FUNCTIONAL DOCUMENTATION

ADVANCED ALGORITHMS

ATHMAR NABIL SHAMHAN – SEYDA KOCLAR
PROJECT GROUP 1

Table of Contents

1. Problem Description

The problem that is going to be solved in this project is about generating a sequence of boxes using dynamic programming. More specifically, we are given N number of 2 dimensional boxes including the width and length of every single one of them and are expected to find the largest set of boxes such that they can be put one into another.

To make the problem clearer:

Given:

- N: Number of 2-D rectangular boxes.
- Dimension of each box: (W,L).

Goal:

- Create a sequence of boxes as large as possible.

Constraints:

- One box can be inside the other only if the lower box has strictly higher dimensions. For example:

  Box1 = (W_1, L_1) and Box2 = (W_2, L_2)

  Box2 can be put into Box1 only if $W_1 > W_2$ and $L_1 > L_2$

- The boxes can only be rotated horizontally or vertically, meaning that there will be no box put into another one diagonally. Since each box has 2 unique instances, the rotation can only be 90 degrees because 180 degrees will give us the same instance:

  (W, L) -> (L, W) (90 degrees rotated)

  (L, W) -> (W, L) (90 degrees rotated)

- None of the boxes contain more than one box inside even though two or more boxes could fit into some of them.
- Multiple instances of boxes are allowed.

2. Solution Description

To solve the problem with dynamic programming, the problem will be converted to a one of the most popular problems called Longest Increasing Subsequence (LIS). LIS is a very similar problem to ours but with only couple of differences. Let us first describe how we are going to make our problem look like LIS then move onto explain the details of this conversion and LIS itself. **Note that, it is assumed that starting indexes of all the arrays in the document is 1 not 0, for simplicity.**

2.1 Converting The Problem

In the problem we are asked to find longest sequence and this sequence must be in decreasing order since the boxes will be put into one in another. The idea is very similar to LIS, but in LIS it is important that the sequence preserves the positions of the elements in correct order. In our case the order of the boxes is not important. Thus, to be able to use LIS we need to do 2 operations at first:

- Generate an array of N elements containing the dimensions of boxes including rotated ones with the following rule for box i where i$\in \{1,2,3, \dots, N\}$:
    - The first coordinate will be considered as width and it will take the max of (W_i, L_i)
    - The second will be considered as length and it will take the min of (W_i, L_i)

  By this way we generate our array using the necessary rotated versions of the boxes and avoid using unnecessary space.

- Sort our array in decreasing order with merge sort. The process is like sorting the boxes and putting them in an order so that they can be chosen in preserved order as well, just like a sequence in LIS. Sorting will be according to first coordinate, which stands for width, i.e., maximum of the dimensions. By doing this we are making sure that whichever sequence of boxes we choose, the widths will always preserve the decreasing order, hence, the problem becomes

only applying LIS according to second coordinate, which is length, i.e., minimum of the dimensions.

2.2 Merge Sort

Merge sort is a sorting algorithm that applies divide and conquer paradigm. Let us show the steps of the algorithm for an array of N elements.

Divide:

Divide the N-element array to be sorted into two subarrays of N/2 elements each.

Conquer:

Sort these two subarrays with using recursive calls of merge sort.

Combine:

Merge the two sorted subarrays to produce the sorted list.

2.3 Longest Increasing Subsequence

Longest increasing subsequence is a problem of finding the largest subsequence of given sequence such that all the elements of the subsequence will be sorted in increasing order.

For example, if the given sequence is: [10, 22, 9, 33, 21, 50, 41, 60, 80] then the length of LIS will be 6 and the sequence will be: [10, 22, 33, 50, 60, 80].

In this project, we choose LIS computing its solution using binary search and we are going to make it as Longest Decreasing Subsequence, which will be denoted as LDC from now on. For the algorithmic explanation of LDS, we will only explain the solution of $\theta(NlogN)$ itself. However, to understand this solution, one should first understand what patience game is.

Patience game is the game also called Solitaire. In this game the purpose is putting the cards to the piles considering the number on the given card must be less than the number on the last card in one of the piles. Only this notion of the game is enough to understand our algorithm. In the case of our problem, we will use the rule that cards in the piles should be in

increasing order. Let us explain how we are going to use this rule in our algorithm with an example.

Say that we have an array [3,10,7,9,8]. These numbers represent our cards individually. We have an empty pile we call Z. For the first card since we have no card in the pile we will put it in Z[1]. Now we have 10 in our hands, and we need to find the place for it. Since 10 > 3 we will put it in Z[1] it is like we are putting our card 10 on top of card 3 but every time we only need to compare the last card of the pile. We do not store all the pile, instead we are taking only the last card which was put in it. It is time for putting 7 and since 7 is not greater than 10 we need to open a new pile which is Z[2] and put 7 here. For 9 we have two cards in piles to look: 10 and 7. Since it is less than 10 and greater than 7 we will put it on top of 7, hence, Z[2] will be 9. Now we have 10 and 9 as the last cards of the piles, our last card 8 is less than both. Therefore, we will open a pile Z[3] and put 8 in here. Realize that we have 3 elements in Z and this length shows the length of the longest decreasing subsequence, Z itself does not store the LDS but with the help of another 2 arrays we can backtrack and find the subsequence itself. Below there is a figure shows these steps:

