rightarrow}(x, y_1)$. Then, replacing y_1 with y_2 we obtain a valid candidate for the value of $LCIS^{\rightarrow}(x, y_2)$. □

Consider all v-pairs with the same x coordinate $(x, y_1), (x, y_2), \ldots, (x, y_{\mathsf{cnt}(v)})$. We binary search for the result of (x, y_i) for $i = \mathsf{cnt}(v), \ldots, 2, 1$. By Lemma 4.3.4, in the i-th step we can start with the result found in the $(i + 1)$-th step. Using exponential search [4], by convexity of the log function the overall complexity becomes $\mathcal{O}(\mathsf{cnt}(v)(1 + \log(n/\mathsf{cnt}(v))))$. This is still too slow, as every step involves a separate invocation of Lemma 4.3.1 and takes $\mathcal{O}(\log n)$ time. To obtain the final speed up, we process all x coordinates $x_1, x_2, \ldots, x_{\mathsf{cnt}(v)}$ together. The high level idea is to synchronise all exponential searches and exploit the possibility of asking a batch of queries.

We start with modifying the proof of Lemma 4.3.1 to allow for more general queries: given x, we want to find the smallest y such that there exists $(x', y') \in S$ with $x' < x$ and $y' < y$ (or detect that there is none). The modification is straightforward and does not increase the time complexity. Now we can restate processing all pairs with the same x coordinates. We start with a counter c initially set to n and i set to $\mathsf{cnt}(v)$. As long as $i \geq 1$, we use exponential search starting at c to find the result for (x, y_i). Let c' be the found result. We use the modified Lemma 4.3.1 to determine the smallest y such that c' is the result for (x, y) and then keep decreasing i as long as $i \geq 1$ and $y_i > y$. Then, we decrease c' by 1 and repeat.

We further reformulate processing all pairs with the same x coordinate. Consider a conceptual complete binary tree on n leaves (without losing generality, n is a power of 2). Every node corresponds to an interval $[a, b]$, and by querying such a node we will understand querying structure $D(a)$ with the current (x, y_i). Consider the leaf corresponding to c. Calculating c' with exponential search can be phrased as starting at the leaf corresponding to c and going up as long as the query at the current node

fails (we only need to ask a query if the previous node was the right child of the current node; otherwise, we can immediately jump to the nearest ancestor with such property). After having reached the first ancestor for which the query succeeds, we descend from its left child to the leaf corresponding to c' by repeating the following step: if querying the right child of the current node succeeds we descend to the right child, and otherwise we descend to the left child. See Figure 4.2.

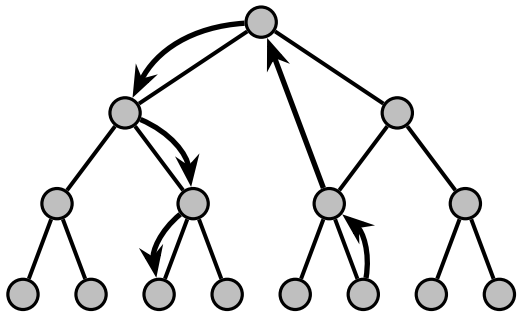


Figure 4.2: Exponential search for the next node phrased as traversing the binary tree.

Now we are able to synchronize the exponential searches as follows. We traverse the conceptual complete binary tree recursively: to traverse the subtree rooted at node u with children u_ℓ and u_r we (i) visit u, (ii) recursively traverse the subtree rooted at u_r, (iii) visit u again, (iv) recursively traverse the subtree rooted at u_ℓ. Thus, every node is visited twice. We claim that when visiting the nodes of the conceptual complete binary tree using this strategy, for any x coordinate we are always able to wait till we encounter the node that should be queried next. This is formalised in the following lemma.

