Problem Set 5 Solutions

This problem set is due **at 11:59pm** on **Friday, March 20, 2015.**

Exercise 5-1. Read CLRS, Chapter 11.

Exercise 5-2. Exercise 11.3-5.

Exercise 5-3. Read CLRS, Chapter 14.

Exercise 5-4. Exercise 14.3-3.

Exercise 5-5. Read CLRS, Chapter 15.

Exercise 5-6. Exercise 15.1-4.

Exercise 5-7. Exercise 15.2-4.

Exercise 5-8. Exercise 15.4-5.

Problem 5-1. New Operations for Skip Lists [25 points]

This problem will demonstrate that skip lists, and some augmentations of skip lists, can answer some queries about "nearby" elements efficiently. In a dynamic-set data structure, the query FINGER-SEARCH(x, k) is given a node x in the data structure and a key k, and it must return a node y in the data structure that contains key k. (You may assume that such a node y in fact appears in the data structure.) The goal is for FINGER-SEARCH to run faster when nodes x and y are nearby in the data structure.

(a) [12 points] Write pseudocode for FINGER-SEARCH(x, k) for skip lists. Assume all keys in the skip list are distinct. Assume that the given node x is in the level-0 list, and the operation should return a node y that stores k in the level-0 list.

Your algorithm should run in $O(\lg m)$ steps with high probability, where $m = 1 + |\text{rank}(x.\mathit{key}) - \text{rank}(k)|$. Here, $\text{rank}(k)$ refers to the rank (index) of a key k in the sorted order of the dynamic set (when the procedure is invoked). "High probability" here is with respect to m; more precisely, for any positive integer α, your algorithm should run in $O(\lg m)$ time with probability at least $1 - \frac{1}{m^\alpha}$. (The constant implicit in this O may depend on α.) Analyze your algorithm carefully.

In writing your code, you may assume that the implementation of skip lists stores both a $-\infty$ at the front of each level as well as a $+\infty$ as the last element on each level.

Solution: For each node x, we use fields:

$key[x]$ the key stored in node x
$level[x]$ the level of the linked list containing x
$next[x]$ the next element in the list containing x
$prev[x]$ the previous element in the list containing x
$up[x]$ the element in the level $level[x] + 1$ list containing key[x]
$down[x]$ the element in the level $level[x] - 1$ list containing key[x]

The procedure call is FINGER-SEARCH(x, k), where x is a level-0 node in the skip list and $k \neq key[x]$ is a key that we assume appears in the skip list. The procedure returns a level-0 node y such that $k = key[y]$.

We present code for the case where $k > key[x]$; the case where $k < key[x]$ is symmetric. The algorithm proceeds in two phases. In the first phase, we ascend levels as quickly as possible, and move to the right, going as far as possible while maintaining the invariant: $key[next[z]] \leq k$.

```
FINGER-SEARCH$(x, k)$
$z = x$
Do forever:
if $up[z] \neq$ NIL and $key[next[up[z]]] \leq k$
    $z = up[z]$
else if $key[next[next[z]]] \leq k$
    $z = next[z]$
else break
```

In the second phase, we move to the right as quickly as possible, and descend, until we reach the level 0 node containing key k. This maintains the invariant $key[z] \leq k$.

```
Do forever:
if $key[next[z]] \leq k$
    $z = next[z]$
else if $level[z] \neq 0$
    $z = down[z]$
else return $z$
```

To see why this takes time $O(\lg m)$ with high probability, we first prove a high-probability bound on the highest level reached during the finger search.

Lemma 1 *Let β be any positive integer. With probability at least $1 - \frac{1}{m^\beta}$, the highest level reached by* FINGER-SEARCH *is at most $(\beta + 1) \lg m)$.*

Proof. The only keys that appear at nodes that the search visits are in the range from $key[x]$ to k, inclusive. There are m keys in this range.

For any c, the probability that any particular key in this range appears at a node in a list at level $> c \lg m$ is at most $\frac{1}{2^{c \lg(m)}} = \frac{1}{(m)^c}$. Therefore, by a union bound, the probability that any of these m keys appears at a level $> c \lg m$ is at most $\frac{m}{m^c} \leq \frac{1}{m^{c-1}}$. Taking $c = \beta + 1$ gives the result.

