COMP 7712 - FALL 2010 - ASSIGNMENTS

1. BONUS Assignment 1 - Due September 15, 2010

This is the first bonus assignment. Bonus assignments give you extra "brownies" points. Each brownie point equals to 1 percent of the grading scale out of 100 percents. You are supposed to work on these problems by yourself, without external help from anyone or any source.

There are no partial credits. You either get a problem correctly or not. Therefore, hand in your solutions only if you think your solution is right. And of course, there can be more than one ways to solve any problem. You can use can techniques you know to solve them.

(1) (2 brownies) Let $T(1) = 1, T(n) = T(\frac{n}{4}) + T(\frac{3n}{4}) + 1$. Show/prove that $T(n) = O(n)$.

(2) (3 brownies) This is a programming assignment. In this assignment, you are supposed to identify the *longest growth* in a list of numbers. The input is a list of numbers; for example **5, 2, 1, 4, 8, 7, 8, 5, 3, 9, 10, 6**. The output is a longest growth (there might be more than one), for example **2, 4, 7, 8, 9, 10**.

A "growth" is a series of increasing numbers. This series of numbers must be a sublist of the input list. In the example above, the longest growth (sublist) is underscored to show you where it is in the input list. Of course, the actual input list is not no underscored. Your program will be tested on many inputs.

2. Assignment 1 - Due September 8, 2010

Chapter 1 (exercises): 1, 12, 15, 28.
Additional problems:

- Let $T(1) = 1, T(n) = n^2 + 9T(\frac{n}{3})$. Find the complexity of $T(n)$.
- Let $T(1) = 1, T(n) = n^2 + 5T(\frac{n}{3})$. Find the complexity of $T(n)$.
- Let $T(1) = 1, T(n) = 10 + T(\frac{n}{3})$. Find the complexity of $T(n)$.

3. How to turn in your assignment

- Hard copies are due at the beginning of the lecture on the due date.
- For programming assignments, you will need
  (1) Send a soft copy by email to me (vphan@memphis.edu). Please put **COMP 7712: hw 2** (or whichever hw it's supposed to be) IN THE SUBJECT of your email.
  (2) Include a README file to tell me how to run your program
  (3) Provide several test cases for each task to show that your program works correctly.

COMP 7712 FALL 2010

Bonus Assignment 1

Date Submitted

September 15, 2010

Submitted by 0/2

Md. Mishfaq Ahmed

U00394646

Graduate Student

Department of Computer Science

University of Memphis

$T(1) = 1$

$T(n) = T(n/4) + T(3n/4) + 1$

Answer

$$T(n) = T(n/4) + T(3n/4) + 1 \quad \ldots\ldots \text{(i)}$$

$$T(n/4) = T(n/16) + T(3n/16) + 1 \quad \ldots \text{(ii)}$$

$$T(3n/4) = T(3n/16) + T(9n/16) + 1 \quad \ldots \text{(iii)}$$

$\therefore$ $T(n)$ can be expressed as follows:

$$T(n) \leq 2T(3n/4) + 1$$

true, but doesn't lead to the expected solution

$$\leq 4T(9n/16) + 2 + 1 \quad [\text{observing equation (iii)}]$$

~~$\leq 2^3 T(3^3 n/16 \times 4$~~

$$\therefore T(n) \leq 2^k T\left(\left(\frac{3}{4}\right)^k n\right) + k \quad \ldots\ldots \text{(iv)}$$

~~To fin~~ Now we choose the value of k in a way such that;

$$\left(\frac{3}{4}\right)^k n = 1$$

$$\text{or,} \quad \left(\frac{3}{4}\right)^k = \frac{1}{n}$$

$$\text{or,} \quad n = \left(\frac{4}{3}\right)^k$$

$$\text{or,} \quad \log_2 n = \log_2 \left(\frac{4}{3}\right)^k$$

$$\text{or,} \quad \log_2 n = k \log_2 4/3$$

$$\therefore \quad k = \frac{\log_2 n}{\log_2 (4/3)}$$

$$= \log_2 (3n/4) = O(\log_2 n)$$

$\therefore$ from equation (iv), using this value of k and the condition that $T(1) = 1$, we observe,

$$T(n) \leq 2^k (T(1)) + k$$

$$= 2^k \cdot 1 + k$$

$$= 2^k + k$$

$$= O(2^{\log_2 n}) + \log_2 n$$

$$= O(n) + \log_2 (n)$$

$= O(n)$. // ~~Observing $\log_2 n$ is $O(n)$~~.

$\therefore$ $T(n)$ is $O(n)$.

Shown

88

COMP 7712 - Fall 2011
Exam 1

Name: Muhammad M Khan

An answer gets a full score only if it is correct, unambiguous and non-redundant.

1. (*10 points*) Is $n^3 \log_2 n - n \in \Omega(n^3)$? Prove your answer using the definition of Ω.

$n^3 \log_2 n - n \in \Omega(n^3)$ if $n^3 \log_2 n - n \geq cn^3$

For $n \geq 4$, $c = 1$ $\quad n^3 \log_2 n - n \geq cn^3$

$\Rightarrow 4^3 \log_2 2^2 - 4 \geq 64$

$\Rightarrow 2 \cdot 64 - 4 \geq 64 \quad \Rightarrow 124 \geq 64$

So, $n^3 \log_2 n - n \in \Omega(n^3)$ is True

10

2. (*20 points*) Determine the running time of algorithms Bar and Foo.

Algorithm 1 Bar(a, b)

```
1: while a < b do
2:     a = a * 2
3: return a * b
```

Step 1, 3 take constant time

Step 2 takes $\log_a b$

So Bar running time is $c + \log_a b$

Algorithm 2 Foo(A) $\rightarrow n$

```
1: for i = 1 to A.length do          ⟶ 1 . . . n
2:     for j = i to A.length do        1 . . . n
3:         sum = sum + i + j + Bar(i, j)   log_a b + c
4: return sum
```

20

For each of the value of i, j runs i times in step 1, 2

So,

$1 \times n = n$

$2 \times n-1 = n-1$

$3 \times n-2 = n-2$

$\vdots$

$n \times 1 = 1$

Total $= \frac{n(n+1)}{2}$

for $\frac{n(n+1)}{2}$ time Step 3 is run Step 3 take $c + \log_a b$ time

Step 4 take constant time

$\therefore Foo(n) = \frac{n(n+1)}{2} \cdot \log_a b + c = \frac{n^2}{2} \log_a b + \frac{n}{2} \log_a b + c = O(n^2 \log_a b)$

3. (*20 points*) Use substitution to find the running time of this algorithm, First, which takes as input an array A and 2 indices L and R.

Algorithm 3 First(A, L, R)

1: **if** $L >= R$ **then**
2: **return** $L + R$
3: $m_1 = L + (R - L) \cdot \frac{1}{3}$
4: $m_2 = L + (R - L) \cdot \frac{2}{3}$
5: **return** $m_1 + m_2 + First(A, L, m_1) + First(A, m_1 + 1, m_2)$

Suppose for n inputs the running time of First algorithm is $T(n)$

steps 1, 2, 3 takes constant time

In step 5, the first algorithm is called recursively for $T(n/3)$, 2 time

So,

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

$= c + 2[c + 2T(n/3^2)] = c + 2c + 2^2 T(n/3^2) = 3c +$

$= c + 2c + 2^2[c + 2T(n/3^3)] = c + 2c + 2^2 c + 2^3 T(n/3^3)$

20

$= c + 2c + 2^2 c + \cdots + 2^{k-1} c + 2^k T(n/3^k)$

$= c[1 + 2 + 2^2 + \cdots + 2^{k-1}] + 2^k T(n/3^k)$

$= c(2^k - 1) + 2^k T(n/3^k)$

$= c(2^{\log_3 n} - 1) + 2^{\log_3 n}$

$= c \cdot 2^{\log_3 n} - 1 + 2^{\log_3 n}$ ✓

$= (c+1) 2^{\log_3 n} - 1$

$= (c+1) n^{\log_3 2} - 1$

$= \Theta(n^{\log_3 2})$

$T(n/3^k) = T(1)$

when $\frac{n}{3^k} = 1$

$\Rightarrow n = 3^k$

$\Rightarrow k = \log_3 n$

4. (*20 points*) Compare the efficiency of the Second algorithm to the First algorithm (in the previous problem). You can use whichever method to determine its running time.

Algorithm 4 Second(A, L, R)

1: **if** $L >= R$ **then**
2: **return** $L + R$
3: $m_1 = L + (R - L) \cdot \frac{1}{3}$
4: $m_2 = L + (R - L) \cdot \frac{2}{3}$
5: **return** $m_1 + m_2 + Second(A, L, m_2)$

Suppose for n inputs Second algorithm takes T(n) running time.

Steps 1, 2, 3, 4 takes constant time

Step 5 calls the second algorithm for $2n/3$ terms and it takes $T(n/3)$ running time.

So, the recurrence relation for algorithm 4 is

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

$= c + T(n/1.5)$

According to Master theorem

$a = 1, \quad b = 1.5, \quad d = 0$

20

& and $\log_{1.5} 1 = 0$

So, $T(n) = \theta(n^0 \log n)$

$= \theta(\log n)$

It takes less time with 2nd algorithm as

$\log n < n^{\log_3 2}$

So, algorithm 2 is efficient

5. (*20 points*) A ternary tree is a tree in which a node can have 1, 2 or 3 children. A full ternary tree is a ternary tree in which internal nodes have exactly 3 children and leaf nodes have no children. Children of every node v can be accessed by *v.left*, *v.middle* and *v.right*. If a node has no left child, for example, then *v.left* is NIL.

Write an algorithm that takes as input the root node of a ternary tree and returns True if the tree is full and False if it is not. You must also determine the running time of your algorithm.

TerTree is the recursive algorithm ~~root~~ called for root, root is changed in each call

```
TerTree(root)
1. if (root.left == NIL) or (root.middle == NIL) or (root.right == NIL)
       return False.
2. else if (root.left == True)
       TerTree(root.left)
3. else if (root.right == True)
       TerTree(root.right)
4. else if (root.middle == True)
       TerTree(root.middle)
5. if (root.left == root.right == root.middle = NIL)
6.     return True.
```

missing return

??

X

5

$T(n) = c + 3T(n)$

$a = 3, \quad b = 1, \quad d = 0$

$\log_1 3 \neq 0.$

6. (*15 points*) Given a sorted array of n distinct integers, $A[1...n]$, write a divide-and-conquer algorithm that returns True if there is an index i such that $A[i] = i$, and False if there is no such index. For example, if $A = [-10, 0, 3, 20, 40]$, then the algorithm returns True (because $A[3] = 3$).

Your algorithm must run in $O(\log n)$ time.

L, H are Lowest & Highest index of the array

Search1(L, H)

```
1. if L > H
       return False

2.     mid = ⌊(L+H)/2⌋;

3.  if A[mid] = mid
       return True;

4.  else if A[mid] > mid
       return Search1( L, mid-1) ;

5.  else
       return Search1( mid+1, H) ;
```

Steps 1, 2, 3, takes constant time

either step 4 or 5 is called each take $T(n/2)$ if Search1 takes $T(n)$ time

$T(n) = T(n/2) + c$

$a = 1, \; b = 2, \; d = 0$ $\qquad \log_2 1 = 0$

so, according to Master theorem

$\Theta(n^0 \log n) = \Theta(\log n)$

COMP 7712 - Fall 2010
Exam 1

Name: MD MISHFAQ AHMED

93/100

Show all work concisely and precisely. Provide answers only for what I ask. No more. No less.

1. (*15 points*)

1. $T(n) = n \log n + 5n$
2. $S(n) = 10n^2 + 20n \log n$

Prove that $T(n) \in \Omega(n)$ and $S(n) \in \theta(n^2)$.

proving $T(n) \in \Omega(n)$:

Taking as constant $c = 1$ we observe

$n \log n + 5n > cn$ for all $n > 1$. ✓

$\therefore \quad T(n) \in \Omega(n)$.

15

2. proving $S(n) = 10n^2 + 20n \log n \in \theta(n^2)$

~~Step 1: proving $S(n) \in O(n^2)$:~~

we can prove it by dividing the two ~~etc~~ things we are comparing and taking limit while $n \to \infty$.

$$\lim_{n \to \infty} \frac{S(n)}{n^2} = \lim_{n \to \infty} \left(\frac{10n^2 + 20n \log n}{n^2} \right)$$

$$= \lim_{n \to \infty} \left(\frac{10n^2}{n^2} \right) + \lim_{n \to \infty} \left(\frac{20n \log n}{n^2} \right)$$

(P.T.O.)

$$= 10 + \lim_{n \to \infty} \left(\frac{20 \log n}{n} \right)$$

$$= 10 + 20 \lim_{n \to \infty} \left(\frac{\log n}{n} \right)$$

$$= 10 + 20 \lim_{n \to \infty} \left(\frac{1/n}{1} \right)$$

// differentiating both numerator & denominator; using L'Hospital's Rule.

$$= 10 + 20 \lim_{n \to \infty} \left(\frac{1}{n} \right)$$

$$= 10 + 20 \times 0$$

$$= 10$$

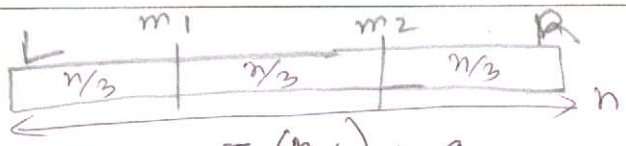
= a constant.

$\therefore S(n) \in \Theta(n^0)$.

2. (*20 points*) Use substitution to determine the complexity of the following algorithm, whose inputs consist of A, an array of size n, L and R are supposed to be indices of A.

Algorithm 1 foo(A, L, R)

1: **if** $L >= R$ **then**
2: **return** $A[R]^3$
3: $m_1 = L + \frac{R-L}{3}; m_2 = L + \frac{2(R-L)}{3}$
4: **return** $foo(A, L, m_1) + foo(A, m_2, R)$.



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

// c = some constant denoting time for finding m_1, m_2 and adding the results of two recursive calls.

$\therefore T(n) = 2T(n/3) + c$

$= 2[2T(n/9) + c] + c$

$= 2^2 T(n/3^2) + 2c + c$

$= 2^2[2T(n/3^3) + c] + 2c + c$

$= 2^3 T(n/3^3) + 2^2 c + 2c + c$

$= 2^k T(n/3^k) + 2^{k-1} c + \ldots + 2c + c$ ---- (i)

now, choosing the value of k such that $\frac{n}{3^k} = 1$ or

or, $n = 3^k$

or, $k = \log_3 n$.

also, $T(1) = 1$.

