CSCI 3110 Assignment 6 Solutions

December 5, 2012

2.4 (4 pts) Suppose you are choosing between the following three algorithms:

1. Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
2. Algorithm B solves problems of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.
3. Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in big-O notation), and which would you choose?

These are

1. $T(n) = 5T(n/2) + \Theta(n)$
   We have $a = 5$, $b = 2$, $f(n) = \Theta(n)$. $n^{\log_b a} = n^{\log_2 5} > n^2$, so $f(n) = O(n^{\log_b a - \epsilon})$. This is Case 1 of the Master Theorem, thus $T(n) = \Theta(n^{\log_2 5})$.
2. $T(n) = 2T(n-1) + O(1)$
   This recurrence does not fit the Master Theorem. However, observe that the number of subproblems doubles n times and each subproblem uses $O(1)$ time. Therefore $T(n) = \Theta(2^n)$.
3. $T(n) = 9T(n/3) + T(n^2)$
   We have $a = 9$, $b = 3$, $f(n) = \Theta(n^2)$. $n^{\log_b a} = n^{\log_3 9} = n^2$, so $f(n) = \Theta(n^{\log_b a})$. This is Case 2 of the Master Theorem. Thus, $T(n) = \Theta(n^2 \log n)$.

The algorithm which grows the slowest asymptotically is the 3rd algorithm with a running time of $\Theta(n^2 \log n)$, so we would choose it.

2.5 (2 pts) Solve the following recurrence relations and give a Θ bound for each of them.

(f) $T(n) = 49T(n/25) + n^{3/2} \log n$

We have $a = 49$, $b = 25$, $f(n) = \Theta(n^{3/2})$. $n^{\log_b a} = n^{\log_{25} 49}$, so $f(n) = \Omega(n^{\log_b a + \epsilon})$. This is Case 3 of the Master Theorem, so we check the regularity condition $af(n/b) \leq cf(n)$ for some $c < 1$ and sufficiently large n. That is, $49((n/25)^{3/2} \log(n/25)) \leq cn^{3/2} \log n$. Since $49((n/25)^{3/2} \log(n/25)) = (49/25^{3/2})(n^{3/2} + (\log n - \log(25))$ the condition holds for $c = (49/25^{3/2}) = 49/125 < 1$. Thus, $T(n) = \Theta(n^{3/2} \log n)$.

(i) $T(n) = T(n-1) + c^n$, where $c > 1$ is some constant

$T(n) = T(n-1) + c^n = T(n-2) + c^{n-1} + c^n$. We can thus unroll this to $T(n) = 1 + c + c^2 + \ldots c^n$. As $c > 1$, this is $\Theta(c^n)$.

2.12 (2 pts) How many lines, as a function of n (in $\Theta(\cdot)$ form), does the following program print? Write a recurrence and solve it. You may assume n is a power of 2.

```
F(n)
1  if n > 1:
2      PRINT_LINE("still going")
3      F(n/2)
4      F(n/2)
```

This program prints one line and then splits into two subprograms with half of the input size each. The recurrence is $T(n) = 2T(n/2) + 1$. Using the Master Theorem, $n^{\log_b a} = n^{\log_2 2} = n$. Since $f(n) = 1$, we have that $f(n) = O(n^{\log_b a - \epsilon})$ so the solution is $\Theta(n)$ lines.

2.25 (4 pts) In Section 2.1 we described an algorithm that multiplies two n-bit binary integers x and y in time n^a, where $a = \log_2 3$. Call this procedure fastmultiply(x, y).

(a) We want to convert the decimal integer 10^n (a 1 followed by n zeros) into binary. Here is the algorithm (assume n is a power of 2):

```
PWR2BIN(n)
1  if n = 1
2      return 1010_2
3  else
4          z =???
5          return FASTMULTIPLY(z, z)
```

Fill in the missing details. Then give a recurrence relation for the running time of the algorithm, and solve the recurrence.

The missing details are $z =$ PWR2BIN$(n/2)$. The recurrence is $T(n) = T(n/2) + n^a$. Unrolling this, we get $\Sigma_{i=0}^{\log_2 n}(n/2^i)^a$ which is $\Theta(n^a)$.

(b) Next, we want to convert any decimal integer x with n digits (where n is a power of 2) into binary. The algorithm is the following:

```
DEC2BIN(x)
1  if n = 1
2      return binary[x]
3  else
4          split x into two decimal numbers x_L, x_R with n/2 digits each
5          return ???
```

Here binary$[\cdot]$ is a vector that contains the binary representation of all one-digit integers. That is, binary$[0] = 0_2$, binary$[1] = 1_2$, up to binary$[9] = 1001_2$. Assume that a lookup in binary takes $O(1)$ time. Fill in the missing details. Once again, give a recurrence for the running time of the algorithm, and solve it.

