Problem Set 3

This problem set is due at the beginning of class on Thursday, March 20, 2003. Note that you have three weeks to do it in.

Each problem is to be done on a separate sheet (or sheets) of paper. Mark the top of each sheet with your name, 6.046J/18.410J, the problem number, your recitation section, the date, and the names of any students with whom you collaborated.

Problem 3-1. Priorities

Brian Dean is a VERY busy TA. He often has so many things to do he can’t even figure out which one he should do first. Nitin Thaper has offered to help him by assigning 'priority' numbers to each of Brian’s tasks. Nitin guarantees no two tasks will have the same priority. Now, content with at least having priorities, Brian writes all his tasks down on paper, and simply works on the one that is most important until he finishes it.

As new tasks appear, Nitin is more than happy to assign a ‘priority’ number to each of them. But Brian would like to be somewhat more efficient in how he processes all that data. Assume that there are $n$ tasks Brian is working on. Each has an integer associated with them that represents the priority. Your task is to analyze the running time of each of these ideas (do not provide an algorithm unless we ask how):

(a) If Brian were to store all his tasks in an array, along with the priority values assigned to them:

- How long does it take to search for a task of specific priority?
- How long does it take to add one more task?
- How long does it take to delete a task? (You can assume you already know where the task is, so searching is unnecessary). Explain how you would do so.
- How long does it take to find the most important task?

Solution:

Searching for a task of specific priority takes $O(n)$ for an array where $n$ is the size of the array.

Adding one more task takes $O(1)$ assuming space was already allocated. If space was not already allocated, by using the doubling the array trick from recitation, adding one more task takes $O(1)$ amortized.

Deleting a task takes $O(1)$. We can switch that task with the last task and then delete the last entry of the array in $O(1)$.

Finding the most important task takes $O(n)$ by traversing the array and computing the task with maximum priority.
(b) If Brian were to store all his tasks in a sorted array, along with the priority values assigned to them (the sorting is done based on priority):

- How long does it take to search for a task of specific priority?
- How long does it take to add one more task?
- How long does it take to delete a task?
- How long does it take to find the most important task?

**Solution:** Searching for a task of specific priority takes $O(\log n)$ for a sorted array where $n$ is the size of the array. To accomplish this we use binary search.

Adding one more task takes $O(n)$. To do this we use the same process as seen in insert sort when adding one more element. Adding one more element is exactly one iteration of insert sort.

To delete one task takes $O(n)$, by doing the reverse of adding one more task. We find the element, delete it and then we have to shift the subarray to the right of that position to the left by one position.

Finding the most important task takes $O(1)$ since it would always be the first or the last task in the array, depending on the order we sorted the elements.

(c) Mihai (who loves complicated ideas) suggests the following: Store the data in multiple arrays. Let $k = \lceil \lg(n + 1) \rceil$. Let the binary representation of $n = \langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. We will keep $k$ sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k-1$, the length of array $A_i$ is $2^i$. Each array is either full or empty, depending of whether $n_i = 1$ or $n_i = 0$. The total number of elements in all $k$ arrays is therefore $\sum_{i=0}^{k-1} n_i 2^i = n$. Each of these arrays is kept sorted, but there is no relationship between elements in different arrays.

- How long does it take to search for a task of specific priority?
- How long does it take to add one more task? Analyze the worst-case for an addition, and analyze the amortized time for a continuous sequence of additions (no deletions in the middle), starting from an empty data structure. Describe how one performs an addition.
- How do you perform a deletion in time $\Theta(n)$?
- How long does it take to find the most important task? Describe how one does this.

**Solution:**

Searching for a specific element takes $O(\log^2 n)$. We need to search in the array for every level. Total time is $\sum_{i=1}^{\log n} i = O(\log^2 n)$

To add one more task takes $O(n)$ since we may have to merge 2 sorted arrays at every level. This takes time linear in the size of the array per level, thus $O(n)$ total time. Amortized, starting for an empty data structure, and inserting $n$ elements, we will have to merge at level 0, $n/2$ times, at level 1, $n/4$ times, and so on. Therefore, the total time is $n/2 + 2n/4 + 4n/8 + \ldots$. There are $O(\log n)$ levels, thus this adds up to $O(n \log n)$. 

To perform deletion in $O(n)$ we delete from the lowest level which is not empty. We delete the element, which takes $O(n)$ and then we will have to move the elements of the array to the lower levels, and that takes $O(n)$ time by just completing the levels in order.

Finding the most important task takes $O(\log n)$. We compute the most important task for every level ($O(1)$ per level) and then we compute the most important task of these $O(\log n)$ tasks.

(d) Yoav (whom we last saw sorting nuts and bolts) suggests that Brian store his tasks in a balanced BST, with the keys being the priorities. Additionally, one more pointer should be used to keep track of the position in the tree of the item with the highest priority:

• How long does it take to search for a task of specific priority?
• How long does it take to add one more task? Describe how one does this.
• How long does it take to delete a completed task? Describe how.
• How long does it take to find the most important task?