$\therefore$ from equation (i),

$T(n) = 2^{\log_3 n} T(1) + 2^{\log_3 n - 1} c + \ldots + 2c + c$

[P.T.O.]

$$= 2^{\log_3 n} + C\left[2^{\log_3 n - 1} + 2^{\log_3 n - 2} + \cdots + 1\right]$$

$$= 2^{\log_3 n} + C\,\frac{2^{\log_3 n - 1}\left(2^{\log_3 n - 1} + 1\right)}{2}$$

$$= 2^{\log_3 n} + C\left[\frac{2^{\log_3 n - 1}\left(2^{\log_3 n - 1} + 1\right)}{2}\right]$$

// since $1+2+3+\cdots+K = \frac{K(K+1)}{2}$

$$= 2^{\log_3 n} + C \cdot 2^{\log_3 n}$$

$$= \theta\left(2^{\log_3 n}\right)$$

$$= \theta\left(n^{\log_3 2}\right)$$

Ans. $\theta\left(n^{\log_3 2}\right)$

3. (*20 points*) Use the Master theorem to find a, b, and d and again determine the complexity of the previous problem.

$T(n) = aT(n/b) + f(n)$

here, $a = 2$, $b = 3$

and $d = 0$, since,

$f(n) = c$, a constant.

$\therefore n^{\log_b a} = n^{\log_3 2}$ -- (i)

& $n^d = n^0 = 1$ -- (ii)

comparing (i) & (ii),

$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_3 2})$

20

4. (*20 points*) In this problem you have to design a sorting algorithm using a new procedure called **partition-in-3**, which takes as inputs an array A and two indices (of A) L and R. It returns 2 indices p and q and after the call the resulting array A is partitioned in such a way that

$$\begin{aligned} A[i] &\le A[p] && \text{for } L \le i \le p \\ A[p] < A[i] &\le A[q] && \text{for } p < i \le q \\ A[i] &> A[q] && \text{for } q \le i \le R \end{aligned}$$

Basically, this is how you call this algorithm: **p, q = partition-in-3(A, L, R)**. You can only use this algorithm to design a new sorting algorithm. You can not use any other algorithms.

```
void 3-partite-sort (A[], index L, index R) {
    if (L ≥ R) return;
    p,q = partition-in-3(A,L,R);
    3-partite-sort (A[], L, p-1);
    3-partite-sort (A[], p+1, q-1);
    3-partite-sort (A[], q+1, R);
}
```

20

[P.T.O.]

~~int p, int q~~

index p,q partition-in-3 (A, L, R) {

pelement1 = A[L];

~~pelement2 = A[R];~~

p = L;

~~q = R;~~

~~for (index i = L to~~

~~if (A[p] > A[q])~~

~~swap (A[p], A[q]); // from this point on A[p] ≤ A[q]~~

q = smallest index in the list A[L] ... A[R] of element A[q] such that A[q] > A[p];

pelement2 = A[q]; index j = L;

for (index i = L+1 to q-2)

{ if (A[i] ≥ pelement1)

{

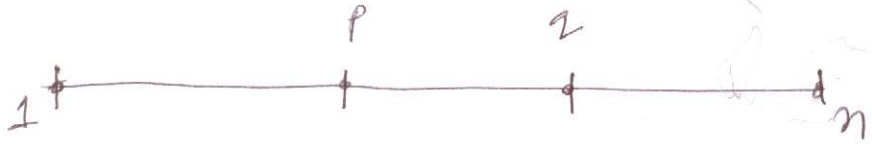
}

else swap { j++;

swap (A[i], A[j])

5. (*10 points*) Determine the running time complexity of the sorting algorithm you designed in the previous problem. You can use either substitution or the master theorem.

$T(n) = 3T(n/3) + Q(\text{partition-in-3})$



p can be any one of the n elements with probability $1/n$.

also q " " " " " " " " " " " $1/n$.

but once p is chosen as the ith smallest element in the sorted array, q has to be among $(i+1)$ st to the nth element in the sorted array. with ~~per~~ probability $1/n-i$

10

$$\therefore T(n) = \sum_{p=1}^{n} \sum_{q=p+1}^{n} \left(\frac{1}{n}\right)\left(\frac{1}{n-p}\right)\{T(p-1)+T($$

$$\therefore T(n) = \sum_{p=1}^{n} \sum_{q=p+1}^{n} \left(\frac{1}{n}\right)\left(\frac{1}{n-p}\right)\{T(p-1)+T(q-p-2) + T(n-q-1)\}$$

. . . (i)

(i) is the expression for average case running time complexity for $T(n)$.

[P.T.O.]

$\therefore T(n) \leq \sum_{1}^{n} 1$

$\therefore T(n) \leq n \sum_{1}^{n} 3T$

$\therefore T(n) \leq 3T(n-2)$

$\leq 3^2 T(n-2^2)$

$\leq 3^3 T(n-2^3)$

$= O(n^{3/2})$.

Ans.

6. (*15 points*) A weighted directed acyclic graph G is a directed graph without any cycle, plus a positive distance $d(i, j)$ for any edge (i, j). You will design an algorithm to find the length of a longest path in G; you can assume that G is connected. You should go through 2 steps.

1. Argue that this problem has an "optimal substructure". In other words, if the longest path from i to j goes through k, then the path from i to k is a longest path (as is the path from k to j.)

   Note: as I mentioned in class, the general longest path problem does not have an optimal substructure. Your argument must explain why this particular version of the longest path has an optimal substructure.

2. To find the length of the longest path, you might use a "**topological sorting**" algorithm. You can call this algorithm like this: **L = topological-sort(G)**.

   Topological sorting returns the list of vertices L of G ordered in the following way: if (i, j) is an edge, then i appears before j in L.

   You should also follow the approach that I discussed in class. For example, start with a clear definition, which is a rephrasing of the original problem in such a way that it allows you do find solutions of a big problem in terms of solutions of smaller problems.

(1) Argument:

This graph G has two properties which allows it to show optimal substructure characteristics.

(i) no cycle: so, the length of the path cannot be lengthened indefinitely by adding the vertices in a cycle again and again to the path.

figure 1

like in figure 1.

(ii)

[P.T.O.]

2. construction

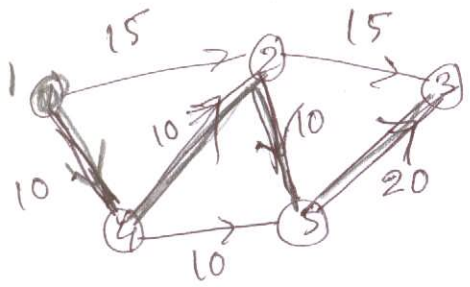


fig 2

(A DAG)

the ~~length of the~~ longest path from vertex 1 and vertex 3 in figure 2 is shown highlighted. It is,

$1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

and length is $D[1,3] = 10+10+10+20$

$= 50$.

Since, V_2 is an intermediate vertex along this path, by the principle of optimal substructure, the path $1 \rightarrow 4 \rightarrow 2$ should be the longest between 1 and 2. As seen in the figure, this is indeed the case.

Defining the problem in terms of subproblems:

Now, to find the longest path between v_i and v_j

let, $P[i,j]^x$ = list of vertices in the longest path in $P[i][j]$ allow first x vertices to be in the path.

Now for some vertex V_K: $(i \leq k \leq n)$ [n = total # of vertices)

case1 if V_K is in $P[i,j]^n$:

$P[i,j]^k = P[i,k]^{k-1} + P[k,j]^{k-1}$ // using principle of optimum subpro

case2 if $V[k]$ not in $P[i,j]^n$:

$$P[i,j]^k = P[i,j]^{k-1}$$

because the list do not change.

// $P[i,j]^n$ is the final optimal path considering all n vertices

~~list topologicalSort (G)~~

~~{~~

~~// $v_1, v_2 \ldots v_n$: set of vertices in G is V.~~

```
max_path (G){
// v1, v2, vn : set of vertices in G is V.
// initially all path length = 0;
// all path list = e;
L^0_ij = e; d^0 = 0;
for (index k=1 to n){
    for (index i=1 to n){
        for (index j=1 to n){
            if (d[i][j]^{k-1} < d[i][k]^{k-1} + d[k][j]^{k-1})
            { d[i][j]^k = d[i][k]^{k-1} + d[k][j]^{k-1};
              L^k = k;
            }
            else { d[i][j]^k = d[i][j]^{k-1};
                   L^k_ij = e  // no vertex added at this step.
                 }
        }
    }
}
```

COMP 7712 - FALL 2011 - ASSIGNMENT 1 - DUE 09/07/2011

1. List reversal

Algorithm 1 *Reverse*(L)

```
1: if L is empty then
2:     return L
3: R = Reverse(L.tail())
4: R.append(L[0])
5: return R
```

1.1. **Correctness.** Define $R(i)$ to mean "Reverse(L), where L is a list with i elements".

(1) $R(0)$ is correct because when the list is empty, the algorithm returns an empty list.

(2) Assume that $R(k)$ is correct, i.e. assume that the algorithm correctly reverses lists with k elements. Suppose that we call $Reverse(L)$ where $L = l_1, \cdots, l_{k+1}$. In step 3, $L.tail() = l_2, \cdots, l_{k+1}$. Based on the assumption, $Reverse(L.tail())$ works correctly, and therefore, $R = l_{k+1}, \cdots, l_2$. When we append the first element of L to R, $R = l_{k+1}, \cdots, l_1$. And this is the correct result.

Thus, "If $R(k)$ is correct, then so is $R(k+1)$". Therefore, $R(i)$ is correct for all i.

1.2. **Efficiency.** Suppose that indexing, tail, and append all take $O(1)$ time to execute. Let $T(n)$ be the running time of *Reverse* on a list with n elements. Then,

$$T(n) = c + T(n-1)$$

for some constant c. This means $T(n-1) = c + T(n-2)$, thus $T(n) = 2c + T(n-2)$. Similarly, $T(n-2) = c + T(n-3)$ and thus $T(n) = 3c + T(n-3)$. Continuing this way, we have $T(n) = nc + T(0)$, where $T(0)$ is the time to reverse an empty list. This takes at most c steps (line 1-2 in the algorithm).

Therefore, $T(n) = (c+1) \cdot n = \Theta(n)$.

1.3. **Notes.** Writing a proof involves many equations. Naturally, proofs are difficult to read. It is best if you *write proofs in complete sentences.* If you do so, when people read your proofs, they read them in complete sentences.

A well-written proof shows clarity of thought.

Proof is a formal word for algorithmic reasoning. When you design or analyze algorithms, you will find yourselve constructing proofs of correctness, or proofs of efficiency, either explicitly on paper or implicitly in your head.

When you are learning, explicit is better than implicit.

I hope you study this carefully and use these ideas and conventions in your future assignments/exams.

125

COMP 7712 - Fall 2011
Exam 2

Name: Muhammad Mahmudur Rahman Khan

An answer gets a full score only if it is correct, unambiguous and non-redundant.

1. (*20 points*)

Algorithm 1 Foo(A)

```
1: sum = 0
2: for i = 1 to n do
3:     for j = i to n do
4:         sum += max(A_i, A_{i+1}, ..., A_j)
5: return sum
```

Determine the worst-case running time of algorithm *Foo*, which takes as input an array A with n elements. Explain very **briefly** (1-2 sentences) how you get your answer.

Step 1, 5 takes constant time $O(1)$

In case of Steps 2,3

for $i = 1$, j runs n times

$i = 2$ j " $n-2$ "

.

.

.

$i = n$ " " 1 time

So, Steps 2,3 runs in total = $n + n-2 + n-3 + \dots 1$

$= \frac{n(n-1)}{2} = O(n^2)$

So, in worst case steps 2,3 runs $O(n^2)$

~~Steps~~ In step 4, $\max(A_i \dots A_j)$ has worst case running time of $O(n)$

sum operations take constant time. So, worst case running time of Step 4 is $O(n)$

Step4 is inside i,j loop so its runs n^2 time in worst cas

So, Worst case running time of FOO(A) is

$F(n) = n^2 \cdot n + O(1) = n^3 + O(1) = O(n^3)$

20

COMP 7712 - FALL 2011 - ASSIGNMENT 1 - DUE 09/07/2011

1. List reversal

Given a list with the following methods:

- Indexing. For example, $L[0]$ is the first item, $L[1]$ is the second item of the list.
- Tail. $L.tail()$ returns the tail of L. The tail of a list is the same list but without the first element. For example, tail of $[1, 2, 3, 4]$ is $[2, 3, 4]$.
- Append. $L.append(x)$ places x at the end of the list. For example, if $L = [1, 2, 3]$, then after executing $L.append(5)$, L is [1,2,3,5].

Algorithm 1 *Reverse*(L)

```
1: if L is empty then
2:     return L
3: R = Reverse(L.tail())
4: R.append(L[0])
5: return R
```

Use mathematical induction to prove that the algorithm Reverse correctly reverses a list. For example, Reverse([1,2,3,4]) returns [4,3,2,1]. Make sure that you use mathematical induction in the same way that we discuss in class.

2. Running Time Analysis

Suppose that indexing, tail, and append all take $O(1)$ time to execute. Compute the running time of Reverse.

3. Turning in your assignment

- Assignment is due in class, at beginning of class on Sept 07, 2011.
- Discuss with your friends, but write your own solutions.
- Write your solution clearly. If I can't read your hand-writing, I can't grade your answers.
- Write your answers concisely and precisely.

125

COMP 7712 - Fall 2011
Exam 2 Name: Muhammad Mahmudur Rahman Khan

An answer gets a full score only if it is correct, unambiguous and non-redundant.

1. (*20 points*)

Algorithm 1 Foo(A)

1: $sum = 0$
2: **for** $i = 1$ to n **do**
3: **for** $j = i$ to n **do**
4: $sum \mathrel{+}= \max(A_i, A_{i+1}, \cdots, A_j)$
5: **return** *sum*

Determine the worst-case running time of algorithm *Foo*, which takes as input an array A with n elements. Explain very **briefly** (1-2 sentences) how you get your answer.

Step 1, 5 taken constant time $O(1)$

In case of Steps 2,3

for $i = 1$, j runs n times
$i = 2$ j " $n-2$ "

20

$i = n$ " " 1 time

So, Steps 2,3 roun in total = $n + n-2 + n-3 + \cdots 1$

$= \frac{n(n-1)}{2} = O(n^2)$

So, in worst case steps 2,3 runs $O(n^2)$

~~Steps~~ In step 4, max $(A_i \cdots A_j)$ has worst case running time of $O(n)$
sum operations take constant time. So, worst case running time of step 4 is $O(n)$

step4 is inside i,j loop so its runs n^2 time in worst c

So, Worst case running time of FOO(A) is

$F(n) = n^2 \cdot n + O(1) = n^3 + O(1) = O(n^3)$

2. (*20 points*)

Algorithm 2 Bar(A, L, R)

```
1: if L >= R then
2:     return 1
3: sum = 0
4: for i = L to R do
5:     for j = i to R do
6:         sum += max(A_i, A_{i+1}, ..., A_j)
7: m = (L+R)/2
8: return sum + Bar(A, L, m-1) + Bar(A, m+1, R)
```

Determine the worst-case running time of algorithm *Bar*, which takes as input an array A with n elements, and L and R are indices of A.

