CS 124 Data Structures and Algorithms — Spring 2015

Problem Set 6 solutions

Problem 1

In class we analyzed QuickSelect using the notion of "good" recursive calls: a recursive call was good if it reduced the number of working elements by a factor of at least 3/4. Instead, we could adopt an analysis for QuickSelect similar to that for QuickSort. Suppose we call QuickSelect(A, k), to find the kth smallest element in an array A of size n. Assume the elements of A are distinct. Note the running time of QuickSelect is proportional to the number of comparisons. Thus if we let $X_{i,j}$ be a random variable which is 1 if ith smallest item and jth smallest item are ever compared throughout the execution of QuickSelect(A, k), then the running time is proportional to $\sum_{i<j} X_{i,j}$.

(a) For $i < j$, give an exact expression for $\mathbb{E}\, X_{i,j}$ in terms of i, j, k, n. You may need to employ case analysis.

(b) Using (a), show that $\mathbb{E} \sum_{i<j} X_{i,j} = O(n)$.

Solution

(a) We will break this into three cases:
Case 1: $i < j < k$. i and j will be compared if and only if i or j is selected as the pivot before i or j are separated from k. i or j will be separated from k if the pivot is chosen from the range $[i, k]$, so we have

$$\mathbb{P}(X_{i,j} = 1) = \mathbb{P}(\text{pivot} \in \{i, j\} \mid \text{pivot} \in [i, k]) = 2/(k - i + 1).$$

Case 2: $k < i < j$. Just as in case 1, we have that i and j will be compared if and only if one of them is chosen as the pivot before they are separated from k. This separation will now occur whenever a pivot is chosen from the range $[k, j]$ so we have

$$\mathbb{P}(X_{i,j} = 1) = \mathbb{P}(\text{pivot} \in \{i, j\} \mid \text{pivot} \in [k, j]) = 2/(j - k + 1).$$

Case 3: $i \leq k \leq j$. This time separation occurs when the pivot is chosen from $[i, j]$ so we have

$$\mathbb{P}(X_{i,j} = 1) = \mathbb{P}(\text{pivot} \in \{i, j\} \mid \text{pivot} \in [i, j]) = 2/(j - i + 1).$$

(b) Using linearity of expectations we get

$$\mathbb{E}\left[\sum_{i<j} X_{i,j}\right] = \sum_{i<j} \mathbb{E}\left[X_{i,j}\right] = \sum_{i<j<k} \mathbb{E}\left[X_{i,j}\right] + \sum_{k<i<j} \mathbb{E}\left[X_{i,j}\right] + \sum_{i\leq k\leq j} \mathbb{E}\left[X_{i,j}\right]$$

We will analyze these three sums separately. For the first sum,

$$\begin{aligned}
\sum_{i<j<k} \mathbb{E}\left[X_{i,j}\right] &= \sum_{i<j<k} \mathbb{P}\left(X_{i,j}=1\right) \\
&= \sum_{i<j<k} \frac{2}{k-i+1} \\
&= \sum_{i=1}^{k-2} \sum_{j=i+1}^{k-1} \frac{2}{k-i+1} \\
&= \sum_{i=1}^{k-2} \frac{2(k-i-1)}{k-i+1} \\
&\leq \sum_{i=1}^{k-2} 2 \\
&= 2(k-2)
\end{aligned}$$

By symmetry, the above analysis also gives us

$$\sum_{k<i<j} \mathbb{E}\left[X_{i,j}\right] \leq 2(n-k-2).$$

$$\begin{aligned}
\sum_{i\leq k\leq j} \mathbb{E}\left[X_{i,j}\right] &= \sum_{i\leq k\leq j} \mathbb{P}\left(X_{i,j}=1\right) \\
&= \sum_{i\leq k\leq j} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{k-1} \sum_{j=k+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{k-1} \sum_{t=k+1-i}^{n-i} \frac{2}{t+1} \\
&= \sum_{t=2}^{n-1} \sum_{i=\max(1,k-t+1)}^{\min(k-1,n-t+1)} \frac{2}{t+1} \\
&\leq \sum_{t=2}^{n-1} \sum_{i=k-t+1}^{k-1} \frac{2}{t+1} \\
&= \sum_{t=2}^{n-1} \frac{2(t-1)}{t+1} \\
&\leq \sum_{t=2}^{n-1} 2 \\
&= 2(n-2)
\end{aligned}$$

We leveraged that fact that we can swap the two sums by making the inner sum over all of the valid i for which there exists a valid j such that $j-i=t$.

