Project Boxes: Functional Documentation

Group Member: Lu Zhicong, Zhang Boxuan

1. Problem Definition

1.1. Original Problem

We have a given set of boxes and need to find the largest set of boxes, such that they can be put one in another (like matryoshka).

1.2. Definitions

1.2.1. We define a box as $B_i = (x_i, y_i)$, where x_i is the width of box and y_i is the length of box. We can see any box B_i could be written as $B_i = (x_i, y_i)\ or\ B_i = (y_i, x_i)$.To make the definition clear, we define a standard format of describing a box that $B_i = (x_i, y_i), x_i \le y_i$.

1.2.2. We define that a box $B_i(x_i, y_i)$ can be put into another box $B_j(x_j, y_j)$ as $B_i \le B_j$, which also means $(x_i \le x_j) \wedge (y_i \le y_j)$.

This relation is **transitive**, that if $B_i \le B_j \wedge B_j \le B_k$, then $B_i \le B_k$.

1.2.3. We define 2 boxes of the same size as $B_i = B_j$, which also means $(x_i = x_j) \wedge (y_i = y_j)$.

This is the only case that satisfies $B_i \le B_j$ and $B_j \le B_i$.

1.2.4. We define "a sequence of boxes" as $\mathbb{S}$:

$$\mathbb{S} \coloneqq \left[B^{(1)}, B^{(2)}, \dots, B^{(k)}\right]; s.t.\ \forall i = 1, \dots, k-1, B^{(i)} \le B^{(i+1)}.$$

1.3. Problem

The original problem equals to find **the longest sequences of boxes $\boldsymbol{S}$** among a set of N boxes $\mathbb{B} = \{B_1(x_1, y_1), B_2(x_2, y_2), \dots, B_i(x_N, y_N)\}, N > 0$.

2. Solution

2.1. Solution Description

2.1.1. STEP 1:

- **Rotate all boxes into standard format that width is longer than length:** $B_i = (x_i, y_i), x_i \le y_i$.
- **Sort the boxes by length $\boldsymbol{y}$ (nondecreasing). If there are boxes with the same length $\boldsymbol{y}$, then we should sort these boxes by width $\boldsymbol{x}$ (nondecreasing):**

$$\mathbb{B}_{rank} = [\boldsymbol{B_1}(\boldsymbol{x_1}, \boldsymbol{y_1}), \boldsymbol{B_2}(\boldsymbol{x_2}, \boldsymbol{y_2}), \dots, \boldsymbol{B_N}(\boldsymbol{x_N}, \boldsymbol{y_N})]$$

, such that $\forall \boldsymbol{i} = \boldsymbol{1}, \dots, \boldsymbol{N-1}$, $(\boldsymbol{y_i} < \boldsymbol{y_{i+1}}) \vee (\boldsymbol{y_i} = \boldsymbol{y_{i+1}} \wedge \boldsymbol{x_i} \le \boldsymbol{x_{i+1}})$.

Therefore, for any boxes B_j and B_k, there are:

$$(j < k) \wedge (x_j \le x_k) \Rightarrow B_j \le B_k$$

$$(j < k) \wedge (\mathrm{B_j} \neq \mathrm{B_k}) \Rightarrow \neg(\mathrm{B_k} \le \mathrm{B_j}).$$

We will construct the sequences by the order of $\mathbb{B}_{rank}$, so any new box can only become the last box of an existing box sequence (unless it is the same with the last box, but in this case, we do not need to discriminate the same boxes).

When we add a new box into the box sequences, we choose the longest sequence of length L that can accept the new box, then we get a sequence of length L+1.

For box sequences of the same length, we only need to record the "best" one whose last box has the shortest width, so that it will be the easiest one to be put into the following boxes.

2.1.2. STEP 2:

- Define DP_{width} array such that the L_{th} elements $DP_{width}[L]$ stores the minimal width x of all the boxes $B^{(L)}$ that satisfies the following condition: $B^{(L)}$ is the last box of a length-L sequence $\mathbb{S} \coloneqq [\, B^{(1)}, B^{(2)}, \dots, B^{(k)}]$.