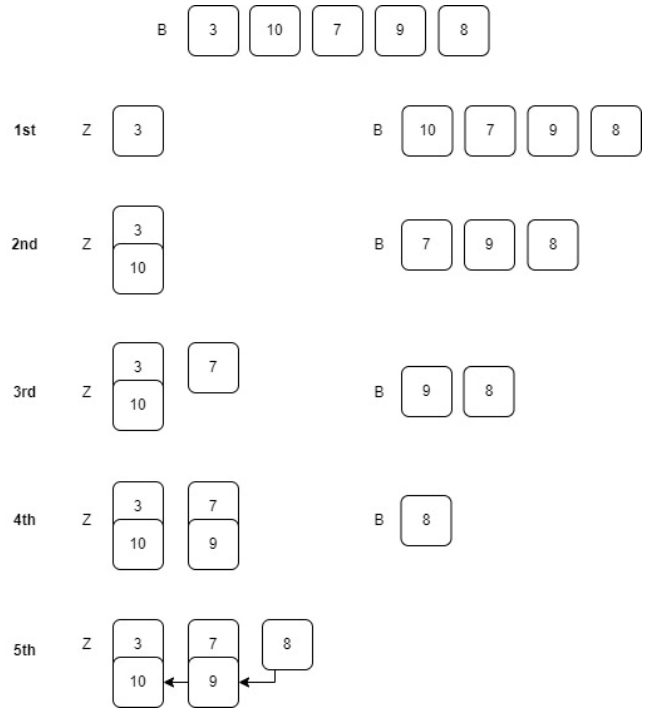

Figure 1: The figure shows the patience sort algorithm

In this figure Z is shown as piles but we have array, and we are not taking the pile but only the last values in it. Moreover, our increasing subsequence resides in this. We can backtrack and get the sequsubence 10,9,8. Luckily, we get the sequence itself in Z but if we have 11 after 8, we will put it in the first pile and realize that Z would not contain the sequence, however, we will still find the sequence, but we need another array to be able to track. Note that the pile k shows the end values of the decreasing

subsequences of length k up considering the sequence until the i-th element.

Definitions (for each 1 ≤ i ≤ N):

B – The array containing lengths of the boxes which is in form $[b_1, b_2, \ldots, b_N]$.

$\boldsymbol{B_i}$ **–** The optimal decreasing subsequence of the sequence $[b_1, b_2, \ldots, b_i]$ ends with b_i of any length, the length could be the values between 1 to i.

Z[k] – The largest last element of LDS of length k found so far or -1 if no LDS of length k was found.

I[k] – Index of the element in Z[k] or empty if Z[k] = -1

P[i] – Index of prelast element of the best LDS (of $[b_1, b_2, \ldots, b_i]$) ending in b_i and -1 if there is no prelast element for the i-th element.

Now let us give an example for better understanding. Say that we are given B = [3,10,7,9,11,8]. For this sequence, $B_i\ values\ will\ be$: B_1 = [3], B_2 = [10], B_3 = [10,7], B_4 = [10,9], B_5 = [11], B_6 = [10,9,8]. The solution will be one of these sequences which is B_6 in this example.

For this sequence, the values in Z will be Z[1] = 11 (the largest possible value for the subsequence of length 1), Z[2] = 9 , Z[3] = 8 and Z[4] = Z[5] = Z[6] = -1. We see that Z[k] contains the largest last element of LDS of length k as we have described above and if there is no such element then it contains -1.

For I[k] values, we will have I[1] = 5 since for the subsequence of length 1 we have Z[1] = 11 at the end and its index is 5. I[2] = 4 since for the subsequence of length 2 we have the sequence [10, 9] as optimal and the last element is 9 which has index 4. I[3] = 6 which shows element 8 is the last value of subsequence of length 3 as it is expected.

For P[i] values, we will have P[1] = -1 since there is no parent for the first element of the subsequence. P[2] = -1 since 10 becomes first element of the subsequence now. P[3] = 2 since 7 is added to the end of 10 which makes 10 as prelast element hence its index is put into P[3]. P[4] = 2 since 9 is greater than 7 and we chose to put 9 and remove 7 at the end of the

subsequence now 10 is prelast for 9. P[5] = -1 since 11 becomes a new subsequence of length 1 which has no prelast. P[6] = 4 since 8 is less than 9 and we added it at the end of the subsequence [10,9] which makes 9 as prelast of 8.

We saw the definitions and an example to make them solid. One can say that in our algorithm, we are basically trying to find the best place for element b_i in each iteration of i. We can see that one of the most important properties of Z is that it is constructed in the algorithm. Since it is a sorted array, for finding the best place for element b_i, we can conduct binary search on Z and get the appropriate index, which we call k throughout the explanation.

For the solution, for 1 ≤ i ≤ N, our substructure property for finding longest subsequence is [1]:

$$B_i = \begin{cases} (LongestOf(B_j), b_i), & 1 \leq j < i \wedge b_j \geq b_i \\ (b_i), & otherwise \end{cases}$$

But for a faster algorithm we get the help of Z, that is why we can construct an algorithm $\theta(NlogN)$ although we have larger number of subsequences.

At the beginning we have no sequence, **array Z is initialized with -1 in all its indices**. **Z[k] = b_i** where k is found with the help of binary search (let us denote as BS), which can be formulated like below:

$$k = BS(start, end) \begin{cases} start, & if\ start > end \\ middle, & if\ Z[middle] = b_i \\ BS(start, middle - 1), & if\ Z[middle] < b_i \\ BS(middle + 1, end), & if\ Z[middle] > b_i \end{cases}$$

where in the beginning of each iteration of i, start = 1 (which is the starting index), end = N which is the last index (in our case it is 6) and middle = (start + end) / 2 stands for middle index in each computation. Note that k is equal to the smallest index of an element which satisfies the condition that the element is less than b_i.

