CSORW4231 HOMEWORK 5

Due Mon, Apr 10

Jun Hu

(jh3846)

Problem 1. Exercise 15.4-5 on Page 397: Longest monotonically increasing subsequence.

Solution.

Algorithm Assuming a sequence X, with n numbers.

LMIS(X)

```
1    copy X to X'
2    Y = QUICK-SORT(X') in increasing order
3    return LCS(X, Y)
```

Correctness LMIS(X) should return the 1 longest 2 monotonically increasing subsequence.

Let Z be a longest monotonically increasing subqeuence of X. If Z holds monotonically increasing, Z must be a subsequence of the sorted sequence Y. If Z is not a sebsequence of Y, and contains $x_i <= x_j$ which is not in Y. However, X still contains $x_i <= x_j$, if we sort X in increasing order Y', which is the same sequence of Y, Y' must contain $x_i <= x_j$, but Y doesn't, this is a contradiction!

Since Z must be a subsequence of Y, as the subsequence of X, we only need to invoke longest common subsequece LCS(X, Y) to obtain the longest subsequence.

In conclusion, LMIS(X) is the longest monotonically increasing subsequence holds.

Running-time Analysis of pseudocode:

Line 1 copy n numbers takes $O(n)$.

Line 2 QUICK-SORT on n numbers takes $O(n^2)$.

Line 3 LCS on 2 sequences of n numbers takes $O(n^2)$.

Therefore, LMIS(X) takes $O(n^2)$.

□

Problem 2. Problem 15-2 on Page 405: Longest palindrome subsequence.

Solution.

<u>Algorithm</u> Assuming a non-empty string sequence S, with n characters.
LPS(S)

```
reverse the sequence S to S'
new sequence Q = LCS(S, S')
if length of Q is odd, the central character is c, before c is
    subsequence M of Q, M' is reversed M
  return string McM'
if length of Q is even, the first half of Q is M, M' is
    reversed M
  return string MM'
```

<u>Correctness</u> When n = 1, suppose $S =< s_0 >$, so $LCS(S, S') = s_0$, s.t. $c = s_0$, M and M' are blank, $LPS(S) = McM' = s_0$, correct.

$n > 1$:

Either McM' or MM' is apparently 'palindromic'.

Assume R is a longest palindromic subsequence of S. according to the feature of palindrome, R is also the subsequence of S'. Length of LCS(S, S') is the upper bound of length of R. The length of LPS(S) equals to Q holds the 'longest'. Combining above, if McM' is a subsequence of S, then everything holds.

The next proof is taking the odd condition, for the even condition, just adding a pseudo center c' to transform to the odd condition. In short, parts split by c for S and S' are the reversed situation to find LCS(S_p, S'_p), but because the order, the results may not just the reverse, but using substitution to solve it. In details, denote notation X' to be the reversed of X: this will think of $S = XcY$ and $S' = AcB$, by noticing that $X' = B$ and $Y = A'$, while M is LCS(X, A) (the part before c), then M' is also a LCS(X', A') which is LCS(B, Y), so using M' to substitute the original LCS(Y, B) (the part after c), the McM' is still a LCS(S, S'). McM' holds a subsequence of S.

In conclusion, LPS(S) is a longest palindrome subsequence.

<u>Running-time</u> Analysis of pseudocode:

Reverse operation takes $O(n)$, condition IF takes $O(1)$, LCS on 2 sequences of n characters takes $O(n^2)$.

Therefore, LPS(S) takes $O(n^2)$.

□

Problem 3. Problem 15.3 on Page 405: Bitonic euclidean traveling-salesman problem.

Solution.

<u>Algorithm</u> First sort n points by x-coordinate, left to right, as $p_1, p_2, \dots, p_n$.
For $1 \leq i \leq j \leq n$, denote $\mathrm{BD}(i,j)$ be the shortest bitonic distance from p_1 though two different path to p_i and p_j, we first solve this sub-problem:

$$\mathrm{BD}(i,j) = \begin{cases} \overline{p_i, p_j} & \text{if } i = 1, j = 2 \\ \mathrm{BD}(i, j-1) + \overline{p_{j-1}, p_j} & \text{elif } i < j-1 \\ \min_{1 \leq k < j}(\mathrm{BD}(k,i) + \overline{p_k, p_j}) & \text{elif } i = j-1 \end{cases}$$

To obtain the shortest path of the input n points. We compute $\mathrm{BD}(n,n)$, which is the shortest bitonic distance of the n points, and in the meanwhile, for each pair (i,j), we store k in $\mathrm{P}(i,j)$, s.t. p_k is the neighbor of p_i to p_j, then retrieve the path from $\mathrm{P}(i,j)$.

<u>Correctness</u> An optimal solution to a problem (instance) contains optimal solutions to sub-problems. We need to prove the sub-problems.

1.For condition $i = 1, j = 2$, trivial.

2.Bitonic tour from $p_1, p_2, \dots, p_n$, then p_{n-1} and p_n must be neighbors. If there are p_m, p_l as neighbors of p_n, then the path will be somewhere like p_m, p_n, p_l and p_{n-1}, which contradicts the sorted points $x_{n-1} < x_n$. This explain the condition that $i < j-1$, $\overline{p_{j-1}, p_j}$ will always be the only path.

3.If $i = j-1$, there must be a point p_k s.t. $k < j-1$ path towards p_j, under this condition, we need to find the p_k to keep the path shortest.

In conclusion, above 3 on the algorithm holds for each recursion, so the algorithm holds.