Now fix $c = \beta + 1$. To analyze the time for FINGER-SEARCH(x, k), we use the following lemma (from lecture, and from Problem Set 4):

Lemma 2 *Let β be any positive integer. With probability at least $1 - \frac{1}{m^\beta}$, the number of coin tosses required to obtain $c \lg m$ heads is $O(\lg m)$. (The constant in the $O()$ depends on $c and \beta$.)*

Now consider the behavior of the algorithm. During the first phase of execution, identify "heads" with moving up a level and "tails" with moving to the right. During the second phase, identify "heads" with moving down a level and "tails" with moving to the right. The lemma implies that, with probability at least $1 - \frac{1}{m^\beta}$, both of the following happen:

1. The first $c \lg m$ levels of the upward part of the search (or the entire upward part of the search if it finishes within $c \lg m$ levels), take time $O(\lg m)$. (The behavior is like the reverse of a skip list search operation conducted from right to left, truncated to consider only the bottom $c \lg m$ levels.)
2. The final $c \lg m$ levels of the downward part of the search take time $O(\lg m)$. (The behavior is like the last steps of an ordinary skip list search operation.)

To complete the proof that the algorithm takes time $O(\lg m)$ with high probability, fix $\beta = \alpha + 1$. A union bound tells us that, with probability at least $1 - \frac{2}{m^\beta}$, both of the following hold: the number of levels visited by the search is at most $c \lg m$, and the bottom $c \lg m$ levels of the search complete within time $O(\lg m)$. In other words, with probability at least $1 - \frac{2}{m^\beta}$, the entire search completes within time $O(\lg m)$. Since $\frac{2}{m^\beta} \leq \frac{1}{m^\alpha}$, this implies that with probability at least $1 - \frac{1}{m^\alpha}$, the search completes within time $O(\lg(m))$.

Another query on a dynamic set is RANK-SEARCH(x, r): given a node x in the data structure and positive integer r, return a node y in the data structure that contains the key whose rank is $\text{rank}(x) + r$. You may assume that such a node appears in the data structure. Rank search can be implemented efficiently in skip lists, but this requires augmenting the nodes of the skip list with new information.

(b) [6 points] Define a way of augmenting skip lists that will support efficient rank search, and show that your augmentation does not increase the usual high-probability order-of-magnitude time bounds for the SEARCH, INSERT, and DELETE operations.

Solution:

Add another field to each node x in the skip list:

$count[x]$ the number of nodes in level 0 of the skip list that have keys k with $key[x] < k \leq key[next[x]]$.

This addition does not affect the SEARCH operation.

For INSERT(k), while finding the right location for k in the level 0 list, we add 1 to the count of every node from which we move down. When we insert the new key in a node x in the level 0 list, we set $count[x] = 1$. Then, when we insert key k in a node y in a higher-level list, we compute $count[y]$ by adding the counts for all nodes in the next lower-level list that contain keys in the range $[k, \ldots, key[next[y]])$. We also reduce $count[prev[y]]$ by the new value of $count[y]$. The additional cost for INSERT is $O(\lg n)$, with high probability with respect to n, as we can see by another application of Lemma **??**.

For DELETE(k), for each node y that gets removed, at any level, we increase $count[prev[y]]$ by $count[y] - 1$.

(c) [7 points] Now write pseudocode for RANK-SEARCH(x, r) for skip lists. Again assume that all keys in the skip list are distinct. Assume that the given node x is in the level-0 list, and the operation should return a node y in the level-0 list.

Your algorithm should run in $O(\lg m)$ steps with high probability with respect to $m = r+1$. More precisely, for any positive integer α, your algorithm should run in $O(\lg m)$ time with probability at least $1 - \frac{1}{m^\alpha}$. Analyze your algorithm carefully.

Solution: This is similar to part (a). We again use a two-phase algorithm for which the first phase ascends as quickly as possible and the second phase moves to the right as quickly as possible.

