Final Exam

Instructions

Please write your answers in the blue book(s). Work alone. Do not use any notes or books. You have approximately three hours to complete this exam.

Unless otherwise specified, you should justify your answers. Running times should be given in asymptotic notation. When describing an algorithm, you should feel free to use as a subroutine any algorithm found in Levitin or the lectures; you do not need to write down the details of such an algorithm or re-prove its properties. Similarly, standard mathematical facts can be stated and used without proof.

1. Lots of ones (20 points)

How many ones does the following procedure print when run with input n? Compute the best bounds you can: the exact value if possible, a big-Θ expression if you can't find the exact value, or big-O and big-Ω bounds if you can't find a big-Θ expression.

```
Ones(n):
    if n = 0:
        print 1
    else:
        for i = 1 to 2^n:
            Ones(n-1)
```

(Note: You should read the upper bound of the `for` loop as 2^n.)

Solution:

The recurrence is $T(n) = 2^n T(n-1)$, with $T(0) = 1$. It is not hard to see that this has the solution $T(n) = \prod_{i=1}^{n} 2^i = 2^{\left(\sum_{i=1}^{n} i\right)} = 2^{n(n+1)/2}$.

Note that this quantity can be written as $\Theta(2^{n(n+1)/2})$ or $2^{\Theta(n^2)}$ but not as $\Theta(2^{n^2})$, since this last quantity is roughly the square of the correct answer and is not bounded by a constant multiple of it.

2. Two-dimensional search (20 points)

The elements of an n-by-n array A are arranged in ascending rows and columns; formally, $A[i][j] < A[i][j']$ whenever $j < j'$ and $A[i][j] < A[i'][j]$ whenever $i < i'$.

Suppose you wish to determine if A contains a particular target element x. Give the most efficient algorithm you can for solving this problem, and compute the best upper bound you can on its asymptotic worst-case running time as a function of n.

Hint: For some algorithms it may be easier to keep track of the number of elements m than the number of rows or columns.

Solution:

Simple $O(n \log n)$ algorithm

Run binary search separately on each row.

$O(n)$ reduce-and-conquer algorithm

Examine $A[1][n]$. If $A[1][n] > x$, then x does not appear anywhere in the first row: recurse on $A[2..n][1..n]$. If $A[1][n] < x$, then x does not appear anywhere in the last column: recurse on $A[1..n][1..n-1]$. In either case we reduce the sum of the number of rows and columns by 1, so if we don't find x we reach an empty matrix in at most $2n$ such steps. Total cost is $O(n)$.

There are other ways to get $O(n)$ time from divide-and-conquer algorithms (e.g. by finding all values less than or greater than x in the middle row by binary search and then recursing on the approximately $n^2/2$ positions that might still hold x, but the analysis of these variants is more complicated. There are no solutions that do better than $O(n)$ in the worst case: see the lower bound below.

Lower bound

It is *not* possible to solve this problem in less than $O(n)$ time in the worst case. Consider an input where every element off the $i = j$ diagonal is either 0 or $2x$, and every element on the diagonal is either x or $x + 1$. Without reading the entire diagonal it is impossible to tell if it contains any instances of x.

3. A sorting-based data structure (20 points)

Professor Sortsnes, who only knows one algorithm, proposes a new sorting-based data structure that eliminates the messy complexity of balanced binary trees while preserving $O(\log m)$ time for FIND operations. The idea is to maintain a sorted array A of m elements together with an unsorted auxiliary array B of up to $\lg m$ elements. To FIND an element, perform binary search on A and exhaustive search on B. To INSERT an element, append it to B; if this makes B contain more than $\lg m$ elements, append all the elements of B to A and sort the new, larger A using Mergesort in $\Theta(m \lg m)$ time.

Compute the amortized cost per operation of performing n INSERT operations using this scheme.

Solution:

If we assume that the n-th INSERT causes a sort, we get for the total cost the rather horrible recurrence $T(n) = \Theta(n \lg n) + T(n - \lg n)$, which expands into a sum of terms of the form $\Theta(n_i \lg n_i)$ where each $n_{i+1} = n_i - \lg n_i$. Solving this sum exactly may be tricky, but we can reasonably guess that it is dominated by the larger terms, e.g., those for which $n_i > n/2$ and $\lg n_i > \lg(n/2) = \lg(n) - 1$. For these terms we have $\Theta(n_i \lg n_i) = \Theta(n \lg n)$. There are somewhere between $\frac{n/2}{\lg n}$ and $\frac{n/2}{\lg n - 1}$ such terms, which we can write as $\Theta(n/\lg n)$. Multiplying the bound on each large term by the bound on the number of large terms gives $\Theta((n \lg n)(n/\lg n)) = \Theta(n^2)$.