<u>Running-time</u> Sorting takes $O(n \lg n)$, the bottom of the recursion takes $O(1)$, the main part is the times of recursion.
Roughly, j is from n to 2, and i is from $j-1$ to 1, the recursion will call $O(n^2)$ times.
Totally, the algorithm takes $O(n^2)$.

□

Problem 4. Exercise 15.1-2 on Page 370: Counterexample.

Solution.

length	$i =$	1	2	3
price	$p_i =$	2	50	60
density	$\frac{p_i}{i} =$	2	25	20

Let the rod to be length 3, the greedy algorithm will cut to rod = 2 and 1, which has price of 52, but the optimal solution is rod = 3, which has price of 60. This is a contradiction.

$\square$

Problem 5. Problem 17-2 on Page 473 (Skip c): Making binary search dynamic.

Solution.

a. In general, just do BST-SEARCH(T, key) on the A_i forest.

Algorithm **OBST-SEARCH(A_i, key)**

```
i = 0
while not find the key, and i <= k-1
  BST-SEARCH(Ai, key)
  i = i + 1
```

Correctness Obviously, because all the elements will be searched before the key is found. If the key is not in the forest, BST-SEARCH(T, key) will return null.

Running-time In the worst case, BST-SEARCH(A_i, key) is called from $i = 0$ to $k - 1$.
In the while loop, A_i size is 2^i BST-SEARCH(A_i, key) takes $O(\lg(2^i)) = O(i)$
Because $k = \lceil \lg(n+1) \rceil$, sum them all:

$$\sum_{i=0}^{k-1} O(i) = O(k^2) = O(\lg^2 n)$$

b. In general, inserting is adding another A_0 to the forest, merge them to A_1 if there is another identical A_0, do this until no such A_i.

Algorithm **OBST-SEARCH(A_i, key)**

```
i = 0
while exists two identical A(i)s
  BST-MERGE two A(i) into A(i+1)
  i = i + 1
```

Correctness Obviously, because the new element will eventually end up somewhere in the A_i.

Running-time In the worst case, the i added from 0 to $k - 1$, and each merge from i to $i + 1$ take $O(i + 1) = O(i)$, $k = \lceil \lg(n+1) \rceil$, so similarly,

$$\sum_{i=0}^{k-1} O(i) = O(k^2) = O(\lg^2 n)$$

For the amortized time, notice that the merge occurs for n_0 every time, but for n_1 every $2^t h$ time, ..., as for n_{k-1}, it is 2^k time. Suppose the insertions operate m times. Totally the running time is:

$$\sum_{i=0}^{k-1} \lfloor \frac{m}{2^i} \rfloor O(2^{i+1}) \leq 2mO(k) = m \lg n$$

Each insertion operation, takes

$$\frac{m \lg n}{m} = \lg n$$

□

Problem 6. Problem 17-3 on Page 473: Amortized weight-balanced trees.

Solution. In general, to minimize the average completion time, we always want to finish the task which is the shortest at the moment.

a. Do in-order walk of the subtree rooted at x and store the sorted array in $\Theta(x.size)$ space.
Take the median of the array as the root. (takes O(1) running time).
Recursively repeat median picking on the new left and right subtrees.
Each operation recurrence guarantees $\frac{1}{2}$-balanced.
Running time: $T(x.size) = 2T(\frac{1}{2}x.size) = \Theta(x.size)$

b. Tree is split associated with α. For each iteration, in worst case, search leads to the larger subtrees with αn nodes. Because $\frac{1}{2} \leq \alpha < 1$, the running time is $T(n) = T(\alpha n) + 1 = O(\lg n)$

c. By definition, as $\Delta(x) \geq 0$ and c is a sufficiently large constant that depends on α non-negative, so potential is always non-negative.

The summation is applies when $\Delta(x) \geq 2$. Suppose $x.left.size$ is larger (counterpart is the same in general),

$$\begin{aligned} x.left.size - x.right.size &\geq 2 \\ x.left.size - (x.size - x.left.size - 1) &\geq 2 \\ x.left.size &\geq \frac{1}{2} + \frac{1}{2}x.size \end{aligned}$$

Which is a contradiction to $\frac{1}{2}$-balanced tree that $x.left.size \leq \frac{1}{2}x.size$, so the summation won't apply, the potential will be 0.

d. Suppose left subtree is larger.
$Delta(x) = x.left.size - x.right.size = \alpha x.size - ((1-\alpha)x.size - 1) = m(2\alpha - 1) + 1$

The amortized cost of rebuilding the subtree is the actual cost with their potential difference: $\widehat{c_i} = c_i + \Phi_i - \Phi_{i-1}$

To take $O(1)$ time, we need $m + \Phi_i - \Phi_{i-1} = O(1)$, because we rebuilt it to be $\frac{1}{2}$-balanced, and we already know the potential after rebuilding is 0, so we get $m = \Phi_{i-1} + O(1) \leq \Phi_{i-1} \leq c(m(2\alpha - 1) + 1)$, which means:

$$c \geq \frac{1}{2\alpha - 1 + \frac{1}{m}} \geq \frac{1}{2\alpha - 1}$$

e. Similarly, the actual cost plus the potential difference is the amortized cost given by $\widehat{c_i} = c_i + \Delta\Phi_i$ for insertion or deletion.

And insertion or deletion are based on search, witch takes $O(\lg n)$, as shown in answer **b.**.

As shown in **d.**, it takes $O(1)$ to rebuilt, for node i in the path of root to insert of delete x node, $\Delta(i)$ will increase $O(1)$, the worst case is this happens for every such node, as shown in **b.**, there are $O(\lg n)$ (depth of the tree) such nodes, such that $\Delta\Phi_i = O(\lg n)$. In sum, the running time takes $O(\lg n)$.

□