The missing details are **return** FASTMULTIPLY(PWR2BIN$(n/2)$, DEC2BIN(x_l)) + DEC2BIN(x_R). The recurrence is $T(n) = 2T(n/2) + n^a$, as there are two recursive calls with half of the input size and both FASTMULTIPLY and DEC2BIN use $\Theta(n^a)$ time.
Using the Master Theorem, we have $a = 2$, $b = 2$, and $f(n) = \Theta(n^a)$. Considering $n^{\log_b a} = n^{\log_2 2} = n$, we see that $f(n) = \Omega(n^{\log_b a + \epsilon})$ so we have Case 3. We check the regularity condition $af(n/b) \leq cf(n)$. This is $2(n/2)^a \leq cn^a$. As $2(n/2)^a = (2/2^a)n^a$, this holds for $c = (2/2^a) < 1$. Therefore, by the Master Theorem, the solution to the recurrence is $\Theta(n^a)$.

2.29 (4 pts) Suppose we want to evaluate the polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$ at point x.

(a) Show that the following simple routine, known as Horner's rule, does the job and leaves the answer in z.

```
1  z = a_n
2  for i = n − 1 downto 0
3        z = zx + a_i
```

Note that for $n = 0$ the loop is not executed and the alg. correctly returns a_0. Now consider the alg. is run for $n + 1$: In the first n iterations of the loop the algorithm computes: $z = a_1 + a_2x + \ldots + a_nx^{n-1}$. On the last iteration, the algorithm computes $a_0 + zx = a_0 + a_1x + \ldots + a_nx^n$. Which is the required computation. Hence, Horner's rule begins with $z = a_n$. It then multiplies z by x and adds a_{n-1}. It thus implements the factorization $p(x) = a_0 + x(a_1 + x(a_2 + x(\ldots)))$ from the inside out and is correct.

(b) How many additions and multiplications does this routine use, as a function of n? Can you find a polynomial for which an alternative method is substantially better?

This routine uses n additions and n multiplications as it does one addition and multiplication for each step and there are n steps. A polynomial with a single non-zero coefficient a_ix^i can be evaluated with a logarithmic number of multiplications and additions using the repeated squaring method of evaluating x^i.

2.16 (2 pts) You are given an infinite array $A[\cdot]$ in which the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. You are not given the value of n. Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, in $O(\log n)$ time. (If you are disturbed by the fact that the array A has infinite length, assume instead that it is of length n, but that you don't know this length, and that the implementation of the array data type in your programming language returns the error message ∞ whenever elements $A[i]$ with $i > n$ are accessed.)

Iterate through the powers of 2, $i = 1, 2, 4, \ldots$ until we find the first location $A[i]$ such that $x \leq A[i]$. As $A[i/2] < x$, we know that $n \leq 2x$. We can then do a binary search from $A[0]$ to $A[i]$ to find x if it is in the array. Both steps take $\Theta(\log x) = \Theta(\log n)$ time.

3 (b) (2 pts) Consider an $m \times n$ array A of integers. Each row and colum of the array are in ascending order: $A[i, j] \leq A[i, j + 1]$, $i = 1 \ldots n$, $j = 1 \ldots m - 1$ and $A[i, j] \leq A[i + 1, j]$, $i = 1 \ldots n - 1$ $j = 1 \ldots m$.

Design an algorithm that determines if a given integer x is in A, if so, return it's position otherwise return a -1. Explain your algorithm with words and diagram(s). Show that your alogrithm terminates and find its worst case complexity. *Hint: Start by comparing x with $A[1, m]$*

A simple observation helps in finding the solution: Since each row and column of A is sorted, x is not in a row i if $x > A[i, m]$ and x is not in a column j if $x < A[1, j]$. Compare x with $A[1, m]$. If $x < A[1, m]$ then x can not be in the last column, so we compare x with $A[1, m - 1]$. If $x > A[1, m]$ then x can not be in the first row, so we compare x with $A[1, m]$. We can thus use the following algorithm:

```
SEARCH2DARRAY(A, x)
1  row = 1
2  col = m
3  while row ≤ n and col ≥ 1
4        if x < A[row, col]
5              col = col − 1
6        elseif x > A[row, col]
7              row = row + 1
8        else return (row, col)
9  return −1
```

Each iteration of the while loop either increases the row number by 1 or decreases the column number by 1, so the algorithm terminates in $\Theta(n+m)$ steps, each of which takes $\Theta(1)$ time. Thus, the algorithm terminates in $\Theta(n+m)$ time.

4. (6 pts) (a) Draw the recursion tree to guess a solution and prove your guess for $T(n) = T(n-a) + T(a) + cn$ for constants $a \geq 1$ and $c > 0$. Assume $T(n) = T(1)$ for $n < a$.

`See Fig ex_4_a and ex_4_b included alongside`

The solution for this recurrence is $T(n) = O(n^2)$. Take the worst case of $a = 1$. Then $T(n) = T(n-a) + T(a) + cn = T(n-1) + T(1) + cn$. This is $\Theta()$ of $T(n-1) + cn$, which can be unrolled to $c(n + (n-1) + (n-2) + \ldots + 1) = \Theta(n^2)$.