By using this formula in each iteration of i the found values of k are:

i=1 -> k=1 i=4 -> k=2

i=2 -> k=1 i=5 -> k=1

i=3 -> k=2 i=6 -> k=3

We cannot write exact formula for Z[k] since it will always be b_i but formula for k can be written since with binary search we are finding the proper k as written above. One can say that in our algorithm the main concern is finding k. After the program terminates length of Z gives us the length of the longest subsequence.

In order to track the sequence itself we will make use of 2 arrays called I, stands for Indexes, and P, stands for Parents. Array I will store the index of box b_i,which is i as we have defined above. We need this array to track the box id placed in Z[k]. Hence, **I[k] = i** is formula for I, while the formula for k is given above.

Array P stores the box id of the box where b_i put into, hence, **P[i] = I[k-1] and it is initialized with -1 in all its indexes at first.** We look k-1 index of I because we are processing the box numbers with array I and for each iteration we calculate k put a box number into I[k] meaning that in one of the previous iterations we found a box for I[k-1] and it will be the parent for box i. Note that for the first box there is no parent hence P[1] = -1 all the time.

For the example given above let us show the first 3 iterations:

1st iteration. b_1= 3, we found k as 1, then Z[1] = 3, I[1] = 1, P[1] = -1 (we put box 1 in Z[1] hence I[1] = 1 no parent for the first element of subsequence)

2nd iteration. b_2= 10, we found k as 1, Z[1] = 10, I[1] = 2, P[2] = -1 (we put box 2 in Z[1] since it is greater than 3, better chance for appending subsequence, hence I[1] = 2. P[2] = -1 since we take 10 as first element of subsequence, thus, it has no parent)

3rd iteration. b_3= 7, we found k as 2, Z[2] = 7, I[2] = 3, P[3] = 2(we append the sequence [10], put 7 in Z[2] hence the sequence of length 2 is ending with 7,which is box 3, we put I[2] = 3, P[3] = 2)

Lastly, let us explain how to print the subsequence itself and then conclude our algorithm. At the end of the iterations for constructing Z, say that we have Z with length = len excluding the indexes containing -1. Our array I will store the box number of the last box in I[len]. This is the last box it is not a parent for any box; hence we need to print it first. We can find the parent box of this last box in P[I[len]]. Then we will have a loop on P goes until the value in its index becomes -1 which means there is no parent anymore, thus, it will be the first box in our sequence. Our loop will look like this:

```
int parentIndex = I[len];
PRINT(B[parentIndex])

while(P[parentIndex] != -1)
{
        PRINT(B[P[parentIndex]]);
        parentIndex = P[parentIndex];
}
```

This concludes our algorithm. For the example sequence we found that the longest decreasing subsequence is [10,9,8] but it will be written as [8,9,10] since we constructed in that way meaning that 8 put into 9, 9 put into 10. The length of Z will be 3 which is the length of the subsequence.

2.4 The Overall Algorithm of the Solution

The overall algorithm of the solution to the problem has 3 main steps:

- Generating array of N elements containing rotations mentioned above.
- Sorting the array decreasing order with merge sort based on width of the boxes, i.e., first dimensions.
- Applying LDS to the array and find the solution.

The assumptions about the problem:

- To be able to be put into a box, a box must have strictly lower dimensions, i.e., B1 = (W_1, L_1) can only be put into B2 = (W_2, L_2) if W_1<$W_2$and L_1<L_2. This is the chosen assumption since taking $W_1 \leq W_2$ or $W_1 < W_2$will make no difference in the complexity of the algorithm.

- The length and the width of the boxes will be integers. This is assumed so that the correctness of the solution can be visible clearly. To choose double or integer dimensions will also make no difference in an algorithmic way.

Algorithm Description

1. By using given array B of boxes containing the dimensions, modify this array in a way that each box will be shown as (W, L) fixed and W will be the max value of the dimensions and L will be the min value of the dimensions.
2. Sort the modified array B in decreasing order of W with merge sort.
3. After sorting the boxes, the problem is the same as LIS.
4. To get overall maximum sequence, apply method LDS derived from LIS explained above.

Example:

Consider we have 5 boxes with the following dimensions:

B1 = (7, 6), B2 = (32, 12), B3 = (4, 6), B4 = (5, 8), B5 = (6, 4).

Solution:

1. **Rotations**

   - Aim: take min and max values of dimensions of boxes and change width to be max value and length to be min value.

   - Result:

Box i	W	L
1	7	6
2	32	12
3	6	4
4	8	5
5	6	4

Table 1: Table showing the boxes taken max value of dimension as width, the other as length

2. **Sorting**

- Aim: using merge sort to sort the boxes based on their first value W_i decreasing where i is the index of the box. The box numbers will be changed according to sorted values. In other words, Box 1 is put to second index and hence now it becomes Box 2 as it can be seen in the table below.

- Result: W_2 > W_4 > W_1> W_3= W_5 -> 32 > 8 > 7 > 6 = 6

Box i	W	L
1	32	12
2	8	5
3	7	6
4	6	4
5	6	4

Table 2: Table showing the sorted versions of the boxes and updated box array

3. LDS

- Aim: Finding the longest subsequence applying LDS with Binary Search. We are going to make use of three auxiliary arrays Z, P and I. The description of the values stored in these arrays explained in the section 2.3. We are going to make use of i and k. We will iterate through our array B. In each iteration of i, we will try to find LDS ending with b_i for the boxes b_1, …,b_i for 1 ≤ i ≤ N.