Putting these three bounds together we get

$$\mathbb{E}\left[\sum_{i<j} X_{i,j}\right] \leq 2k - 4 + 2n - 2k - 4 + 2n - 4 = 4n - 12 = O(n)$$

as desired.

Problem 2

Consider the following implementation of a binary search tree (BST). When searching, we search based on key as in a normal BST. When inserting an element with key k, we assign the element a uniformly random ID in $[0, 1]$. We first insert the element into the BST based on its key as normal. We then *rotate* the node it lands in upward to preserve the invariant that, when looking at node IDs instead of node keys, the tree should be a min heap.
Suppose the keys in the BST at some point are $k_1 < k_2 < \ldots < k_n$. When searching for key k_r, note that we touch the node with key k_i if and only if it is an ancestor of the node with k_r in the BST. Using this fact, show that the expected time to perform a query is $O(\log n)$. Also show that the expected time to insert a new key into the data structure is also $O(\log n)$.
Solution This data structure is called a *treap* in the literature (it is a combination of a heap and a tree).
Let X_i be the indicator r.v. for k_i being ancestor of k_r. By the min heap invariant, this happens if and only if k_i has the smallest ID among $\{k_i, k_{i+1}, ...k_r\}$. Indeed, notice that the lowest common ancestor of k_i, k_r is precisely the element in $\{k_i, ..., k_r\}$ with the smallest ID. It follows that $\mathbb{E}\, X_i = \frac{1}{|i-r|+1}$.
Then the total search time for k_r is $\sum_{i=1}^{n} X_i$. By linearity of expectation, the average query time is

$$\sum_{i=1}^{n} \mathbb{E}\, X_i = \sum_{i=1}^{n} \frac{1}{|i-r|+1}$$

Every term in the summation is in the set $\{\frac{1}{1}, \frac{1}{2}, ..., \frac{1}{n}\}$, and each value in this set appears at most twice. Then the sum is at most $2\sum_{i=1}^{n} \frac{1}{i} = 2H_n = O(\log n)$.
For insertion time, note the time to inserta new key k is exactly the same as the time to search for the successor of k in the data structure. Thus we can bound the insertion time by search time, which we know is $O(\log n)$.

Problem 3

Suppose we use a hash table with chaining, and we hash n distinct items in the set $\{1, \ldots, U\}$ into a table of size $m = n$. Furthermore suppose that the hash function is *totally random*. We saw in class that for any item, its expected query time is $O(1)$. Show that if the hash function is totally random, then in addition we have the stronger property that with probability 99%, no linked list in the entire table has size larger than $O(\log n / \log \log n)$. You can use the following two facts without proof:

(1) for all $1 \leq k \leq n$, $(n/k)^k \leq \binom{n}{k} \leq (en/k)^k$, and

(2) the "union bound": for any set of probabilistic events $\{\mathcal{E}_i\}_{i=1}^{t}$,

$$\mathbb{P}(\mathcal{E}_1 \text{ or } \mathcal{E}_2 \text{ or } \ldots \text{ or } \mathcal{E}_t) \leq \sum_{i=1}^{t} \mathbb{P}(\mathcal{E}_i).$$

Solution Let A_k be the event that the k^{th} list has size at least t, where $t = O\left(\frac{\log n}{\log \log n}\right)$ is to be made more precise later. We need to show that for sufficiently large n,

$$\mathbb{P}(A_1 \cup \cdots \cup A_n) \leq 0.01$$

In fact, we will show that this probability tends to zero as $n \to \infty$. Fix some particular set of t items. The probability that these items all land in the same list k is $\left(\frac{1}{n}\right)^t$. Therefore by union bounding over all $\binom{n}{t}$ sets of t items, the probability that at least t items land in the k^{th} list is at most

$$\mathbb{P}(A_k) \leq \binom{n}{t}\left(\frac{1}{n}\right)^t \leq \left(\frac{e}{t}\right)^t$$