(b) Draw the recursion tree to guess a solution and prove your guess for $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$ for constants $0 < \alpha < 1$ and $c > 0$.

The solution for this recurrence is $T(n) = O(n \log n)$. We can prove this with substition by claiming that $T(n) \leq dn \log n$ for all n. Assume without loss of generality that $\alpha \geq 0.5$. The base case is $T(1)$ which is clearly $O(n \log n)$. Assume that the claim holds for all $T(i), 1 \leq i < n$. Now, consider

$$\begin{aligned}
T(i+1) &= T(\alpha(i+1)) + T((1-\alpha)(i+1) + c(i+1) \\
&\leq d\alpha(i+1)\log(\alpha(i+1)) + d(1-\alpha)(i+1)\log((1-\alpha)(i+1)) + c(i+1) \\
&\leq d\alpha(i+1)\log(\alpha(i+1)) + d(1-\alpha)(i+1)\log(\alpha(i+1)) + c(i+1) \\
&\leq d(i+1)\log(\alpha(i+1))) + c(i+1) \\
&\leq d(i+1)\log(i+1), \text{ for sufficient } d
\end{aligned}$$

(c) Suppose that the splits at each level of quicksort are in the proportion $1-a$ to a where $0 < a = 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approx. $-\log n / \log a$ and the maximum depth is approx $-\log n / \log(1-a)$.

The minimum depth is attained if we pick children of proportion α, and the maximum depth is obtained if we pick children of proportion $1-\alpha$. After k steps down the tree in the first case, the size of the child sub-array is $n \times \alpha^k$. If this sub-array is a leaf in the recursion tree, then its size is 1 and we get the equation $1 = n \times \alpha^k$. From here,

$$k = \log_\alpha 1/n = -\log n / \log \alpha.$$

The other case is symmetric. After k steps down the tree, the size of the child sub-array is $n \times (1-\alpha)^k$. If this sub-array is a leaf in the recursion tree, then its size is 1 and we get the equation $1 = n \times (1-\alpha)^k$. From here,

$$k = \log_{1-\alpha} 1/n = -\log n / \log(1-\alpha)$$

5. (4 pts) (a) Give an algorithm that sorts an array of size n, only containing the elements 1,2 and 3, in linear time. Give a plausible argument that your algorithm sorts the array in linear time

Since we only have 1, 2 or 3 as array elements:

- Call Partition specifying the pivot element to be the integer 2.
- Now we have: $A[1 \ldots q-1] \leq 2$ and $A[q \ldots n] > 2$; the latter sub-array only contains threes.
- Call `Pivot-Partition` again on the sub-array $A[1 \ldots q-1]$ with pivot element = 1.
- Now we have: $A[1 \ldots q'-1] \leq 1$ and $A[q' \ldots q] > 1$; the latter sub-array only contains twos and the former only contains ones.

Since Partition is a $\Theta(n)$ algorithm, and there are two calls to Partition with only constant overhead ($\Theta(1)$) for assigning the pivot, the algorithm described above is in $\Theta(n)$.

(b) Construct an example input for which quicksort will use $\Omega(n^2)$ comparisions when the pivot is chosen as the median of the first, last and middle elements of the sequence.

For the median-of-three pivot based Partition to cause quicksort to take $\Omega(n^2)$ time, the partition must be lop-sided on EVERY call. This will happen if two of the three elements examined (as potential pivots) are the two smallest or the two largest elements in the file. This must happen consistently through all the partitions. For example, consider an EIGHT element array:
(1, (the rest of the numbers must be larger than two), 2)
pivot = Median(1,2, something larger than 2) = 2
move 2 to end as pivot.
(1, (the rest of the numbers must be larger than two), 2)
PARTITION will swap $A[8]$ with $A[2]$ (the element next to 1). Since all other elements are larger than 2, no more swaps will occur and the partition will move to after 2 (now in $A[2]$). Quicksort will recursively sort from the third index to the last one. To continue the pattern of bad splits, the third smallest element must be in $A[3]$. and since the element originally in $A[8]$ was originally in $A[2]$, the element originally in $A[2]$ must be the fourth smallest. Continuing this pattern we see that the sixth smallest must be in $A[4]$ and fifth smallest in $A[5]$ So, the original array must be:

[1, 4, 3, 6, 5, 8, 7, 2]

Running quicksort:
pivot = 2, so array becomes ( )[denotes partition loc.):

[1, 2,][3, 6, 5, 8, 7, 4]

pivot = 4, so array becomes ( )[denotes partition loc.):

[1, 2, 3, 4][5, 8, 7, 6]

pivot = 6, so array becomes ( )[denotes partition loc.):

[1, 2, 3, 4, 5, 6][7, 8]

etc. etc.