We used k to represent the index returned from binary search which is the index that we are going to put our b_i in iteration i, in other words with the help of binary search we will find k as it is explained above.

We will start our iteration from 1. In each iteration we will get result from binary search on the array Z and find which position we need to put element b_i on array Z. We will do our operations based only on the length of the boxes since the widths are already sorted, we do not need to consider them. Hence, in the figure below, only lengths of the boxes are shown:

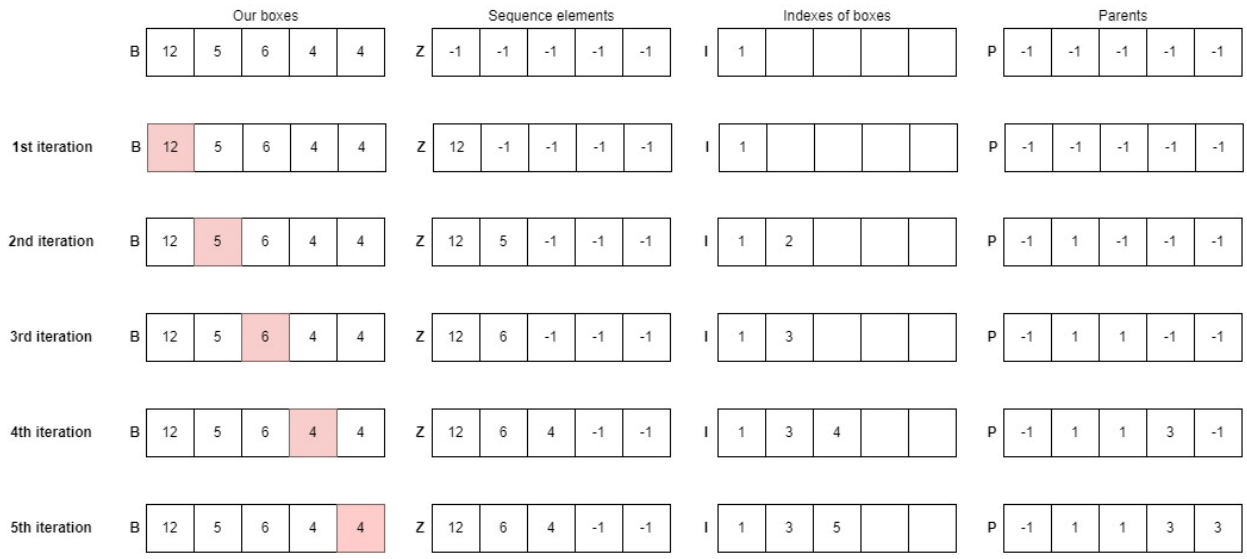

Figure 2: Figure showing the iterations of LDS algorithm

In each iteration we check Z array and find the position for the value of the current box (shown as pink) and place it in Z, at the same time we put the index of the box in the same position in I array then we update P accordingly:

P[i] = I[k-1], if k = 1 then leave it as -1

Here we get the subsequence of length 3, which is the number of elements Z and I contain. We can use the array P to write sequence itself.

At first, we will check the last element of array I which is 5, then we will start with box 5. After that we will iterate by looking the values in P itself:

I[3] = 5	write box 5	(6,4)
P[5] = 3	write box 3	(7,6)
P[3] = 1	write box 1	(32,12)
P[1] = -1	terminate	

Hence our longest subsequence of boxes will be (6,4) -> (7,6) -> (32,12) from inner box to outer box.

2.5 Pseudocode of the Algorithm

Pseudocode for Merge Sort

```
MERGESORT (B, p, r)
  IF p < r
    q <- ⌊(p + r)/2⌋
    MERGESORT (B, p, q)
    MERGESORT (B, q+1, r)
    MERGE(B, p, q, r)

MERGE (B, p, q, r)
  n1 <- q – p + 1
  n2 <- r – q
  Define L[1 .. n1]
  Define R[1 .. n2]                    /*L stands for Left R stands for Right*/
  FOR i <- 1 to n1
    L[i] <- B[p + i - 1]
  FOR j <- 1 to n2
    R[j] <- B[q + j]
  j <- 1
  i <- 1
  FOR h <- p TO r
    IF LENGTH(L) > i and LENGTH(R) > j  /*check arrays still have elements that are not processed*/
      IF WIDTH(L[i]) ≥ WIDTH(R[j])
      THEN
        B[h] = L[i]
        i <- i + 1
      ELSE
        B[h] = R[j]
        j <- j + 1
    ELSE IF LENGTH(L) > i      /* means that right array fully transferred but left one still has values*/
      B[h] = L[i]
      i <- I + 1
    ELSE                       /* means that left array fully transferred but right one still has values*/
      B[h] = R[j]
      j <- j + 1
```