Here
$n = R - L$

Steps 1, 2, 3, 7 take constant time $O(1)$

Steps 4, 5 has worst case running time of $O(n^2)$
and step6 has worst case running time $O(n^3)$ as it is inside two for loops in steps4, 5

20

In step 8 Barz(n) is called for two parts with size $n/2$ recursively

So,
$$\text{Barz}(n) = O(n^3) + 2\,\text{Barz}(n/2) + O(1)$$
$$= n^3 + 2\,\text{Barz}(n/2) = O(n^3)$$

According to master theorem

$$a = 2,\ b = 2,\ d = 3$$

$$\log_2 2 \neq 3$$

So $\theta(n) = \theta(n^{\max\{3, \log_2 2\}})$

$$= \theta(n^3)$$

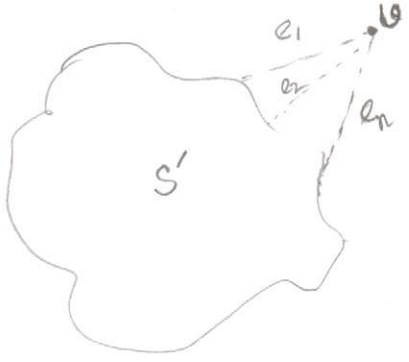
3. (*20 points*) Design an algorithm to find the Maximum Spanning Tree of a weighted graph G. Describe your algorithm in pseudocode very clearly. Use indentation appropriately. It might be better for you to write your algorithm in working paper first, before writing the final solution on the exam paper.

MaxSpanTree(G)

1. ST is initialized to empty or ST = ∅
2. Sort the edge list E with decreasing weight
3. For every edge e ∈ E
4. delete_max(E)
5. ST = ST ∪ e if e does not create a cycle
6. return ST

20

4. (*10 points*) Your algorithm in the previous problem makes a sequence of greedy selection $e_1, e_2, \cdots, e_n$. Explain why the choice e_1 is always correct. You might use an example to illustrate your explanation.



Suppose we have a maximum spanning tree S'. Now we have n options $e_1, e_2, \ldots e_n$ towards a vertex v.

Lets assume $e_1, e_2, \ldots e_n$ are sorted according to decreasing order of weight.

$e_1 > e_2 > e_3 > e_4 \ldots > e_n$

Now, suppose we get a maximum spanning tree adding e_2

So, Maximum spanning tree $S = S' \cup e_2$

So, weight(S) is minimum

We get another spanning tree $S'' = S' \cup e_1$

Weight of (S'') > weight of (S) since weight(S) + e_1 > weight(S') + e_2

if contradicts the definition.

so e_1 will be always the right choice

5

5. (*10 points*) A string is palindromic if it is the same as its reverse; for example, $ATCGCTA$ is palindromic. Design an algorithm that takes as input a string S of length n, and find the *longest palindromic substring* of S. You can use any strategy you want as long as the running time is $O(n^2)$.

Describe your algorithm in pseudocode very clearly. Use indentation appropriately.

```
LPS(S)
1. SUBST ← empty string
2. for i = 1 to length(S)
3.     for j = i to length(S)
4.         if
```

10

6. (*10 points*) In this problem, we are interested in *palindromic subsequences* of a string S. (Recall that a subsequence is different from a substring in that characters in a subsequence might not be consecutive in the string of which it is a subsequence).

As a first step, we can observe that if $S_1 = S_n$, then a longest palindromic subsequence of $S_1 S_2 \cdots S_{n-1} S_n$ must include S_1 and S_n.

Base on this hint, define a formula that can be used to find the **length** of the longest ~~common~~ palindromic subsequences using dynamic programming.

$L_{i,j}$ = the maximum length of the palindromic subsequence in S that starts at i and ends at j

10

7. (*10 points*) Fully specify the formula you obtained in the previous problem. Do not forget about the initial cases.

We can specify the formula as

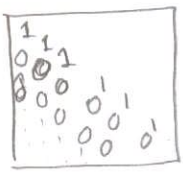
$$L_{i,j} = \begin{cases} 2 + L_{i+1,j-1} & \text{if } S[i] = S[j] \\ \max\{L_{i,j-1}, L_{i+1,j}\} & \text{otherwise} \end{cases}$$

10

We have two initial cases to define when $i == j$ and $i > j$

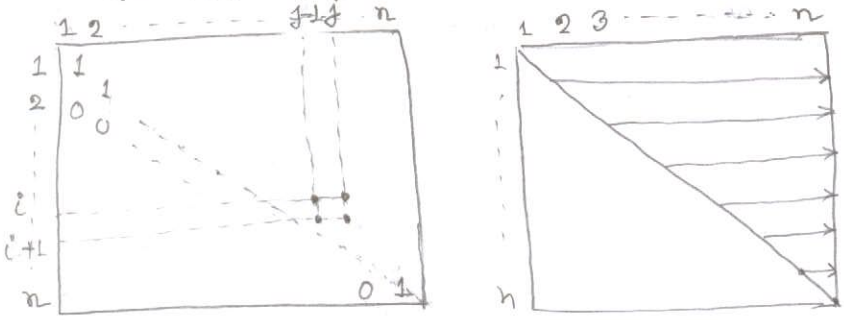
we initialize, $S[i,i] = 1$, so all the diagonal elements are assigned to

Since we are moving i,j index simultaneously for case where $j = i+1$ we can reach $L_{i,j}$ which leads to $L_{i+1,i}$. However it is not valid for our formula. so, we initialise bottom diagonal to zero.



8. (*10 points*) Continuing from the previous problem, representing your formula as an array, explain the "direction" from which to compute values for the array so that all values are available for any computation.

We have to find $L_{1,n}$ finally according to our formula. For $L_{i,j}$ we need three points $L_{i+1,j-1}$, $L_{i+1,j}$ and $L_{i,j-1}$



From bottom and left to right

10

9. (*10 points*) Describe in pseudocode very clearly an algorithm to find the length of the longest ~~common~~ palindromic subsequences of a string S.

```
LPS(S)
1. n = |S|,
2. for i = 1 to n
       L[i,i] = 1;
3. for i = 2 to n
       for j = 1 to i-1
           L[i,j] = 0,
4. for i = n-1 to 1
5.     for j = i+1 to n
6.         if S[i] == S[j]
               L[i][j] = 2 + L[i+1][j-1]
7.         else
               L[i][j] = max{ L[i][j-1], L[i+1][j] }
8. return L[1][n]
```

10

10. (*10 points*) In the previous problem, you only find the length of a longest common subsequence. In this problem, describe an algorithm clearly in pseudocode to find an actual longest ~~common~~ palindromic subsequence subproblem.

```
Print LPS(S, L, i, j)
1. if (i == j)
2.     print S[i]
       return
3. if (i > j) return
4. else if L[i][j] = L[i+1][j-1] + 2
       print S[i]
5.     Print LPS(S, L, i+1, j-1)
6.     print S[j]
7. else if L[i][j] > L[i+1][j]
8.     print LPS(S, L, i, j-1)
9. else print LPS(S, L, i+1, j)
```

10

COMP: 7712

HW # 1

Muhammad M Khan

U00333660

15/10

Algorithm 1: Reverse (L)

```
1. if L is empty then
2.     return L.
3. R = Reverse(L.tail())
4. R.append(L[0])
5. return R
```

Proof by Induction

We have to prove with mathematical induction that Reverse(L) algorithm reverses a list with n elements correctly. Here n = number of elements in the list L.

Basis

~~The algorithm has~~

For $n = 1$, $L = L[0]$

R = Reverse(L.tail()) = Reverse(ϵ) = ϵ

R.append(L[0]) = ϵ, L[0] = L[0]

So, Reverse(L) ~~proves~~ provides the reverse of a list L with 1 element

Induction Hypothesis

Now assume $n=k$ and Reverse (L) algorithm reverses L_k correctly. Here L_k is a list with k elements

We have to prove that if Reverse (L) reverses L_k correctly it will reverse L_{k+1} ~~correct~~ correctly.

Where L_k = List with k elements

L_{k+1} = List with k+1 elements

~~Proof~~

Induction

For L_{k+1}, $n = k+1$ and 1 element is appended to L_k.

Reverse (L_{k+1}) algorithm has following steps

Step 1: R = Reverse (L_{k+1}.tail())

= Reverse (L_k)

Since ~~appen~~ tail() leaves the first element of the list.

Step 2: R.append(L[0]) = Reverse (L_k).append(L[0])

~~From~~

Step1 gives the reverse of the tail of L_{k+1} or reverse of a list with k element leaving the first element L[0]

From basis step we know that Reverse(L[0]) = L[0]

Now, Step 2 gives a list with L[0] appended to the Reverse(L[k+1].tail()). ~~or a Reverse~~ So, we get a reverse for k+1 elements.

So, Reverse(L) reverses a list with k+1 elements correctly

Running Time Analysis

Reverse(n) causes to call

Reverse(n-1)

tail() ⟶ O(1)

append(L[0]) ⟶ O(1)

⟶ O(1)

~~T(0)=O(1)~~ T(0) = O(1) = constant

$T(n) = T(n-1) + O(1) + O(1) + O(1) = T(n-1) + 3O(1)$ (+c)

$= T(n-1) + O(1)$

$= T(n-2) + O(1) + O(1)$

$= T(n-3) + O(1) + O(1) + O(1)$

$= T(1) + (n-1) \cdot O(1)$

$= T(0) + n \cdot O(1)$

$= O(n)$

LATE

COMP 8712: Algorithms and Problem Solving

Assignment 1

100/100 -10

Submitted to,

Dr. Vinhthuy Phan
Associate Professor,
Dept of CS, University of Memphis

Submitted by,

Syed Monowar Hossain
UID: U00397127
Email: smhssain@memphis.edu

Chapter 1, Exercise 1

10/10

Let A[ ] - array containing all numbers starting from A[1]
n - No of elements

Pseudocode:

```
int findlargest( A[ ], n)
{
    max = 1;
    for (i=2; i<=n; i++)
        if (A[max] < A[i])
            max = i;
    return A[max];
}
```

output: max – index of the largest element
A[max] - largest element in the array.

Chapter 1, Exercise 12

15/15

Let n - number of elements in the array
B[n*k] - an array initialized with 0
A[n] - input array (array index starts from 1)
C[n] - output array (array index starts from 1)

Pseudocode:

```
int sortarray( int A[ ], int n)
{
    for (i= 1 to kxn)
        B[i] = 0;
    for (i= 1 to n)
        B[A[i]] = 1;
    j=1;
    for (i= 1 to kn){
        if (B[i] == 1){
            C[j]= i;
            j++;
        }
    }
    return C
}
```

Chapter 1 Exercise 15

15/15

Here $g(n) = n^2 + 3n^3$

$f(n) = n^3$

So, $n^2 + 3n^3 \leq n^3 + 3n^3 \leq 4n^3$ ✓

$\therefore g(n) \leq c.f(n)$ where $c = 4$ for all $N \geq 1$

$\therefore n^2 + 3n^3 \in O(n^3)$

Again $n^2 + 3n^3 \geq 2n^3$ ✓

$\therefore g(n) \geq c.f(n)$ where $c = 2$, for all $N \geq 1$

$\therefore n^2 + 3n^3 \in \Omega(n^3)$

So, $\theta(n) = O(n^3) \cap \Omega(n^3) = \theta(n^3)$

Proved

Chapter 1 Exercise 28

15/15

	Complexity
`for(i=1;i<=n;i++){`	$\theta(n)$
`j=n;`	$\theta(1)$
`while(j>=1){`	$\theta(\log_2 n)$
`<body of the while loop>`	$\theta(1)$
`j=[j/2]`	
`}`	
`}`	

So overall complexity is $T(n) = \theta(n \log_2 n)$

Additional problem 1

15/15

$T(1) = 1$

$T(n) = n^2 + 9T\left(\frac{n}{3}\right)$

Here $a = 9$

$b = 3$

$f(n) = n^2$

$\therefore \log_b a = \log_3 9 = 2$

$\therefore n^{\log_b a} = n^2 = \theta(n^2)$

$\therefore f(n) = \theta(n^{\log_b a})$

$\therefore$ By master method $T(n) = \theta(n^2 \lg_3 n)$

case 2

Additional Problem 2

15/15

$T(1) = 1$

$T(n) = n^2 + 5T\left(\frac{n}{3}\right)$

Here $a = 5$

$b = 3$

$f(n) = n^2$

$\log_b a = \log_3 5 = 1.09$

$\therefore n^{\log_b a} = n^{1.09}$

$\therefore f(n) = n^2 = \Omega(n^{1.09+\epsilon})$ where $\epsilon \approx 0.91$

again $af(n/b) = 5f\left(\frac{n}{3}\right) = 5\frac{n^2}{9} \leq c \cdot n^2$ where $c = \frac{5}{9}$

$\therefore$ By master method case 3 will apply

$\therefore T(n) = \theta(n^2)$

Additional Problem 3

(15/15)

$$T(1) = 1$$

$$T(n) = 10 + T\left(\frac{n}{3}\right)$$

Using substitution method,

$$T(n) = 10 + T\left(\frac{n}{3}\right)$$

$$= 10 + 10 + T\left(\frac{n}{9}\right)$$

$$= 10 + 10 + 10 + T\left(\frac{n}{27}\right)$$

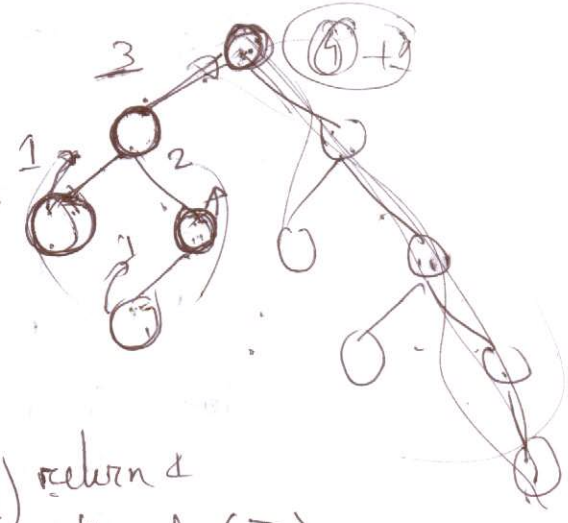
$$= \underbrace{10 + 10 + \ldots + 10}_{\log_3 n \text{ times}} + T(1)$$

$$= 10 \cdot \log_3 n + 1$$

$$= \theta(\log_3 n)$$

$$\therefore T(n) = \theta(\log_3 n)$$

Height(root)




```
Height(T)
{
  if (T.left == null && T.right == null) return 0;
  if (T.left != null)
      l = Height(T.left);
  if (T.right != null)
      r = Height(T.right);
  res = max(l, r) + 1;
  return res;
}
```

```
Height(T)
{
  leaf & → 1
  l = Height(T.l)
  r =
}
```

COMP 7712

ALGORITHMS/PROB SOLV

ASSIGNMENT NO 1

81/100

SUBMITTED BY,

NAME: MD MISHFAQ AHMED

UID: 00394646

GRADUATE STUDENT

DEPT.: COMPUTER SCIENCE

SEMESTER: FALL, 2010

DATE: SEPTEMBER 08, 2010

COMP 7712

hw 1

Submitted by
Md. Mishfaq Ahmed
UID: U00394646
Graduate Student
Department of Computer Science
University of Memphis

Exercises 1

Write an algorithm that finds the largest number in a list (an array) of n numbers.

