CSE 5500 Algorithms. Fall 2018

Home work 1 – Solutions

1. Proceeding along the lines of the problem done in class -
   Probability of success in one attempt is $= (\frac{\sqrt{n}}{n}) * (\frac{\sqrt{n}-1}{n}) \approx (\frac{1}{n})$
   Probability of failure in one attempt is $\approx (1 - \frac{1}{n})$
   Hence, the probability of failure in k successive attempts is $\approx (1 - \frac{1}{n})^k$
   We want this probability to be no more than $n^{-\alpha}$. I.e., we want $(1 - \frac{1}{n})^k \leq n^{-\alpha}$
   Using the inequality $(1 - \frac{1}{x})^x \leq \frac{1}{e}$, $(1 - \frac{1}{n})^k \leq e^{-k/n}$. Equating $e^{-k/n}$ and $n^{-\alpha}$ we get: $k = \frac{\alpha n \log n}{\log e}$. Thus the number of attempts needed is no more than $\frac{\alpha n \log n}{\log e}$ with high probability. In other words, the tun time is $\widetilde{O}(n \log n)$. If we only make $O(\log n)$ attempts, the probability of success can be verified to be not as high as what we want.

2. Algorithm $\mathcal{A}$ is called $d\alpha \log_e n$ (d being a constant) times in the new algorithm. A count of the number of YESs and the number of NOs output by $\mathcal{A}$ is kept. Let u be the total number of YESs and v the total number of NOs. If $u > v$, the new algorithm (call it $\mathcal{B}$) outputs YES; otherwise it will output NO.

   $\mathcal{B}$'s output will be correct if there are more correct answers than incorrect answers in the $d\alpha \log_e n$ answers obtained from $\mathcal{A}$. Let X be the number of times $\mathcal{A}$ outputs the correct answer and let Y be the number of times $\mathcal{A}$ outputs the incorrect answer. We want to show that Prob.$[X > Y]$ is $\geq (1 - n^{-\alpha})$.

   Note that X is a binomial random variable with parameters $(d\alpha \log_e n, c)$. Similarly, Y is a random variable with parameters $(d\alpha \log_e n, (1-c))$. Since c is greater than $\frac{1}{2}$, let $c = \frac{1}{2} + \delta$ for some some constant $\delta > 0$. The expected value of X is $d\alpha \log_e n \left(\frac{1}{2} + \delta\right)$ and the expected value of Y is $d\alpha \log_e n \left(\frac{1}{2} - \delta\right)$.

   Chernoff's bounds are typically used to show that if the mean of a binomial random variable is μ, then, with high probability, the actual value of the random variable can not be signinficantly greater (or less) than μ. This is what we are going to use to prove our result.

   Using Chernoff's inequality,

   $$\text{Prob.}\left[X \leq \left(1 - \frac{\delta}{2}\right) cd\alpha \log_e n\right] \leq e^{-\frac{\delta^2}{8} cd\alpha \log_e n}$$

   The RHS of the above inequality will be $\leq n^{-\alpha}$ if $d \geq \frac{8}{c\delta^2}$.

   Thus we have

   $$\text{Prob.}\left[X \leq \left(\frac{1}{2} + \frac{3}{4}\delta - \frac{\delta^2}{2}\right) d\alpha \log_e n\right] \leq n^{-\alpha} \tag{1}$$

   if $d \geq \frac{8}{c\delta^2}$.

   From equation 1 we see that X will be more than Y with high probability if we pick $d \geq \frac{8}{c\delta^2}$.

3. We can make use of any data structure that supports the following operations: 1) **SEARCH** for an arbitrary element; 2) **INSERT** an arbitrary element; and 3) **DELETE** the minimum. One such data structure is a 2-3 tree. We can use a modified version of any such data structure. Each node in this data structure will be a linked list of identical keys. Sorting, as pointed out in class, amounts to inserting the given keys into the empty data structure and deleting the minimum one by one. When inserting a key, we first check if this key is present in the data structure. If it is, the new key will be inserted as the head of the correcponding list.

If it is not, a new node will be created (using the **INSERT** algorithm of the data structure) and a list with one key will be stored in the node.

Note that the data structure will contain at most d nodes at any time. Time for inserting a single element into the data structure is $\log d + c$, where c is a constant time needed to insert an element into linked list. Time for inserting all the elements into the data structure is $O(n \log d)$. Total time for all the deletes is $O(n + d \log d)$. Therefore, the run time of the algorithm is $O(n \log d + n + d \log d) = O(n \log d)$.