Union bounding again, we have that

$$\mathbb{P}(A_1 \cup \cdots \cup A_n) \leq \sum_{k=1}^{n} \mathbb{P}(A_k) \leq n\left(\frac{e}{t}\right)^t$$

To show that the last expression tends to zero, we'll show that its logarithm tends to $-\infty$, i.e. that

$$\log n + t - t\log t \to -\infty$$

If $t \geq \frac{c\log n}{\log\log n}$, then

$$\begin{aligned}
\log n + t - t\log t &\leq \log n + \frac{c\log n}{\log\log n} - \frac{c\log n}{\log\log n}\log\frac{c\log n}{\log\log n} \\
&= \log n\left(1 + \frac{c}{\log\log n} - c\left(\frac{\log c + \log\log n - \log\log\log n}{\log\log n}\right)\right) \\
&= \log n\left((1-c) + \frac{c - c\log c + \log\log\log n}{\log\log n}\right)
\end{aligned}$$

The term in the parantheses tends to $1-c$ as $n \to \infty$. So if we take $c > 1$, the limit will be $-\infty$, as desired.

Problem 4

We will use the following scheme to hash $n \leq m/2$ items into a hash table A of size m. Each item x with key k is associated with a permutation π_k chosen uniformly at random from all $m!$ permutations of the elements of $\{1, \ldots, m\}$. When we insert x, we first try putting it in $A[\pi_k(1)]$, then $A[\pi_k(2)]$ if that cell was already occpuied, etc. A search is also done by probing locations in that order.

(a) Show that for any of the n insertions, the probability the insertion makes more than t probes is at most $1/2^t$.

(b) Let X_i be the number of probes performed when inserting item i. Show that $\mathbb{E}\max_{1\leq i\leq n} X_i = O(\log n)$.

Solution (a) Let B_j be the event that $A[\pi_k(j)]$ is nonempty. Since the table is never more than half full, $\mathbb{P}(B_j) \leq \frac{1}{2}$. Furthermore, note that $\mathbb{P}(B_{j+1}|B_1, ..., B_j) \leq \mathbb{P}(B_{j+1}) \leq \frac{1}{2}$ since $A[\pi_k(j+1)]$ is uniformly distributed among all locations except the nonempty $A[\pi_k(1)], ..., A[\pi_k(j)]$. Then the probability we make more than t probes is

$$\mathbb{P}(B_1, ..., B_t) = \mathbb{P}(B_t|B_1, ..., B_{t-1})\,\mathbb{P}(B_{t-1}|B_1, ..., B_{t-2})\cdots\mathbb{P}(B_2|B_1)\,\mathbb{P}(B_1) \leq \left(\frac{1}{2}\right)^t$$

As desired.

(b) Let M be the maximum. We have

$$\mathbb{E}\,M = \sum_{k=0}^{\infty}\mathbb{P}(M > k) = \underbrace{\sum_{k=0}^{k^*}\underbrace{\mathbb{P}(M>k)}_{\leq 1}}_{\leq k^*} + \sum_{k=k^*+1}^{\infty}\mathbb{P}(M > k)$$

For the second sum above, note $M > k$ occurs if and only if $\vee_{i=1}^{n}(X_i > k)$, where $\vee$ denotes logical or. Then by the union bound,

$$\mathbb{P}(\vee_{i=1}^{n}(X_i > k)) \leq \sum_{i=1}^{n} \mathbb{P}(X_i > k) \leq \frac{n}{2^k}$$

by part (a). Therefore

$$\sum_{k=k^*+1}^{\infty} \mathbb{P}(M > k) \leq \sum_{k=k^*+1}^{\infty} \frac{n}{2^k} = \frac{n}{2^{k^*}}.$$

By choosing $k^* = \log_2 n$, we thus have that $\mathbb{E}\, M \leq k^* + n/2^{k^*} = O(\log n)$.