Answer

Input: A[1], A[2], ... A[n] a list of n numbers

```
int function findlargest (A[]) {
        max = A[1]; maxpos = 1;
        for (int i = 1 to n)
            if (A[i] > max)
                {
                max = A[i];
                maxpos = i;
                }
        return max, maxpos;
        }
```

here, max is the value of the largest number in the array.

maxpos is the position of the largest number in the list.

(10/10)

Exercise 12

Write a $\Theta(n)$ algorithm that sorts n distinct integers, ranging in size between 1 and kn inclusive, where k is a constant positive integer.

Answer

Input: A[1], A[2], ... A[n]; a list of n ~~numb~~ distinct integers, ranging in size between 1 and kn inclusive, where k is a constant positive integer.

```
function linear_complexity_sort (A[]) {
    Create an array C[1], C[2], ..... C[nk];
    Initialize C[1], C[2], C[3], ... C[nk] to zero;
    for (int i=1 to n)
        C[A[i]] = C[A[i]] + 1
```

[P.T.O.]

~~Create output array B[], B[1], B[2], ... B[nk];~~

```
    Create output array B[1], B[2], ... B[n];
    index = 1;
    for (int i=1 to kn) {
        if (C[i] ≠ 0) {
            B[index] = i;
            index = index + 1;
            C[i] = C[i] - 1;
        }
    }
```

// j = 2 to kn

for (j = 2 to ~~kn~~)

C[j] = C[j] + C[j-1]; // cumulative counting

create output array B[1], B[2], ... B[~~kn~~]

~~for (int k~~

~~for (int x = kn to 1)~~

~~{~~

~~B[C[A[x]]] =~~

for (int x = n to 1)

{

B[C[A[x]]] = A[x]; // construct the output array in reverse order

C[A[x]] = C[A[x]] - 1;

}

10/15

}

Exercise

15. Show that, $f(n) = n^2 + 3n^3 \in \Theta(n^3)$; That is, use the definitions of O and Ω to show that $f(n)$ is both $O(n^3)$ and $\Omega(n^3)$.

Ans.

To show that $n^2 + 3n^3 \in \Omega(n^3)$, we must show that there is a positive c and an integer N such that

$$n^2 + 3n^3 \geq c \cdot n^3 \quad \text{for} \quad n \geq N.$$

Since, $n^2 + 3n^3 \geq 3 \cdot n^3$ ✓ for real $n \geq 1$, setting $c = 3$ and $n = 1$ works.

$\therefore \quad n^2 + 3n^3 \in \Omega(n^3)$.

To show that $n^2 + 3n^3 \in O(n^3)$, we must show that there is a positive c and an N such that for $n \geq N$, we have

$$n^2 + 3n^3 \leq c \cdot n^3.$$

Since, $n^2 + 3n^3 \leq n^3 + 3n^3 \quad (\text{for } n \geq 1)$

$$= 4n^3, \quad ✓$$

setting $c = 4$ and $N = 1$ works.

$\therefore \quad n^2 + 3n^3 \in O(n^3)$

$\therefore \quad n^2 + 3n^3 \in O(n^3) \cap \Omega(n^3) = \Theta(n^3)$

15/15

[Shown]

Exercise
28

```
:
for (i = 1; i <= n; i++) {
    j = n;
    while (j >= 1) {
        <body of the while loop>   // Needs Θ(1)
        j = ⌊j/2⌋;
    }
}
:
```

$n = 2^k$ for some positive integer k.

So the inner while loop runs with the value of j being

$2^k, 2^{k-1}, 2^{k-2}, \ldots, 2^{k-k}$

$\therefore$ inner while loop runs $k+1$ times.

Outer for loop runs n times.

$\therefore$ time complexity, $T(n) = \Theta(n \times (k+1))$

$= \Theta(n(\log_2 n + 1))$

$= \Theta(n \log_2 n)$.

15/15

Additional Problems

① $T(1) = 1$, $T(n) = n^2 + 9T(n/3)$. Find the complexity of $T(n)$.

Here we will using the Master Method. here,

$a = 9$, $b = 3$, $f(n) = n^2$, $n^{\log_b a} = n^{\log_3 9} = n^2$

show work!

$\therefore f(n) = n^{\log_b a}$

$\therefore T(n) = \Theta(n^2 \log_2 n)$ ✗ (8/15)

Ans.

② $T(1) = 1$, $T(n) = n^2 + 5T(n/3)$. Find the complexity of $T(n)$.

Here, $a = 5$, $b = 3$, $f(n) = n^2$, $n^{\log_b a} = n^{\log_3 5} = n^{1+\epsilon}$

for some positive ϵ. $(0 < \epsilon < 1)$

comparing $f(n) = n^2$ with $n^{1+\epsilon}$

$n^2 = \Omega(n^{1+\epsilon})$.

Since $f(n)$ is a polynomial in n,

$T(n) = \Theta(n^2)$. (15/15)

Ans

(3) $T(1)=1$, $T(n)=10+T(n/3)$, Find complexity of $T(n)$

Here, using master method,

$a=1$, $b=3$, $f(n)=10$, $n^{\log_b a} = n^{\log 3^1} = n^0 = 1$

compare $f(n)=10$ with 1.

$10 = \theta(n^0)$.

$\therefore f(n) = \theta(n^{\log_b a})$.

$\therefore T(n) = \theta(n^{\log_b a} \log_2 n) = \theta(1 \cdot \log_2 n) = \theta(\lg n)$

$\therefore T(n) = \theta(\lg n)$. (8/15)

Ans

COMP 7712 - FALL 2011 - ASSIGNMENT 2 - DUE 09/19/2011

(1) Use substitution to find the complexity of these recurrences. Then, verify your answer by applying the Master's theorem (make sure you list what a, b and d are for each equation).
   (a) $T(n) = n^3 + 7T(\frac{n}{2})$
   (b) $T(n) = n^2 + 7T(\frac{n}{2})$

(2) Algorithm Foo's inputs include an array A and indices L and R of A. Use substitution to find the complexity of Foo and verify it using the Master's theorem. You must explain how you got the equation for the running time ($T(n)$) of Foo; in particular, explain the number of steps in each line of the algorithm.

Algorithm 1 Foo(A, L, R)

```
1: if L > R then
2:    return L
3: p = (R−L)/4
4: a = Foo(A, L, L + p)
5: b = Foo(A, L + p + 1, L + 2p)
6: c = Foo(A, L + 2p + 1, L + 3p)
7: d = Foo(A, L + 3p + 1, L + 4p)
8: sum = 0
9: for i = L to R do
10:    sum = sum + i · (a + b + c − d)
11: return sum
```

Assignment #2

COMP 7712

10

By

Muhammad M. Khan

U00333660

1. (a)

$$T(n) = n^3 + 7T(n/2)$$

$$= n^3 + 7\left[(n/2)^3 + 7T(n/2^2)\right]$$

$$= n^3 + 7 \cdot \frac{n^3}{2^3} + 7^2 T(n/2^2)$$

$$= n^3 + 7 \cdot \frac{n^3}{2^3} + 7^2\left[(n/2^2)^3 + 7T(n/2^3)\right]$$

$$= n^3 + 7 \cdot \frac{n^3}{2^3} + 7^2 \frac{n}{(2^2)^3} + 7^3 \cdot T(n/2^3)$$

$$= n^3 + 7 \cdot \frac{n^3}{2^3} + 7^2 \frac{n^3}{(2^2)^3} + 7^3\left[\left(\frac{n}{2^3}\right)^3 + 7T\left(\frac{n}{2^4}\right)\right]$$

$$= n^3 + 7 \cdot \frac{n^3}{2^3} + 7^2 \cdot \frac{n^3}{(2^3)^3} + 7^3 \cdot \frac{n^3}{(2^3)^3} + 7^4 \cdot T\left(\frac{n}{2^4}\right)$$

$$\vdots$$

$$= n^3 + 7 \cdot \frac{n^3}{2^3} + 7^2 \cdot \frac{n^3}{(2^3)^3} + 7^3 \cdot \frac{n^3}{(2^3)^3} + \cdots + 7^{k-1} \cdot \frac{n^3}{(2^{k-1})^3} + 7^k T\left(\frac{n}{2^k}\right)$$

$$= n^3\left[1 + \frac{7}{2^3} + \left(\frac{7}{2^3}\right)^2 + \left(\frac{7}{2^3}\right)^3 + \cdots + \left(\frac{7}{2^3}\right)^{k-1}\right] + 7^k T\left(\frac{n}{2^k}\right)$$

Lets assume $2^k = n$ then $k = \log_2 n$

Now,

$$T(n) = n^3\left[1 + \frac{7}{2^3} + \left(\frac{7}{2^3}\right)^2 + \left(\frac{7}{2^3}\right)^3 + \cdots + \left(\frac{7}{2^3}\right)^{\log_2 n - 1}\right] + 7^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right)$$

$$= n^3\left[\frac{1 - \left(\frac{7}{2^3}\right)^{\log_2 n}}{1 - 7/2^3}\right] + 7^{\log_2 n} T(1)$$

$$T(n) = n^3 \left[\frac{1 - (7/8)^{\log_2 n}}{1 - 7/8} \right] + 7^{\log_2 n} \qquad [\text{as } T(1) = 1]$$

$$= 8n^3 \left[1 - \frac{7^{\log_2 n}}{(2^3)^{\log_2 n}} \right] + 7^{\log_2 n}$$

$$= 8n^3 - 8n^3 \cdot \frac{7^{\log_2 n}}{n^3} + 7^{\log_2 n}$$

$$= 8n^3 - 8 \cdot 7^{\log_2 n} + 7^{\log_2 n}$$

$$= \cancel{8n^3 - 7 \log_2 n}$$

$$= 8n^3 - 7^{\log_2 n}$$

So, $T(n) = 8n^3 - 7^{\log_2 n}$

$T(n) \in O(n^3)$ since $8n^3 - 7^{\log_2 n} \leq 8n^3$ where $c = 8$ & $n \geq 1$

and

$T(n) \in \Omega(n^3)$ since $8n^3 - 7^{\log_2 n} \geq 7n^3$ where $c = 7$ & $n \geq 1$

So, $T(n) = \Theta(n^3)$ ✓

According to Masters theorem

$$d=3 \;,\quad a=7 \;,\quad b=2$$

$$\log_2 7 \neq 3$$

$$\text{So,} \quad T(n) = \theta\left(n^{\max\{3, \log_2 7\}} \right)$$

$$= \theta(n^3)$$

1(b)

$$T(n) = n^2 + 7T(n/2)$$

$$= n^2 + 7\left[(n/2)^2 + 7T(n/2^2)\right]$$

$$= n^2 + 7 \cdot \frac{n^2}{2^2} + 7^2 \cdot T(n/2^2)$$

$$= n^2 + 7 \cdot \frac{n^2}{2^2} + 7^2\left[(n/2^2)^2 + 7^2 T(n/2^3)\right]$$

$$= n^2 + 7 \cdot \frac{n^2}{2^2} + 7^2 \cdot \frac{n^2}{(2^2)^2} + 7^3 T(n/2^3)$$

$$= n^2 + \frac{7}{4} \cdot n^2 + \left(\frac{7}{4}\right)^2 n^2 + 7^3\left[\left(\frac{n}{2^3}\right)^2 + 7T(n/2^4)\right]$$

$$= n^2 + \frac{7}{4} \cdot n^2 + \left(\frac{7}{4}\right)^2 n^2 + \left(\frac{7}{4}\right)^3 n^2 + 7^4 T(n/2^4)$$

$$\vdots$$

$$= n^2 + \frac{7}{4} \cdot n^2 + \left(\frac{7}{4}\right)^2 n^2 + \left(\frac{7}{4}\right)^3 n^2 + \cdots + \left(\frac{7}{4}\right)^{K-1} n^2 + 7^K T\left(\frac{n}{2^K}\right)$$

Lets assume $2^K = n$ then $K = \log_2 n$

Now,

$$T(n) = n^2 + \frac{7}{4} n^2 + \left(\frac{7}{4}\right)^2 n^2 + \left(\frac{7}{4}\right)^3 n^2 + \cdots + \left(\frac{7}{4}\right)^{\log_2 n - 1} n^2 + 7^{\log_2 n} T(1)$$

$$= n^2\left[1 + \frac{7}{4} + \left(\frac{7}{4}\right)^2 + \left(\frac{7}{4}\right)^3 + \cdots + \left(\frac{7}{4}\right)^{\log_2 n - 1}\right] + 7^{\log_2 n}$$

Since $T(1) = 1$

$$T(n) = n^2\left[\frac{(7/4)^{\log_2 n} - 1}{7/4 - 1}\right] + 7^{\log_2 n}$$

$$= \frac{4}{3}n^2\left[\frac{7^{\log_2 n}}{(2^2)^{\log_2 n}} - 1\right] + 7^{\log_2 n}$$

$$= \frac{4}{3}n^2\left[\frac{7^{\log_2 n}}{n^2} - 1\right] + 7^{\log_2 n}$$

$$= \frac{4}{3}7^{\log_2 n} + 7^{\log_2 n} - \frac{4}{3}n^2$$

$$= \frac{7}{3}\cdot 7^{\log_2 n} - \frac{4}{3}n^2$$

$$\therefore T(n) = \frac{7}{3}\cdot 7^{\log_2 n} - \frac{4}{3}n^2$$

$$= \frac{7}{3}n^{\log_2 7} - \frac{4}{3}n^2$$

If, $7^{\log_2 n} = n^x$

$\Rightarrow \log_2 7^{\log_2 n} = \log_2 n^x$

$\Rightarrow \log_2 7 \cdot \log_2 n = x \log_2 n$

$\Rightarrow x = \log_2 7$

$T(n) \in O(n^{\log_2 7})$

since

$$7/3\, n^{\log_2 7} - \frac{4}{3} n^2 \leq c n^{\log_2 7}$$

for $c = \frac{7}{3}$ and $n \geq 0$

$T(n) \in \Omega(n^{\log_2 7})$

since $7/3\, n^{\log_2 7} - 4/3\, n^2 \leq c n^{\log_2 7}$

for $c = 1$ and $n \geq 0$

$\therefore T(n) = \Theta(n^{\log_2 7})$ ✓

According to Masters theorem

$d = 2, \quad a = 7, \quad b = 2$

$\log_2 7 \neq 2$

So, $T(n) = \Theta\left(n^{\max\{2, \log_2 7\}}\right)$

$= \Theta(n^{\log_2 7})$ ✓

2.

