CS3510 Design & Analysis of Algorithms Section B

Test 1 Review and Practice Questions

Instructor: Richard Peng Test 1 in class, Friday, Sep 8, 2016

- Main Topics
  - Asymptotic complexity: O, Ω, and Θ.
  - Designing divide-and-conquer algorithms.
  - Setting up runtime recurrences.
  - Solving recurrences using Master theorem (other methods are optional).
  - Applications of fast multiplication.
- NOT included:
  - Definition and algorithm of inversion counting.
  - Details of how to multiply numbers faster than n^2.
  - Guess and check / recursion tree: master theorem works for everything on test.
- Proving big-O bounds (Homework 1, Problem 1. Ex 0.1 in Textbook):
  - If $f_1 = O(g_1(n))$, $f_2 = O(g_2(n))$, then
    * $f_1 f_2 = O(g_1 g_2)$.
    * $f_1 + f_2 = O(\max\{g_1, g_2\})$.
  - For any constants a, b, and c, $O(\log^a(n)) \leq O(n^b) \leq O(c^n)$,
  - $\ln(n) = \Theta(\log n) = \Theta(\log_2 n) = \Theta(\log_c n)$.
- Divide-and-conquer and setting up running time recurrences (Homework 1, Problems 2 and 3. Ex 2.12, 2.16, 2.17, 2.23 in textbook)
  - General structure of a recursive algorithm:
    * Split the problem up.
    * Make a recursive calls to problems of size n/b
    * Combine the results.
  - $T(n)$: running time when given input of size n.

- If total cost of split/combine is $O(n^d)$, runtime recurrence is:

$$T(n) = aT(n/b) + O(n^d).$$

- Master Theorem:

  The recurrence:

$$T(n) \;=\; a\,T\left(\frac{n}{b}\right) \;+\; c \cdot n^d, \quad T(1) \;=\; e$$

  where $a > 0$, $b > 1$, $c > 0$, $d \geq 0$ and $e \geq 0$ are constants, has the solution given below:

  Case 1: If $d = \log_b a$ then $T(n) = O(n^d \log n)$.

  Case 2: If $d > \log_b a$ then $T(n) = O(n^d)$.

  Case 3: If $d < \log_b a$ then $T(n) = O(\, n^{\log_b a})$.

- Example of using Master theorem to analyze recursion (Homework 1 Problems 2ac, 3. Ex 2.4, 2.5abcde in textbook):
  - Multiplies 2 n-digit numbers using 3 multiplies of $n/2$-digit numbers, and addition/subtraction. Recurrence: $T(n) = 3T(n/2) + O(n)$.
  - Master theorem: this is $a = 3$, $b = 2$, $d = 1$, so $d < log_2 3$, and the overall runtime is $O(n^{\log_2 3}) = O(n^{1.59})$.

- Applications of fast (polynomial) multiplication:
  - Computing sumsets by multiplying indicator polynomials (include x^i if i is in the set). Set of sums given by non-zeros in product without carry.
  - Counting mismatches in strings (Homework 1, Problem 4)

1. Let A and B be two matrices to be multiplied. We saw an algorithm for this problem that takes time $O(n^{\log_2 7})$. Suppose someone discovers a way to obtain the product of two order $n \times n$ matrices by doing 24 multiplications of two matrices of order $n/5$ and combining the results in $O(n^2)$ time. Write the recurrence for the running time of this new algorithm. What is the solution to this recurrence? Is this running time better than the running time for Strassen's algorithm?

   Solution: $T(n) = 24T(n/5) + cn^2$ whose solution is $O(n^2)$. This is certainly better than Strassen's algorithm whose complexity is $O(n^{\log_2 7})$.

2. (From Fall 2015 Test 1) Let x be an n-bit number. The following recursive algorithm computes x^k for some integer k. (Assume k is a power of 2.) It uses a bit-multiplication algorithm MULTIPLY(y,z) that takes two numbers y and z and returns their product. The numbers y, z and their product are all in bits.

   POWER(x, k)

   IF $k = 1$ then Return (x)

   If $k > 1$ Then

   Let $y =$ POWER (x, $k/2$)

   Return (MULTIPLY (y, y))

   Assuming that the number of bit operations for multiplying two n bit numbers is n^2, set up a recurrence for the number of bit operations used by this algorithm to compute x^n and solve the recurrence.

