CS4231: Analysis of Algorithms, I

Midterm Exam Solutions, Thursday October 25, 2018

Problem 1 [18 points, 6 points per part]

Give asymptotic solutions $T(n) = \Theta(f(n))$ for the following recurrences. Assume that $T(n)$ is constant for sufficiently small n. State in each case the method that you used (eg. Master method, etc.). No justification is needed. For example, you can say, "$T(n) = \Theta(n)$, I used the Master method." No more explanation is required.

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

Answer: $T(n) = \Theta(n^{\log_2 3})$

Master Theorem
a=3, b=2, $f(n) = n$, $n^{\log_b a} = n^{\log_2 3}$. Note that $1 < \log_2 3 \Rightarrow n = O(n^{\log_2 3 - \varepsilon})$ for ε sufficiently small, for example for ε<0.5 . Therefore, case 1 applies and $T(n) = \Theta(n^{\log_2 3})$.

b. $T(n) = 2T(n-1) + 1$

Answer: $T(n) = \Theta(2^n)$.
Expanding the recurrence: $T(n) = 2T(n-1) + 1 = 2[2T(n-2) + 1] + 1 = 2^2 T(n-2) + 2 + 1 =$

$$= 2^3 T(n-3) + 2^2 + 2 + 1 = \cdots = 2^{n-1} T(1) + \sum_{i=0}^{i=n-2} 2^i = 2^{n-1} T(1) + 2^{n-1} - 1 = \Theta(2^n)$$

The solution can be computed also with the recursion tree method. In this case the recursion tree is a complete binary tree with height n-1, the internal nodes have cost 1 and the leaves have cost $T(1)$=constant.

c. $T(n) = 2T(n/3) + T(n/4) + n$

Answer: $T(n) = \Theta(n)$

This is similar to Homework 1, problem 3, part f. Note that $\frac{2n}{3} + \frac{n}{4} = \frac{11}{12} n$ and 11/12 < 1. We can show that the solution is $\Theta(n)$ by the Substitution method or by the Recursion tree method; see the solutions to Homework 1.

Problem 2 [16 points, 4 points per part]
For each pair of functions *f, g* below, determine whether $f=o(g)$, $f=\Theta(g)$, or $f=\omega(g)$ and circle the correct answer. No justification is needed. All logarithms are with base 2.

1. $f(n) = 4n^4 + 2n^3 + 10$, $g(n) = (n^2 - 1)^2$ — $f=\Theta(g)$

Justification: $g(n) = n^4 - 2n^2 + 1$. Both functions are polynomials of degree 4, thus they are both $\Theta(n^4)$. Hence, $f=\Theta(g)$.

2. $f(n) = 2^{n+5}$, $g(n) = 2^{2n}$ — $f=o(g)$

Justification: $f(n) = 2^5 \cdot 2^n = 32 \cdot 2^n$ and $g(n) = 4^n$.
Therefore, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{32}{2^n} = 0.$

3. $f(n) = n\sqrt{n}$, $g(n) = n(\log n)^4$ — $f=\omega(g)$

Justification: We know that any (positive) power of *n* grows faster than any power of log*n* , i.e. $n^c = \omega((\log n)^d)$ for all *c,d* >0. In particular, $\sqrt{n} = n^{0.5} = \omega((\log n)^4)$.
Therefore, $n\sqrt{n} = \omega(n(\log n)^4)$.

4. $f(n) = n^4$, $g(n) = 2^{3\log n}$ — $f=\omega(g)$

Justification: $g(n) = (2^{\log n})^3 = n^3$. Thus, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} n = \infty$. Hence $f=\omega(g)$.

Problem 3 [10 points]

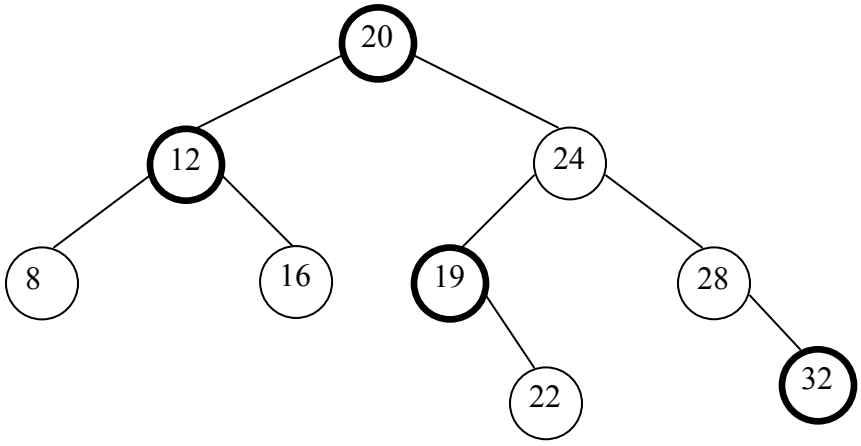
See solutions to Homework 2 that will be posted on Monday.

