

CS 141 Homework Assignment #1

Set on Jan. 14, 2008

Due on Jan. 28, 2008

**Problem 1.** [15 pts] Analyze the worst-case time complexity of the following algorithms, and give tight bounds using the Theta notation. You need show the key steps involved in the analyses.

a)

	cost	times
procedure matrixvector( $n$ : integer)		
var $i, j$ : integer;		
begin		
for $i \leftarrow 1$ to $n$ do begin	$c_1$	$n$
$B[i] \leftarrow 0$ ;	1	$n$
$C[i] \leftarrow 0$ ;	1	$n$
for $j \leftarrow 1$ to $i$ do	$c_2$	$\sum_{i=1}^n i$
$B[i] \leftarrow B[i] + A[i, j]$ ;	$c_3$	$\sum_{i=1}^n i$
for $j \leftarrow n$ downto $i + 1$ do	$c_4$	$\sum_{i=1}^n (n - i)$
$C[i] \leftarrow C[i] + A[i, j]$ ;	$c_5$	$\sum_{i=1}^n (n - i)$
end		
end		

The first *for* loop runs  $n$  times. The second *for* loop runs  $\sum_{i=1}^n i = n(n+1)/2$  times while the third *for* loop runs  $\sum_{i=1}^n (n-i) = n(n-1)/2$  times. The total running time will therefore be:  $c_1n + n + n + (c_2 + c_3) \cdot n(n+1)/2 + (c_4 + c_5) \cdot n(n-1)/2$ . [3 pts]

The whole expression can be re-written as:  $a_1n + a_2n^2$ , which is  $\Theta(n^2)$ . [2 pts]

b)

```

Algorithm MysteryPrint( $A[1 \dots n]$ )
begin
  if  $n = 1$  then
    print  $A[1]$ ;
  else begin
    print  $A[n]$ ;
    call MysteryPrint( $A[1 \dots n - 1]$ );
    call MysteryPrint( $A[2 \dots n]$ );
  end
end;

```

Assume that the time complexity of the above algorithm is  $T(n)$ . The recursion calls itself with parameters of  $n-1$  elements twice (MysteryPrint( $A[1 \dots n-1]$ ) and MysteryPrint( $A[2 \dots n]$ )).

the number of elements in the array gets reduced by 1 every time this function call is made). Then *if* statement and `print A[n]` are both called once in each recursion. We can express this relationship as:

$$\begin{cases} T(n) = 2T(n-1) + c, & \text{if } n \geq 2 \\ T(1) = c \end{cases} \quad [2 \text{ pts}]$$

Solving the recurrence, we can expand it as:

$$\begin{aligned} T(n) &= 2T(n-1) + c \\ &= 2(2T(n-2) + c) + c = 2^2T(n-2) + 2^1c + c \\ &= 2^2(2T(n-3) + c) + 2^1c + c = 2^3T(n-3) + 2^2c + 2^1c + c \\ &= \dots \\ &= 2^{n-1}T(1) + 2^{n-2}c + 2^{n-3}c + \dots + 2^1c + c \\ &= c \cdot \sum_{i=0}^{n-1} 2^i = c \cdot (2^n - 1) \end{aligned} \quad [2 \text{ pts}]$$

Hence, the time complexity of this algorithm is  $\Theta(2^n)$ . [1 pt]

c)

	cost	times
procedure whileloop( <i>n</i> : integer)		
var <i>i</i> , <i>count</i> : integer;		
begin		
<i>count</i> ← 0;	$c_1$	1
<i>i</i> ← 1;	$c_2$	1
while <i>i</i> < <i>n</i> do begin	$c_3$	$\log_3 n$
<i>i</i> ← 3 <i>i</i> ;	$c_4$	$\log_3 n$
<i>count</i> ← <i>count</i> + 1;	$c_5$	$\log_3 n$
end		
end		