  DP_{width} array is **nondecreasing** because if $DP_{width}[L-1] > DP_{width}[L]$, then consider sequences $\mathbb{S}_L = [B^{(1)}, B^{(2)}, \dots, B^{(L-1)}, B^{(L)}]$ and $\mathbb{S}_{L-1} = [B^{(1)}, B^{(2)}, \dots, B^{(L-1)}]$. Then the width of boxes $x^{(L-1)} \le x^{(L)} = DP_{width}[L] < DP_{width}[L-1]$, which means $DP_{width}[L-1]$ is not the minimal element.

  The longest sequence's length $MaxLength$ is the max index of DP_{width} that has a value.

- Define $SeqLength$ array such that $SeqLength[i]$ stores the length of the longest sequence $\mathbb{S}$ end with $B_i(x_i, y_i)$.

  This array can help us restore the longest sequence. We will illustrate the procedure in Step 4.

2.1.3. STEP 3:

- **Initialize DP_{width} as an array of length $N+1$, and $SeqLength$ array as an array of length N. Fill $DP_{width}[0]$ with -1. Store the current longest sequence length to variable L_{max}.**

- **Iterate through the boxes of $\mathbb{B}_{rank} = [\, \boldsymbol{B_1(x_1, y_1), B_2(x_2, y_2), \dots, B_N(x_N, y_N)}]$ in order, and complete the $\boldsymbol{DP_{width}}$ array and $\boldsymbol{SeqLength}$ array. During this step, the length of longest sequence should be found (L_{max}).**

  How to update $\boldsymbol{DP_{width}}$ array: Catch a new element $B_i(x_i, y_i)$. If $x_i \ge DP_{width}[L_{max}]$, then set $DP_{width}[L_{max}+1]$ as x_i and increase L_{max} by 1. Otherwise, search in the subarray of $DP_{width}[0 \sim L_{max}]$ for m such that $DP_{width}[m] \le x_i < DP_{width}[m+1]$. Noting that DP_{width} is nondecreasing, we can use **binary search** to find it, then replace $DP_{width}[m+1]$ with x_i.

  How to update $\boldsymbol{SeqLength}$: After updating DP_{width} array, notice that $B_i(x_i, y_i)$ is larger and only larger than the minimal last box of existing sequences of length **m**, which means there is a length **m** sequence can accept B_i and no **m+1** sequence accepting B_i. Thus, the longest sequence end with

$B_i(x_i, y_i)$ should be **m+1**.

- **Pseudocode:**

```
Input: 𝔹_rank = [B_1(x_1, y_1), B_2(x_2, y_2), ..., B_N(x_N, y_N)]
Output: DP_width array, SeqLength array, L_max variable
L_max ← 0;
DP_width ← array of N+1, DP_width[0] ← -1;
SeqLength ← array of N;
for i ← 1 to N do
    if x_i ≥ DP_width[L_max] then
        L_max ← L_max + 1 ;
        DP_width[L_max] ← x_i ;
        SeqLength[i] ← L_max;
    else
        m ← BinarySearch(DP_width , x_i);
        DP_width[m+1] ← x_i ;
        SeqLength[i] ← m+1;
    endif
return L_max;
```

2.1.4. STEP 4:

- **Restore the longest increasing sequence of boxes.**
- **Iterate through the $SeqLength$ array backwards. Find an element $SeqLength[i]$ equals to $MaxLength$, then box with identical index $B_i(x_i, y_i)$ is the last box of target sequence. Decrease the $MaxLength$ by 1, continue the iteration, and repeat the operation to construct target sequence.**
- **Pseudocode:**

```
Input: SeqLength array, L_max, 𝔹_rank = [B_1(x_1, y_1), B_2(x_2, y_2), ..., B_N(x_N, y_N)]
Output: LongestSequence array
l ← L_max;
LongestSequence ← array of L_max;
for i ← N to 1 do
    if SeqLength[i] = l then
        LongestSequence[l] ← B_i(x_i, y_i);
        l ← l–1;
    else
        continue;
return LongestSequence;
```

3. Correctness

3.1. Proof

The Dynamic Programming:

