Midterm Exam

Instructions

Please put your name at the top of every page (1 point).

Please write your answers on the exam if possible. More paper is available if you need it.

Work alone. Do not use any notes or books. You have approximately 75 minutes to complete this exam. There are four problems on this exam, for a total of 100 possible points (including the point for putting your name at the top of every page).

Unless otherwise specified, you should justify your answers and give running times for all algorithms. 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. Asymptotic notation (20 points)

Prove or disprove: $3^n = O(2^n)$.

Solution:

Here's a disproof. From the definition of $O(2^n)$ we have that 3^n is only in $O(2^n)$ if there exists some n_0 and c such that $3^n \leq c2^n$ for all $n > n_0$. Fix some c. We will find an n for which $3^n > c2^n$. Let $n = 2 \lg c$; then $3^n = 3^{2 \lg c} = c^{2 \lg 3}$, where $2 \lg 3 = 3.16992\ldots$, but $c2^n = c \cdot c^2 = c^3 < c^{3.1}$. It follows that if n exceeds $\max(n_0, 2 \lg c)$, that $3^n > c2^n$, which proves that 3^n is not $O(n^2)$.

Another disproof uses limits: $\lim_{n\to\infty} \frac{3^n}{2^n} = \lim_{n\to\infty}(3/2)^n$ which diverges. So again 3^n is not in $O(2^n)$.

2. Some recurrences (6 points each)

Below are some recurrences. Give the solution (in big Θ form) for each. In each case you should assume that $T(n)$ is bounded by a constant for small values of n.

1. $T(n) = 3T(n/2) + 1$. *Solution:* $T(n) = \Theta(n^{\lg 3})$ *by Master Theorem.*
2. $T(n) = 3T(n/2) + n$. *Solution:* $T(n) = \Theta(n^{\lg 3})$ *by Master Theorem.*
3. $T(n) = 3T(n/2) + 3^n$. *Solution:* $T(n) = \Theta(3^n)$ *by Master Theorem.*
4. $T(n) = 3T(n-1) + 1$. *Solution:* $T(n) = \Theta(3^n)$. *Proof: First we will show that* $T(n) = \sum_{i=0}^{n-1} 3^i + 3^n T(0)$. *This equation holds when* $n = 0$*; for larger* n*, we have* $T(n) = 3T(n-1) + 1 = 3(\sum_{i=0}^{n-1} 3^i + 3^{n-1}T(0)) + 1 = 1 + \sum_{i=1}^{n} 3^i + 3^n T(0) = \sum_{i=0}^{n} 3^i + 3^n T(0)$. *Since the second term is* $\Theta(n)$*, we have* $T(n) = \Omega(3^n)$. *To show that it is in* $O(n)$*, bound the summation by* $\sum_{i=0}^{n} 3^i = \sum_{j=0}^{n} 3^{n-j} \leq 3^n \sum_{j=0}^{\infty} 3^{-j} = 3^n/(1 - 1/3) = \Theta(3^n)$.

3. Graph 2-colorability (25 points)

Give an algorithm for determining if a graph is two-colorable, i.e. if it is possible to color every vertex red or blue so that no two vertices of the same color have an edge between them. Your algorithm should run in time $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. You should assume that the graph is undirected and that the input is presented in adjacency-list form.

Solution:

The key insight is that for any graph that is 2-colorable, choosing a color for one node in a connected component fixes the colors of all other nodes in the component: any node that is an odd distance away gets the opposite color, while any node that is an even distance away gets the same color. So we do not need to consider all possible colorings, and it is enough to color all nodes using DFS or BFS.

Here is one possible algorithm based on DFS. We assume we have an array `mark` with indexes spanning V, initialized to `none`.

```
ColorDFS(G, v, color):
    mark[v] = color
    for each neighbor u of v in G:
        if mark[u] = color:
            return false
        else if mark[u] = none:
            if ColorDFS(G, u, opposite(color)) = false
                return false
    // else
    return true

IsTwoColorable(G):
    for each vertex v of G:
        if mark[v] = none:
            if ColorDFS(G, v, red) = false:
                    return false
    // else
    return true
```

The running time of this procedure is $O(V + E)$ as in standard depth-first search.

4. Finding a missing number (30 points)

An array of n elements contains all but one of the integers from 1 to $n+1$.

1. Give the best algorithm you can for determining which number is missing if the array is sorted, and analyze its asymptotic worst-case running time.

2. Give the best algorithm you can for determining which number is missing if the array is *not* sorted, and analyze its asymptotic worst-case running time.

Solution:

1. If the array is sorted, we can use binary search. We are looking for the smallest index i for which $A[i] = i+1$; this will be our missing number. If $A[n/2] = n/2+1$, i is less than or equal to $n/2$, and we can recurse on the first half of A; otherwise it is greater than $n/2$, and we can recurse on the second half of A. In either case we get the recurrence $T(n) = T(n/2)+\Theta(1)$ which gives a worst-case running time of $\Theta(\log n)$.

2. If the array is not sorted, then in the worst case we will have to look at every location in the array (otherwise what we think is the missing element could be in the location we didn't look in). So the best we can hope to do is get an $O(n)$ algorithm. One such algorithm creates an auxiliary array B with indices 1 to $n+1$, initializes all locations to 0, scans through A setting $B[A[i]] = 1$ for each i from 1 to n, and finally scans through B looking for a zero. Each of these three steps takes $O(n)$ time, so we have an $O(n)$ algorithm for this case.