Pseudocode LDS

```
LDS (B, Z)                     /*Z is the same array we used above*/
  DEFINE P[N]
  FOR i <- 1 TO N              /*Initialize P*/
    P[i] <- -1
  FOR i <- 1 TO N
```

```
    k <- BINARYSEARCH(Z, B[i])
    Z[k] <- B[i]
    I[k] <- i
    IF k != 1 THEN P[i] = I[k – 1]
  len <- 0              /*len is described as length of Z excluding the indexes containing -1*/
  WHILE len ≤ N and Z[len] > -1 DO
    len <- len + 1
  PRINT len             /*len becomes the length of LDS*/
  parentIndex <- I[len]
  PRINT B[parentIndex]
  WHILE P[parentIndex] != -1 DO
    PRINT B[P[parentIndex]]
    parentIndex = P[parenIndex]

BINARYSEARCH(Z, B[i])
  start <- 1
  end <- SIZE(Z)
  WHILE start ≤ end DO
    middle <- (start + end) / 2
    IF Z[middle] < B[i] THEN end <- middle - 1
    ELSE IF Z[middle] > B[i] THEN start <- middle + 1
    ELSE RETURN middle
  RETURN start
```

Pseudocode for Main Program

```
DEFINE Z[N]                 /*for subsequence information mentioned above*/
FOR i <- 1 TO N
   READ(B[i])               /* B stores boxes in the format of (W,L) as mentioned*/
   W_i <- max(W_i, L_i)
   L_i <- min(W_i, L_i)
   Z[i] <- -1
MERGESORT (B, 1, N)
LDS(B, Z)
```

3. Correctness Analysis

To prove that the approach we used to solve our problem is correct, dividing the solution into parts and prove the correctness of each part then combining them will be an easier way. Hence, we will first analyze merge sort then LDS and lastly our main algorithm.

Correctness Analysis of Merge Sort

We have two different functions called MERGE and MERGESORT. MERGESORT uses MERGE and that is why we first analyze this function and then move onto MERGESORT.

Correctness of MERGE

Since this function has iterative method to combine the arrays, we need to find loop invariant, condition that holds in each iteration, to be able to prove correctness. In the loop we sort the two parts of our array using variable h starts from p iterates to r.

Loop invariant: The subarray B[p .. h-1] has the h – p largest elements in decreasing sorted order.

Initialization: Prior to the first iteration we have h = p, so that subarray B[p .. h-1] is empty. This subarray contains h - p = 0 largest elements of L and R, since I = j = 1, both L[i] and R[j] are the largest elements of their arrays but have not been copied back into B.

Maintenance: To see in each iteration the loop invariant is maintained, let us first suppose that L[i] ≥ R[j]. then L[i] is the largest element but not yet copied into B. Since B[p .. h-1] contains h – p largest elements after L[i] is copied into B[h] the subarray B[p .. h] will contain the h – p + 1 largest elements. Incrementing h and i in the loop will help re-establishing the loop invariant for the next iteration. In the case of R[j] ≥ L[i] the same actions are taken but now for j not i, hence the invariant will be re-established again.

Termination: On the last iteration h = r + 1. By the invariant the subarray B[p .. h-1] which is B[p .. r] contains h – p = r – p + 1 largest elements of L[1 .. n_1] and R[1 .. n_2] in sorted order. L and R together contains $n_1 + n_2$ = N elements which covers all the array meaning that the whole array B will be combined in the correct order.

These steps show that we preserve our condition in each iteration and successfully terminate our loop, thus, we say this algorithm is correct [2].

As we have proven that MERGE is correct, we can now prove the correctness of MERGESORT using proof by induction.

Base: We have array of 1 element, and it is already sorted since only contains one element. Base case is correct.

Inductive Hypothesis: Assume that MERGESORT correctly sorts array of 2m+1 elements.

Then we need to prove it can manage to sort array of 2m+2 elements.

Inductive Step: The function will call itself two times for 2 arrays of size m+1. We know that by the induction hypothesis these arrays are already sorted. This shows that after recursive calls, array B will be sorted between indices p to q and q+1 to r respectively. We already showed that MERGE woks correctly, meaning that these two parts will be combined correctly and the array between indices p to r, which is the whole array, will be sorted.

These conclude the proof of merge sort.

Correctness Analysis of LDS

Since the algorithm uses iteration, we can use the loop invariant to prove that it is correct.

Loop invariant:

1. Z is decreasing after each loop. (This property is the one that allows us to use binary search.)
2. After each loop the value of k is given as the smallest index of Z such that Z[k] < b_i (this is the definition of k)
3. After each loop, Z[k] always saves the last element of the decreasing subsequence of length k if there are multiple choices for the same length it stores the largest one.
4. After each loop I[k] stores the box id, which is i, for the box that is chosen as the largest value for the ending of the decreasing subsequence of length k.
5. After each loop P[i] stores the box id, which is I[k-1], shows that current box i is put into the box with id I[k-1].

Initialization:

1. Before entering the loop all elements of Z is -1 meaning that Z is not increasing. After the first iteration Z[1] = b_1 which is always greater than -1, hence, Z becomes a decreasing array.
2. k = 1 is indeed the smallest index for b_1 satisfies the condition Z[1] = -1 ≤ b_1.
3. Z[1] = b_1 also stores the last element of the decreasing subsequence when length is 1 and the length is one since there is only one decreasing subsequence and it is also the largest.
4. I[1] = 1 since the length of the decreasing sequence is 1, which is k and it is 1, we have b_1 as the element and the id of it is also 1.
5. Before the loop all the values of P are -1 since we have not processed anything they are not decided and for our box 1 since it is the only element there is no parent for it.