Lets assume $T(n)$ is the running time of Algorithm 1 foo (A, L, R) for input size n

Step 1, 2, 8 take constant time $\#(1)$

Step 4,5,6,7 each take $T(n/4)$ time $\#(\frac{n}{4})$

Step 9,10 take n time $\#(n)$

If we ignore the constant time,

$$T(n) = \#(n) + 4T(n/4) \quad \checkmark$$

$$= n + 4\left[n/4 + 4T(n/4^2)\right]$$

$$= n + n + 4^2 T(n/4^2)$$

$$= 2n + 4^2\left[n/4^2 + T(n/4^3)\right]$$

$$= 3n + 4^3 T(n/4^3)$$

$$\vdots$$

$$= kn + 4^k T(n/4^k)$$

lets assume $n = 4^k$

$$\Rightarrow k = \log_4 n$$

$$\therefore T(n) = n\log_4 n + 4^{\log_4 n} T(1)$$

$$= n\log_4 n + n$$

(—)

$T(n) \in O(n \log_4 n)$

since $n \log_4 n + n \geq c \cdot n \log_4 n$

for $c = 1$, and $n \geq 0$

$T(n) \in \Omega(n \log_4 n)$

since $n \log_4 n + n \leq cn \log_4 n$

for $c = 0$ and $n \geq 0$

$\therefore$ ~~$T(n) = \Theta(n^{\log_4 n}$~~

$= \Theta(n$

$\therefore T(n) = \Theta(n \log_4 n) = \Theta(n \log n)$ ✓

According to Masters theorem

$d = 1, \quad a = 4, \quad b = 4$

since $\log_4 4 = 1$

$T(n) = \Theta(n \log n)$ ✓

95/100

COMP 7712 - Fall 2010
Exam 2

Name: MD. MISHFAQ AHMED

Show all work concisely and precisely. Provide answers only for what I ask. No more. No less.

1. (*20 points*) Determine the running time of the following algorithm *mystery*, which takes as inputs an array A and two indices of A, L and R. This algorithm, *mystery*, calls another algorithm *foo* which runs in $\theta(n)$ time. You can use the master's theorem or substitution (but, preferably not both).

Algorithm 1 $mystery(A, L, R)$

```
1: if L >= R then
2:     return 1
3: m = (L+R)/2
4: a = foo(A, L, m)
5: b = foo(A, m, R)
6: return a + b + mystery(A, L, m − 1) + mystery(A, m + 1, R)
```

20

$T(n) = 2T(n/2) + 2\theta(n/2)$ ... (i)

~~or, $T(n) = 2T(n/2) +$~~

$T(n/2) = 2T(n/4) + 2\theta(n/4)$

∴ substituting in (i) gives,

$T(n) = 2\left[2T(n/4) + 2\theta(n/4)\right] + 2\theta(n/2)$

$= 2^2 T(n/2^2) + 2^2\theta(n/2^2) + 2\theta(n/2)$

$= 2^3 T(n/2^3) + 2^3\theta(n/2^3) + 2^2\theta(n/2^2) + 2\theta(n/2)$

$= \cdots$

$= 2^k T(n/2^k) + \cdots + 2^3\theta(n/2^3) + 2^2\theta(n/2^2) + 2\theta(n/2)$ ... (ii)

here, mystery calls itself recursively twice with $n/2$ array elements. Also it calls foo ~~with~~ ~~a~~ twice with ~~$n/2$ array~~ & size $n/2$

[Please turn over]

for $\frac{n}{2^k} = 1$, we get,

$2^k = n$

or, $k = \lg n$

$\therefore T(n) = nT(1) + \frac{n}{2}$

Using master's theorem:

from Master's theorem

$n^{\log_b a} = n^{\log_2 2} = n^1 = n^d$

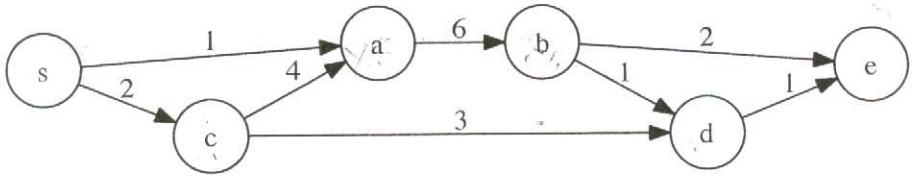
here, $a = 2$
$b = 2$
$d = 1$

$\therefore$ complexity is,

$T(n) = \theta(n \lg n)$.

Ans: $T(n) = \theta(n \lg n)$.

2. (*20 points*) Use Dijkstra's algorithm to find the shortest paths from s to all vertices. You do this by filling out the table below.



1/3

Show the list of vertices visited by Dijkstra's algorithm in the right order of exploration (starting with s).

2/3

20

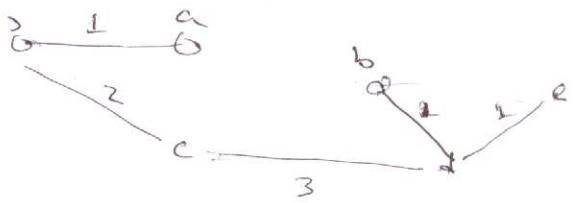
Table to keep track of the current shortest length from s. In the textbook, this is $length(v)$; in the lecture, I used $d(v)$.

	s	a	b	c	d	e
1st iteration	0	∞	∞	∞	∞	∞
2nd iteration	0	1 s	~~∞~~	2 s	∞	∞
3rd iteration	0	1 s	~~7~~ a	2 s	∞	∞
4th iteration	0	1 s	7 a	2 s	5 e	∞
5th iteration	0	1 s	7 a	2 s	5 c	6 d
6th iteration	0	1 s	7 a	2 s	5 c	6 d

labeled:

s = 0
a = 1
c = 2
d = 5
e = 6
b = 7

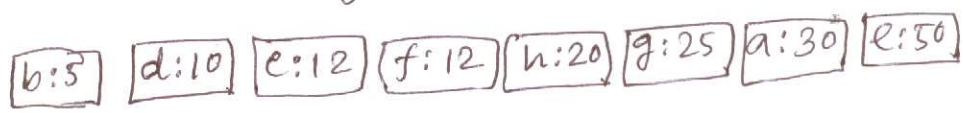
s a 1 ✓
b d 1 ✓
d e 1 ✓
s c 2 ✓
b e 2 ✓
c d 3 ✗
a b c ✗



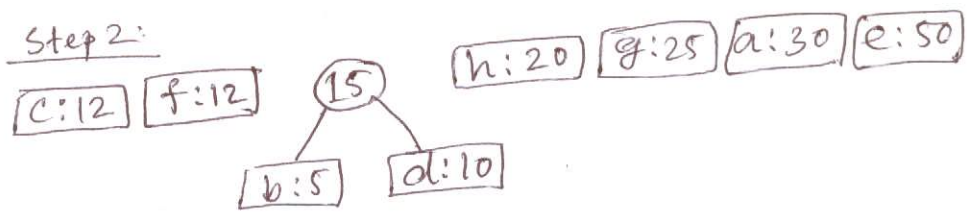
3. (*20 points*) Construct the Huffman code (Huffman tree + actual codewords) for the following alphabet and frequencies.

	freq.	codeword
a	30	00
b	5	0100
c	12	1000
d	10	0101
e	50	11
f	12	1001
g	25	101
h	20	011

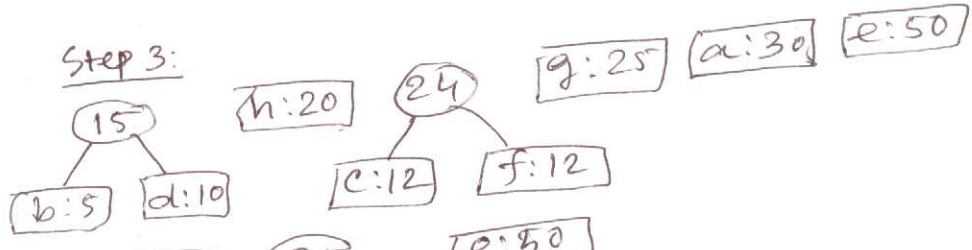
Step 1
In nondecreasing frequency:



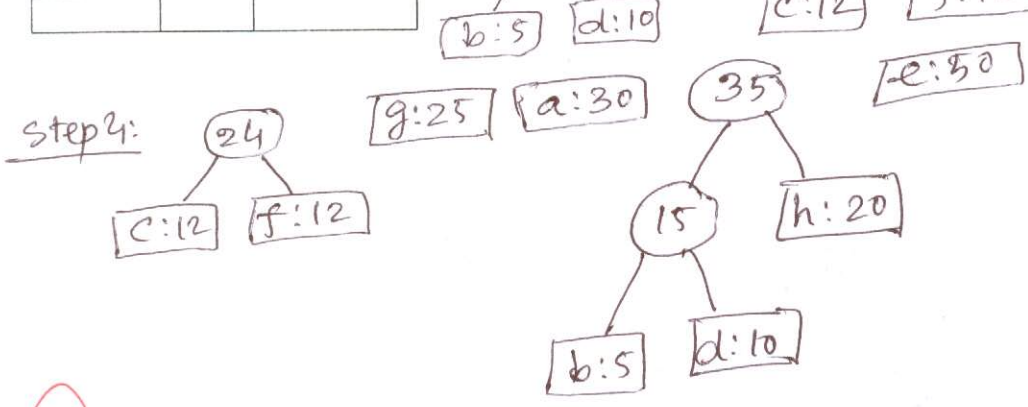
Step 2:



Step 3:

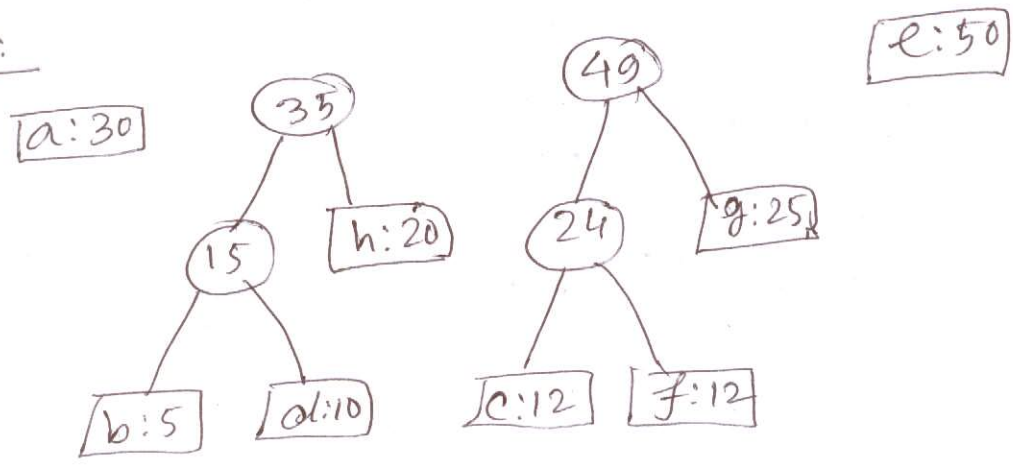


Step 4:



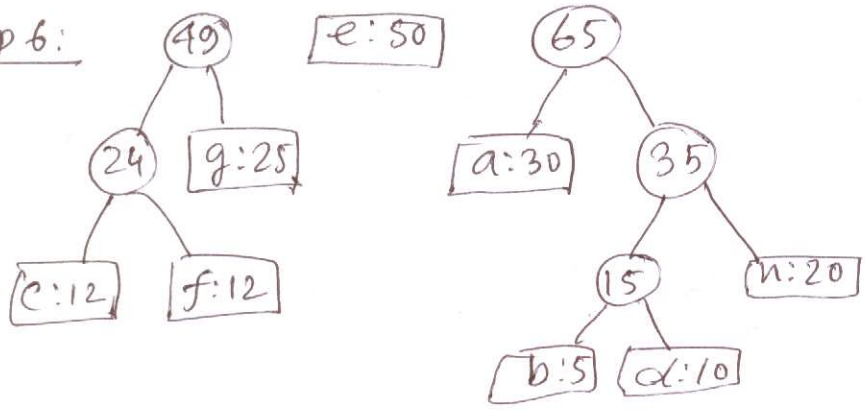
20

Step 5:

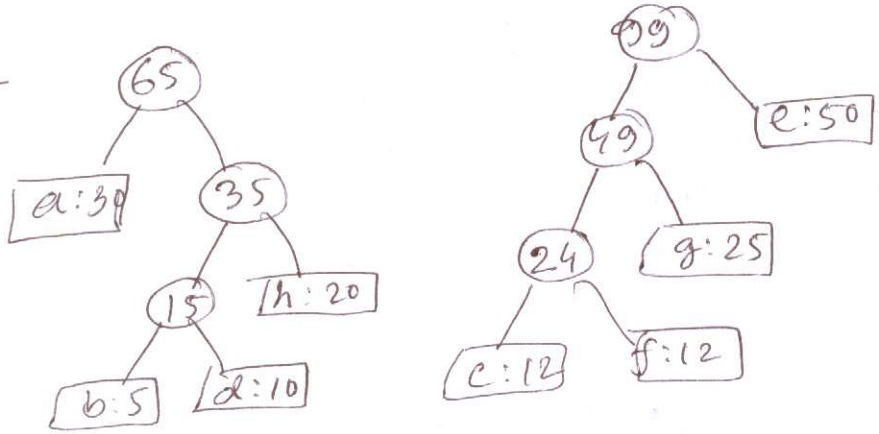


[P.T.O.]