Solution:
Searching for a task of specific priority takes $O(\log n)$.
Adding one more task takes $O(\log n)$. We add the task using the algorithm from the lecture. Updating the pointer takes $O(1)$.
Deleting one task takes $O(\log n)$. We delete the task using the algorithm from the lecture. Updating the pointer takes $O(1)$.
Finding the most important task takes $O(1)$, by checking the pointer to the item with highest priority.

Problem 3-2. Dynamic random number generator
You want to build a dynamic random number generator. The generator should maintain an array $W[1...n]$ of non-negative (but not necessarily integer) weights. The generator should support the following operations:

• $Modify(i, w)$: assigns $W[i] = w$
• $Generate$: returns a random number from $\{1...n\}$, such that the probability of returning $i$ is equal to $\frac{W[i]}{\sum_j W[j]}$

Show how to implement the generator such that both operations take $O(\log n)$ time. You can use procedure $Rand(a, b)$, which (in unit time) returns a number $r$ chosen uniformly at random from the (continuous) interval $[a, b]$. The initial state of the data structure should be defined by weights $W[1] = 1$, $W[i] = 0$, for $i > 1$, but you can ignore the time needed to initialize the data structure.

Solution: The solution for this problem is similar to the BST augmentation which has been
covered in recitations. We augment the tree in the following way: for every node we keep
the size of the subtree rooted at that node, where every node has a weigh. To avoid proving
that insertions, deletions, and rotations still work, we notice that the data structure is rather
static and use the following trick: at the initialization step we insert \( n \) nodes in the tree each
with weight 0. We balance the tree using any of the methods from the book. We notice that
this tree is static now, and we never perform insertions or deletions. The operation we have
to implement is changing of weights. When we change the weight of a node \( x \), we have to
update the size of every subtree. We notice that only the sizes of the subtrees rooted at the
nodes on the path from \( x \) to the root change by the difference between the new weight and
the old weight of \( x \). Thus, we can update this values in \( O(\log n) \). Therefore, when we modify
the weight of a node \( i \), we search for the node and update the tree, in total time \( O(\log n) \).

To generate a random number \( \{1 \ldots n\} \), such that the probability of returning \( i \) is equal to
\[
\frac{w[i]}{\sum_{j=1}^{n} w[j]},
\]
we first generate a random number \( z \) between 0 and \( \sum_{j=1}^{n} w[j] \). We return an element
\( x \) for which \( \sum_{j=1}^{z-1} W[j] < z \) and \( \sum_{j=1}^{z} W[j] \geq z \). The probability of return the element \( y \) of
weight \( W[y] \) is exactly \( \frac{w[y]}{\sum_{j=1}^{n} w[j]} \). The procedure of returning this element is very similar to
the procedure of returning an element of rank \( k \) which has been covered in recitation. We
generate \( z \) between 0 and \( \sum_{j=1}^{n} W[j] \) and we begin at the root. Let \( i \) be the current node. Let
\( A \) be the size of the left subtree. If \( A > z \) then we recurse in that subtree. If \( A + W[i] < z \)
then we recurse on the right subtree with \( z' = z - A - W[i] \). Otherwise we return \( i \).

Claim 1 We return an element \( x \) for which \( \sum_{j=1}^{z-1} W[j] < z \) and \( \sum_{j=1}^{z} W[j] \geq z \).

Proof. If the element we’re looking for in on the left subtree then the sum of the weights
of the elements of the left subtree is greater than \( z \), so we can recurse on the left subtree.
If the elements is on the right subtree then the sum of the weights of the left subtree plus
the weight of the root has to be less than \( z \), so we can recurse on the right subtree. If the
element we’re looking for is the current node then the sum of the weights of the left subtree
is less than \( z \) and the sum of the weights of the left subtree plus the weight of the root is
greater or equal to \( z \). In this case, we correctly return the root.

Since the tree is balanced, generate finishes after \( O(\log n) \) steps.

Problem 3-3. Product Finder

The following problem should be highly reminiscent of problem 1-3.

Design an algorithm, which, given an input array \( A[1], \ldots, A[n] \) of different integers from the
range \( \{1 \ldots n^4\} \), and a target integer value \( x \), prints all pairs \((i, j)\) such that \( A[i] + A[j] = x \).
Your algorithm should have an expected running time of \( O(n) \). Your algorithm should
allocate no more space than \( O(n) \).

You do not need to prove correctness, but you should justify the running time. Note that
the running time is different from before. Also, again note that we want the indices, not the
values in the array.
**Food for thought:** Assume now that you would like to check if there is any triple \((i, j, k)\) such that \(A[i] + A[j] + A[k] = x\). Can you design an algorithm that solves this task in time \(O(n^{2-\epsilon})\) for some constant \(\epsilon > 0\)? Note: none of us can. If you discover one, apply for a professor position at MIT immediately.

**Solution:** There are two main ideas on how to solve this problem. One idea is to hash the numbers and their complement \((A[i] and x - A[i])\) and to check for every collision of \(a\) and \(b\) if \(a + b = z\). Another idea is to sort the array using radix sort and then to use two pointers traversing the array one from 0 to \(n\) and the other one from \(n\) to 0 checking for \(a + b = z\).