   Solution:

   Since x^k has kn bits, multiplying $x^{k/2}$ with itself takes time $O(k^2n^2)$.

   Let $T(k)$ be the runtime of calling $POWER(x, k)$. We get the recurrence:

$$T(k) = T(k/2) + O(k^2n^2), \quad T(1) = 0$$

   Since n does not depend on k, we can first solve the modified runtime recurrence;

$$T'(k) = T'(k/2) + O(k^2),$$

   which by master's theorem with $a = 1$, $b = 2$, and $d = 2$ solves to $T'(k) = O(k^2)$. Multiplying the n^2 back in then gives $T(k) = O(k^2n^2)$, which when $k = n$ gives $O(n^4)$.

3. (From Fall 2015 Test 1, also on Homework 1, Problem 3) Assume that you are given an $O(n)$-time algorithm $MEDIAN$ that takes as input an array A of n distinct positive integers and returns the median element in A.

   Using the algorithm $MEDIAN$ design an $O(n)$ algorithm that, given an array A of n distinct positive integers and an index $1 \leq k \leq n$, determines the k-th smallest element in A.

   (The median element in A is the $\lceil n/2 \rceil$-th smallest element of A.)

   Solution: Here is an $O(n)$ algorithm for the task at hand:

   - Use $MEDIAN$ algorithm to find the median m of the array A in $O(n)$ time.
   - If $k = \lceil n/2 \rceil$ then return m.

- Partition A using m into two arrays A_L and A_R such that A_L has all elements of A that are $\leq m$ and A_R has all elements of A that are $> m$. This takes $O(n)$ time.
- If $k \leq m$, search A_L for the k-th element recursively.
- If $k > m$, search A_R for $(k - m)$-th element recursively.

The running time of this algorithm is given by the recurrence:

$$T(n) = T\left(\frac{n}{2}\right) + c\,n, \quad T(1) = e$$

whose solution is $O(n)$.

4. Given as input an array A of n integers, describe an $O(n \log n)$ time algorithm to decide if the entries of A are distinct. Why does your algorithm run in time $O(n \log n)$?

Solution:

Algorithm idea: Sort the array A. Compare consecutive elements to to see any element is repeated. If so, the elements are not distinct.

Algorithm:

(a) Sort the array A.

(b) $i = 1$. $distinctsofar = True$.

(c) WHILE $i < n$ and $distinctsofar = True$ do:

   i. If $A[i] \neq A[i+1]$ Then $i = i + 1$ ELSE $distinctsofar = False$

(d) Output the array B.

This algorithm takes time $O(n \log n))$ time: **Step 1** takes $O(n \log n)$ time. The WHILE loop is executed at most $n - 1$ times and each time through the loop the time takes is $O(1)$. All other steps take time $O(1)$.

5. Given a sorted array of n distinct integers $A[1] \cdots A[n]$, describe an $O(\log n)$ divide-and-conquer algorithm to find out whether there is an index i such that $A[i] = i$. Why does your algorithm run in the claimed time bound?

Solution 1: consider the array B with

$$B[i] = A[i] - i.$$

Since $A[i] < A[i+1]$, we get

$$B[i] = A[i] - i \leq A[i+1] - 1 - i = B[i+1],$$

so the array B is non-decreasing and sorted.

An index where $A[i] = i$ corresponds to one where $B[i] = 0$, so it suffices to binary search for 0 in B, which takes $O(\log n)$ time.

Solution 2 : Use binary search as follows, assume $n = 2^k$. Search(A,1,n) gives the required answer.

Search(A,lb,ub)
{
mid := (ub + lb)/2
if((lb==ub)&&(a[mid] != mid)) return **NO**
if (A[mid] == mid) output **YES**
if (A[mid] > mid) **Search**(A,lb,mid)
if (A[mid] < mid) **Search**(A,mid,ub)
}

Since we are using binary search, for each call to **Search** the difference between ub and lb is halved. Hence, the running time is $O(\log n)$.

6. You are given an infinite array A in which the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. You are *not* given the value of n. Describe an $O(\log n)$ algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists.

