Algoritmer og Datastrukturer opg 4

chs380, zts900

09-03-2025

---

1 Dynamic programming

In our function we loop thorugh indexes i=1 upto i=j-1 in A. Each itteration we check the condition $A[j] \not< A[i]$ same as $A[j] \geq A[i]$. r is a list of longest observed subsequences upto each A[j], so the invariant for the outer loop is that at the start of the loop r contains all the longest non decreasing subsequences for each j upto j-1. We loop through all j and make r a list of all j subsequences lenghts minium 1 for each j. We return the max of this list, meaning the longest non decreasing subsequence in A.

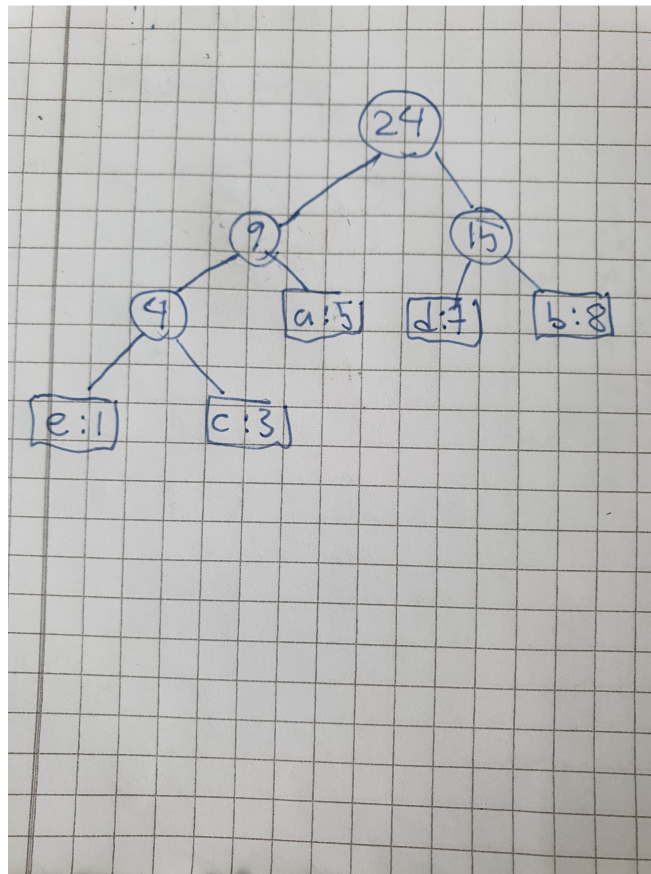
1. False we return the max of a list which is a number not a list of indexes
2. False, same as 1
3. False, the condition we check is $A[j] \not< A[i]$same as $A[j] \geq A[i]$. Which indicates non decreasing seuqnce
4. True, we loop throgh each index from 2 to j in A, we record the longest non decreasing subsequence up to each j, and record it in r. We then take the max from r, returning the longest non decreasing subseqeunece in r.

---

2 Huffman codes

We have an alphabet with frequences for each letter: $a : 5, b : 8, c : 3, d : 7, e : 1$. We draw the following Huffman tree.

Figure 1:



3 More dynamic programming

a)

We use induction on $\ell = j - i$. This means that we consider the smallest possible difference between i and j, which occurs when they are equal, $i = j$, meaning $\ell = j - i = 0$.
We know that if $j = i$, then $t_{i,j} = 1$
We want to show that

$$t_{i,j} \geq 2^{j-i+1} - 1.$$

Base Case

Inserting $j - i = 0$ in the right side of the inequality, we get:

$$2^{0+1} - 1 = 2^1 - 1 = 1.$$

Since $t_{i,i} = 1$, we see that $1 \geq 1$, proving that the base case holds.

Inductive Hypothesis

We assume that the statement holds for a given value $\ell = m$ and show that it also holds for $\ell = m + 1$.
This means that, we assume that the inequality already holds for all intervals up to length m:

$$t_{i,j} \geq 2^{j-i+1} - 1, \quad \forall j - i \leq m.$$

Inductive Step

We now need to show that it also holds for $\ell = m + 1$, meaning for $t_{i,j}$ where $j - i = m + 1$.
From the recurrence relation, we know that:

$$t_{i,j} = 1 + \sum_{k=i}^{j-1} (t_{i,k} + t_{k+1,j}).$$

And from our inductive hypothesis, we know:

$$t_{i,k} \geq 2^{k-i+1} - 1, \quad t_{k+1,j} \geq 2^{j-k} - 1.$$

We can therefore estimate the sum as:

$$\sum_{k=i}^{j-1} (t_{i,k} + t_{k+1,j}) \geq \sum_{k=i}^{j-1} \left((2^{k-i+1} - 1) + (2^{j-k} - 1) \right).$$