For simplicity, we’ll show a slightly different solution. This solution is harder to implement but easier to understand/visualize. First, we can use radix sort to sort the elements of the array in time \(O(n)\) since every element is an integer from 0 to \(n^{O(1)}\). This has been covered in the class. We will keep track of the original indices in \(A'\). In the second step we will compute the array \(B\) as the complement of \(A\) \((B[i] = x - A[i])\). We will keep track of the original indices in \(B'\). \(B\) will be sorted in decreasing order. In the third step we will invert \(B\) (and \(B'\)). In the forth step we will merge \(A\) and \(B\) in \(C\) (keeping track of the indices in \(C'\)). In the last step for every two consecutive elements with the same value \(a \leq x/2\), we know that \(a\) came from \(A\) and the other \(a\) came from \(B\) so the sum of these two elements is \(x\). By checking \(a \leq x/2\) we make sure we don’t double count. Because we keep track of the original indices we can print their original indices \((i, j)\) such that \(A[i] + A[j] = x\). This procedure prints all the pairs because if \(A[i] + A[j] = x\) (and let’s say \(A[i] < A[j]\)) then \(A[i]\) will appear twice in \(C\) and we will print the pair \((i, j)\). All the steps of the algorithm take \(O(n)\).

**Problem 3-4. Professor Indyk’s Sock Drawer**

Professor Indyk does his laundry fortnightly. Because he hates matching socks after the laundry session, he instead grabs all the socks and throws them in his special sock drawer. The reason his sock drawer is special is because until Professor Indyk pulls out a sock, he has no clue which one he will end up with, and the sock is always a random sock from those remaining.

Say Professor Indyk has \(n\) pairs of socks, each unique from the other pairs. Professor Indyk’s idea is to pull out a sock, then pull another, and on and on, until he finds at least one pair that matches, at which point he returns all the other socks to the drawer, and puts on the matching pair.

Clearly, this algorithm has a worst-case number of sock pulls of \(\Theta(n)\) (On an unlucky day, Prof Indyk could indeed pull out one of every sock). However, what is the expected number of pulls before getting a matching pair? Note: we are only interested in the asymptotic (i.e., \(\Theta(\cdot)\)) answer.

**Solution:** The answer is \(\Theta(\sqrt{n})\).

Let \(s_1 \ldots s_{2n}\) be a random permutation of socks. To be more formal: let \(s_1 \ldots s_{2n}\) be a result of permuting the sequence \(<1,1,2,2,3,3 \ldots n,n>\) using a permutation chosen uniformly
at random from the set of all permutations of sequences of length $2n$. Let $T$ be a random variable equal to smallest $i$ such that $s_i = s_j$ for some $j < i$. We want to show that $E[T] = \Theta(\sqrt{n})$.

We start from showing $E[T] = O(\sqrt{n})$. For this purpose, we define an event $A$, that holds if and only if the prefix $s_1 \ldots s_{\sqrt{n}}$ contains a pair of matching socks. By standard probability we know that

$$E[T] \leq \Pr(A)\sqrt{n} + \Pr(\bar{A})E[T|\bar{A}]$$

Thus, it suffices to show that $E[T|\bar{A}] = O(\sqrt{n})$. For this purpose, define a random variable $T'$ to be the smallest $i > \sqrt{n}$ such that $s_i = s_j$ for some $j \in \{1 \ldots \sqrt{n}\}$. Clearly, $T' \geq T$, so it suffices to bound $E[T'|\bar{A}]$.

We introduce the following notation: a sock $s_i, i > \sqrt{n}$, is called good if $s_i = s_j$ for some $j \in \{1 \ldots \sqrt{n}\}$; otherwise, we call it bad. When we condition on $\bar{A}$, there are precisely $\sqrt{n}$ good socks among $s_{\sqrt{n}+1} \ldots s_{2n}$. We pick those socks one after another until we find the first good sock. Whenever we pick the next sock $s_i$, the probability of picking a good one is equal to $\frac{\sqrt{n}}{2n-i} \geq \frac{1}{2\sqrt{n}} = p$. Thus, the expected waiting time for the first good sock is at most $\sqrt{n} + 1/p = 3\sqrt{n} = O(\sqrt{n})$.

To show that $E[T] = \Omega(\sqrt{n})$, it suffices to show that the event $\bar{A}$ has a constant probability $q$ of success, since $E[T] \geq q\sqrt{n}$. The latter is easy. Let $q_i$ be the probability that $s_1 \ldots s_i$ are all different. We have $q_{i+1} \geq q_i \cdot \left(1 - \frac{i}{2n-i}\right) \geq q_i \left(1 - 1/\sqrt{n}\right)$. Thus, $q = q_{\sqrt{n}} \geq (1-1/\sqrt{n})^{\sqrt{n}} \approx 1/e$.

**Optional Extra Credit:** To improve the running time, volunteer to sort Professor Indyk’s socks for him.

**Solution:** To our surprise, no one approached this problem. We expect more interest after Quiz 2 :)