   Solution: Find an upper bound for n in $\log_2 n$ rounds by checking in the i-th round if $A[2^i]$ is ∞.

   (This upper bound cannot be more than twice the actual value of n. Why?)

   With this upper bound do a binary search to find if x is in the array.

7. Describe an $O(n \log_2 n)$ time algorithm that, given a set S of n real numbers and another real number x, determines whether or not there exist two elements in S whose sum is exactly x.

   (Hint: Doing a binary search in a sorted list can be done in $O(\log n)$ time.)

   Solution:

   1. Use mergesort to sort the set S in ascending order. This takes $\O(n \log_2 n)$ time.

   2. For each element $a \in S$ perform a binary search of the sorted array S to find $x - a$, if it is present. Each search takes $\O(\log_2 n)$ time and at most n searches are performed, so this entire step will take at most $O(n \log_2 n)$ time.

   Combining steps 1 and 2 causes the algorithm to take $O(n \log_2 n)$ time.

8. Let $A_1, A_2, \cdots, A_k$ be k sorted arrays, each with n elements. Give an $O(nk \log k)$ algorithm to combine them into a single sorted array of kn elements. (Assume k is a power of 2.)

   Solution: Merge them pairwise: A_i with A_{i+1} for $i = 1, 3, \cdots, k - 1$. Assuming that merging two arrays of size n takes $c.n$ comparisons for some constant c, the $k/2$ merges take $cn(k/2)$ comparisons. In the next stage, merge pairwise the resulting $k/2$ arrays each with $2n$ elements. This takes $c(2n)(k/4) = cn(k/2)$ comparisons. Repeat this process until there is only one array of kn elements. There are $\log_2 k$ stages and each stage takes $cn(k/2)$ comparisons.

9. Let A be an array of n positive integers. Let n be a multiple of 5. Describe an $O(n)$ algorithm to find if there is an element in A that occurs at least $n/5$ times in A. (Hint: Use the linear time algorithm $SELECTION$ for computing the k-th smallest element discussed in class.)

   Solution: An element that occurs at least $n/5$ times must be one of the following four elements: $n/5$-th smallest, $2n/5$-th smallest, $3n/5$-th smallest, and $4n/5$-th smallest element. Why? Suppose there is an element x that occurs at least $n/5$ times in A. Let AS be the array that results when A is sorted. Let i be the least index where x appears in AS. Then, $AS[i]$ to $AS[i + n/5 - 1]$ are all equal to x.

   The algorithm: Use the linear time algorithm $SELECTION$ for computing the k-th smallest element (discussed in class) to compute the $n/5$-th smallest, $2n/5$-th smallest, $3n/5$-th smallest, and $4n/5$-th smallest elements. For each one go through the array A to see if it occurs $n/5$ times. If none of these elements occur $n/5$ times then there is no such element. The algorithm takes $O(n)$ time.

10. Describe an $O(n^{1.6})$ time algorithm for computing the number of Pythagorean triples modulo n. Specifically the number of triples (a, b, c) with $0 \leq a, b, c < n$ such that

$$a^2 + b^2 \equiv c^2 \ mod \ n.$$

   Here $x \ mod \ n$ denotes the remainder of x when divided by n. You may assume that it can also be computed in $O(1)$ time.

   Solution: First compute for each i the number of ways to get

$$a^2 \equiv i \ mod \ n$$

   with some $0 \leq a < n$. This takes $O(n)$ time because we can just loop through all values of a. Let these values be p_i. Note that since we have the same choices of b and c, they will produce the same vector.

   Then form the polynomial

$$P(x) = \sum_{i=0}^{n-1} p_i x^i,$$

and compute using fast multiplication

$$R(x) = P(x)^2$$

in $O(n^{1.6})$ time.

The coefficients of x^k in R is the number of pairs of $0 \le a < n$ and $0 \le b < n$ such that

$$\left(a^2 \ mod \ n\right) + \left(b^2 \ mod \ n\right) = k,$$

For each of these coefficients, we have equality when

$$c^2 \equiv k \ mod \ n.$$

This value is precisely

$$p_{k \ mod \ n},$$

so looping through all $O(n)$ coefficients of R gives the answer.