Adv Data Struct & Algorithm: Homework 5

Zimian Li

June 2, 2018

1. Union of dynamic sets. Let T_1 and T_2 be BSTs with the property that if $x_1 \in T_1$ and $x_2 \in T_2$ then $key[x_1] < key[x_2]$.

   (a) Show how to efficiently create a single randomized search treap $T = T_1 \cup T_2$ when T_1 and T_2 are valid randomized search treaps.

   Add a new node x as the root, then put T_1 as left sub-tree and T_2 as right sub-tree. Change $\rho(x)$ to ∞, and rotate x down to be a leaf, then remove x.

   (b) Show how to efficiently create a single RB-tree $T = T_1 \cup T_2$ when T_1 and T_2 are both valid RB-trees.

   Assume $height(T_1) >= height(T_2)$.
   At first, get the leftmost node of T_2, remove it from T_2, let it be node x. Then from the root of T_1, following the right children, find a node whose height is $height(T_2)$. When find it, let it be node y, replace node x with y, then make y be the left child of x, and put the rest of T_2 as the right sub-tree. Then I think in some conditions it also need to recolor some nodes.

   (c) Analyze the running time in both cases.

   Suppose $h_1 = height(T_1), h_2 = height(T_2)$.
   For (a), the time complexity is $O(h_2)$, because the root would rotate from top to the bottom of T_2.
   For (b), I'm not sure with the recoloring part, if it's similar with the insertion of RB tree, I think the time complexity should be $O(max(h_1, h_2))$, because it need to go from the root of the shorter tree to the leftmost end, and then go from the root of the higher tree to the node with the same height of the shorter tree.

2. Supppose that we wish to keep track of a point of maximum overlap in a set of intervals, that is, a point with the largest number of intervals in the set that contain it.

   (a) Show that there will always be a point of maximum overlap that is an endpoint of one of the segments.

Proof: Assume none of endpoints is a point of maximum overlap. Let an interval of all contiguous points of maximum be $[x, y]$. As I assumed, no segment ends in $[x, y]$, in other words, all segments should end after y. Suppose the endpoint of the segment that ends first is z, and $z > y$. Then all other segments should end after z. Therefore, I can increase the interval of all contiguous points of maximum to $[x, z]$, and there's an endpoint on z. That's a contradiction! Hence, there will always be a point of maximum overlap that is an endpoint of one of the segments.

(b) Design a data structure that efficiently supports the operations INTERVALINSERT, INTERVALDELETE, and FIND-POM, which returns a point of maximum overlap.

First keep a red-black tree of all endpoints. Then about the additional information, we need three more. For an arbitrary node x:
1) Let $v(x) = 1$ if it's a low endpoint, and $v(x) = -1$ if it's a high endpoint. Then we can find when looking for the POM in a sorted array of endpoints, just add all these values $+1, -1$ until finding the maximum.
2) Let $sum(x) = sum(v(left(x))) + v(x) + sum(v(right(x)))$. It records the sum of all values of its all children plus itself.
3) Let $max(x) = \max\{max(left(x)), sum(v(left(x))) + v(x), sum(v(left(x))) + v(x) + max(right(x))\}$. If $max(x) == max(left(x))$, the POM is in the left subtree; If $max(x) == sum(v(left(x))) + v(x)$, the POM is x; if $max(x) == sum(v(left(x))) + v(x) + max(right(x))$, the POM is in the right subtree.
The operations INTERVALINSERT, INTERVALDELETE are simliar with the regular red-black tree, except we need to maintain these extra information, it still takes $O(h)$ time, because these extra information are only related with their left and right children. And the FIND-POM also takes $O(h)$ time, as the rule (3) above, we can look from the root to the right branches.

3. Consider the problem of neatly printing a paragraph on a printer, The input text is a sequence of n words of lengths $l_1, l_2, ..., l_n$, measured in characters. All words are alphanmeric and no punctuation is allowed. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each, Our criterion of "neatness" is as follows. If a given line contains words i through j and we leave exactly one space between words, the number of extra space characters at the end of the line is $extra[i, j] = M - j + i - \sum_{k=i}^{j} l_k$. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines.

(a) Explain why a dynamic programming solution is appropriate. In your answer, consider the two key ingredients of dynamic programming: optimal substructure and overlapping subproblems.