Consider a primitive 2-dimension Dynamic Programming Function $DP_{width}(L,\ k)$, which means the least width $x^{(L)}$ of $B^{(L)}$ of the Box Sequences consisting of boxes from the first k elements subset of $\mathbb{B}_{rank}$: $\{B_1(x1, y1),\ B_2(x2, y2), \dots,\ B_k(x_k, y_k)\}$,

$$DP_{width}(L,\ k) \coloneqq DP_{width}[L], in\ the\ k_{th}\ iteration\big(when\ processing\ B_k(x_k, y_k)\big).$$

, then we can write the **State Transition Equation:**

$$DP_{width}(L,\ k) = \begin{cases} x_k,\ if\ x_k > DP_{width}(L-1, k-1)\ \wedge\ x_k \le DP_{width}(L,\ k-1); \\ \qquad\qquad DP_{width}(L,\ k-1),\ otherwise; \end{cases}$$

Therefore, $DP_{width}(L,\ k)$ only depends on $B_k(x_k,\ y_k)$ and $DP_{width}(L,\ k-1)\ or\ DP_{width}(L-1,\ k-1)$.

Thus, we can simply represent it as 1-dimension Array $DP_{width}[L]$ and iterate k from 1 to N.

In the end, the last element of $DP_{width}[L_{max}]$ represents $DP_{width}(L_{max},\ N)$, which is the least width of the $x^{(L)}$ of the longest box sequences consisting of all the boxes. The L_{max} is the length of the longest box sequence.

3.2. Time Complexity Analysis

- The total time complexity is O(N log N).

  Step 1:

  Rotating the boxes costs O (N).

  The Merge Sort costs O (N log N).

  Step 2:

  The initialization of DP_{width} and $SeqLength$ array: O (N)

  Step 3:

  We can prove that DP_{width} Array is always nondecreasing by induction:

    The DP Array is nondecreasing at the beginning: DP_{width} = [].

    If the DP Array is already nondecreasing, then we add a new box of width x_k.

    We find the maximal L such that DP_{width} [L] $\le x_k$, then DP_{width} [L] must be the maximal element of DP_{width} such that $x \le x_k$(because DP_{width} Array is nondecreasing), and DP_{width} [L+1] must be the minimal element such that $x > x_k$. After replacing DP_{width} [L+1] with x_k, the DP Array is still nondecreasing:

$$DP_{width}\ [\mathrm{L}] \leq \mathrm{x_k} < DP_{width}\ [\mathrm{L}+1]_{\mathrm{old}} \leq DP_{width}\ [\mathrm{L}+2]\ \leq \cdots$$

Therefore, we can use binary-search in Step 3 during the process of adding a new box into DP Array, so the time complexity of each iteration is O (log N), and the total time complexity of Step 3 is O (N log N)

Step 4:

Return the longest box sequence: O (N).

Iterate the $SeqLength$ array of length N.

4. Input and Output Description

4.1. Input Format

First Line: **N** (integer that shows the number of boxes)

Next **N** lines: **index x y** (index is the index of box (from 1 to N),x and y are double values larger than 0; split char: ‘ ’)

4.2. Input Example with comment

3	//number of boxes
1 2.4 2.0	$//\mathrm{B_1}(\mathrm{x_1}=2.4, \mathrm{y_1}=2.0)$
2 5.0 6.0	$//\mathrm{B_2}(\mathrm{x_2}=5.0, \mathrm{y_2}=6.0)$
3 1.0 10.0	$//\mathrm{B_3}(\mathrm{x_3}=1.0, \mathrm{y_3}=10.0)$

4.3. Output Format

First Line: L_{max} (integer that shows the length of the longest sequence)

Next L_{max} lines: **index x y** (index is the index of box getting from input file, x and y are width and length of box).

4.4. Output Example with comment

2	//length of longest sequences
1 2.4 2.0	$//\mathrm{B_1}(\mathrm{x_1}=2.4, \mathrm{y_1}=2.0)$
2 5.0 6.0	$//\mathrm{B_2}(\mathrm{x_2}=5.0, \mathrm{y_2}=6.0)$