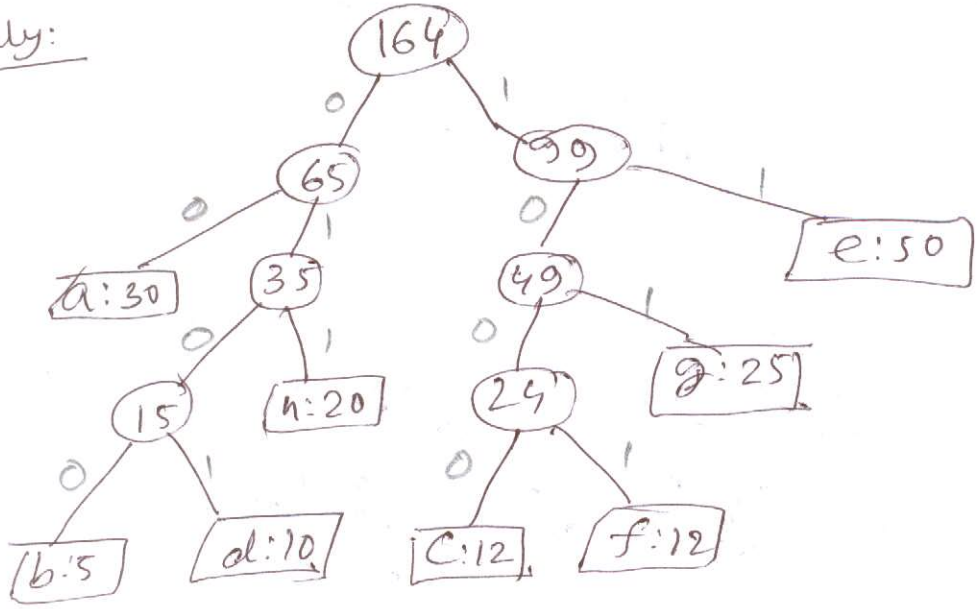
Step 6:



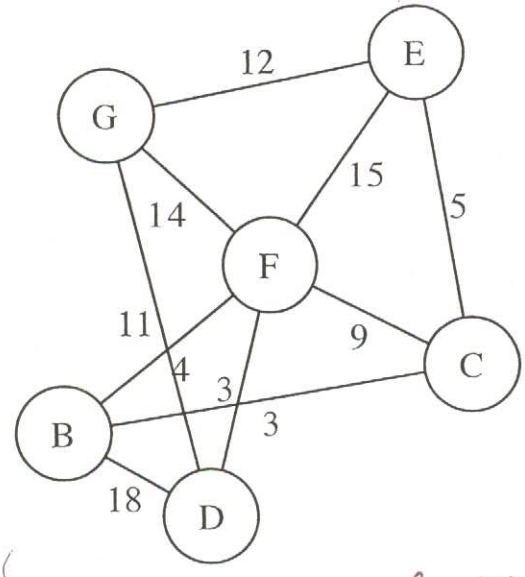
Step 7:



finally:

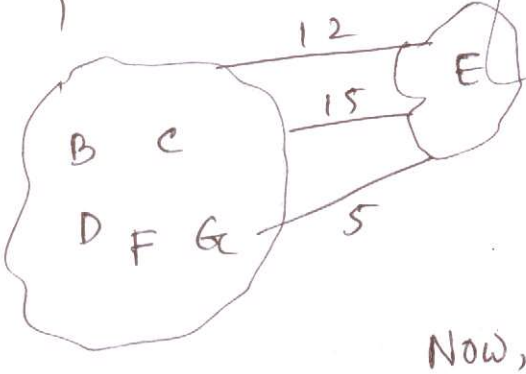


4. (*20 points*) Explain clearly (using the same reasoning we discussed in class about the correctness of greedy algorithms) why the edge (C,E) must be part of any minimum spanning tree.



20

consider the two disjoint set of vertices {E} and {B, C, D, F, G}



Assume, we already have a minimum spanning tree ~~for~~ ~~the~~ T' for the set {B, C, D, F, G}.

Now, to build the final ~~span~~ minimum spanning tree T for the whole graph we can take either edge GE, FE or CE.

consider the two subsets $S_1 \equiv \{B, D, F, G\}$ and $S_2 \equiv \{C, E\}$ and assume that we already have a minimum spanning tree T_1 for S_1.

Now, to connect $\{C, E\}$ to this tree T_1, we have the following options: (without forming cycle)

1) Add GE, EC to T_1
2) Add FE, EC to T_1
3) Add BC, CE to T_1
4) Add FC, CE to T_1.

or,

5) Add GE, FC
6) Add GE, BC
7) Add FE, FC
8) Add FE, BC

One of this 8 paths must be taken and, Calculating the weights, it is easy to show that the final must contain one of the first 4 combinations.

$\therefore$ CE must be part of the minimum tree.

[The optimal path adding C, E to T_1 is: BC, CE].

[Shown]

5. (*10 points*) A subsequence is *palindromic* if it is the same whether you read it left-to-right, or right-to-left. For example, in the sequence **A,C,G,T,G,T,C,A,A,A,A,T,C,G**, the subsequence A,G,G,A is palindromic; so is the subsequence A,C,A. They are not longest, however. Your task is to design an algorithm that finds *the length of a longest palindromic subsequence.*

Hint: a dynamic programming solution to this problem will need a definition with two parameters (e.g. $L[i][j]$).

To get full credits, you need to provide (1) a precise definition for $L[i][j]$, (2) an equation to compute $L[i][j]$, (3) an actual algorithm in pseudocode based on the equation.

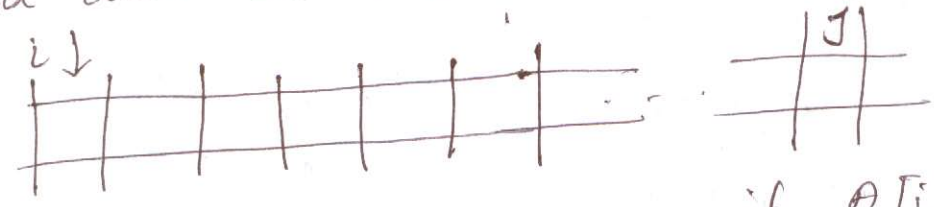
(i) Defining $L[i][j]$

we observe $L[i][i] = 1$ for $\forall i$

$$L[i][i+1] = \begin{cases} 2 & ; \text{ if } A[i] = A[i+1] \\ 1 & \text{if } A[i] \neq A[i+1] \end{cases}$$

where $A[i]$ is the i-th element in the sequence.

in order to compute $L[i][j]$, $(i<j)$, we observe that we shall be needing $L[i][i+1], L[i][i+2] \ldots L[i][j-1]$ and also $L[i+1][j], L[i+2][j] \ldots L[j-1][j]$.



$$L[i][i+2] = \begin{cases} 3 & ; \text{ if } A[i] = A[i+2] \\ L[i]2 & ; \text{ if } A[i] \neq A[i+2] \text{ and } (A[i] = A[i+1] \text{ or } A[i+1] = A[i+2]) \\ 1 & : \text{ if } A[i] \neq A[i+1] \neq A[i+2] \end{cases}$$

$$\therefore L[i][j] = \begin{cases} \max_{i<k<j} \{ L[k][j-1] + 2 \}; & \text{if } A[j] = A[k-1] \text{ for some } k \text{ } i<k<j \\ \max_{i<k<j} L[i][k] \end{cases}$$

Formal definition:

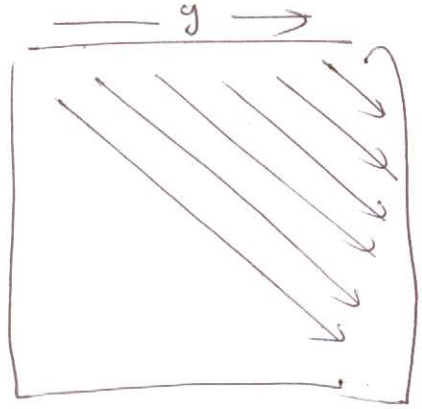
$$L[i][j] = \begin{cases} \max_{i<k<j} L[k][j-1] + 2 ; & \text{if } A[j] = A[k-1] \text{ for some } k \\ \max_{i<k<j} L[i][k] ; & \text{otherwise} \end{cases}$$

$L[i][i] = 1$ for all i

$L[i][i+1] = \begin{cases} 2 & \text{if } A[i] = A[i+1] \\ 1 & \text{if } A[i] \neq A[i+1] \end{cases}$

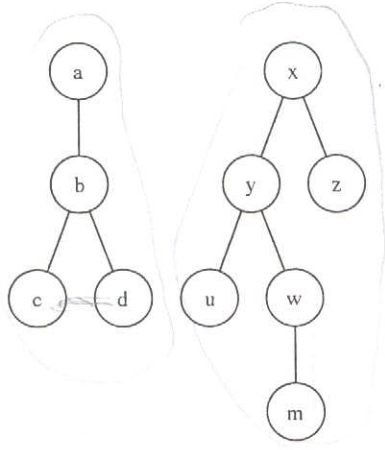
Algorithm: // compute diagonally

1) Initialize all $L[i][i] = 1$.
2) Init all $L[i][i+1] = \begin{cases} 2 \\ 1 \end{cases}$ $i \in (1, n)$ if $A[i] = A[i+1]$ if $A[i] \neq A[i+1]$
3) for diagonal = 1 to n



{ compute $L[i][j]$ as shown order in the figure }

6. (*10 points*) Design an algorithm that takes a tree T as input and returns True if T has a set of edges that touches each node exactly once. For example, for the tree on the left, the algorithm returns False. For the tree on the right, the algorithm returns True; in this case, the set of edges that satisfies the requirement is $\{(x, z), (u, y), (w, m)\}$.



10

1) Sort the vertices in nondecreasing order of degrees in a list named ~~order V~~ ordered_V.

2) ~~for~~ While Ordered_V is not empty:

pick a vertex v with lowest degree from ordered_V.

if degree(v) = 0: return false.

else,

{ pick an edge connecting v to its highest degree neighbour. remove both from the list Ordered_V

~~reduce the degree of the neighbour by 1 and~~

if no such neighbour found: return false

}

3) return true.

COMP 7712

FALL 2010

ASSIGNMENT 3

MD. MISHFAQ AHMED

ID : 00394646

100/100

15

20/20

The Algorithm:

```
Multiplication_number (int i, int j) {
    if (i == j)  return 0;
    minimum = ∞;
    for (k = i to j-1, k++) {
        number = Multiplication_number (i, k) +
                 Multiplication_number (k+1, j) +
                 d[i-1] * d[k] * d[j];   // assume d a
                                            global array
        if (number < minimum)
            minimum = number;
    }
    return minimum;
}
```

complexity:

let $T(n)$ = running time complexity for n matrices

$\therefore T(n)$ = running time complexity of Multiplication_number $(1, n)$

[P.T.O.]

But, two recursive calls will occur within each iteration of the loop,

$$\therefore T(n) = \sum_{k=1}^{n-1} \{T(k) + T(n-k) + O(1)\} + O(1)$$

$$\therefore T(n) \geq 2 \sum_{k=1}^{n-1} T(k)$$

$$\geq 2 \{T(n-1) + T(n-2) + \ldots + \ldots + T(1)\}$$

$$\geq 2\,T(n-1) \qquad \text{// ignoring all lower order values in terms of } n$$

$$\geq 2^2 T(n-2)$$

$$\geq 2^3 T(n-3)$$

from which we find:

$$T(n) = \Omega(2^n)$$

$\therefore$ A divide and conquer algorithm based on eqn. 3.5 is having exponential time complexity.

16(a) (20/20)

The algorithm:

```
NOD (int n) {
    if (n==1) return 1;
    total = 0;
    for (int i = 1 to n-1, i++)
        total += NOD (i) * NOD (n-i);
    return total;
}
```

// idea:

$(M_1 \cdot M_2 \cdot M_3 \ldots M_i)(M_{i+1} \ldots M_{n-2} \cdot M_{n-1} \cdot M_n)$

↑ ↑

possible splits

For all possible splits we have to consider the number of ways in which the left subsequence of Matrices can be multiplied and the number of ways in which the right subsequence of Matrices can be multiplied and finally multiply this two values and add it to the total.

21 (20/20)

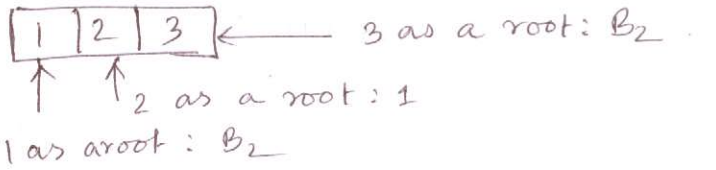
The keys are: 1 2 3 4 5 6 (let)

~~let B_i be the number of ways i keys can be used to construct a binary~~

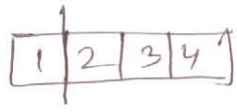
let B_i be the number of subtrees can be constructed using i distinct keys.

$\therefore$ $B_1 = 1$; $B_2 = 2$:

$B_3 = 2B_2 + 1 = 5$



for B_4:



B_4 = number of trees with (1) as a root

+ number " " " (2) " " "

+ " " " " (3) " " "

+ " " " " (4) " " "

= B_3

+ B_2

+ B_2

+ B_3

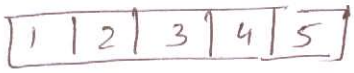
= $2B_3 + 2B_2$

= $2 \times 5 + 2 \times 2$

$\therefore B_4 = 14$

// with (1) root left subtree is empty and right subtree can be arranged in B_3 ways. ~~same~~ or similar argument follows for other roots

for B_5:



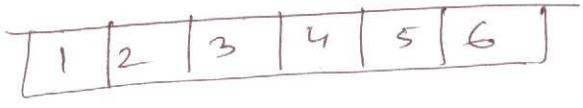
B_5 = number of trees with (1) as root +
" " " " (2) " " +
" " " " (3) " " +
" " " " (4) " " +
" " " " (5) " "

$= B_4 + B_3 + B_2 * B_2 + B_3 + B_4$

$= 2*14 + 2*5 + 2*2$

$\therefore B_5 = 42$

for B_6:



B_6 = number of trees with (1) as root +
" " " " (2) " " +
" " " " (3) " " +
" " " " (4) " " +
" " " " (5) " " +
" " " " (6) " " +

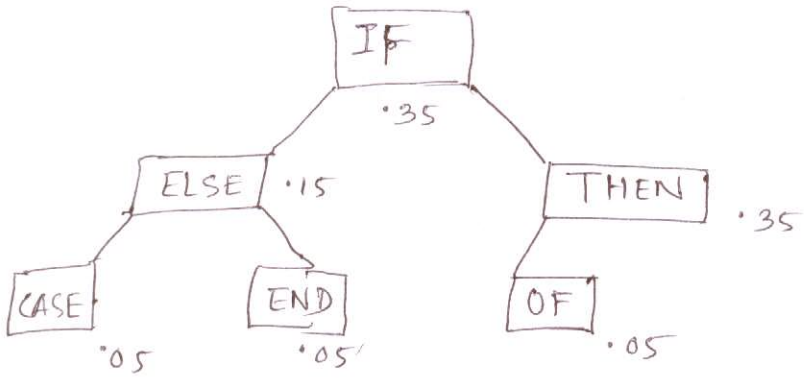
$= B_5 + B_4 + B_2 * B_3 + B_3 * B_2 + B_4 + B_5$

$= 2*42 + 2*14 + 2*5*2$

$\therefore B_6 = 132$

Answer: 132

22. 20/20



~~27.~~

~~The equations~~:

~~A[i][j]~~

(20/20)

27 The equations are:

$$A[i][j] = \underset{i \le k \le j}{\text{minimum}} \left(A[i][k-1] + A[k+1][j] \right) + \sum_{m=i}^{j} p_m ; \quad [i < j]$$

$$A[i][i] = p_i$$

$$A[i][i-1] = 0 , \quad A[j+1][j] = 0$$

To find the value of $T(n)$: {running time for input size n}.

$$T(n) = \sum_{K=1}^{n} T(k-1) + T(n-k) + \sum_{k=1}^{n} p_k \; O(n)$$

// since any of the n key can be root, so k can be 1, 2 ... ... or n.

$$\therefore T(n) \geq 2 \sum_{K=0}^{n-1} T(k)$$

$$\geq 2 \left[T(n-1) + T(n-2) + \cdots + T(1) + T(0) \right]$$

$$\therefore T(n) \geq 2T(n-1)$$

$$\geq 2^2 T(n-2)$$

$$\geq 2^3 T(n-3)$$

$$\vdots$$

$$\therefore T(n) = \Omega(2^n) ,$$ hence the diviide and conquer algorithm would be exponential.