Lemma 4.3.5. *Let the result for (x, y_{i+1}) be c and the result for (x, y_i) be $c' < c$. All queries necessary to calculate c' can be answered during the traversal after the second visit to c and before the second visit to c'.*

Proof. The calculation consists of two phases. First, we need to ascend from the leaf corresponding to c, reaching its first ancestor u at which the query fails. Recall we only need to ask queries if the previous node is the right child of the current node. For each such node v we will be able to use second visit to v in the traversal. Thus, we will process all such queries after the second visit to u. Then, we need to descend from the left child of u. In every step, we query the right child v_r of the current node v, and continue either in the left or in the right subtree of v. To this end, we use the first visit to v_r in the traversal. □

We start calculating time complexity, by noting that we have to traverse the whole tree, which can be done in $\mathcal{O}(n)$. For each x coordinate, by convexity of the log function, we need to query at most $\mathcal{O}(\mathsf{cnt}(v)(1 + \log(n/\mathsf{cnt}(v))))$ nodes of the

conceptual binary tree. Denoting by q_u the number of queries to a node u, we thus have $\sum_u q_u = s = \mathcal{O}(\mathsf{cnt}(v)^2(1 + \log(n/\mathsf{cnt}(v))))$. Invoking Lemma 4.3.1, the total time to answer all these queries is $\sum_u q_u(1+\log(n/q_u))$. By convexity of the function $f(x) = x\log(n/x)$, this is maximised when all q_us are equal, but there are only n of them, making the total time :

$$n + \sum_u q_u(1 + \log(n/q_u)) \leq n + s(1 + \log(n^2/s)) \leq s(1 + \log(n^2/\mathsf{cnt}(v)^2))$$
$$= \mathcal{O}(n + \mathsf{cnt}(v)^2(1 + \log(n/\mathsf{cnt}(v)))^2).$$

For $\mathsf{cnt}(v) < \sqrt{n}$, we invoke the $\mathcal{O}(\mathsf{cnt}(v)^2 \log^2 n)$ solution. As such sum over all elements is at most $\mathcal{O}(n\sqrt{n}\log^2 n)$, which is strongly subquadratic, we can further ignore it. Thus, we obtain for a single element the $\mathcal{O}(\mathsf{cnt}(v)^2(1 + \log(n/\mathsf{cnt}(v)))^2)$ complexity.

4.4 Combining Solutions

Let c be a parameter to be fixed later. We call v *frequent* if $\frac{n}{c} < \mathsf{cnt}(v)$, and *rare* otherwise.

We partition the sequence σ into fragments. Each fragment is either a single frequent element or a maximal range of rare elements. By definition of a frequent element and maximality of fragments consisting of rare elements, we have $\mathcal{O}(c)$ fragments. We maintain the dp_v table as in the first solution, but we only update it after having processed a whole fragment. So, when considering a fragment starting at v we only assume that the values of dp_{v-1} can be accessed in constant time. For a fragment consisting of a single frequent element, we proceed exactly as in the first solution. In the remaining part of the description we describe how to process a fragment consisting of rare elements $v, v+1, \ldots$.

We consider all v'-pairs, for $v' = v, v+1, \ldots$. We will compute $LCIS^{\rightarrow}(x, y)$ for each such matching pair (x, y), and store it in the appropriate structure $D(r)$ implemented as described in Lemma 4.3.1. To compute the values of $LCIS^{\rightarrow}(x, y)$ for all v'-pairs, we use parallel exponential search as in the second solution with the following modification. To check if $LCIS^{\rightarrow}(x, y_i) > r$, we need to consider two possibilities for the corresponding sequence C ending at $A[x] = B[y_i] = v'$:

1. If $C[|C| - 1]$ belongs to the same fragment then it is enough to check if $D(r)$ contains a pair (x', y') with $x' < x$ and $y' < y_i$.
2. Otherwise, it is enough to check if $\mathsf{dp}_{v-1}[x][y_i] \geq r$.

Additionally, after having found c' we need to keep decreasing i as long as $i \geq 1$ and the answer for (x, y_i) is c', and this needs to be tested in constant time per each such

i. We again need to consider two possibilities, and either compare y_i with the value of y' found by querying $D(c'-1)$ with x, or test if $\mathsf{dp}_{v-1}[x][y_i] \geq r$ in constant time. Overall, this incurs only additional constant time per every step of the exponential search for every considered matching pair.

After having considered all v'-pairs for the last element v' in the current fragment, we need to compute $\mathsf{dp}_{v'}$ from dp_{v-1} and the calculated values of $LCIS^{\rightarrow}$. Of course, we want to operate on $\mathsf{dp}'_{v'}$ and dp'_{v-1} instead of $\mathsf{dp}_{v'}$ and dp_{v-1}. This is done row-by-row. The i-th row is computed in two steps.

