CPSC 320 Sample Midterm 2
November 2010

[6] 1. Short Answers

[3] a. If we can not use a recursion tree to prove a tight bound on the value of a function $T(n)$ defined by a recurrence relation, can we use the Master theorem instead? Explain why or why not.

Solution : The proof of the Master theorem used recursion trees, so if we can use the Master theorem to prove a bound on $T(n)$ then we can also use recursion trees. Hence if we can not use recursion trees, then we can not use the Master theorem either.

[3] b. The first (and only) result we looked at in our outline of the proof of the Master theorem stated that

$$T(n) = n^{\log_b a} + \sum_{i=0}^{t-1} a^i f(n/b^i)$$

Explain where the term $a^i f(n/b^i)$ came from, by relating each factor to the tool we used to prove the lemma.

Solution : The i^{th} row of the recursion tree contained a^i nodes, since each node of the tree has a children. Each node worked on n/b^i elements (the size of the subproblems at one level are $1/b^{th}$ the size of the subproblems at the next higher level), hence doing $f(n/b^i)$ work.

[5] 2. In class, we looked at an implementation of a binary counter, and proved that if we use as potential function $\Phi(D_i)$ the number of 1 bits in the counter, then the amortized cost of an INCREMENT operation is in $\Theta(1)$. Suppose instead that we had used the number of 0 bits in the counter as the potential function (note: this breaks the requirement that $\Phi(D_0) = 0$, so it would not be a valid choice). What would the amortized cost of INCREMENT be in this case?

Solution : Suppose the counter contains k bits, and that INCREMENT looks at x bits of the counter. The real cost of the operation is thus x. The first $x - 1$ bits are changed from 1 to 0, thus increasing the potential by $x - 1$. Then the last bit is changed from 0 to 1, which decreases the potential by 1. The potential change is thus $x - 1 - 1$ which is $x - 2$. Hence the amortized cost of the operation is $2x - 2$, which is $\Theta(k)$ in the worst-case. We could therefore *not* use this potential function to prove a good bound on the worst-case running time of a sequence of n operations on the counter.

[14] 3. Given a set T of n teams competing in some sport, a *round-robin tournament* is a collection of games in which each team plays each other team exactly once. We can describe such a

schedule by an array of triplets, where the triplet (d, i, j) means that team i will play against team j on day d. A schedule is *reasonable* if no team plays more than one game per day, and *optimal* if it uses the smallest number of days out of all reasonable schedules.

[8] a. Design a **divide-and-conquer** algorithm that takes as input an integer n, and produces an optimal schedule as long as n is a power of 2.

Solution :

```
Algorithm RoundRobinSchedule(n)
  if n = 1 then
    return { }

  //
  // This is not needed, as the rest of the pseudo-code
  // would handle  this situation  correctly,  but  you
  // could also use this as a base case.
  //
  if n = 2 then
    return { (1, 1, 2) }

  //
  // Schedule for the first half of the teams.
  //
  list1 ← RoundRobinSchedule(n/2)

  //
  // Schedule for the second half of the teams.
  //
  list2 ← RoundRobinSchedule(n/2)
  for each game in list2 do
    game.team1 ← game.team1 + n/2
    game.team2 ← game.team2 + n/2
  endfor

  //
  // Every team in the first half must play against
  // every team in the second half.
  //
  list3 ← { }
  for i ← 1 to n/2 do
    for j ← n/2+1 to n do
      add (n/2 + (j - i) mod (n/2), i, j)
    endfor
  endfor
```

```
    return append(list1, list2, list3)
```

[3] b. Prove that your algorithm produces an optimal schedule whenever n is a power of 2.

Solution : If n is a power of 2, then we produce a schedule that runs for $n-1$ days, and every team plays one game every day. We can not produce a shorter reasonable schedule, since teams would then need to play more than one game on at least one day.

[3] c. Analyze the running time of the algorithm you described in your answer to part (a).

Solution : The running time satisfies the recurrence

$$T(n) = \begin{cases} 2T(n/2) + \Theta(n^2) & \text{if } n > 2 \\ \Theta(1) & \text{if } n \leq 2 \end{cases}$$

and so by case 3 of the Master Theorem (the regularity condition clearly holds for $f(n) = n^2$), $T(n) \in \Theta(n^2)$.