This problem of optimally printing n words can be splitted into a subproblem that optimally printing i words $(1 \leq i < n)$ and the rest lines. And because one problem that printing i words is based on the results of all subproblems that printing $j(1 \leq j < i)$ words, therefore many subproblems are referred many times.

(b) Give a formula for the cost $lc[i, j]$ of including a line containing words i through j in the sum we want to minimize. Make sure your formula handles properly the last line (when j = n) and the case where words i through j do not fit on a single line.

$$lc[i,j] = \begin{cases} 0, if j = n, extra[i,j] \geq 0 \\ extra[i,j], if j \neq n, extra[i,j] \geq 0 \\ \infty, if extra[i,j] < 0 \end{cases}$$

(c) Let $c[j]$ be the cost of an optimal arrangement of words 1 through j. The cost of the optimal arrangement we want is thus $c[n]$. Define $c[j]$ recursively in a form suitable for dynamic programming.

$c[j] = \min_{1<i<j}\{c[i] + lc[i,j]\}$

(d) Give a dynamic programming algorithm to print a paragraph of n words as neatly as possible.

Algorithm 1 DPCOST

```
1: procedure DPCOST(A, n, M)
2:     c[1] = 0
3:     for j ← 2 to n do
4:         c[j] ← ∞
5:         for i ← 2 to j do
6:             extra = M - j + i
7:             for k ← i to j do
8:                 extra ← extra - A[i]
9:             end for
10:            if extra ≥ 0 and j == n then
11:                lc ← 0
12:            end if
13:            if extra ≥ 0 and j ≠ n then
14:                lc ← extra
15:            end if
16:            if extra < 0 then
17:                lc ← ∞
18:            end if
19:            v ← c[i] + lc
20:            if v < c[j] then
21:                c[j] ← v
22:            end if
23:        end for
24:    end for
25:    return c
26: end procedure
```

Algorithm 2 PRINT

```
1: procedure PRINT(W, c, n)
2:     for i ← 1 to n do
3:         Print W[i] + ' '
4:         if i ≠ n and c[i] ≠ c[i+1] then
5:             Print endl
6:         end if
7:     end for
8: end procedure
```

(e) Analyze the running time and space requirements of your algorithm.

Running time is the sum of both algorithms above:
Time Complexity = $\sum_{j=2}^{n}\sum_{i=2}^{j}\sum_{k=i}^{j} 1+O(n) = \sum_{j=2}^{n}\sum_{i=2}^{j}(j-i)+O(n) = \sum_{j=2}^{n} O(n^2)+$ $O(n) = O(n^3) + O(n) = O(n^3)$.
Space requirement is an additional array:
Space Complexity = $O(n)$.

4. Earlier in the course we described a problem where a robot repeatedly visit a set P of n points, performing an assigned task at each point, The path followed by the robot is a loop and we wish to find a loop whose total length is small. Finding the shortest loop efficiently proved to be a difficult problem (one of the so called millennium problems). Instead, we considered various heuristics. Here we consider only loops that start at the rightmost point and proceed leftwards, visiting a subset of the points in decreasing x-coordinate order, until the leftmost point is reached. Then the robot turns around and visits the remaining points in increasing x-coordinate order. A loop with this property is called an LR-loop. We wish to find the shortest LR-loop using dynamic programming.

   (a) Assume $P = \{P_1, ..., P_n\}$ has been sorted by x-coordinate so that P_1 is the leftmost and P_n is the rightmost point. What points is P_1 connected to in an LR-loop? What can you say about those points if the loop is optimal?

   There should only be 2 points connected to P_1 in an LR-loop, they could be P_2 to P_n. If the loop is optimal, I think these points should be roughly divided to half and half, and the loop should look more like a circle.

   (b) Explain why a dynamic programming solution is appropriate. In your answer, consider the two key ingredients of dynamic programming: optimal substructure and overlapping subproblems.

   This problem can be splitted into optimal subproblems, when deciding the routine of P_i, it has existed an optimal subproblem with P_1 to P_i and P_n, we need to choose to put P_i on the upper half or the lower half. And every step depends on the last step, these last steps are overlapping subproblems.

(c) Give a recursive formula for the length of the shortest LR-loop in a form suitable for dynamic programming.

$l(i) = l(i-1) + \min_{2 \le j < i}\{d(j,i) + d(i,n) - d(j,n)\}$
$d(x, y)$ is getting the Euclidean distance of P_x and P_y.