The first phase maintains the invariant: $count[z] \leq rem$.

```
RANK-SEARCH($x, r$)
$z = x$
$rem = r$
Do forever:
if $up[z] \neq$ NIL and $count[up[z]] \leq rem$
    $z = up[z]$
else if $count[z] + count[next[z]] \leq rem$
    $rem = rem - count[z]$
    $z = next[z]$
else break
```

At the end of the first phase, we know that: $count[z] \leq rem < count[z] + count[next[z]]$. Now the second phase moves right as quickly as possible then down.

```
Do forever:
if $count[z] \leq rem$
    $rem = rem - count[z]$
    $z = next[z]$
else if $level[z] \neq 0$
    $z = down[z]$
else return $z$
```

The analysis is very similar to that for part (a).

Problem 5-2. Choosing Prizes [25 points]

In this problem, you are presented with a collection of n ***prizes*** from which you are allowed to select at most m prizes, where $m < n$. Each prize p has a nonnegative integer ***value***, denoted $p.value$. Your objective is to maximize the total value of your chosen prizes.

The problem has several variations, described in parts (a)–(d) below. In each case, you should give an efficient algorithm to solve the problem, and analyze your algorithm's time and space requirements.

In parts (a)–(c), the prizes are presented to you as a sequence $P = \langle p_1, p_2, \ldots, p_n \rangle$, and your algorithm must output a *subsequence* S of P. In other words, the selected prizes S ($|S| = m$) must be listed in the same order as they are in P.

(a) [4 points] Give an algorithm that returns a subsequence $S = \langle s_1, s_2, \ldots \rangle$ of P of length at most m, for which $\sum_j s_j.value$ is maximum. Analyze your algorithm in terms of n and m.

Solution: This simply involves outputting the m prizes with the largest values. We use the SELECT algorithm from lecture to find the $k = (n - m - 1)$-th order statistic of the input list. We then partition the input list around the k-th rank prize and output all prizes to its right, there should be exactly m of them.

SELECT and partitioning both take $O(n)$ time for a total runtime of $O(n)$. An in-place implementation of SELECT only uses an additional $O(1)$ space.

Solution: (Aternate, Sub-Optimal) This can also be done in time $O(n \lg m)$ and space $O(m)$, by scanning a list representing P, using a min-heap to keep track of the m highest-valued prizes seen so far.

In more detail: For the first m elements of the prize list, we just insert the elements into the min-heap. For each subsequent element, we compare the value of the new prize with the value of the minimum (top) element in the heap. If the new element is greater, replace the old element with the new one and adjust its position. This takes time $O(\lg m)$, for a total of $O(n \lg m)$ for processing the entire prize list P.

In order to output the final list S efficiently, it is helpful to augment this construction with two-way pointers between the elements of P and the corresponding heap nodes. Then after we have the final heap, we can simply scan P once again, in order, outputting the values that appear in heap nodes. This part takes time $O(n)$.

The total time complexity is $O(n \lg m)$. Since this requires a heap of size m, the space required is $O(m)$.

(b) [7 points] Now suppose there are two types of prizes, type A and type B. Each prize's type is given as an attribute $p.\mathit{type}$. Give an algorithm that returns a subsequence $S = \langle s_1, s_2, \ldots \rangle$ of P of length at most m, for which $\sum_j s_j.\mathit{value}$ is maximum, subject to the new constraint that, in S, *all the prizes of type A must precede all the prizes of type B*. Analyze your algorithm in terms of n and m.

Solution: Use Dynamic Programming, based on prefixes of the prize list. We define two functions, $C_A(i,k)$ and $C_B(i,k)$, for $0 \le i \le n$ and $0 \le k \le m$:

- $C_A(i,k)$ is the maximum total value achievable by a subsequence of $\langle p_1, \ldots, p_i \rangle$ consisting of at most k type A prizes.
- $C_B(i,k)$ is the maximum total value achievable by a subsequence of $\langle p_1, \ldots, p_i \rangle$ consisting of any number (possibly 0) of type A prizes followed by any number (possibly 0) of type B prizes, such that the total number of prizes is at most k.