Maintenance:

1. If Z is decreasing before the ith cycle, in the ith cycle the index of where we put b_i will be decided with the formula given for k above in the descriptions part, k, which will be j for $\boldsymbol{smallest\ j < N\ where\ b_i > Z[j]}$, means that in the loop we have Z[k+1] > Z[k] = b_i > Z[k-1]. Hence, after updating the value of Z[k] we will have a decreasing sequence again.
2. In the previous cycles we found the proper k for all the values up until $b_{i\text{-}1}$, which makes the array Z decreasing, hence for b_i we will look for 3 conditions until we find proper k.
    a. Z[middle] = b_i then for sure k = middle since it means that for the same value of b_i we have already found the k before, and we will use the same k now.
    b. Z[middle] < b_i then we will look for the left subarray which is Z[start .. middle-1] since it is decreasing, and we might be able to find smaller index that has a value smaller than b_i and it is obvious that that index will be more suitable for k.
    c. Z[middle] > b_i then we will look for the right subarray which is Z[middle+1 .. end] since the array is decreasing the value which is smaller than b_i can be found in the other indexes of Z which are greater than middle index.

d. For the formula of k, we continue to do these operations until we find b_i in the array as Z[k] = b_i or we have a subarray with only one element in it which is Z[k] < b_i . There is no way we could not find a k that satisfies Z[k] < b_i since we initialized all elements as -1. At one point this condition must hold.

As a result, we can see that our binary search algorithm will find the proper h, which is the smallest index can be found, for b_i to be put in Z satisfying the condition we have.

3. Z[k] also become the largest value until now because we replace the value in Z[k] with b_i which is already larger than Z[k] and we have stored the largest one before replacing with b_i , thus this makes our value $Z[k]$ is the largest possible one so far.
4. During the loop, array I will change according to k and we showed that the found k works in the step above, hence, storing the box id i, which is the box we put at the end of the decreasing subsequence with length, in I[k] will gives us the right notation since we define I in that way, i.e. storing the box id where the box is the last largest element in the decreasing subsequence of length k.
5. During the iteration j, P will change also according to j and say that we find a value k-1 for this box to be put as the largest last box of subsequence with length k-1 > 0. We put this id of this box in I[k-1]. In one of the next iterations say in iteration i note that i > j, we will find a box that can be appended the sequence of length k-1, hence the calculations above will give us k as length, we have $the\ id\ of\ b_j, which\ is\ i\ in\ I[k-1]$ and now we found a box b_i that can be put in b_j , hence the parent of current box will be b_j which we store its id in I[k-1]. As a result, our current box, box i, will have the parent box j which can be denoted as P[i] = I[k-1] = j.

Termination:

At the end of the iterations, we will be covering all the boxes through 1 to N considering all the possible sequences hence we will get the length of the sequence looking at length of Z since the indexes of Z shows the length of decreasing subsequences. By using the array, I and P we can

write the sequence itself. Hence, it can be said that the algorithm of LDS works properly as expected.

Correctness Analysis of Main Algorithm

Assuming the dimension of a box is (x, y) then if we call length of this box is the minimum of (x, y) and width is the maximum of (x, y) since we can rotate a box and rotating gives us the ability to set length and width according to our choice and this way, we choose the rotation that makes width larger and length smaller. After then we are taking the rotations of the boxes that their first dimension as width and second as length. This is also an appropriate operation since we have already chosen the rotations and we want the dimensions aligned so that we can only consider one dimension while applying LDS. After these we have merge sort which has proven and LDS which has also proven then our algorithm terminates. This means that overall algorithm is correct.

4. Time Complexity Analysis

For the time complexity analysis, let us first analyze time complexities of each function used in the main program and then summing up those time complexities will naturally give us the overall time complexity of the whole solution.

Time Complexity of Merge Sort

Merge sort is a well-known algorithm having the following recurrence equation of time complexity in general:

$$T(N) = \begin{cases} \theta(1), & if\ N = 1 \\ 2T\left(\frac{N}{2}\right) + \theta(N), & if\ N > 1 \end{cases}$$

The equation of merge sort is appropriate to apply master theorem. Hence, let us compute the time complexity by using master theorem. Recall the master theorem [3]:

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Figure 3: Figure showing the definition of the master theorem

In our case a = 2, b = 2 and f(N) = $\theta(N^{\log_2 2}) = \theta(N)$, then the time complexity of merge sort is: T(N) = $\theta(N \log N)$.

Time Complexity of LDS

```
1.  LDS (B, Z)
2.    I[1] <- B[1]
3.    DEFINE P[N]
4.    FOR i <- 1 TO N                    /*N iterations which is θ(N)*/
5.      P[i] <- (-1)
6.    FOR i <- 1 TO N                    /*N iterations having logN in each -> θ(NlogN)*/
7.      k <- BINARYSEARCH(Z, B[i])
8.      Z[k] <- B[i]
9.      I[k] <- i
10.     IF k != 1 THEN P[i] = I[k – 1]
11.   len <- 0
12.   WHILE len ≤ N and Z[len] > -1 DO
13.     len <- len + 1
14.   PRINT len
15.   parentIndex <- I[len]
16.   PRINT B[parentIndex]
17.   WHILE P[parentIndex] != -1 DO
18.     PRINT B[P[parentIndex]]
19.     parentIndex = P[parenIndex]

20. BINARYSEARCH(Z, B[i])
21.   start <- 1
22.   end <- SIZE(Z)
23.   WHILE start ≤ end DO
```

```
24.     middle <- (start + end) / 2
25.     IF Z[middle] < B[i] THEN end <- middle - 1
26.     ELSE IF Z[middle] > B[i] THEN start <- middle + 1
27.     ELSE RETURN middle
28.   RETURN start
```

Since BINASRYSEARCH is a used function inside LDS let us first explain the time complexity of this function. Binary search is a searching algorithm works on already sorted array. The logic behind it is always comparing the middle element of the array with the desired value. If it is greater or smaller it sets the pointers accordingly such that it splits up the array half and starts looking only the half that the element might be in it. For example:

Let Y is an array, Y = {9, 7, 6, 5, 3, 2, 1} and we are looking for 1. Then the algorithm sets middle as 5 and compares it with 3. Since 3 is smaller than the algorithm starts to consider the right-hand side of this array starting the next index of where 5 is placed. This means that every iteration we break our array into half and this process continues until the element is found or there is no remaining element in the array.