We can rewrite it as:

$$\sum_{k=i}^{j-1} 2^{k-i+1} + \sum_{k=i}^{j-1} 2^{j-k} - 2(j-i).$$

Using the sum formulas we have that:

$$\sum_{k=i}^{j-1} 2^{k-i+1} = 2^{m+1} - 2, \quad \sum_{k=i}^{j-1} 2^{j-k} = 2^{m+1} - 2.$$

Therefore, we obtain:

$$t_{i,j} \geq 1 + (2^{m+1} - 2) + (2^{m+1} - 2) - 2(m+1).$$

Which can be simplified as:

$$t_{i,j} \geq 1 + 2^{m+1} + 2^{m+1} - 4 - 2(m+1).$$

For sufficiently large m, we get:

$$t_{i,j} \geq 2^{m+1} - 1 = 2^{(j-i+1)} - 1.$$

This proves that the statement also holds for $j - i = m + 1$.
We have therefore shown,by induction,that:

$$t_{i,j} \geq 2^{j-i+1} - 1, \quad \forall i, j \text{ where } i \leq j.$$

b)

In the problem we can see that $m_{i,j}$ is defined recursively. But if we handle it with a naive recursive approach, it would repeatedly compute the same values, leading to exponential runtime.
By using dynamic programming, we can store previously computed values in a table and significantly reduce the computation time.

We will fill a 2D table $dp[i][j]$, where $dp[i][j]$ represents $m_{i,j}$.
According to the given recurrence formula:

$$m_{i,j} = \begin{cases} 1, & \text{if } i = j, \\ \max\{m_{i,k} + m_{k+1,j} + x_{i-1}x_kx_j \mid i \leq k < j\}, & \text{if } i < j. \end{cases}$$

Pseudocode

```
Algorithm DP-Compute-M:
Input:  n, and the array x[0..n].
Output:  M[i][j] = m_{i,j} for all 1 <= i <= j <= n.

1) Initialize an nÃn array M, indexed by (i,j).

2) For i from 1 to n:
```

```
        M[i][i] = 1

3) For length l from 2 to n:                    // subproblem sizes
        for i from 1 to n - l + 1:
            j = i + l - 1
            // Compute M[i,j] using the recurrence
            best_val = -â
            for k from i to j-1:
                cost = M[i][k] + M[k+1][j] + x[i-1] * x[k] * x[j]
                if cost > best_val:
                    best_val = cost
            M[i][j] = best_val

4) // At the end, M[1][n] holds m_{1,n}, but we have M[i][j] for all i,j
```

We observe that the table dp has dimensions $(n+1) \times (n+1)$, so the space complexity is:

$$O(n^2)$$

Which is because we store all subproblem solutions in a two-dimensional table.

We observe that the algorithm contains three loops:

- Outer loop: Iterates over interval lengths $O(n)$.
- Middle loop: Iterates over starting points $O(n)$.
- Inner loop: Iterates over possible split points $O(n)$.

The total time complexity is therefore

$$O(n) \times O(n) \times O(n) = O(n^3)$$

This is significantly faster than the naive recursive approach, which had exponential time complexity.

We can therefore conclude that by storing previously computed results, we avoid redundant calculations and achieve polynomial time complexity $O(n^3)$, making dynamic programming much more efficient than a naive recursive approach.

---

4 Greedy algorithms

a)

For a solution to the MaxProduct problem i.e a a max sequence over some elements $x_1 ... x_n \in R$ which we call I^*, we have a sub-set of all our negative x_i. $S^* = \{i \in I^* \mid s_i < 0\}$. We know that this set contains an even number of elements s_i since an optimal solution always will contain a number

of negative elements such that their negative signs cancel out, i.e if we factor out $(-1)^k(s_1 \cdot s_2...s_k)$ from all s_i, we will have for $k : even$ that $(-1)^k(s_1 \cdot s_2...s_k) = (s_1 \cdot s_2...s_k)$ and for $k : odd$ that $-(s_1 \cdot s_2...s_k)$. Hence an optimal solution maximising the product will always have $k : even$.