To compute the values of these functions, we proceed iteratively, in terms of successively larger values of i. For the base case, we have $C_A(0,k) = C_B(0,k) = 0$ for every k; that is with no prizes to choose from, we can't achieve a positive value. Inductively, we compute the values of $C_A(i+1,k)$ and $C_B(i+1,k)$, for every i, $0 \le i \le n-1$, and every k, in terms of values of $C_A(i,*)$ and $C_B(i,*)$.

For C_A, we define $C_A(i+1,0) = 0$; that is, without being allowed to select any prizes, we can't achieve a postive value. For $k \ge 1$, we define $C_A(i+1,k)$ using two cases:

1. If $p_{i+1}.\mathit{type} = A$, then

$$C_A(i+1,k) = \max\left(p_{i+1}.\mathit{value} + C_A(i,k-1), C_A(i,k)\right).$$

   The first expression within the max corresponds to selecting the new prize, and the second expression corresponds to not selecting it.

2. If $p_{i+1}.\mathit{type} = B$, then

$$C_A(i+1,k) = C_A(i,k).$$

   Here there is no choice—we cannot select the new prize since C_A considers only sequences of type A prizes.

For C_B, we define $C_B(i+1,0) = 0$, and for $k \ge 1$, we define $C_B(i+1,k)$ using two cases:

1. If $p_{i+1}.\mathit{type} = A$, then

   $$C_B(i+1,k) = \max\left(p_{i+1}.\mathit{value} + C_A(i,k-1), C_B(i,k)\right).$$

   The first expression within the max corresponds to selecting the new prize; if we do this, then we can consider only sequences consisting of type A prizes up to position i. The second expression corresponds to not selecting thte new prize.
2. If $p_{i+1}.\mathit{type} = B$, then

   $$C_B(i+1,k) = \max\left(p_{i+1}.\mathit{value} + C_B(i,k-1), C_B(i,k)\right).$$

The best total prize value for the entire prize sequence is $C_B(n,m)$.

There are $(n+1)\cdot(m+1)$ subproblems, and each subproblem takes $O(1)$ time to solve. Therefore, the overall running time of the DP is $O(m\cdot n)$. In this iterative construction, we need only store the values of the functions for index i when computing the values for $i+1$, so the space requirement is $O(m)$.

So far, this returns only the maximum achievable prize value. In order for the DP to return the actual prize selection sequence, we let each subproblem entry contain a pointer to a linked list containing a prize sequence that led to the prize value recorded for that subproblem. The time complexity is still $O(m \cdot n)$, since we can construct each new list simply by (possibly) appending one item to an old list. Since each new subproblem adds only a constant number of nodes to the collection of lists, the total space is $O(m \cdot n)$.

(c) [7 points] As in part (a), there is only one type of prize. Give an algorithm that returns a subsequence $S = \langle s_1, s_2, \ldots \rangle$ of P of length at most m, for which $\sum_j s_j.\mathit{value}$ is maximum, subject to the new constraint that, in S, *the values of the prizes must form a non-decreasing sequence*. Analyze your algorithm in terms of n and m.

Solution: Use Dynamic Programming. As in part (b), we based the construction on prefixes of the prize list. This time, we define one function, $P(i,k,z)$, for $0 \le i \le n$, $0 \le k \le m$, and z any integer in $\{p_i.\mathit{value} \| 1 \le i \le n\}$:

- $P(i,k,z)$ is the maximum total value achievable by a subsequence of $\langle p_1, \ldots, p_i \rangle$ consisting of at most k prizes, in which the successive values of the chosen prizes do not decrease and all chosen prizes have values $\le z$.

To compute the values of P, we proceed iteratively, in terms of successively larger values of i. For the base case, we have $P(0,k,z) = 0$ for every k and z. We compute the values of $P(i+1,k,z)$ in terms of values of $P(i,*,*)$, as follows.

We define $P(i+1,0,z) = 0$ for every z. For $k \ge 1$, we define $P(i+1,k,z)$ using two cases:

1. If $p_{i+1}.\mathit{value} > z$ then

   $$P(i+1,k,z) = P(i,k,z).$$

   In this case the new value cannot be used, since all chosen values must be $\le z$.