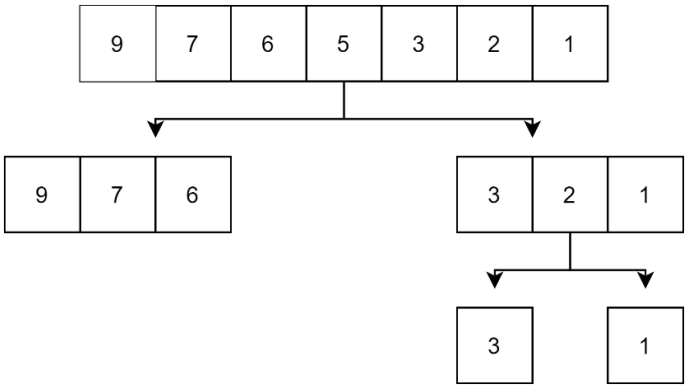

Figure 4: Figure showing an example of the logic behind Binary Search

In every iteration we have one comparison then going on with the other half. We can plot a tree to see it clearly.

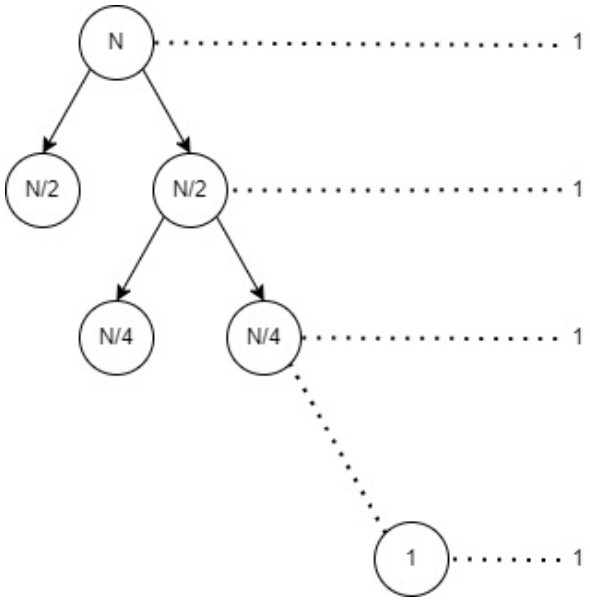

Figure 5: Figure showing the tree consisting of subproblems in Binary Search

In each iteration we have only 1 operation the 1s in the figure represents this. The nodes show the length of the arrays. In order to compute how many ones we have, we need to know how many times we can divide N to 2 until it becomes 1.

$1 = N / 2^x$

$2^x = N$

$x = \log N$

Thus, we must run our operation of $\theta(1)$ for logN times. This makes time complexity of Binary Search $\theta(logN)$.

For the lines between 7 – 11 there is a loop which iterates N times and for each iteration we have a BINARYSEARCH function with $\theta(logN)$.

Lines 13-15 we have a loop iterates according to the maximum number of boxes. If max sequence length is 5 it will iterate 5 times. As a result, it can be said that this loop can take at most N times in worst case, i.e., if all the boxes can be added to our sequence. Hence, we say that time complexity for this loop is O(N).

For the loop in lines 17-19 we are writing the sequence itself, hence, time complexity of this loop is the same as the loop in 13-15, it will take N times in worst case. This gives us O(N).

If we sum up these complexities it will give us the complexity of LDS which is:

$\theta(NlogN)$+O(N)+O(N)+O(N) = $\theta(NlogN + 3N)$ = $\theta(N(logN + 3))$ in here we can omit 3 and it becomes $\theta(NlogN)$. As a result, time complexity of LDS is $\theta(NlogN)$.

Time Complexity of Main Program

1. DEFINE Z[N]
2. FOR i <- 1 TO N
3. READ(B[i])
4. W_i <- max(W_i, L_i)
5. L_i <- min(W_i, L_i)
6. Z[i] <- -1
7. MERGESORT (B, 1, N)
8. LDS(B, Z)

In the lines 2-6 for loop iterating N times is used to initialize Z and read B. Line 7 calls merge sort which runs in $\theta(NlogN)$ and line 8 calls LDS which also has the complexity of $\theta(NlogN)$. If we sum up these complexities, we will get:

$\theta(2NlogN) + \theta(N) = \theta(N(2logN + 1))$ we can omit 1 and 2 here since they will be negligible when N becomes larger and larger.

To summarize, our overall complexity will be $\theta(NlogN)$.

5. Input and Output Description

1. Input Description:

The input file is a .txt file with the following format:

- First line contains a number N showing the number of given boxes.
- For lines starting from 2 to N+1, each line contains two numbers representing the dimensions of a box. First dimension and second dimension are separated with comma and no space in between.