4. Find the median M of the n given keys in $O(n)$ time. If X is the given input sequence partition X into X_1 and X_2 such that $X_1 = \{q \in X : q < M\}$ and $X_2 = \{q \in X : q > M\}$. If $|X_1|$ and $|X_2|$ are both even, then M is the unique element. In this case output M and quit. If not, if $|X_1|$ is odd, recursively look for the unique element in X_1. If none of the above cases applies, recursively look for the unique element in X_2. Let $T(n)$ be the run time of this algorithm on any input of size n. Then, it takes $O(n)$ time for the initial selection and the partitioning. The time for the recursive call is $T(n/2)$. Therefore, it follows that $T(n) = T(n/2) + O(n)$ which solves to: $T(n) = O(n)$.

5. We can see that the smallest element appears $2n - 1$ times in C $(\min\{k_1, k_1\}, \min\{k_1, k_2\}, \ldots, \min\{k_1, k_n\}, \min\{k_2, k_1\}, \min\{k_3, k_1\}, \ldots, \min\{k_n, k_1\})$. Similarly, the second smallest element apperas $2n-3$ times in C and the ith smallest element appears $2n - 2i + 1$ times in C. Now the median of C is the jth smallest element such that
$\sum_{i=1}^{j} 2n - 2i + 1 = \frac{n^2}{2}$
j can be obtained by solving the above equation, i.e., $j^2 - 2nj + \frac{n^2}{2} = 0$.
Now, the median of C can be obtained by finding the jth smallest element in S.
Complexity $= O(n)$.

6. Do a radix sort on the elements in A and B separately. Complexity $= O(n)$.
Merge the two sets to check whether the sets are disjoint. Complexity $= O(n)$.
Total complexity $= O(n)$.

7. We assign a polynomial to each node in each of the two trees, by the following rules:

   - Every leaf gets polynomial $P = x_0$
   - An internal vertex v at height h having children $v_1, v_2, \ldots, v_k$ gets polynomial $P_v = (x_h - P_{v_1})(x_h - P_{v_2}) \ldots (x_h - P_{v_k})$

   We claim that the two trees are isomorphic if and only if the polynomials at their roots are equal. The left to right implication is immediate: if the trees are isomorphic then the polynomials will be identical by virtue of multiplication being commutative.

   If the polynomials are equal, then we can prove by induction that the trees are isomorphic. The base case is trivial: if two trees have polynomial x_0 then they are single node trees and are isomorphic. If the polynomial at the root of both trees is $P_v = (x_h - P_{v_1})(x_h - P_{v_2}) \ldots (x_h - P_{v_k})$, since x_h does not appear in any of $P_{v_i}, i = 1, k$ it must be the case that both trees have k children which can be paired based on their polynomials. By induction, since the polynomials of the children are equal, the children's subtrees are isomorphic and thus the two trees are isomorphic.

   So the problem of checking tree isomorphism has been reduced to checking equality of (at most) degree n multivariate polynomials. This can be done in $\tilde{O}(n)$ as presented in class.

8. Let $G(V,E)$ be the input graph. A trivial algorithm is Algorithm 1. The worst case runtime is $O(mT_c)$ where T_c is the time needed to check if a graph has a perfect matching, which is the same as the time needed to multiply two matrices. Here $m = |E|$.

Algorithm 1 Algorithm P3

```
1. pick edge $e = (u,v) \in E$
remove edge $e$ and nodes $u$ and $v$ from the graph
if there exists a perfect matching for the new graph then
  recursively compute perfect matching $M$ on the remaining graph
  return $M \cup (u,v)$
else
  restore graph to initial state
  remove edge $(u,v)$ but not nodes $u$ and $v$
  go to 1
end if
```

9. For each submatrix of size m^2, the probability of giving an incorrect answer is $\leq \frac{m^2}{t/\log t}$ where $[1,t]$ is the range out of which we choose prime p. The probability of giving an incorrect answer for any of the $(n-m+1)^2$ submatrices is $\leq \frac{(n-m+1)^2 m^2}{t/\log t}$. The max of $(n-m+1)^2 m^2$ is obtained for $m = n/2$ and is $O(n^4)$. So, if we choose $t = n^{\alpha+4.1}$ then the previous probability is less than $n^{-\alpha}$.

   The runtime is $O(n^2)$ using the following observation. Let $B[i,j]$ be the fingerprint of the submatrix having the lower right corner at (i,j) and let $C[i,j]$ be the fingerprint of m contiguous values in row i, ending at position j . Then

$$B[i,j] = (B[i-1,j] - 2^{m^2-m} C[i-m+1,j])2^m + C[i,j](mod\ p)$$

   For row i, we only need values of B from row $i-1$, so the extra memory for B is linear. The values of C for row i and $i-m+1$ at column j can be computed on the fly as we scan the current row from left to right, so for C we only add constant memory overhead.

   Thus, we can compute the fingerprint of the submatrix ending at (i,j) in constant time from the fingerprint of the submatrix ending at position $(i-1,j)$. Since testing each fingerprint takes $O(1)$ time, the total runtime is $O(n^2)$ (including the cost of fingerprinting the initial submatrices ending at positions in row $m-1$).