Sample Solution to Midterm

Problem 1. Short answer questions (5 × 3 = 9 points)

(a) Arrange the following functions in order from the slowest growing function to the fastest growing function. You may simply state the answer without any additional justification.

$$\lg n \quad (\lg \lg n)^2 \quad n^{1/4}$$

Answer:

$$(\lg \lg n)^2 \quad \lg n \quad n^{1/4}$$

(b) True or false: Given a heap A of n integers, and an integer k, one can determine in $O(\log n)$ time whether k is in A. Justify your answer.

Answer: False. Suppose all of the internal nodes of the heap have higher values than all of the leaves. Then, if we are given an element that is smaller than all of the internal nodes, we have to essentially compare with all of the leaves to determine whether the element is in the heap.

(c) True or false: The worst-case running time of Quicksort is worse than that of Mergesort, while the best-case running time of Quicksort is better than that of Mergesort. Justify your answer.

Answer: The worst-case running times of Quicksort and Mergesort are $\Theta(n^2)$ and $\Theta(n \lg n)$, respectively. The best-case running times of Quicksort and Mergesort are $\Theta(n \lg n)$ and $\Theta(n \lg n)$, respectively. So, in terms of asymptotic running time, the statement is false since the best-case running time of Quicksort is not *better* than that of Mergesort. Full credit will be given to whoever got the worst- and best-case running times of both algorithms correct.

Huffman codes. Consider a set S of 8 characters $a, b, c, \ldots, h$. For each of these two frequency distributions (parts d and e below), give the optimal Huffman code. You may simply draw the Huffman code tree without any additional justification.

(d) All the characters have the same frequencies.

Answer: Balanced binary tree with eight leaves. The code set is the set of all 3-bit numbers.

(e) $a : 1 \;\; b : 1 \;\; c : 2 \;\; d : 3 \;\; e : 5 \;\; f : 8 \;\; g : 13 \;\; h : 21.$

Answer: For the character set a, b, c, d, e, f, g, and h, with the frequencies 1, 1, 2, 3, 5, 8, 13, and 21, respectively, the Huffman code is 0000000, 0000001, 000001, 00001, 0001, 001, 01, and 1, respectively. In general, for an n-length Fibonacci sequence, the Huffman encoding is $n-1$ 0's for a_1, and $n-i$ 0's followed by a 1 for a_i, $2 \leq i \leq n$.

Problem 2. (5 points) Recurrence

Derive a tight bound for the following recurrence relation. Assume that $T(n)$ is $\Theta(1)$ for $n \leq 3$.

$$T(n) = T(n/2) + T(n/3) + 2n.$$

Answer: When we write the recursion tree for the above recurrence, we obtain a contribution of $2n$ from level 0, $2n(5/6)$ from level 1, $2n(5/6)^2$ for level 2, and so on. In general, level i contributes $(5/6)^i 2n$. Therefore, total contribution is at most $2n(1 + (5/6) + (5/6)^2 + \ldots)$, which is the sum of a geometrically decreasing sequence. Taking the sum over infinite terms yields the upper bound $2n(1 - 5/6) = 12n$.

For the lower bound, it is easy to see from the recurrence that $T(n)$ is at least $2n$. Therefore, $T(n) = \Theta(n)$.

Problem 3. (5 + 5 = 10 points) Exponentiation

Consider the following algorithm for calculating the nth power of a number a.

```
POWER(a, n)
1. if n = 0
2.     return 1
3. else
4.     return a × POWER(a, n − 1)
```

(a) Write a recurrence relation for the number of multiplications performed by POWER. Solve the recurrence.

Answer: Line 4 performs one multiplication and recurses from n to $n-1$. Lines 1 and 2 indicate that $T(0) = 0$. So we obtain the following recurrence.

$$T(n) = \begin{cases} T(n-1)+1 & n > 0 \\ 0 & n = 0 \end{cases}$$

By iteration or the recursion tree approach, we get $T(n) = n$.

(b) Describe a faster algorithm for the problem that performs $O(\lg n)$ multiplications. Justify the upper bound on the number of multiplications performed by your algorithm. (*Hint:* Think divide-and-conquer.)

Answer: If n is even, we can write $a^n = (a^{n/2})^2$. If n is odd, we can write $a \times (a^{\lfloor n/2 \rfloor})^2$. Thus, we can reduce a problem of size n to a problem of size $\lfloor n/2 \rfloor$ and then at most 2 additional multiplications. So the algorithm is:

```
FASTPOWER(a, n)
1. if n = 0
2.     return 1
3. else
4.     x = FASTPOWER(a, ⌊n/2⌋)
5.     if n is odd
6.         return a × x × x
7.     else
8.         return x × x
```

The recurrence for the number of multiplications is $T(n) \le T(\lfloor n/2 \rfloor) + 2$, and $T(0) = 0$. Replacing $\lfloor n/2 \rfloor$ by $n/2$ and using the Master theorem, we get $T(n) = O(\lg n)$.

Problem 4. (5 + 5 = 10 points) Comparing two lists

You and your friend each have a long list of favorite restaurants, and would like to determine the intersection of the two lists; i.e., all restaurants that are in both of your lists. Assume that each list is of size n.

