Advanced Algorithms

Warsaw University of Technology

Functional Documentation

November 16th, 2019

Problem Description

We are given a set of boxes, for each box we have a width w_i and length l_j. Consider two boxes b_i and b_j with width w_i, w_j and length l_i , l_j and respectively. Box b_i can only be packed inside of box b_j if and only if $w_i < w_j$ and $l_i < l_j$.

We are tasked with finding the longest set of boxes that fit inside one another. The solution algorithm to the aforementioned problem must be based on Dynamic Programming algorithms.

Solution Description

Description

As a solution to the box packing problem, let us combine two well-known algorithms to help us achieve a fast algorithm that solves our problem. Let us assume that we have 2 vectors as inputs, w and l for width and length respectively. Let us also assume that $l_i \geq w_i, \forall i \in \{1, \ldots, n\}$ where n is the length of l and w.

- Start off by sorting the vector l ascendingly (do note that sorting the vector w instead of l would work as well in this case - since the width and the length of the boxes are interchangeable) using merge sort. The same exact adjustments will be made on the vector w. For instance, replacing l_i with l_j while sorting the vector l would result in replacing w_i with w_j in the vector w.
- Then apply the "Longest Increasing Subsequence" dynamic programming algorithm on the vector w, and the return values from the algorithm are: 1) a new vector p holding the longest increasing subsequence of indexes corresponding to the vector w, and 2) the length of the longest increasing subsequence res, taken from the vector d, which has the same length as w, and holds the values of the length of the longest increasing subsequence for each value ending with w_i.
- Fill out the matrix A where the first row vector is $row_1 = l[p_i]$ and the second row vector is $row_2 = w[p_i]$, with $i = 1, \ldots, n$.
- Finally, output is the solution as res, and the box dimensions as $(l[p_1], w[p_1]) < \ldots < (l[p_n], w[p_n])$.

Pseudocode

Below is the pseudocode of the above described solution:

```
Function fulfill_assumption(vector a, vector b)
	Var x := 0
	For i = 0 to length of a do
		If a[i] < b[i] then
			x := a[i]
			a[i] := b[i]
			b[i] := x

	Return a, b

Function merge(vector left_a, vector right_a,
		vector left_b, vector right_b)
	Var result_a := empty list
	Var result_b := empty list

	While left_a is not empty and right_a is not empty do
		If first(left_a) ≤ first(right_a) then
			Append first(left_a) to result_a
			Append first(left_b) to result_b
			left_a := rest(left_a)
			left_b := rest(left_b)
		Else
			Append first(right_a) to result_a
			Append first(right_b) to result_b
			right_a:= rest(right_a)
			right_b:= rest(right_b)

	While left_a is not empty do
		Append first(left_a) to result_a
		Append first(left_b) to result_b
		left_a := rest(left_a)
		left_b := rest(left_b)
	While left_a is not empty do
		Append first(right_a) to result_a
		Append first(right_b) to result_b
		right_a := rest(right_a)
		right_b := rest(right_b)

	Return result_a, result_b

Function merge_sort(vector a, vector b)
	If length of a ≤ 1 then
		Return a, b

	Var left_a, right_a := empty list
	Var left_b, right_b := empty list
	For each x with index i in a and y with index j in b do
		If i < (length of a)/2 then
			Add x to left_a
			Add y to left_b
```

```
            Else
                  Add x to right_a
                  Add y to right_b

      left_a, left_b := merge_sort(left_a, left_b)
      right_a, right_b := merge_sort(right_a, right_b)

      Return merge(left_a, right_a, left_b, right_b)

Function lis(vector a)
      Var n := length of a
      Var d := list of size n, filled with 1s
      Var p := list of size n, filled with -1s
      For i = 1 to n do
            For j = 1 to i do
                  If a[j] < a[i] and d[i] < (d[j] + 1) then
                        d[i] = d[j] + 1
                        p[i] = j

      Var res := d[0]
      Var pos := 0
      For i = 1 to n do
            If d[i] > res then
                  res := d[i]
                  pos := i

      Var p
      While pos ≠ -1
            Append a[pos] to p
            pos := p[pos]
      Reverse(p)

      Return p, res

Var l := list of input of integers
Var w := list of input of integers
l, w := fulfill_assumption(l, w)
l, w := merge_sort(l, w)

Var p := empty list
Var res := 0
p, res := lis(w)

Print p, res
```