This will be our guess for $T(n)$. We have just shown that $T(n) = \Omega(n^2)$, but we still need to prove that the guess works as an upper bound even if we throw in the smaller terms.

Suppose that $T(k) \le ak^2$ for $k < n$; then $T(n) = \Theta(n \lg n) + T(n - \lg n) \le cn \lg n + a(n - \lg n)^2 = cn \lg n + an^2 - 2an \lg n + a \lg^2 n \le an^2$ when $a \gg \frac{1}{2}c$, giving $T(n) = O(n^2)$.

4. A frustrating exam (20 points)

You are given an exam with questions numbered $1, 2, 3, \ldots, n$. Each question i is worth p_i points. You must answer the questions in order, but you may choose

to skip some questions. The reason you might choose to do this is that even though you can solve any individual question i and obtain the p_i points, some questions are so frustrating that after solving them you will be unable to solve any of the following f_i questions.

Suppose that you are given the p_i and f_i values for all the questions as input. Devise the most efficient algorithm you can for choosing set of questions to answer that maximizes your total points, and compute its asymptotic worst-case running time as a function of n.

Solution:

Short version: use dynamic programming.

More details: Let $S(i)$ be the maximum total score that can be obtained from questions i through n. Any such score is obtained from a set of questions that either includes i or not; in the first case, the best score is $p_i + S(i + f_i + 1)$, and in the second case, the best score is $S(i + 1)$. The following loop calculates the best possible total score, given a large array S with all entries initialized to 0:

```
for i = n downto 1:
    S[i] = max(p[i] + s[i+f[i]+1], s[i+1])
return S[1]
```

The running time is easily seen to be $O(n)$ (possibly with some additional tinkering to catch indices off of the end of the array).

5. AVERAGE-SUM (20 points)

Recall that the NP-complete SUBSET-SUM problems asks, given a set of non-negative integers S and a target K, whether S has a subset S' with $\sum_{i \in S'} i = K$.

The AVERAGE-SUM problem asks, given just a set of non-negative integers S, whether S has a subset S' with $\sum_{i \in S'} i = \frac{1}{|S|} \sum_{i \in S} i$, where $|S|$ is the number of elements in S. It is similar to the SUBSET-SUM problem, except now the target value is always the average value in S.

Give a polynomial-time algorithm for AVERAGE-SUM, or prove that it is NP-complete.

Solution:

We'll show that it's NP-complete. It's easy to see that it is in NP (guess S' and verify). To show that it is NP-hard, reduce from SUBSET-SUM. Given an input (S, K) to SUBSET-SUM, we will construct a new set T in polynomial time such that (S, K) is in SUBSET-SUM if and only if T is in AVERAGE-SUM, by first removing elements of S that are too big, and then adding a single huge new element to S to bring the average up to K.

Let $S_K = \{x \in S : x \leq K\}$. Let $n = |S_K|$ and let $S_K = \{x_1, x_2, \ldots x_n\}$. Let $T = \{x_1, x_2, \ldots x_n, y\}$, where $y = (n+1)K - \sum_{i=1}^{n} x_i$. Then $|T| = n + 1$ and $\sum_{i \in T} i = (n+1)K$, so $\frac{1}{T} \sum_{i \in T} i = K$.

If there is a subset S' of S that sums to K, then this same S' is a subset of T; it follows that (S, K) in SUBSET-SUM implies T in AVERAGE-SUM. Conversely, suppose that T is in AVERAGE-SUM, i.e., that there is some subset T' of T that sums to K. If $y > K$ then y is not in T' and T' is a subset of S that sums to K. If $y \leq K$ then $\sum_{i \in S_K} i \geq nK$ which implies that every $x_i = K$. In this case there is also a subset S' of S that sums to K (any single x_i will do). So T in AVERAGE-SUM implies (S, K) in SUBSET-SUM as well, and the reduction works as advertised.

CS 365 home page: http://zoo.cs.yale.edu/classes/cs365/