2. If $p_{i+1}.value \leq z$ then

$$P(i+1, k, z) = \max\left(p_{i+1}.value + P(i, k-1, p_{i+1}.value), P(i, k, z)\right).$$

Here, the first expression corresponds to selecting the new prize; if we do this, then the values of all the previous prizes must be $\leq p_{i+1}.value$. The second term represents the option of not selecting the new prize.

The best total prize value for the entire prize sequence is $P(n, k, maxZ)$, where $maxZ$ is the maximum of the prize values, $maxZ = \max_{1 \leq i \leq n} p_i.value$.

There are $(n+1) \cdot (m+1) \cdot n$ subproblems, and each subproblem takes time $O(1)$. Therefore, the overall running time of the DP is $O(m \cdot n^2)$. In this iterative construction, we need only store the values of the functions for index i when computing the values for $i+1$, so the space requirement is $O(m \cdot n)$.

As in part (b), in order for the DP return the actual prize selection sequence, we let it maintain pointers to linked lists containing the prize sequences. The new time complexity is still $O(m \cdot n^2)$. Since each new subproblem adds only a constant number of nodes to the collection of lists, the total space is $O(m \cdot n^2)$.

In part (d), the prizes are represented by a *rooted binary tree* T, with root vertex r, where each vertex u has an associated prize, $u.prize$. Let P be the set of prizes in the tree. As before, each prize p has a nonnegative integer attribute $p.value$.

(d) [7 points] Give an algorithm that returns a set S of at most m prizes for which $\sum_{s \in S} s.value$ is maximum, subject to the new constraint that, for any $s \in S$ that is associated with a non-root node u of T, the prize at node $u.parent$ is also in S. (This implies that the selected prizes must be associated with nodes that form a connected subtree of T rooted at r.)

Solution: Use Dynamic Programming, based on rooted subtrees. For any vertex u, let T_u be the subtree rooted at u. We define $C(u, k)$, for any vertex u, and any integer k where $0 \leq k \leq m$:

- $C(u, k)$ is the maximum total value achievable by a set S_u of at most k prizes from subtree T_u, subject to the constraint that for any $s \in S_u$ at vertex $v \neq u$ in T_u, the prize at node $v.parent$ is also in S_u.

We can define C recursively as follows. As a base case, we define $C(u, 0) = 0$ for every u. We also extend C and define $C(NIL, j) = 0$ for every j: a nonexistent node cannot contribute any value. For vertex u and $0 < k \leq m$, define:

$$C(u, k) = \max_{0 \leq j \leq k-1}(u.prize.value + C(u.left, j) + C(u.right, k-1-j)).$$

That is, we consider the best selection obtainable by choosing the prize at the root node u and splitting up the choice of the remaining $k-1$ prizes between the two subtrees. The best total value for the entire input binary tree T is $C(r, m)$.

To compute the $C(u, k)$ values, the algorithm can perform a Depth-First Search of the tree and compute the values of $C()$ in reverse DFS order. There are $n \cdot (m+1)$ subproblems and each subproblem takes $O(m)$ time because $k \leq m$, for a total time complexity of $O(n \cdot m^2)$. For each subproblem $C(u, k)$, we need only keep track of $C(u.\mathit{left}, j)$ and $C(u.\mathit{right}, j)$ for all values of j, $0 \leq j \leq k-1$; therefore, at worst, we need to store $O(m)$ values per subproblem for a total space requirement of $O(m \cdot n)$.

In order to return the actual set of prizes instead of just the largest total value, we keep track of some additional information for each subproblem: the actual value of j used to obtain the split of k yielding the largest value. (If two different values of j yield the largest value, we pick one arbitrarily.) Once we have these, we can traverse the tree T from the top down, say, in breadth-first order, assigning to each node u a count of the number of resources that will be chosen from T_u in an optimal selection. While doing this, we output $u.\mathit{prize}$ for every node u that gets assigned a nonzero count.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015