Below an example of input file can be seen:

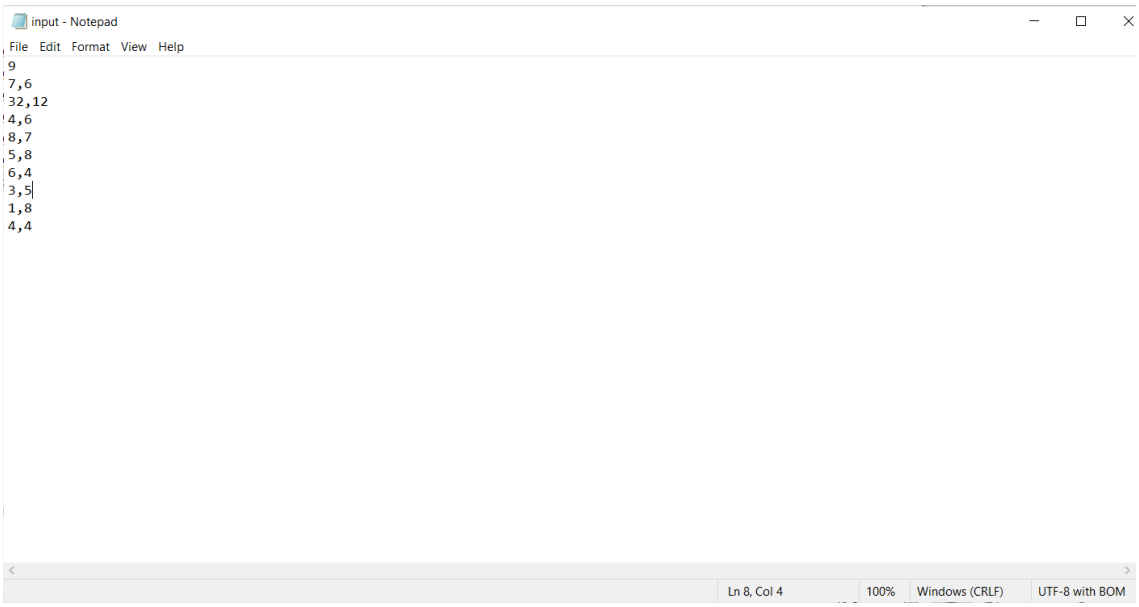

Figure 6: Figure showing an example input file

2. Output Description:

The output file is a .txt file with the following format:

- First line contains a number len, which we have described and used above, which is the maximum number of boxes put one in other.
- For lines starting from 2 to len+1, each line contains a tuple representing the dimensions of a box. First dimension and second dimension are separated with comma and no space in between. The order of the lines shows the generated subsequence from innermost box to outermost box, i.e., the box in line j is the one put inside into the box in line j+1 for $2 \leq j < len+1$

Below an example of an output file can be seen:

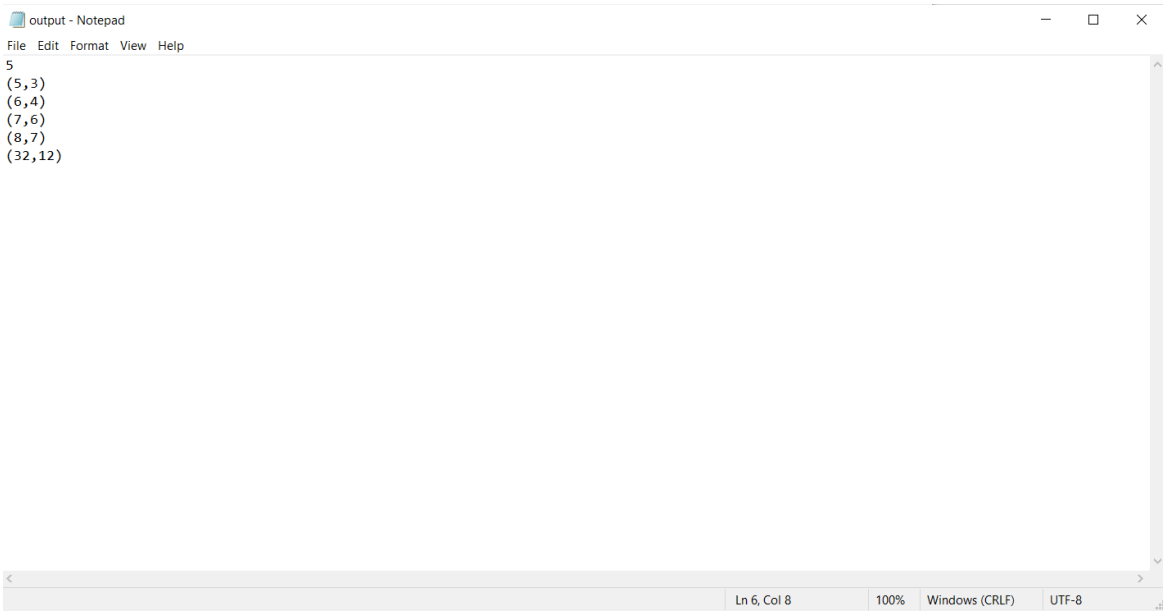

Figure 7: Figure showing an example output file

6. References

[1] http://www.dei.unipd.it/~geppo/DA2/DOCS/lis.pdf

[2] https://www.cs.mcgill.ca/~dprecup/courses/IntroCS/Lectures/comp250-lecture16.pdf

[3] Charles E. Leiserson. Ronald L. Rivest. Clifford Stein. **Introduction to Algorithms**. Third Edition. The MIT Press. Cambridge, Massachusetts London, England