CS344: MidTerm Solution

November 5, 2017

1. *(Rapid Answers)*

   (a) If $f = O(g(n))$ and $f = \Omega(g(n))$, then $f(n) = g(n)$. True or False? Provide reasons.

   False. $f(n) = 10g(n)$ would satisfy both properties and yet $f(n) \neq g(n)$.

   (b) Solve the recurrence $T(n) = 3T(n/4) + n \log n$ and $T(1) = 1$, using any method you prefer.

   We have

   $$T(n) = 3(3T(n/16) + (n/4)\log(n/4)) + n \log n \leq 9T(n/16) + (3n/4)\log n + n \log n$$

   Continuing this way with each iteration, we will have

   $$n \log n + (3/4)n \log n + (3/4)^2 n \log n + \cdots$$

   so in total $T(n) = O(n \log n)$.

   (c) We implemented dictionary using uniform hashing, collision resolution by chaining, and hash functions that can be calculated in $O(1)$ time each. If we are happy with $O(\log n)$ time on average for search, what will be m, the number of hash buckets you will choose for storing n elements?

   Average search time is $O(1+n/m)$, so if we want this to $O(\log n)$, we can choose $m = O(n/\log n)$.

   (d) Given a polynomial of 10 variables, degree 20 and suppose we choose randomly values for the variables from a set S of size 1000, what is an upper bound on the probability that we get a false outcome (that the polynomial is NOT identically 0, but these values make the polynomial turn out to be 0).

   We showed that the probability is at most $d/|S|$ where d is the degree. So, the probability is at most $20/1000$, independent of number of variables.

   (e) Muthu has a binary string s of length n and you have a binary string t of length n. What should Muthu communicate to you to check if $s = t$, and how many bits are needed? Assume we are good with a small probability of error.

   We should use Karp-Rabin fingerprint $f(x)$ for string x. In this case, Muthu should communicate q (the random prime), and $f(s)$. Both of them are $O(\log n)$ in size.

2. *(Dynamic Programming)* A tree T consists of a root node and zero or more children, each of which is a tree. Its *vertex cover* is a subset S of nodes such that for each (node u,child v) pair in T, at least one of u or v is in S. The problem is to find the minimum sized vertex cover. Solve it using dynamic programming.

   - For each node v in T, what function will you compute?

     For every v in T, we compute two functions:
     - $f_1(v)$, the smallest vertex cover for tree rooted at v when v IS in the vertex cover, and
     - $f_2(v)$, the smallest vertex cover for tree rooted at v when v is NOT in the vertex cover.

   - What is the recursive definition for that function. Consider the two cases when v is part of the minimum vertex cover and when v is NOT a part of it.

     We have

     $$f_1(v) = \left(\sum_{w \text{ children of } v} \min\{f_1(w), f_2(w)\} \right) + 1$$

     and

     $$f_2(v) = \sum_{w \text{ children of } v} f_1(w)$$

   - Complete the dynamic programming solution by stating in what order you evaluate this function and how much time, space does the whole algorithm take?

     We calculate this bottom-up in T so $f_1(w)$ and $f_2(w)$ will be available for all w when we evaluate for v. Like we discussed in the maximum independent case, time and space is $O(n)$, the number of nodes in T.

3. *(Divide and Conquer)* **30 points** Suppose you are given an array of positive integers $A[1,n]$. Each $A[i]$ is an element.

   - For $1 \le i \le j \le n$, a *sub-array* $A[i,j]$ of A is $A[i, i+1, \ldots, j]$
   - A prefix of A is a sub-array $A[1, 2, \ldots, i]$
   - A suffix of A is a sub-array $A[i, i+1, \ldots, n]$.
   - The sum of a sub-array $A[i,j]$ is the sum of all its elements $\sum_{k=i}^{k=j} A[k]$.

   Design a divide and conquer algorithm that counts the number of sub-arrays with even sum.

   - A naive approach is to calculate the sum of all possible sub-arrays ($A[i,j]$ for all i,j) and then returning the number of these of even sum. What is the running time of this algorithm?

     There are $O(n^2)$ such sub-arrays, and for each, it takes $O(n)$ time to calculate the sum, so the running time is $O(n^3)$.

- Show how to compute the sums of all prefixes of A in $O(n)$ time. Given these values, show that the sum of any sub-array $A[i, j]$ can be computed in $O(1)$ time. How long does it take now to go through all possible sub-arrays and return the ones of even sum like in the case above?

  Let sum of the ith prefix $A[1, 2, \ldots, i]$ be $P[i] = \sum_{k+1}^{k=i} A[k]$. We have

  $$P[i+1] = P[i] + A[i+1]$$

  so all P's can be computed in $O(n)$ time.
  The sum of any sub-array $A[i, j]$ is

  $$P[j] - P[i-1]$$

  so it can be computed in $O(1)$ time from P's.
  The running time of the algorithm above is $O(n^2)$ because the sum of each of the $O(n^2)$ sub-arrays can be computed in $O(1)$ time as shown.

- We will use divide-and-conquer approach. Suppose that you divide A into two disjoint sub-arrays L and R of length $n/2$, and determine the number of sub-arrays of even sum for both of these two sub-arrays recursively. Why is this information not sufficient to compute the number of even sum sub-arrays of A? Be concise.

  There are sub-arrays of A like $A[i, i+1, \ldots, n/2, n/2+1, \ldots, j]$ where for example, the sum of the sum-array

  $$A[i, i+1, \ldots, n/2]$$

  and the sum of the sub-array

  $$A[n/2+1, \ldots, j]$$

  can both be odd, and the subarray

  $$A[i, i+1, \ldots, n/2, n/2+1, \ldots, j]$$

  has even sum, which means it is not sufficient to only determine the sub-arrays of even sum in L or R.

- Suppose now that your recursive algorithm computes and returns extra information beyond the number of even sum sub-arrays: what constant size extra information can be returned so that from the recursive solutions for L and R one can recover in $O(1)$ time (1) the number of even sum sub-arrays of A and (2) also the extra information to return when you pop out of the recursion for A? (Hint: Think about the number of prefixes and suffixes with even and odd sum in R and L.)

  For the subproblem on L, return

  - L_1, the number of sub-arrays with even sum,
  - L_2, the number of suffixes with odd sum,
  - L_3, the number of suffixes with even sum.
  - L_4, the number of prefixes with odd sum,
  - L_5, the number of prefixes with even sum

  Do the same for R. Then the number of even sum sub-arrays is

  $$L_1 + L_2R_4 + L_3R_5 + R_1$$

  (be careful about corner cases if any). Now we can devise similar formula for all other sums we need recursively.

- Write a recurrence for your divide-and-conquer algorithm above. Solve the recurrence using your favorite method. What is the running time of the algorithm?

  The discussion above gives the recurrence for a subarray in terms of the 5 terms we calculate and return for each of L and R. These recurrences can be calculated in $O(1)$ time once the L and R problems have been solved. So, the recurrence is

  $$T(n) = 2T(n/2) + O(1)$$

  which solves to $O(n)$.