Bonus Assignment 2

Syed Monowar Hossain
UUID: 00397127

2

This problem can be solved using the idea of "hamming code" that can detect and correct single bit error.

In this problem, 3 bits are required to represent any number between 1 to 8. Let "r" check bits are required to correct all single bit errors. Let $n = 3+r$. Each of the 2^3 legal numbers has "n" illegal codewords at a distance 1 from it (we have to ask "n" questions to determine "x" assuming that, among "n" questions at most 1 answers is a lie.

So, each of the 2^3 numbers requires $n+1$ bit patterns dedicated to it. Since the total number of bit patterns is 2^n, we must have

$$(n+1)2^3 \leq 2^n \quad \because n = m+r = 3+r$$

$$(3+r+1)2^3 \leq 2^{r+3} \Rightarrow r+4 \leq 2^r$$

we can find the least "r" that supports the above equation and here $r = 3$

So, total questions that are required to ask is $3+3 = 6$, among which one answer may be a lie.

Questions that are to be asked:- First convert $(x-1)$ to binary.

Q1, Q2, Q3: Is the "i"th bit of $(x-1)$ is 1? where $i = 1, 2, 3$
(Here $x-1$ is used to make the range of the number between 0 to 7 instead of 1 to 8.)

Q4: Is the {1st bit of $(x-1)$ + 2nd bit of $(x-1)$} % 2 = 1?

Q5: is the {2nd bit of $(x-1)$ + 3rd bit of $(x-1)$} % 2 = 1?

Q6: Is the {3rd bit of $(x-1)$ + 1st bit of $(x-1)$} % 2 = 1?

By the answer of above 6 questions and using the idea of hamming code it can be easily identified if there's any lie among this 6 questions and after correcting, "x" can be found.

Sudip Saha
Homework #3
Algorithm

Problem 1:

$$T(n) = n^2 + T\left(\frac{2n}{5}\right) + T\left(\frac{3n}{5}\right)$$

(8+2)

We have to prove that, $T(n) = O(n^2 \log n)$

Base case:

Here no base case has been provided in the equation. However, the condition of being $O(n^2 \log n)$ requires that n should be at least greater than 1. Because, $1^2 \log 1 = 0$.

If we assume that $T(n) = 0$, for $n < 2$, then we can prove the above statement for base case, $n = 2$.

2

$$T(2) = 2^2 + T(0.8) + T(1.2)$$
$$= 4 + 0 + 0$$
$$= 4 \leq c \cdot 2^2 \log 2, \text{ if } c \text{ is at least } 1.$$

Inductive step:

We assume that,

$$T(n) = O(n^2 \log n)$$

for $n = 2, 3, \ldots, K$

Now we have to prove it for $n = K + 1$.

Proof:

$$T(K+1) = (K+1)^2 + T\left(\frac{2(K+1)}{5}\right) + T\left(\frac{9(K+1)}{5}\right)$$

$$\therefore T(K+1) \le (K+1)^2 + c.\frac{4(K+1)^2}{25}\log\frac{2(K+1)}{5} + c.\frac{9(K+1)^2}{25}\log\frac{3(K+1)}{5}$$

$$= (K+1)^2 + \frac{c(K+1)^2}{25}\left[4\log\frac{2(K+1)}{5} + 9\log\frac{3(K+1)}{5}\right]$$

$$= (K+1)^2 + \frac{c(K+1)^2}{25}\left[4\log(K+1) + 4\log\frac{2}{5} + 9\log(K+1) + 9\log\frac{3}{5}\right]$$

$$= (K+1)^2 + \frac{c(K+1)^2}{25}\left[13\log(K+1) + 4\log\frac{2}{5} + 9\log\frac{3}{5}\right]$$

$$= (K+1)^2 + \frac{13c}{25}(K+1)^2\log(K+1) + \frac{c(K+1)^2}{25}\left[4\log\frac{2}{5} + 9\log\frac{3}{5}\right]$$

$$= \frac{13c}{25}(K+1)^2\log(K+1) + (K+1)^2\left[1 + \frac{c}{25}\left(4\log\frac{2}{5} + 9\log\frac{3}{5}\right)\right]$$

$$= \frac{13c}{25}(K+1)^2\log(K+1) + (K+1)^2\left[1 + \frac{c}{25}(-12)\right] \qquad \left[\because 4\log\tfrac{2}{5} + 9\log\tfrac{3}{5} \approx -12\right]$$

$$= \frac{13c}{25}(K+1)^2\log(K+1) + (K+1)^2\left(1 - \frac{12c}{25}\right)$$

Now, $T(K+1) = O((K+1)^2\log(K+1))$ if,

$$\frac{13c}{25}(K+1)^2\log(K+1) + (K+1)^2\left(1 - \frac{12c}{25}\right) \le c(K+1)^2\log(K+1)$$

$$\Rightarrow \frac{12c}{25}(K+1)^2\log(K+1) \ge (K+1)^2\left(1 - \frac{12c}{25}\right)$$

$$\Rightarrow \frac{12c}{25}\log(K+1) \ge 1 - \frac{12c}{25}$$

$$\Rightarrow \frac{12c}{25}\left(1 + \log(K+1)\right) \ge 1$$

$$\therefore c \ge \frac{25}{12\left[1 + \log(K+1)\right]}$$

Highest value of $\frac{25}{12[1+\log(K+1)]}$ occurs when $K=0$

If $K=0$, $\frac{25}{12(1+\log(K+1))} = \frac{25}{12(1+\log 1)} = \frac{25}{12}$

$\therefore c \geq \frac{25}{12}$

for the sake of safety, we can take $c=3$ ANS.

— o —

Problem 2

Assumption:

First n elements of $A[\cdot]$ are integers and the rest are α. We can perform the comparison $(A[i] = \alpha)$. The procedure returns p, if $A[p] = x$ for any p; otherwise it returns -1.

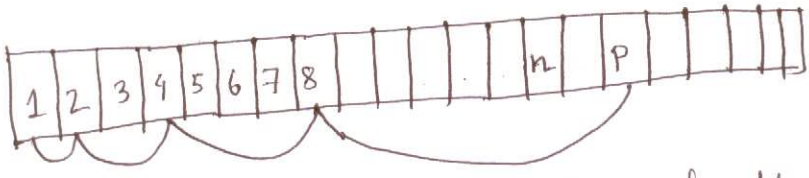
Procedure:

```
Procedure findX (A[·], x)
{
    if (A[1] = α)
        return -1;
    K = boundary(A[·], 1);          [ O(⌈log n⌉) ]
    p = binary_search (A[·], 1, K, x);   [ O(⌈log n⌉) ]
    return p;
}
Procedure boundary (A[·], p)
{
    if (A[p] ≠ α and A[p+1] = α) return p;
    else if (A[p] = α) return p;
    else return boundary (A[·], 2p);
}
```

```
Procedure binary_Search (A[·], F, L, x)
{ if (F<L) return -1;
  else if (F==L)
    { if (A[F]=x)  return F;
      else return -1;
    }
  c = ⌊(F+L)/2⌋;
  if(A[c]==x)
        return c;
  else if (A[c]>x) return binary_Search (A[·], F, c-1, x);
  else             return binary_search (A[·], c+1, L, x);
}
```

Complexity:

The procedure boundary(A[·], p) works ~~like~~ as follows,



So, it returns the smallest P such that,

$P \geq n$ and $P = 2^q$, for some q.

that is, $2^{q-1} < n \leq 2^q$

$\Rightarrow q-1 < \log n \leq q$ $\therefore q = \lceil \log n \rceil$

For example, if $n = 14$, it returns 16.

if $n = 16$, it returns exactly 16.

So, the complexity of procedure boundary(A[·], P) is

$$O(\log P)$$
$$= O(q)$$
$$= O(\lceil \log n \rceil)$$

As the input of procedure binary_search(A[·], f, l, x) is also an array of size P. so it's complexity is also $O(\lceil \log n \rceil)$

So, overall complexity of procedure findX(A[·], x) is

$$O(\lceil \log n \rceil) + O(\lceil \log n \rceil)$$
$$= O(\lceil \log n \rceil)$$
$$= O(\log n)$$

Problem 3:

Assumptions:

We assume that, the two lists have different distinct numbers. We assume that, $K<m$ and $K<n$, where m and n are the length of array A and B.

Procedure:

```
Procedure findKSmallest ( A, B, K)        [Input: Two arrays A and
{                                          B of size m and n respectively
                                           Output: Kth smallest elemen
                                                   of AUB ]
  if ( B[K/2] < A[K/2] < B[K/2 + 1] )
                          return A[K/2] ;
  else if ( A[K/2] < B[K/2] < A[K/2 + 1] )
                          return B[K/2] ;
  else if ( A[K/2] > B[K/2 + 1] )
          {   a1 = 1 ;   a2 = K/2 - 1 ;  }
  else    {   a1 = K/2 + 1 ;  a2 = K ;   }
  While (1)
  {  am = ⌊(a1 + a2)/2⌋ ;
     diff = am - a1 ;
     bm = K - diff ;
     if ( B[bm] < A[am] < B[bm + 1] )  return A[am];
     else if ( A[am] < B[bm] < A[am + 1] )  return B[bm];
     else if  A[am] > B[bm + 1]
                          a2 = am - 1 ;
     else
                          a1 = am + 1 ;
  }
}
```

Illustration:

The concept here is that,

we are tracking two indices am and bm to lists A and B respectively such that,

$$a_m + b_m = K$$

But turning those am, bm right or left, we are trying to come to a point when either,

$A[a_m] < B[b_m] < A[a_m+1]$, when b_m is the kth smallest elmnt

Or,

$B[b_m] < A[a_m] < B[b_m+1]$, when a_m is the kth smallest elem

And we start from $\frac{K}{2}$ for A and B, as well.

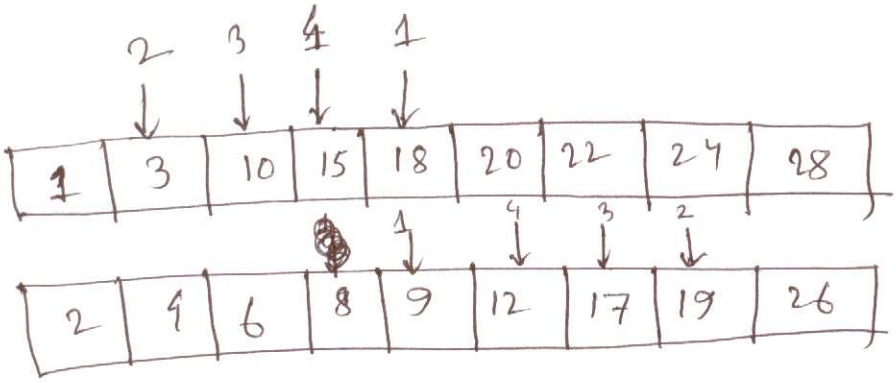
For example, for the following two arrays,

A	1	3	10	15	18	20	22	24	28

B	2	4	6	8	9	12	17	19	26

We want to find the 10th smallest element,

We start from A[5] and B[5]



Here $\overset{1}{\downarrow}$ means first iteration, $\overset{2}{\downarrow}$ means 2nd, so on.

Complexity:

Both of A and B's size gets reduced to half in each ~~subpro~~ iteration and it starts for ~~&~~ K.

So, the complexity is

$O(K)$ ~~whe~~

$O(K)$ ~~wl~~ is also $O(\log m + \log n)$

Because $K \leq m$, $K \leq n$.

In fact, ~~if~~ we take the assumption of distinct elements in the arrays, then the complexity becomes $O(K)$, which is also $O(\log m + \log n)$.

Problem 4:

```
Procedure Circle (G1, e)
{
    for each u∈V, u.visited=false;
    u, v ← the two end nodes of e;
    explore (G1, u, v);
    if (u.visited == true) return 1;
    else return 0;
}
```

[Input: $G_1 = (V, E)$ is an undirected graph with V and E as vertex and edge set
e ← the edge
Output: 1, if there is a cycle
0, if there is no cycle]

```
Procedure explore (G1, u, v)
{ v.visited = true;
  if (v == u) ret return;
  for each (v, x) ∈ E
      if (x.visited == false)
          explore (G1, u, x);
}
```

Illustration:

If e has vertices u and v,
then we start from v and explore,
Eventually if we encounter u, we stop
and at this, we know that there is a cycle containing e

Complexity:

This is ~~the~~ almost the same as DFS.
And its complexity is,

$$O(V+E) .$$

Problem 5:

```
Procedure topsort (G)
{
    for each u ∈ V,  u.visited = false;
    S ← an empty stack
    L ← an empty list to store sorted nodes;
    for each u ∈ V,
        if (u.visited == false)
            explore (G, u, S);
    While (S not empty)
        L.insert (S.pull() );
}

procedure explore (G, u, S)
{
    u.visited = true;
    for each (u,v) ∈ E.
        if (v.visited == false)
            explore (G, v, S)
    S.push (u);
}
```

2

Complexity:

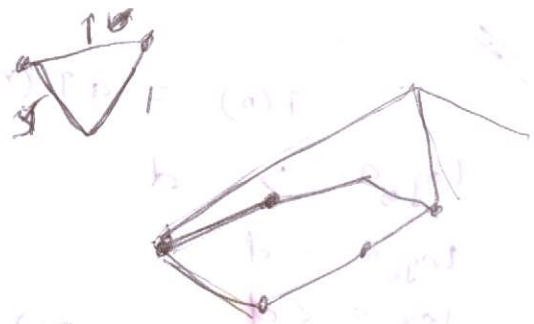
It's almost a DFS.