First, we need to set $\mathsf{dp}_{v'}[i][j] = \max\{\mathsf{dp}_{v'}[i-1][j], \mathsf{dp}_{v-1}[i][j]\}$ for every $j = 1, 2, \ldots, n$. This is done by processing whole blocks in constant time and precomputing the result for every possible combination of the following information:

1. $\mathsf{dp}'_{v'}[i-1][j], \mathsf{dp}'_{v'}[i-1][j+1], \ldots, \mathsf{dp}'_{v'}[i-1][j+b-1]$,
2. $\mathsf{dp}'_{v-1}[i][j], \mathsf{dp}'_{v-1}[i][j+1], \ldots, \mathsf{dp}'_{v-1}[i][j+b-1]$,
3. $\mathsf{dp}_{v'}[i-1][j-1]$,
4. $\mathsf{dp}_{v-1}[i][j-1]$.

This can be preprocessed in $\mathcal{O}(4^b \cdot b^2)$ time after observing that, as in the first solution, only the difference $\mathsf{dp}_{v'}[i-1][j-1] - \mathsf{dp}_{v-1}[i][j-1]$ is relevant and, additionally, it can be capped at b (if it is bigger than b then we can set it to b). The time is $\mathcal{O}(n/b)$.

Second, we need to consider the values of $LCIS^{\rightarrow}(i,j)$ computed for the current fragment. If the result computed for a matching pair (i,j) is r then we need to update $\mathsf{dp}_v[i][j'] = \max\{\mathsf{dp}_v[i][j'], r\}$, for every $j' \geq j$. This can be done by simultaneously scanning all such js and the blocks. By maintaining the maximum r, we can update the value of $\mathsf{dp}_v[i][j]$ at the beginning of the block. Then, we consider all other j's belonging to the same block, and consider its corresponding result r'. If $\mathsf{dp}_v[i][j'] \geq r'$ then this result is irrelevant, and otherwise we must increase some of the values in the block by 1 (as $\mathsf{dp}_v[i][j'-1]$ is assumed to have been already updated and due to Lemma 4.2.1). As in the first solution, this is implemented by setting $\mathsf{dp}'_v[i][j'] = 1$ and changing the nearest 1 into 0. Overall, the time is bounded by the number of considered matching pairs plus additional $\mathcal{O}(n/b)$ time.

We set $b = \frac{\log n}{5}$ so that the preprocessing time is $o(n)$. For each frequent element we spend $\mathcal{O}(n^2/b)$ time, so $\mathcal{O}(n^2/b \cdot c)$ overall. For each fragment consisting of rare elements, the time is $\mathcal{O}(\mathsf{cnt}(v)^2 \log^2(n/\mathsf{cnt}(v)))$ for every v to compute the results, and then $\mathcal{O}(n^2/b)$ plus the number of results. Using $\mathsf{cnt}(v) \leq n/c$, where c is sufficiently large, and calculating the derivative of $f(x) = x \log^2(n/x)$ we upper bound $\mathsf{cnt}(v) \log^2(n/\mathsf{cnt}(v)) \leq n/c \cdot \log^2 c$ for every rare v, so the overall time is $\mathcal{O}(n^2/b \cdot c + n/c \cdot \log^2 c \sum_v \mathsf{cnt}(v)) = \mathcal{O}(n^2/b \cdot c + n^2/c \cdot \log^2 c)$.

Choosing $c = \sqrt{\log n} \log\log n$ we obtain an algorithm working in $\mathcal{O}(n^2 \log\log n / \sqrt{\log n})$ time.

Chapter 5

Improving the First Solution

5.1 Redefining the Blocks in Dynamic Programming

Definition 5.1.1 (Special entry)**.** We call an entry (x, y) special for dp_v if $\mathsf{dp}_v[x][y] > \max\{\mathsf{dp}_v[x-1][y], \mathsf{dp}_v[x][y-1]\}$.