The important thing to understand in this algorithm is the fact that the *while* loop will run only  $k$  times where  $3^k < n$ . This gives  $k < \log_3 n$ . Hence, the running time of this algorithm will be:  $c_1 + c_2 + (c_3 + c_4 + c_5) \cdot \log_3 n$ . [3 pts]

Hence, the time complexity is  $\Theta(\log_3 n)$  or  $\Theta(\log n)$  (Base of the log is irrelevant for asymptotic analysis). [2 pts]

**Problem 2.** [10 pts] [Levitin, p. 60; same for the 1st ed] Question 7 (a, c).

a) If  $t(n) \in O(g(n))$ , then  $g(n) \in \Omega(t(n))$  [5 pts]

*Proof:* Since  $t(n) \in O(g(n))$ , there exist some positive constant  $c_1$  and some nonnegative integer  $n_1$  such that:

$$t(n) \leq c_1 g(n) \quad \text{for all } n \geq n_1$$

Then, we have

$$g(n) \geq \frac{1}{c_1} t(n) \quad \text{for all } n \geq n_1$$

Hence,  $g(n) \in \Omega(t(n))$  with constant  $c = 1/c_1$  and  $n_0 = n_1$ . **Q.E.D.**

c)  $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$  [5 pts]

*Proof:* For any  $t(n)$ ,

$$\begin{aligned} & t(n) \in O(g(n)) \cap \Omega(g(n)) \\ \text{iff } & t(n) \in O(g(n)) \text{ and } t(n) \in \Omega(g(n)) \\ \text{iff } & \text{there exist some positive constant } c_1, c_2 \text{ and some nonnegative integer } n_1, n_2, \\ & \text{such that: } t(n) \leq c_1 g(n) \text{ for all } n \geq n_1 \quad \text{and} \quad t(n) \geq c_2 g(n) \text{ for all } n \geq n_2 \\ \text{iff } & c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0, \text{ where } n_0 = \max\{n_1, n_2\} \\ \text{iff } & t(n) \in \Theta(g(n)). \quad \mathbf{Q.E.D.} \end{aligned}$$

**Problem 3.** [10 pts] Insertion sort can be expressed as a recursive procedure as follows. Given  $A[1 \dots n]$ , we recursively sort  $A[1 \dots n - 1]$  and then insert element  $A[n]$  into the sorted array  $A[1 \dots n - 1]$ . Write a recurrence relation for the running time of this recursive version of insertion sort.

Feel free to use the big-O notation and assume the worst case. Can you also solve the recurrence relation? Is your result the same as the one obtained by the summation analysis in the textbook (pp. 160-162)?

*Answer:*

```
procedure InsertionSort( $A[1 \dots n]$ )
  begin
    if  $n = 1$  then
      return  $A[n]$ ;
    else begin
      InsertionSort( $A[1 \dots n - 1]$ );
      Insert( $A[n], A[1 \dots n - 1]$ );
    end
```

end  
end;

The InsertionSort calls itself with parameter of  $n - 1$  once, then Insert function is called to insert  $A[n]$  into the sorted array. Assuming that array data structure is being used, in the worst case, the running time for each Insert is  $\Theta(n)$  because we need to shift elements to make room for  $A[n]$  to be inserted. So we can express the running time with the recurrence relation as follows:

$$\begin{cases} T(n) = T(n - 1) + \Theta(n), & \text{for } n \geq 2 \\ T(1) = c \end{cases} \quad [4 \text{ pts}]$$

Solving the recurrence relation

$$\begin{aligned} T(n) &= T(n - 1) + \Theta(n) \\ &= T(n - 2) + \Theta(n - 1) + \Theta(n) \\ &= \dots \\ &= T(1) + \Theta(2) + \dots + \Theta(n - 1) + \Theta(n) \\ &= \Theta(1) + \Theta(2) + \dots + \Theta(n - 1) + \Theta(n) \\ &= \sum_{i=1}^n \Theta(i) = \Theta\left(\sum_{i=1}^n i\right) \\ &= \Theta(n(n + 1)/2) = \Theta(n^2) \end{aligned} \quad [2 \text{ pts}]$$

So the time complexity is  $\Theta(n^2)$ , the same as the one obtained by summation analysis in the textbook. [4 pts]

**Problem 4.** [15 pts] Give a tight asymptotic bound for  $T(n)$  in each of the following recurrences. Assume that  $T(n)$  is a constant for  $n \leq 2$ . Justify your answers.