(a) Design an efficient algorithm for the above problem. State the worst-case running-time of your algorithm. The more efficient your algorithm is in terms of its worst-case running time, the more credit you will get.

Answer: We first note that names can be compared against each other (alphebatically); so there is a notion of ordering among restaurant names in the same way as there is a notion of ordering among numbers. Let the two lists be M and V. A simple $\Theta(n^2)$ time algorithm can be obtained by checking for each pair of elements $a \in M$ and $b \in V$ to see whether $a = b$. Since there are n^2 pairs, this takes $O(n^2)$ time in the worst-case. Since we will have to go through all of the pairs to determine the final result, the worst-case input will require $\Omega(n^2)$ time. Thus, the running time of the algorithm is $\Theta(n^2)$. But we can do better.

We sort each of the lists M and V and then scanning through the sorted lists for identical elements using a procedure similar to the merge procedure of mergesort.

1. Sort M and V separately, both in nonincreasing order
2. $i = j = 1$
3. **while** $i \leq n$ and $j \leq n$
4. **if** $M[i] = V[j]$ print $M[i]$
5. **if** $M[i] \geq V[j]$ $j = j + 1$
6. **if** $M[i] \leq V[j]$ $i = i + 1$

For the running time of the algorithm, we note that sorting of the two lists takes $O(n \lg n)$ time using heapsort or mergesort. Finally the scan through the two lists (steps 4 through 6) takes $O(n)$ time. Therefore, the total running time of the algorithm is $O(n \lg n)$.

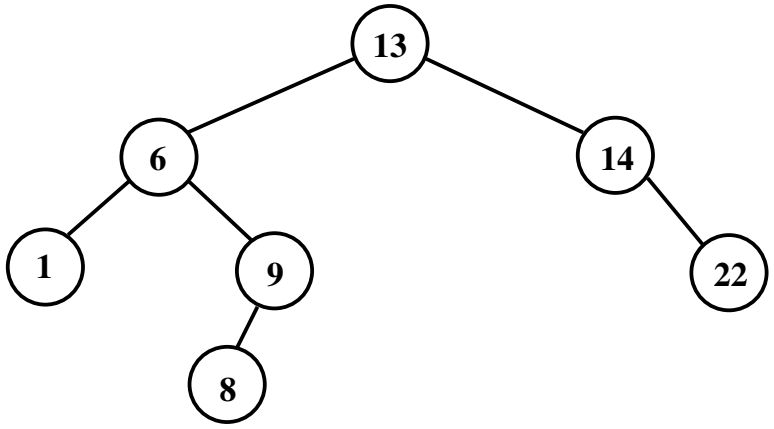
(b) Design an algorithm for the same problem with smaller *expected* running time. State the expected running time of your algorithm.

Answer: We hash the elements of M into a table of size n, using a suitable hash function. Note that there may be collisions, but if the hash function is chosen well, the expected time will be $\Theta(n)$. Next, for each element x of V, we search whether x is in M. Since the expected time for search is $\Theta(1)$ (constant), we obtain that the total expected time is $\Theta(n)$.

Problem 5. (3 + 7 = 10 points) Binary search trees

(a) Label the following binary tree with numbers from the set $\{6, 22, 9, 14, 13, 1, 8\}$ so that it is a legal binary search tree.

Answer:



(b) Recall that all of the binary search tree operations studied in class, namely, INSERT, DELETE, SEARCH, MAXIMUM, MINIMUM, SUCCESSOR, and PREDECESSOR take $O(h)$ time in the worst-case, where h is the height of the binary tree. In this exercise, we would like to augment the binary search tree in such a way that the operation MINIMUM can be calculated in $O(1)$ time.

We add a new field called SUBTREEMIN$[x]$, which stores for each node x the minimum key value among all of the nodes in the subtree rooted at x. This way, for a given binary search tree T, we can perform MINIMUM$[T]$ by simply returning SUBTREEMIN$[root[T]]$.

Describe how to modify the INSERT and DELETE operations so as to maintain the SUBTREEMIN$[\cdot]$ field and yet keep the worst-case running time of the operations to be $O(h)$. (A clear description in words would suffice.)

Answer: For inserting key k, just go through the path as in regular insertion, and change the SUBTREEMIN$[x]$ value at any node x to k if k is smaller than SUBTREEMIN$[x]$. Finally, when a new (leaf) node is created with key k, initialize the SUBTREEMIN field for this node to be k. This modification adds only a constant number of operations per level to the original insert algorithm; hence the running time is still $O(h)$.

For deleting, we consider three different cases. If the deleted node, say x, is a leaf node, then we trace the path from x to the root until we reach the first node y such that y is the right child of its parent. During this traversal, we update the SUBTREEMIN value for each node traversed (until and including y) to be the key of the parent of x.

If the node to be deleted, say x, has only one child, then we delete the node as usual and set its child, say u to be a child of the parent of x. Then, as in the above case, we trace the path from u to the root until we reach the first node y such that y is the right child of its parent. During this traversal, we update the SUBTREEMIN value for each node traversed (until and including y) to be the SUBTREEMIN value in u.

Finally, if the node to be deleted, say x, has both children, we delete the node and replace it by the predecessor, then set the SUBTREEMIN value for the replacement node to be the

minimum of the predecessor key and the SUBTREEMIN value stored in the left child of x in the original tree.