Lemma 5.1.2 (Significant pairs)**.** *If there exist v such that (x, y) is special for dp_v, then (x, y) is a significant pair.*

Proof. The value of an entry $\mathsf{dp}_v[x][y]$ is equal to the maximum of previous states or $LCIS^{\rightarrow}(x, y)$ if $A[x] = B[y] \leq v$. Since $\mathsf{dp}_v[x][y]$ is larger than previous states, the (x, y) is a v'-pair for $v' \leq v$. Additionally, $LCIS^{\rightarrow}(x, y) = \mathsf{dp}_v[x][y]$.

Fix any other v'-pair (x', y') which fulfills $x' \leq x, y' \leq y$. Since $v' \leq v$, we know that $LCIS^{\rightarrow}(x', y') \leq \mathsf{dp}_v[x'][y'] < \mathsf{dp}_v[x][y] = LCIS^{\rightarrow}(x, y)$. From Definition 2.5.1 we conclude that (x, y) is a significant pair. □

Fix $b = \log^{2/3} n / \log\log n$. Divide the dp table into blocks of size $b \times b$ as shown in Figure 5.1 (here $b = 2$).

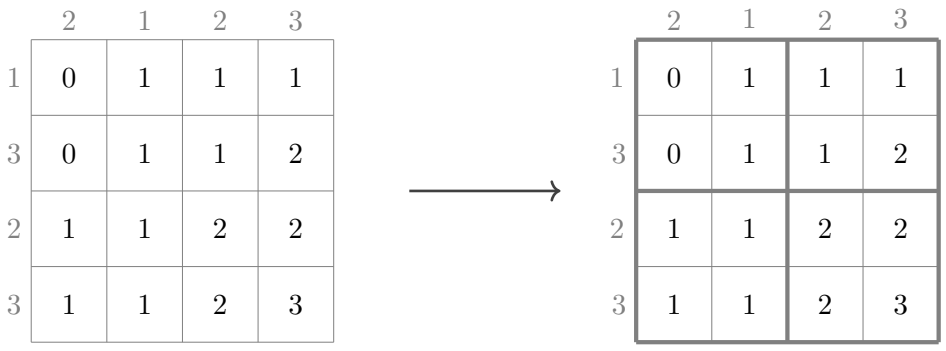


Figure 5.1: The division of dp into blocks

Fix a block with the left upper corner in entry (i, j). The information we want to keep in each block is enumerated below:

1. $\mathsf{dp}_v[i-1][j-1]$
2. $\mathsf{dp}'_v[i-1][k]$ for $k \in \{j, j+1, \ldots, j+b-1\}$
3. $\mathsf{dp}'_v[k][j-1]$ for $k \in \{i, i+1, \ldots, i+b-1\}$
4. For each special entry (k, l), memorize this entry. Notice that we need just $2 \log b \leq 2 \log \log n$ bits to do so, as it is enough to keep $k - i$ and $l - j$.

Here, the references to dp' are to its value. We do not keep dp' in the optimised solution.

Notice, that these information are enough to restore the value of each entry inside the block. Additionally, the information from points $2, 3, 4$ can be kept in $\mathcal{O}(b + s \log b)$ bits, where s denotes the number of special entries inside this block. Thus, we can keep the dp_v table using just $\mathcal{O}(\frac{n^2}{b} + S \log b)$ bits, where S denotes the number of special entries in the whole table. From Lemma 2.5.2 we bound S by $\mathcal{O}(n^2/\log^{1/3} n)$, thus number of required bits is $\mathcal{O}(n^2 \log \log n/\log^{1/3} n)$.

5.2 Cutting Blocks

As in the previous solution, we want at some point to show how to update such blocks quickly. Unfortunately, in the previous solution those blocks had fixed size as the amount of information kept there was constant, and that does not hold on anymore. Here, the size of the block depends on s which can be up to $b^2 = \log^{4/3} n/(\log \log n)^2$.

In this section we show how to ensure that size of these blocks is smaller than $\log n$. In particular, we keep the number of special entries s strictly less than $M = \frac{1}{5} \log n/\log \log n$.

The idea is to cut blocks which contain too many ones. All blocks are greedily cut only in the second dimension, thus their first dimension remains b. Figure 5.2 presents how to cut blocks (special entries are marked with s).