Problem 4 [18 points; 6 points per part]

Answer the following questions. In the True-False questions, circle True or False and justify your answer by either providing an algorithm (or pointing to a known algorithm), or by providing an argument why the task in the claim is impossible. Your justifications are *more important* than the True/False designations, and must be ***succinct, precise and convincing***.
All the parts can be adequately answered and justified in a few lines.

All the times below refer to worst-case time.

1. Consider the following tree where the dark nodes are black and the light nodes are red. Give three reasons why this is not a legal red-black tree.



Reason 1: There is a red node with a red child: the nodes with keys 24 and 28.

Reason 2: The right subtree of the root contains a node with key 19, which is smaller than the key of the root.

Reason 3: The path from the root to the missing (nil) left link of the node with key 28 has only one black node, while all the other paths from the root to a missing link have two black nodes.

2. *Claim*: Given an (unsorted) array A with n integers in the range 1 to n^2, we can determine in O(n) worst-case time whether there exist two indices i,j such that $A[i]+A[j]= n^2$.

True

First, sort the array A. As shown in class, a set of integers in the range 0 to n^d , where d is constant, can be sorted in O(n) time using Radix Sort. After the array is sorted, the problem is similar to a homework problem.

```
i=1; j=n;
while ( i < j ) do
    if  (A[i] + A[j] < n^2) then i =i+1
     else  if  (A[i] + A[j] > n^2) then j = j-1
          else return "Yes";
return "No"
```

3. The new congress will simplify the tax code as follows.
There will be only 3 tax brackets:
- the top 10% of people with highest income will pay 35% tax
- the next 20% of people in order of income will pay 20% tax
- the bottom 70% of people will pay 5% tax

We are given an array $A[1\ldots n]$ with the records of all the people and their incomes; the array is sorted by name. Assume for simplicity that no two people have the same income.

Claim: We can compute the taxes of all the people in O(n) worst-case time.

True

Use the linear time selection algorithm on the incomes of the people twice: to compute the income x that is 10% from the top, and to compute the income y that is 30% from the top (i.e., the incomes x, y of rank $9n/10$ and $7n/10$ respectively, counting from smallest to largest). Compare the income in every record with x and y and compute accordingly the tax for each person.

Problem 5 [18 points]

Consider the following algorithm.

Input: Array $A[1\ldots n]$ of numbers
Output: Array $B[2\ldots n]$

```
for i=2 to n do
  {  B[i] = ∞;
     for j= 1 to i-1 do
        if ( 0 ≤ A[i]-A[j] < B[i] ) then B[i] = A[i]-A[j]
  }
return B
```

1. [6 points]
Give the time complexity of this algorithm in $\Theta(.)$ form. Justify your answer.

The time complexity is $\Theta(n^2)$:
The time is proportional to $\sum_{i=2}^{n}\sum_{j=1}^{i-1}1 = \sum_{i=2}^{n}(i-1) = 1+2+3+\ldots+(n-1) = \frac{n(n-1)}{2} = \Theta(n^2)$

2. [12 points]
Give a more efficient algorithm that computes the same output. You do not have to give pseudocode; clear, precise description in English suffices. State explicitly the (worst-case) running time of your algorithm, and justify its correctness and the running time. (*Hint:* Use a suitable data structure.)