Problem 5

Solution If we simply compared T with every shift of S in two nested for loops, there would be $n - m + 1$ iterations, and the time per iteration would be m. Thus the total time would be $O(m(n-m+1)) = O(nm)$. This is too slow.

The key is to speed up the time to compare T with a shift of S in a single iteration. The main observation is that a length-m string over an alphabet Σ can be interpreted as a number between 0 and $|\Sigma|^{m-1} - 1$. Then, we are really just testing equality of two numbers. When are two numbers A, B equal? They are equal when $A - B = 0$. Define $C = A - B$. To test whether C is 0, we compute $C \mod p$ for a prime p and say it's zero if this mod is 0; else we say it's non-zero. Note though that we can be fooled if C isn't actually zero. Let us say that C has $\alpha(C)$ prime divisors. Recall from problem set 1 (!) that $\alpha(C) \leq \log_2 C$ (actually even fewer!). Thus if we take a *random* prime p amongst the first $\alpha(C)/P$ primes, then if C is not zero, the probability that we are fooled into thinking that it is zero is at most P. Interpreting T as a number, we can calculate $T \mod p$ in pre-processing. It turns out also that once we have $S[i..i+m-1] \mod p$, it's easy to get $S[i+1..i+m] \mod p$ in a constant number of arithmetic operations! This leads to constant time checks for whether T matches a shift of S. Details follow (also see our implementations). Some students also realized that a deterministic algorithm in the textbook, Knuth-Morris-Pratt (KMP), can also be used to solve this problem in linear time; students interested in details can read about it in the recommended textbooks or on Wikipedia. The solution outlined here though more naturally fits into the current theme of using randomization.

First, consider any string R of length m containing only lower case alphabetic charactes. We can interpret R as a base 26 number. Let R_p denote the value of R modulo p, and b_p^m denote the value of $26^m \pmod p$. Note that we can compute b_p^m and R_p in time $O(\log p)$ given b_p^{m-1} and $R[1,m]_p$. To do this, just note that $b_p^m \equiv b_p^{m-1} \cdot 26 \pmod p$ and $R_p \equiv R[0] \cdot b_p^m + R[1,m]_p \pmod p$. Then, computing R_p and b_p^m from scratch takes time $O(m \log p)$.

Now, we compute T_p and $S[i, m+i-1]_p$. for $1 \leq i \leq n-m$. Computing T_p and $S[1,m]$ takes time $O(m \log p)$ as above. We can compute $S[i+1, m+i]_p \equiv (S[i, m+i-1]_p - S[i] \cdot b_m^p) \cdot 26 + S[m+i] \pmod p$ from $S[i, m+i-1]$ in time $O(\log p)$. Then the total time spent here is $O((m+n)\log p) = O(n \log p)$.

For each T_p and $S[i, m+i-1]_p$, disagreement implies that T and $S[i, m+i-1]$ differ. If they agree, then T and $S[i, m+i-1]$ may or may not differ. The idea is to select several primes p from some distribution so that the probability that T_p and $S[i, m+i-1]_p$ agree for every p but T and $S[i, m+i-1]$ differ is small. In practice for this problem, it's enough to choose two primes that are around 10^7. The probability of failure can be reduced by using more primes, or by choosing larger primes.

If we more quantitative bounds, we can use the following technical analysis - for us to have a false match, we need $p | T - S[i, m+i-1]$ but $T \neq S[i, m_i - 1]$. Then $0 < |T - S[i, m+i-1]| < 26^m$.

There are more than 5×10^7 primes between 10^7 and 10^9. Since $26^{5,000,000}$ has approximately 7×10^6 digits, at most 1×10^6 of these primes can divide the difference. Thus if we choose k primes at random in this range, the probability that they both divide the difference is at most $\frac{1}{50^k}$. Union bounding over the at most $5,000,000$ differences and the 16 test cases, to get a failure probability of less than 1%, it's enough to take k such that $\frac{16 \cdot 5 \times 10^6}{50^k} < 0.01$, or 6 such primes.