Correctness and Time Complexity Analysis

Proof of Correctness

Correctness of Fulfill_Assumption

As this is an iterative function, we want to find a loop invariant which is a condition that holds every time the internal loop (or loops) is executed and helps us prove correctness. In this case, let us focus on the a list. At iteration k, our loop invariant will be: $a[l] \leq b[l], \forall l \in \{1, \ldots, k\}$. In other words, the elements we switched to a up to position k, are all less than every element in b. Why does this condition help us? Since the a list is filled with elements that are less than or equal to the corresponding elements in b, all the remaining larger elements in b will be switched to a later, at positions $a[k+1], \ldots, a[n]$. Hence this condition means that at the end of the Fulfill_Assumption, $a[k] \leq b[k], \forall k$, which means that after copying, every element in a will be less than or equal to every corresponding element in b.

Correctness of Merge

To prove that Merge_Sort works correctly we have to first look at the correctness of the Merge function. As this is an iterative function similar to the Fulfill_Assumption function, let us focus on the $result_a$ list which, in this case, will be out loop invariant. At iteration k, suppose that the indices in the two parts of the list a are i and j. Then, our loop invariant will be:$result_a \leq a_l, \forall l \in \{i, \ldots, m\}$and $result_a \leq a_l, \forall l \in \{j, \ldots, p\}$. In other words, the element we just copy at position k is the minimum of the remaining elements. Since the $result_a$ list is filled from left to right, all the elements left in a will be put in later, at positions $result_a[k+1], \ldots, result_a[q-p+1]$. Hence, this condition means that at the end of the Merge, $result_a[k] \leq result_a[l], \forall k < l$, which means that after copying, a will be correctly sorted between p and q.

Correctness of Merge_Sort

Now that we know Merge works correctly, we will show that Merge_Sort works correctly, using a proof by induction.

For the base case, consider a list of 1 element (which is the base case of the algorithm). Such a list is already sorted, so the base case is correct.

For the induction step, suppose that Merge_Sort will correctly sort any list of length less than n. Suppose we call Merge_Sort on a list of size n. It will recursively call Merge_Sort on two lists of size $\frac{n}{2}$. By the induction hypothesis, these calls will sort these lists correctly. Hence, after the recursive calls, list a will be sorted between indices $p, \ldots, m$ and $m+1, \ldots, q$ respectively. We have already showed that Merge works correctly, hence after executing it, a will be correctly sorted between p and q. This concludes our proof of Merge_Sort.

Correctness of Lis

Next, we would like to prove that Lis works correctly. Given a list $X = (x_1, \ldots, x_n)$ of n integers drawn from the set of positive integers Z^+, an Increasing Subsequence (IS) of X is a subsequence $Z = (z_1, \ldots, z_k)$ of X (that is, $Z = (x_{i_1}, \ldots, x_{i_k})$, with $1 \leq i_1 < \ldots < i_k \leq n$) with $z_i < z_{i+1}$ for $1 \leq i < k$. Our goal is to show that Lis returns the Longest Increasing Subsequence (LIS) of the input string X, denoted $LIS(X)$.

For $1 \leq i \leq n$, define $\underline{LIS}(X_i)$ as the longest among all those increasing subsequences of prefix X_i which end with x_i. (Observe that $\underline{LIS}(X_i)$ may be much shorter than $LIS(X_i)$.) Moreover, let $l[i]$ be the length of a $\underline{LIS}(X_i)$. Clearly, the length of $LIS(X)$ is $max\{l[i]: 1 \leq i \leq n\}$, since any $\underline{LIS}(X_i)$ is just an increasing subsequence of X, hence no larger than $LIS(X)$, but also, if $Z = LIS(X) = (x_{i_1}, \ldots, x_{i_k})$ then Z must be a $\underline{LIS}(X_{i_k})$.