From the above algorithm we see that for each i, $B[i] = \infty$ if $A[i]$ is smaller than all the previous elements $A[j]$, $j<i$, and otherwise $B[i] = \min\{ A[i]-A[j] \mid 1\le j<i, A[i]\ge A[j] \}$, equivalently, $B[i]$ is equal to the difference between $A[i]$ and a previous element $A[j]$, $j<i$, that is smaller than or equal to $A[i]$ and is as close to $A[i]$ as possible. In other words, if we let $S(i\text{-}1)$ denote the set consisting of the first i-1 elements of the array A, and we let $\text{pred}(S(i\text{-}1), A[i]) = \max\{s \in S(i-1) \mid s \le A[i]\}$ denote the element of $S(i\text{-}1)$ that is $\le$ and closest to $A[i]$ (the "predecessor" of $A[i]$ in $S(i\text{-}1)$), then

$$B[i] = \begin{cases} \infty & \text{if } A[i] < \min S(i-1) \\ A[i] - \text{pred}(S(i-1), A[i]) & \text{otherwise} \end{cases}$$

Thus, we can determine easily $B[i]$ if we represent the sets $S(.)$ using a data structure that supports efficiently the operations min, predecessor, and insertions. A balanced tree such as a red-black tree is a suitable such data structure. In case of duplicates in the array A, we can choose to keep just one copy of the key in the red-black tree or alternatively we can keep duplicate keys, it does not matter – either way will yield the same result here.

Given a set S of elements represented by a red-black tree with root *root* and another key a, the following function returns *pred*(S,a) or ∞ if all elements of S are greater than a.

```
function pred(root,a)
x=root;
y=∞;
while x≠null do
    if (x.key > a) then  x=x.left
      else { y=x.key; x=x.right }
return y
```

The algorithm to compute the array B is then as follows.

```
root = new empty red-black tree;
INSERT(root, A[1]);
for i=2 to n do
   {  y = pred(root, A[i]);
      if (y=∞)  then B[i] =∞  else B[i] = A[i]- y;
      INSERT(root, A[i])
   }
return B
```

The algorithm performs n insertions and n-1 *pred* calls, each of which takes O(log$n$) time, therefore the time complexity of the algorithm is O(n log$n$).

Problem 6. [20 points]

We have n coins, all of which have the same weight except for one that is fake and weighs less. We have a scale which we can use to compare the weights of any two (disjoint) subsets of coins: if we place a subset A of coins on one side and subset B on the other side, the scale tells us whether the two sides have the same weight, and if not, it tells us which side is heavier. Note that every weighing has *three* possible outcomes: A is heavier, or B is heavier, or A and B have equal weight. Each use of the scale counts as one step. We would like to identify the fake coin in the minimum number of steps.

1. [12 points] Give an algorithm that identifies the fake coin using as few steps as you can. Give the exact number of steps (not just an asymptotic expression). Justify the correctness of your algorithm and the running time.

We can find the fake coin in $\lceil \log_3 n \rceil$ steps. We use Divide and Conquer.

- If there is only one coin, then this is the fake coin.
- Otherwise, i.e. if $n>1$, then we divide the given set of coins into three sets A, B, C, where A and B have the same cardinality, as close as possible to $n/3$. Specifically, if n is a multiple of 3, then A, B, C are all of size $n/3$. If $n=1 \bmod 3$, i.e. $n = 3k+1$ then A and B have size k and C has size $k+1$. If $n=3k+2$, then A and B have size $k+1$ and C has size k.
- Weigh A against B. If A weighs less, then we know that A contains the fake coin and we recurse on A. Similarly, if B weighs less, then we recurse on B. If A and B have the same weight, then we know that C must contain the fake coin and we recurse on C.

The recurrence for the number of steps is $T(n) = T(\lceil n/3 \rceil) + 1$ if $n>1$, and $T(1)=0$. The solution is $T(n) = \lceil \log_3 n \rceil$.

2. [8 points] Show a lower bound on the number of steps required to identify the fake coin. The lower bound should be again in the form of an exact expression (not just asymptotic).

Every algorithm for the problem corresponds to a decision (comparison) tree G, which at each internal node compares two subsets of coins and at each leaf identifies one of the coins as the fake coin. Note that when we compare the weights of two subsets A, B of coins, there are 3 possible outcomes: A is lighter, B is lighter, A and B have the same weight. Accordingly, every internal node of the tree can have three children, i.e. G is a ternary tree. There are n possibilities for the fake coin, hence G has at least n leaves. A ternary tree of height h can have at most 3^h leaves. Since G has n leaves, G must have height at least $\lceil \log_3 n \rceil$. Therefore, every algorithm requires at least $\lceil \log_3 n \rceil$ steps in the worst case.