b)

Constraints for optimal solution

I^*will include the following elements

1. Possitive: always all $x_i \geq 1$
2. Negative: anlways an even nr of negative elements or no negative elements.
3. Negative: maximum one negative element in rage $-1 \leq 0$, (in the case where having a pair f.ex $-4 \cdot -0.5 > 1$ where -4 is the only negative less than -1)
4. Empty set $\{\} = 1$ if all elements are between 1 and -1 i.e $\mid x_i \mid \in [0, 1]$ or no elements

Proposed algorithm

The main idea is to (1) find approximately the limits a and b that contain the range $x_a \leq -1 < 1 \leq x_b$we want to exclude, we search for them with binary search but save information about closest larger element and index to b and closest smaller to a, this takes logn time, (2) then adjust this approximate solution with a finite set of operations in $O(1)$ time. Leaving us with a very fast algorithm $O(lgn)$.

PseudoCode

- Check that input is not empty array, if so return empty set.
- Finding the limits to the approximate range to exclude:
  - Find with binary search index b and elementx_b, st x_b is $min\{x_b \geq 1\}$
  - Find with binary search index a and element x_a, st x_a is $max\{x_a \leq -1\}$
- Adjusting the a limit with finite operations to guarantee optimal solution:
  - if $a = 1$ // we have only one negative smaller or equal to -1.
    - if $x_1 \cdot x_2 > 1$ // is the right element (i=2) also negative?
      - $a = a + 1 = 2$ // move pointer to include x_2
    - else
      - $a = 0$ // move pointer to 0 exclude the one negative
  - if a odd && $a > 1$: // we have odd nr elems smaller or equal to -1
    - $a = a - 1$ //move pointer left (exclude largest element, so even negative nrs)
  - if a even: // even nr negative nrs less than or euqal to -1
    - keep a in place

- Return indexes for largest product
  - if $x_a > -1 \& x_b < 1$: // all nrs in range $\mid x_i \mid \in [0, 1]$
    - reutrn $I^* = \{\}$ // empty product is 1.
  - else:
    - return $I^* = \{1, ..., a, b, ..., n\}$// we dont actually return this whole set of nrs we just print the relevant ranges.

Runtime

We have two binary searches each in $O(lgn)$ time, and refining limit operations all doing simple operations f.ex moving a pointer or doing if comparisons this takes $O(1)$ time. The overall runtime is hence $O(lgn)$ time.

Correctness

We have an optimal solution

- $I^* = max \{\{1, ..., a, b, ..., n\}, \{2, b, ..., n\}, \{0, b, ..., n\}, \{1, ..., (a-1), b, ..., n\}, \{\}\}$
- For any set of indexes including our proposed set: $I^* \geq max \{\{1, ..., a, b, ..., n\}, \{2, b, ..., n\}, \{0, b, ..., n\}, \{1, ..., (a-1)$
- Also for our proposed set if our set of solutions contain all possible outcomes $I^* \leq max \{\{1, ..., a, b, ..., n\}, \{2, b, ..., n\},$
- From this follows: $I^* = max \{\{1, ..., a, b, ..., n\}, \{2, b, ..., n\}, \{0, b, ..., n\}, \{1, ..., (a-1), b, ..., n\}, \{\}\}$

To show that our proposed set contains all outcomes observe that the following structure is fitted into the solutions. I will also argue that these cases make sure we find en optimal solution under constraints (1), (2), (3), (4).

Possitives in b to n:

- 2 ways:
  - we have possitive nrs range b to n (b to n not empty) (1)
  - we dont have possitive nrs range b to n (b to n empty) (1)

Negatives in 0 to a:

- More than one negatives range 0 to a
  - We have an even nr negatives in range 0 to a. (0 to a not empty) (2)
  - We have an odd nr negatives in range 0 to a (we adjust a-1, making it an even nr) (2)
- One negative in range 0 to a
  - We have one odd negative smaller than 1 in range 0 to a, and including it gives higher product (we adjust a=2, so we include it) (3)
  - We have one odd negative smaller than 1 in range 0 to a, and including it gives smaller product (we adjust a=0, no negatives included) (3)

- No negatives
  - Range 0 to a is empty

No nrs outside range a to b

- Empty set gives 1 which is larger, we return the empty set (4)

No nrs

- We return empty set (4)

Our cases covers all outcomes and make sure (1),(2),(3),(4) is fullfilled.