(d) Write a dynamic programming algorithm to find the shortest LR-loop for the set $P = \{P_1, ..., P_n\}$.

Algorithm 3 DPCOST

```
1: procedure DPCOST(P, n)
2:     L[2] = d(1, 2) + d(2, n) + d(1, n)
3:     M[0] = 2
4:     for i ← 3 to n-1 do
5:         L[i] = ∞
6:         for j ← 2 to i-1 do
7:             v = L[i-1] + d(j, i) + d(i, n) - d(j, n)
8:             if v ¡ L[i] then
9:                 L[i] = v
10:                M[i-2] = j
11:            end if
12:        end for
13:    end for
14:    return L, M
15: end procedure
```

(e) Analyze the running time and space requirements of your algorithm.

Time complexity = $\sum_{i=3}^{n-1} \sum_{j=2}^{i-1} 1 = O(n)$.
Space complexity = $O(n)$.

5. Consider the following the task scheduling problem. You are given n tasks, competing for the same resource. Each task is specified by a pair (p_i, t_i) meaning that it requires p_i units of processing time and that it can start no earlier than t_i. Assume that active tasks can be suspended and finished later with no penalty. We are interested in finding solutions that minimize the sum of completion times of all tasks involved. For example if n = 2 with tasks (5, 2) and (3, 0) then a valid schedule runs task 2 from time 0 to time 2, suspend it, run task 1 to completion from time 2 to time 7, and finally complete task 2 from time 7 to time 8. The cost of this schedule is 7+8 = 15. Describe an efficient algorithm to find an optimal schedule, i.e., one that minimizes the sum of completion times. Make sure to prove that you algorithm is correct and analyze its running time.

Algorithm 4 SCHEDULING

```
1: procedure SCHEDULING(A, n)
2:     sort A by t
3:     sum ← 0
4:     for i ← 0 to n-2 do
5:         if A[i].t + A[i].p ≤ A[i+1].t then
6:             sum ← sum + A[i].t + A[i].p
7:         else
8:             if A[i].t + A[i].p ≥ A[i+1].t + A[i+1].p then
9:                 A[i].t = A[i+1].t + A[i+1].p
10:                A[i].p = A[i].p - (A[i+1].t - A[i].t)
11:                find a new position k for A[i] by binary search, and put A[i] in the new position
12:                i ← i - 1
13:            else
14:                sum ← sum + A[i].t + A[i].p
15:            end if
16:        end if
17:    end for
18: end procedure
```

The core part of this algorithm is :
When meeting a overlapped task, compare $p + t$ with it, if the current task is smaller, then complete the current one first, otherwise suspend the current task, run the new one, and update the t and p of the suspended task.

Proof of Correctness: if there's no overlapped task, then just run all tasks one by one per their arrival time. It's obviously optimal.
And it's also obvious that those tasks need to be executed by arrival time. The only problem is that it should suspend or not when encounter an overlapped task.
Suppose two overlapped tasks (p_i, t_i) and (p_j, t_j), let $t_i \leq t_j \leq t_i + p_i$.
The scenario is like task i is running, and then task j arrives.
If task i doesn't suspend, the sum of completion time of these two tasks is $s_1 = t_i + p_i + (t_i + p_i + p_j) = 2t_i + 2p_i + p_j$.
If task i suspends, then the sum of completion time of these two tasks is $s_2 = t_i + p_i + p_j + t_j + p_j = t_i + p_i + t_j + 2p_j$
If $s_1 > s_2$, then $s_1 - s_2 = (t_i + p_i) - (t_j + p_j) > 0 \implies t_i + p_i > t_j + p_j$. And similarly, if $s_1 < s_2$, $t_i + p_i < t_j + p_j$.
Then I can conclude that if $t_i + p_i > t_j + p_j$, task i should suspend to decrease the sum, otherwise task i should complete itself first. This is also the choice of my algorithm.

Running Time: For best case that no overlapped tasks, the running time is $O(n \log n)$ because of the sorting.
The difficult part is the worst case, think all tasks as segments, if all segments construct like a pyramid, it would be the worst case. In worst case, line 11 is the main problem, if A is a linear array, inserting $A[i]$ to a new position would also move all other elements in front of it. Then the time complexity would be $O(n^2)$. If A is some kind of container that inserting $A[i]$ only cost no more than $O(\log n)$ time (tree or hash table), then the time complexity would

also be $O(n \log n)$.