Thus, we distinguish two types of blocks. The initial blocks, are the blocks in the first partition of dp (the ones of size $b \times b$). The actual blocks, are the blocks into which initial blocks are cut. Each actual block is contained by some initial block.

We start with calculating how many actual blocks we can have. For two neighbouring actual blocks we notice, that either one of them is an initial block, or they contain at least M special entries. Thus, the number of actual blocks is bounded by $n^2/b^2 + 2 \cdot n^2/(M \cdot \log^{1/3} n) = \mathcal{O}(n^2(\log \log n)^2/\log^{4/3} n)$.

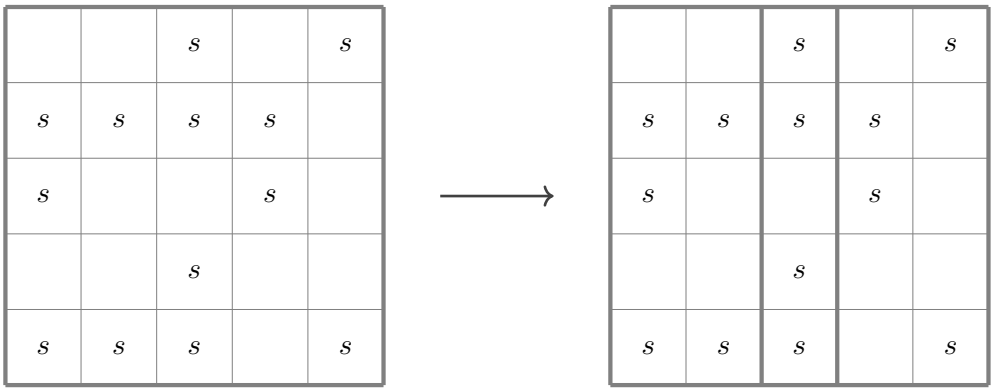


Figure 5.2: How to cut blocks for $M = 6$.

This construction keeps dp in the, up to constant multiplication, same number of blocks, such that information describing each block can be stored in at most $\frac{2}{5}\log n + o(\log n)$ bits. The remaining part is querying dp in constant time. For each initial block, we additionally keep the bitmask which memorises in which column, the new actual block start. With such a bitmask it is easy to restore, which block we want to query. Additionally, since the information describing each block is smaller than $\log n$, we can preprocess for each entry within a block its value, thus we obtain constant query time.

5.3 Updating the Blocks

There are two types of updates we make. The first one is when we encounter a frequent element, and we need to update dp with all its matching pairs. The second one is when we update dp with all results from rare elements.

We start with describing the first type of updates. The solution is roughly the same as previously. We gather the following information (the block has the left, upper corner in (i, j)):

1. The current block mask
2. A mask from the block to the left
3. dp' values from the previous row in the corresponding column
4. $\mathsf{dp}_{v+1}[i-1][j-1] - \mathsf{dp}_v[i-1][j-1]$
5. For which elements $A[k] = v$, where $k \in \{i, i+1, \ldots, i+b-1\}$
6. For which elements $B[k] = v$, where k iterates over columns which have a common part with current block (at most b such)

Such information can be encoded using $\frac{4}{5}\log n + o(\log n)$ bits, so we can run preprocessing in $o(n)$, and then in constant time update this block. The problem raises when after such update the number of special entries exceeds M. The solution is that, the preprocessing returns the list of newly created blocks, so the update can be bounded by $\mathcal{O}(1 + \texttt{\#new blocks})$. Since the number of blocks is bounded separetely, the first type of update can be done in $\mathcal{O}(n^2(\log\log n)^2/\log^{4/3} n)$.

Similarly, we deal with the second kind of update. For each block, we produce a list of matching pairs to update within it, sorted by coordinates (in order by the second one, and the first one). This can be done in $\mathcal{O}(\texttt{\#number of matching pairs} + \texttt{\#number of blocks})$ using bucket sort. Then, we iterate over blocks, and for each block we start with updating it with values from the previous block to the left, and the values from $\texttt{dp}'$ from the row just above the current block. All of these information can be gathered in constant time, and has sum of at most $o(\log n)$. Then, we iterate over matching pairs, and update dp with its result using preprocessing. If at some point, the number of special entries exceeds M, we deal with them in the same way, as in the first type of updates. Because the matching pairs are sorted by the column first, it is easy to allocate these to respective block in constant time. This update also works in $\mathcal{O}(n^2(\log\log n)^2/\log^{4/3} n + \texttt{\#number of matching pairs})$.