- a.  $T(n) = 16T(n/4) + n^2$
- b.  $T(n) = 7T(n/3) + n^2$
- c.  $T(n) = T(n - 1) + n^2$

You may solve these recurrences using either the Master Theorem or the backward substitution method.

*Answer:*

- a.  $a = 16, b = 4, d = 2$ , so  $a = 16 = b^d$ . By applying case 2 of Master Theorem ([Levitin, p. 483]),  $T(n) \in \Theta(n^2 \log n)$  [5 pts]
- b.  $a = 7, b = 3, d = 2$ , so  $a = 7 < 9 = b^d$ . By applying case 1 of Master Theorem,  $T(n) \in \Theta(n^2)$  [5 pts]

c. By using backward substitution method,

$$\begin{aligned}T(n) &= T(n-1) + n^2 \\&= T(n-2) + (n-1)^2 + n^2 \\&= T(n-3) + (n-2)^2 + (n-1)^2 + n^2 \\&= \dots \\&= T(1) + 2^2 + 3^2 + \dots + (n-1)^2 + n^2 \\&= c - 1 + \sum_{i=1}^n i^2 = c - 1 + n(n+1)(2n+1)/6\end{aligned}$$

So  $T(n) \in \Theta(n^3)$ . [5 pts]

**Problem 5.** [15 pts] Describe a  $\Theta(n \log n)$  time algorithm that, given a set  $S$  of  $n$  integers and another integer  $x$ , determines if or not there exist two elements in  $S$  whose sum is exactly  $x$ .

Analyze the running time of your algorithm to make sure that it is really  $\Theta(n \log n)$  (it would be sufficient to show that the running time is  $O(n \log n)$ ).

Before you present the pseudo-code, use a few words to describe the basic idea(s) in your algorithm to help the reader to understand your algorithm.

Hint: Merge/heap sort and binary search.

*Answer:*

Describing your basic idea: [2 pts]

Consider a small example:

$S = \{5, 13, 8, 20, 2, 6, 10, 1\}$

$x = 22$

First, we need to sort  $S$ , and get  $S = \{1, 2, 5, 6, 8, 10, 13, 20\}$

Then, we scan the elements in  $S$  from left to right, and for each element  $S[i]$ , we search for  $x - S[i]$  in  $S[i+1 \dots n]$ . In the example,

$S[1] = 1$ , so we try to search for  $x - S[1] = 21$  in the rest of the list  $S[2 \dots 8]$  (not found)

$S[2] = 2$ , so we try to search for  $x - S[2] = 20$  in the rest of the list  $S[3 \dots 8]$  (found)

Then we are done.

The main thing here is that we need to perform a binary search to avoid a linear scan of the list each time.

Pseudo-code: [8 pts]

```

procedure sumQuery( $S[1 \dots n], x$ )
  var  $i$ : integer;
  begin
    Sort( $S[1 \dots n]$ )           // any  $O(n \log n)$  sorting algorithm
     $i \leftarrow 1$ ;
    while( $i < n$ ) do begin
      BinarySearch( $x - S[i], S[i + 1 \dots n]$ );
      if found, return "yes";
      else  $i \leftarrow i + 1$ 
    end
    return "no";
  end;

```

Running time:      **[5 pts]**

- Sorting takes  $O(n \log n)$
- Each BinarySearch takes  $O(\log n)$
- Perform the BinarySearch  $n - 1$  times, which would make the whole *while* loop  $O(n \log n)$

Combining both Sort and BinarySearch, the total running time for this algorithm will be  $O(n \log n) + O(n \log n) = O(n \log n)$