[6] 4. Write a recurrence relation that describes the worst-case running time of the following algorithm as a function of n. Recall that the call `BinarySearch(A, first, last, x)` runs in $O(\log(\texttt{last} - \texttt{first} + 1))$ time.

```
Algorithm Armadillo(A, first, n, x)
   y ← 1
   s ← ⌊√n⌋
   if (s > 1) then
      while (first + s < n) do
         y ← y * Armadillo(A, first, s, x) +
                BinarySearch(A, first, first + s - 1, x)
         first ← first + s
      endwhile
   endif
   return y
```

Solution :

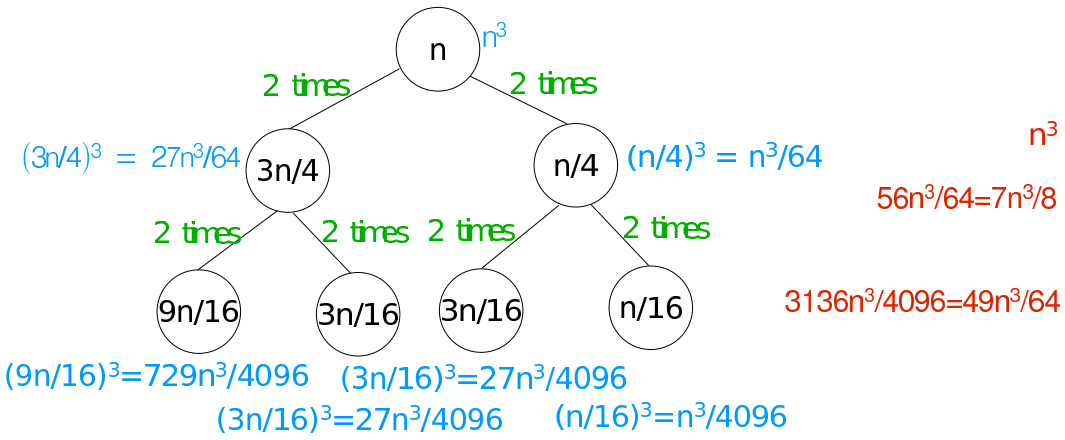
$$T(n) = \begin{cases} \lfloor\sqrt{n}\rfloor T(\lfloor\sqrt{n}\rfloor) + \Theta(\lfloor\sqrt{n}\rfloor \log n) & \text{if } n \geq 4 \\ \Theta(1) & \text{if } n \leq 3 \end{cases}$$

[9] 5. Prove upper and lower bounds on the function $T(n)$ defined by

$$T(n) = \begin{cases} 2T(3n/4) + 2T(n/4) + n^3 & \text{if } n \geq 4 \\ 1 & \text{if } n \leq 3 \end{cases}$$

Your grade will depend on the quality of the bounds you provide (that is, showing that $T(n) \in \Omega(1)$ and $T(n) \in O(100^n)$, while true, will not give you many marks).

Solution : Let us first establish an upper bound for $T(n)$, by drawing a recursion tree (duplicates nodes where not all drawn to make the picture easier to read).



As we can see from the recursion tree in the figure, the children of each node N do seven-eighth the amount of work done at N (note: I was not expecting you to actually compute the exact amount of work done on the third row of the tree). Thus the amount of work done by row i of the recursion tree is at most $(7/8)^i n^3$, up to the level where the last leaf occurs. Thus the total amount of work is

$$\sum_{i=0}^{\log_{4/3} n} (7/8)^i n^3 \leq \sum_{i=0}^{\infty} (7/8)^i n^3 = n^3 \sum_{i=0}^{\infty} (7/8)^i = \frac{1}{1-7/8} n^3 = 8n^3.$$

This means that $T(n) \in O(n^3)$.

For the lower bound, observe that the root of the recursion tree performs n^3 work, and hence $T(n) \in \Omega(n^3)$. Putting the upper and lower bounds together, we conclude that $T(n) \in \Theta(n^3)$.