5.4 Time Complexity

We repeat the same analysis of time complexity, as in the previous algorithm, but this time we pick $c = \log^{2/3} n$. For rare elements we have the same complexity as previously $\mathcal{O}(n^2/c\log^2 c)$. For frequent elements, we obtain a faster update time $\mathcal{O}(c \cdot n^2(\log\log n)^2/\log^{4/3} n)$. Summing these up, and inserting the value of c, we obtain the $\mathcal{O}(n^2(\log\log n)^2/\log^{2/3} n)$ complexity.

Chapter 6

Experiments

In this chapter, we compare the implemented solutions on the chosen tests. All of these experiments were conducted on a laptop with a processor `Intel Core i7-10750H CPU @ 2.60GHz`.

6.1 Solutions

Three solutions for LCIS problem were implemented and compared:

1. The $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ space solution, described in Section 2.4, implemented in file `sol1.cpp`
2. The $\mathcal{O}(n^2)$ time and space solution, described in Section 2.3, implemented in file `sol3.cpp`
3. The $\mathcal{O}(n^2 \log\log n/\sqrt{\log n})$ time solution, described in Chapter 4, implemented in file `sol4.cpp`

Additionally, to check correctness, the $\mathcal{O}(n^4)$ time solution was implemented in file `sol2.cpp`, along with generator `stress-gen.cpp`, and testing script `stress-test.sh`.

6.2 Testing

There are two types of tests:

1. Randomly generated tests, two sequences of the same length n with elements from $[1, C]$, for given n, C.
2. Sequences described in [6], which have a lot of significant pairs.

There are 9 tests in total. The exact parameters of these tests are given beneath:

1. `random1a.in` – Random test with $n = 20\,000$, $C = 2$
2. `random1b.in` – Random test with $n = 20\,000$, $C = 3$
3. `random1c.in` – Random test with $n = 20\,000$, $C = 5$
4. `random1d.in` – Random test with $n = 20\,000$, $C = 10$
5. `random1e.in` – Random test with $n = 20\,000$, $C = 25$
6. `random1f.in` – Random test with $n = 20\,000$, $C = 100$
7. `random1g.in` – Random test with $n = 20\,000$, $C = 2\,500$
8. `special1.in` – The second type of test, $n = 7\,424$
9. `special2.in` – The second type of test, $n = 16\,384$