Let us now derive an optimal substructure property for the space $\{\underline{LIS}(X_i): 1 \leq i \leq n\}$. For $1 \leq i \leq n$, define S_i as the set $\{j: 1 \leq j < i \text{ and } x_j < x_i\}$. Then, the following recurrence must hold:

$$l[i] = \begin{cases} 1, & \text{if } S_i = \emptyset, \\ 1 + max\{l[j]: j \in S_i\}, & \text{otherwise.} \end{cases}$$

To show that the above relation holds, note that if $S_i = \emptyset$, then all elements x_j with $1 \leq j < i$ are no smaller than x_i, hence the only IS of X_i with last element x_i is (x_i). When $S_i \neq \emptyset$, consider a given $\underline{LIS}(X_i)$. Such a $\underline{LIS}$ has clearly length at least 2. Now, if the penultimate element of such subsequence is x_k, then $k < i$ and $x_k < x_i$ (by the definition of increasing subsequence), therefore $k \in S_i$. Moreover, all the first $k - 1$ elements of $\underline{LIS}(X_i)$ must form a $\underline{LIS}(X_k)$. If this were not the case, we could find an increasing subsequence ending with x_i longer than the $\underline{LIS}(X_i)$ itself, thus a contradiction. Needless to say, $\underline{LIS}(X_k)$ must be the longest among all the $\underline{LIS}$ ending with elements x_j with $j \in S_i$, or, again, we could obtain a longer $\underline{LIS}(X_i)$. This concludes our proof.

Correctness of the Overall Algorithm

The overall algorithm is correct following through from the correctness of all the above functions.

Time Complexity Analysis

Complexity of Fulfill_Assumption

The time of Fulfill_Assumption function is clearly upper bounded by the length of a and b which are in turn n. Thus, it has $O(n)$.

Complexity of Merge

The Merge function goes sequentially on the part of the list that it receives, and then copies it over. So the complexity of this step is $O(q - p + 1)$. To see this, note that either i or j must increase by 1 every time the loop is visited, so each element will be "touched exactly once" in the loop.

Complexity of Merge_Sort

Let us think intuitively what the complexity of Merge_Sort might be. As seen above, the Merge function goes sequentially on the part of the list that it receives, and then copies it over. So the complexity of this step is $O(q - p + 1)$. Merge_Sort does two recursive calls, each on a list which is half the size of the original.

In order to get a better handle of what the resulting complexity might be, suppose that we denoted by $T(n)$ the amount of time that Merge_Sort uses on a list of size n. Note that executing a sequence of instructions will cause us to add the running time. Hence, the running time will obey the following equation:

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + T_{merge}(n) = 2T(\frac{n}{2}) + cn$$

where c is a constant, reflecting the actual number of basic operations (comparisons, tests, arithmetic operations, assignments) in the merge routine. To compute the running time, we would like to get a closed formula for T, or at least figure out what its fastest-growing term is (so that we can figure out $O()$ for the algorithm).

To get a better grip on the problem, let us unwind T for a couple more steps:

$$T(n) = 2T(\frac{n}{2}) + cn = 2(2T(\frac{n}{4}) + c\frac{n}{2}) + cn = 2^2T(\frac{n}{4}) + c\frac{n}{4} + 2cn = 2^2(2T(\frac{n}{8}) + c\frac{n}{4}) + 2cn$$
$$= 2^3T(\frac{n}{8}) + 3cn$$

We continue this expansion until we get to $T(1)$ which is 1 (this corresponds to the base case of running on a list of size 1). Since in each step we halve n, we will reach $T(1)$ in $log_2 n$ steps. At that point, we will have:

$$T(n) = 2^{log_2 n}T(1) + cn\ log_2 n$$

The first term above is $O(n)$, the second is $O(nlogn)$, so the whole Merge_Sort algorithm is $O(nlogn)$.

Complexity of Lis

The running time $T(n)$ of the above Lis algorithm is upper bounded by the number of iterations of the two nested loops needed to compute vectors d and p. Therefore:

$$T(n) = O(\sum_{i=2}^{n}\sum_{j=1}^{i-1} 1) = O(\sum_{i=2}^{n}(i-1)) = O(n^2)$$

Complexity of the Overall Algorithm

Since the Fulfill_Assumption has $O(n)$, the Merge_Sort has $O(nlogn)$ and Lis has $O(n^2)$, then the whole algorithm is $O(n^2)$.

Input and Output Description

Input Description

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

- On row_1: an integer should be written, which is n - the number of boxes inside the file.
- From $row_2, \ldots, row_{n+1}$: two integers should be written separated by a space, the first integer is l_i- the length of the ith box - and the second integer is w_i - the width of the ith box - where $i = 1, \ldots, n$.

Output Description

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

- On row_1: the following string is written "The solution is: res" where res is the result of the solution.
- On row_2: the following string is composed "box b_1 with $(l_1, w_1) < \ldots <$ box b_n with (l_n, w_n).