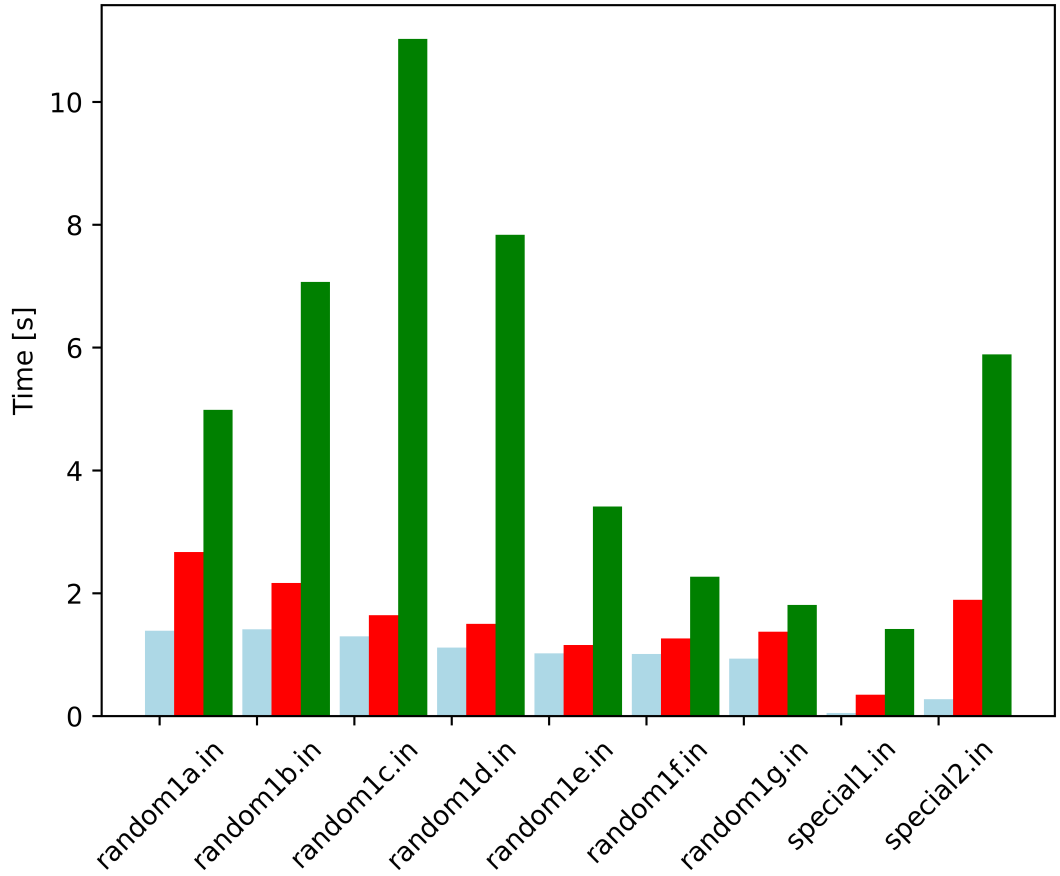
The exact implementation is given in `gen.cpp`, and uses pseudorandom number generator from `testlib.h`, implemented by Codeforces.

For testing purposes, two scripts were implemented `avgtime.sh`, `check.sh`. The first one computes an average time of n runs of a given solution on a given test, and the second one produces a log file with these times. Eash test was run $n = 10$ times. To produce all log files, one has to type command `make test && make run`.

6.3 Results

The following chart presents the comparison of these solution on the given set. The blue, red, and green colors represent `sol1.cpp`, `sol3.cpp`, and `sol4.cpp` respectively.

Without a doubt, the best solution is the `sol1.cpp`. Not much worse is the `sol3.cpp`. Except for the `special2.in` and `random1a.in`, its performance is almost the same. The slowest solution is `sol4.cpp`, even though it achieves theoretically the best bounds. The gap between this solution and the previous is pretty large, especially on the test `random1c.in` (10 seconds to 2 seconds).


Time [s]
0
2
4
6
8
10
random1a.in
random1b.in
random1c.in
random1d.in
random1e.in
random1f.in
random1g.in
special1.in
special2.in

Bibliography

[1] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *56th FOCS*, pages 59–78, 2015.

[2] Amir Abboud and Karl Bringmann. Tighter connections between formula-SAT and shaving logs. In *45th ICALP*, pages 8:1–8:18, 2018.

[3] Anadi Agrawal and Pawel Gawrychowski. A faster subquadratic algorithm for the longest common increasing subsequence problem. In *31st ISAAC*, LIPIcs, pages 4:1–4:12, 2020.

[4] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–87, 1976.

[5] Mark R. Brown and Robert Endre Tarjan. A fast merging algorithm. *J. ACM*, 26(2):211–226, 1979.

[6] Lech Duraj. A sub-quadratic algorithm for the longest common increasing subsequence problem. In *37th STACS*, pages 41:1–41:18, 2020.

[7] Lech Duraj, Marvin Künnemann, and Adam Polak. Tight conditional lower bounds for longest common increasing subsequence. *Algorithmica*, 81(10):3968–3992, 2019.

[8] Szymon Grabowski. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discret. Appl. Math.*, 212:96–103, 2016.

[9] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[10] William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

[11] Yoshifumi Sakai. A linear space algorithm for computing a longest common increasing subsequence. *Inf. Process. Lett.*, 99(5):203–207, 2006.

[12] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[13] I-Hsuan Yang, Chien-Pin Huang, and Kun-Mao Chao. A fast algorithm for computing a longest common increasing subsequence. *Inf. Process. Lett.*, 93(5):249–253, 2005.