So, its complexity is $O(V+E)$.

Problem 5:

Input: $G = (V, E)$,
$u, v \leftarrow$ source and destination nodes,
Output: number of shortest paths

```
Procedure BFS (G, u, v)
{
  for each x ∈ V, dist(x) = ∞;
  dist(u) = 0;
  Q ← empty Queue;
  counter = 0;
  Q.inject(u);
  while (Q not empty)
    {  x = Q.eject();
       for all (x,y) ∈ E
        {
          if (dist(y) == ∞)
            {
              Q.inject(y);
              dist(y) = dist(x) + 1;
              if (y == v) counter = 1;
            }
          else if (y == v)
            {
              if (dist(v) == dist(x) + 1)
                counter ++;
            }
        }
    }
  return counter;
}
```



Complexity:

This works ~~almost~~ the same way as BFS.
So, its complexity is $O(V+E)$.

—— o ——

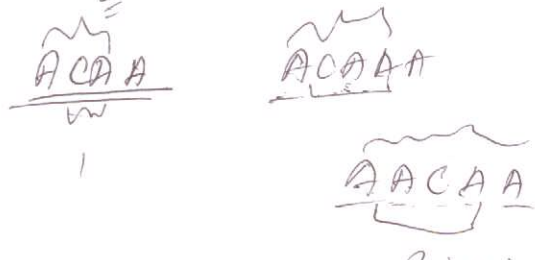
$$T(n) = aT(n/b) + O(n^d)$$

$\log_b a > d$		$O(n^{\log_b a})$
$\log_b a = d$		$O(n^d \log n)$
$\log_b a < d$	→	$O(n^d)$

Merge sort $a[1 \cdots n]$

if $n > 1$:
  return merge(mergesort($a[1, \ldots \lfloor n/2 \rfloor]$), mergesort($a[\lfloor n/2 \rfloor + 1, \ldots n]$))
else
  return a

merge ~~sort~~ ($x[1 \cdots k]$, $y[1, \cdots l]$)
  if $k = 0$ ret $y[1 \cdots l]$
  if $l = 0$ ret $x[1 \cdots k]$
  if $x[1] \leq y[1]$
    return $x[1] \circ$ merge ($x[2 \cdots k]$, $y[1 \cdots l]$)
  else return $y[1] \circ$ merge ($x[1, \cdots k] \circ y[2 \cdots l]$)



COMP 7712

HW # 3

Submitted by

Muhammad M. Khan

U00333660

9.5/10

Ch2. Problem 7

L is a list with n elements

low ← left most index of list

high ← righ most index of list

~~largest (L, l)~~

largest (low, high)

1. if (low == high)
   return L[low] ;
2. else { max1 = largest (low, $\lfloor \frac{low+high}{2} \rfloor$) ;
3. max2 = largest ($\lceil \frac{low+high}{2} \rceil$, high) ;
4. if (max1 > max2)
5. return max1 ;
6. else return max2;

Complexity

Let us assume that for n elements the running time of 'largest' algorithm T(n)

steps 1, 4, 5, 6 take constant time

steps 2, 3 take T(n/2) step each

$\therefore$ $T(n) = c + 2T(n/2)$

$$T(n) = c.n^0 + 2T(n/2)$$

$$d=0, \quad a=2, \quad b=2$$

$$\log_2 2 \neq 0$$

According to Masters theorem

$$T(n) = \Theta\left(n^{\max\{0, \log_2 2\}}\right)$$

$$= \Theta(n)$$ ✓

Ch2. Problem 10

~~The~~

Given a list of n ~~dist~~ distinct positive integers. A.

Partitioning the list into two sublist where the difference between the sums of the two sublists are maximum. It is possible by sorting the list A in nlogn as the time constraint ~~for~~ worst case is nlogn.

Mergesort has a worst case time complexity of nlogn.

So, this problem can be solved with a mergesort algorithm.

```
MergeSort (low, high)
1.  {  if (low < high)
2.          mid = ⌊(low+high)/2⌋ ;
3.          mergesort(low, mid);        → merge.
4.          mergesort (mid+1, high);
5.          merge(low, mid, high)
    }
```

~~Merge~~

```
merge (low, mid, high)
{ List U[low ... high]
  i = low; j = mid+1; K = low;
  while (i ≤ mid && j ≤ high) {
        if (S[i] < S[j]) {
              U[k] = A[i];
              i++
        }
        else {
              U[k] = A[j];
              j++;
        }
        K++;
  }
  if (i > mid)
        move A[j ... high] to U[k ..... high]
  else
        move A[i ... mid] to U[k ... high]
  move U[low ... high] to A[low ... high]
```

$T(n)$ is the running time for merge sort for n elements n is multiple of 2.

So, Time take for merge is $n-1$

$$T(n) = 2T(n/2) + n - 1$$

$$d = 1, \quad a = 2, \quad b = 2$$

$$\log_2 2 = 1$$

According to Masters theorem

$$T(n) = \Theta(n \log n)$$ ✓

Chapter 2. Problem 40

Let $M[n][m]$ is a 2-D array, x is the element to search

top ← index of the top row

bottom ← index of the bottom row

left ← index of the leftmost column

right ← index of the rightmost column

2DBinarySearch (M, top, bottom, left, right, x)

1. if (top > bottom) or (left > right)

   return

2. if ($x = M(\lfloor \frac{top+bottom}{2} \rfloor, \lfloor \frac{left+right}{2} \rfloor)$).

   return True

3. else if ($x < M(\lfloor \frac{top+bottom}{2} \rfloor, \lfloor \frac{left+right}{2} \rfloor)$)

4. return 2DBinarySearch (M, $\lfloor \frac{top+bottom}{2} \rfloor$, bottom, $\lfloor \frac{left+right}{2} \rfloor$, right, x)

5. else

   return 2DBinarySearch (M, top, $\lfloor \frac{top+bottom}{2} \rfloor - 1$, left, right, x)

6. return 2DBinarySearch (M, $\lfloor \frac{top+bottom}{2} \rfloor$, bottom, left, $\lfloor \frac{left+right}{2} \rfloor - 1$, x)

Time Complexity n, m are the row and column size of array M.

2DBinarySearch takes $T(n,m)$ time

Steps 1, 2, 3 takes constant time

steps 4, 6 take $T(\frac{nm}{4})$

Step 5 takes $T(nm/2)$ or $2T(nm/4)$

So, $T(nm) = c + 4T(\frac{nm}{4})$

$= c + 4T(\frac{N}{4})$ $N = nm$

$d=0, \quad a=4, \quad b=4$

$\log_4 4 \neq 0$

So, according to Masters theorem

$$T(n) = \Theta\left(N^{\max\{0, \log_4 4\}}\right)$$

$= \Theta(N)$

$= \Theta(nm)$ ✓

efficient ?
↓
reduce
↓
log

-0.5

Chapter 2. Problem 45

A is an array of real values

L ← leftmost index of array A

R ← rightmost index of array A

Mid ← the point where we split the array into two subarray

LeftSum ← the maximum 'contiguous sum' in the left sublist or sub array

RightSum ← the maximum 'contiguous sum' in the right sublist or subarray

MaxLeftBorderSum ← the maximum 'contiguous sum' whose corresponding values make up the last 'l' positions of left sublist

MaxRightBorderSum ← the maximum 'contiguous sum' whose corresponding values make up the first 'r' positions of right sublist

```
MaxContSum (A, L, R)
1. if (L == R)  return A[L];
2. Mid = ⌊(L+R)/2⌋
3. LeftSum = MaxContSum (A, L, Mid)
4. RightSum = MaxContSum (A, Mid+1, R)
5. MaxLeftBorderSum = 0;
6. LeftBorderSum = 0;
```

```
7.  for (i= mid ; i>=0 ; i--)
8.          LeftBorderSum += A[i];
9.          if MaxLeftBorderSum < LeftBorderSum
10.             MaxLeftBorderSum = LeftBorderSum
            end if
    end for
11. MaxRightBorderSum =0
12. RightBorderSum=0
13. for (i=mid+1 ; i<=R ; i++)
14.         RightBorderSum += A[i]
15.         if MaxRightBorderSum < RightBorderSum
16.             MaxRightBorderSum = RightBorderSum
            endif
    end for
17. return max (LeftSum , RightSum , MaxLeftBorderSum +
                                     MaxRightBorderSum)
```

Procedure Max(A, B, C)

1. Mx = A;
2. if Mx < B
3. Mx = B
4. if Mx < C
5. Mx = C
6. return Mx

~~Complexi~~

Complexity Lets assume for n elements the runningtime is T(n)

Steps 1, 2, 5, 6, 11, 12 take constant steps

Steps 3, 4 take T(n/2) each

Steps 7 to 10 and 13 to 16 each take n/2 steps

Steps 17 takes constant steps

$$\therefore T(n) = c + n/2 + n/2 + 2T(n/2)$$
$$= c + n + 2T(n/2)$$

$d = 1, \quad a = 2, \quad b = 2$

$\log_2 2 = 1$

According to Master theorem

$$T(n) = \Theta(n \log n)$$

COMP 7712

Algorithm ~~Ass~~ Assignment # 4

7+0.5/10
= 7.5

Submitted by
Muhammad M. Khan
(U00333660)

Ex 16(a)

Solution

For $n=1$ Result is 1 ; (A_1)

For $n=2$ Result is 1 : $(A_1 A_2)$

For $n=3$ Result is 2: $(A_1)(A_2A_3)$ or $(A_1 A_2) A_3$

The relation can be formulated as combination for n

$$\text{combination}(n) = \begin{cases} 1 & \text{if } n=1 \\ \sum_{i=1}^{n-1} \text{combination}(i) * \text{combination}(n-i) \end{cases}$$

We can write the formula in a recursive algorithm. Suppose we call this comb(n) where n is the number matrices to be multiplied.

Now.

```
comb(n) {
1.      result=0;
2.  if n == 1
        return 1;
3.  for (i=1 to n-1)
4.      result = result + comb(i) * comb(n-i);
5.  return result;}
```

~~Proste~~
Ex. 10

```
void order (index i, index j) {
1. if (i==j)
        cout << "A" << i;
2. else {
        K = P[i][j];
3.      cout << "(";
4.      order(i, K);
5.      order(K+1, j);
6.      cout << ")"; }}
```

Recursively.

$T(n) = T(n-k) + T(k) + \ldots$

Steps 1, 2, 3 & 6 takes constant time or $O(1)$

Step 5 & 6 divide the problem into two disconnected problem of size k and $n-k$. So, the problem is divided into n disjoint problems and each disjoint problem runs 1, 2, 3 & 6

need to / more clearly

so,

$$T(n) = n * 4O(1)$$
$$= 4n\,O(1)$$
$$= O(1)$$

So, the algorithm takes linear time

induction?

-1.0

Problem: Ex 38

Soution

b Suppose we have n real values in a list

$$R = r_1, r_2, \ldots, r_n$$

$S_{i,j}$ is the sum of the elements from r_i to r_j

Mx is the maximum sum.

If $S_{i,j-1} + r_j < 0$ then j-th element cannot be in the maximum ~~subt~~ contiguous sublist. Moreover it splits the ~~problem~~ list into $r_1, r_2, \ldots r_{j-1}$ and $r_{j+1}, \ldots r_n$. The maximum sublist can be ~~any~~ in any of the sublist. ~~This~~

The algorithm can be written as

MxSum (R_n)

1. $Mx = 0$, $S = 0$;
2. for (i=1 to n) {
3. $S = S + R[i]$;
4. if (S<0)
5. $S = 0$;
6. $Mx = max(Mx, S)$; }
7. return M ?

$\rightarrow max\{S_{i-1} + R_i, R_i\}$

S

* max is a simple algorithm that returns maximum of Mx & S

Time Complexity

Steps 1 & 3-7 takes constant time $O(1)$

The for loop iterate n times.

So, the time complexity is linear

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

Ex.39

Solution

We have two strings ~~S1 & S2~~ S_1 & S_2

$S_1 = x_1 x_2 \ldots x_n$

$S_2 = y_1 y_2 \ldots y_m$

Let us assume

$L_{i,j}$ = length of the longest common subsequence of S_1 & S_2

$$L_{ij} = \begin{cases} L_{i-1,j-1} + 1 & \text{if } S_1[i] == S_2[j] \longrightarrow \text{case 1} \\ \max(L_{i,j-1}, L_{i-1,j}) & \text{otherwise} \longrightarrow \text{case 2} \end{cases}$$

If we return $L[n][m]$, we get the length of LCS

If we analyse the above condition, we see that for case 1 we have found an element of LCS. We have to save each element found & return result.

The algorithm can be written as

LCS (S_1, S_2)

1. $n = S_1.length$
2. $m = S_2.length$
3. lcs_length = 0 , $L[n][m]$ a 2-D matrix for storing length values
4. result = new list created empty.
5. for ($i = 0$ to n)
6. $L[i][0] = 0$;

```
7.  for (i = 0 to m)
8.      L[0][j] = 0 ;
9.  for (i = 1 to n)
10.     for (j = 1 to m)
11.         if (S1[i] == S2[j])
12.             L[i][j] = L[i-1][j-1] + 1 ;
13              lcs_length = L[i][j] ;
14.             result[lcs_length] = S1[i] ;
15.         else
16.             L[i][j] = max (L[i][j-1], L[i-1][j]);
17. return result ;
```

?

(...)

Construct sequence ?

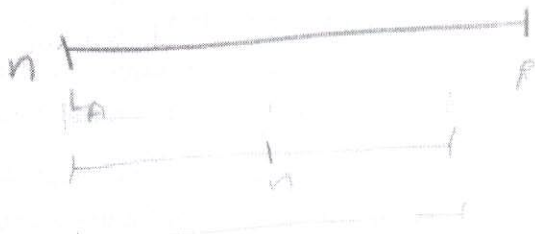
96

Exam 3 - COMP 7712 - Fall 2010 Name: Dagi Dong

1. (*10 points*) The algorithm f takes as inputs two arrays A and B, and indices L_A, R_A and L_B, R_B of A and B respectively. A and B have the same number of elements. You can also assume that there are n elements between L_A and R_A (and between L_B and R_B).

Algorithm 1 $f(A, L_A, R_A, B, L_B, R_B)$

1: **if** $L_A >= R_A$ **then**
2: **return** 1
3: Sort $A[L_A...R_A]$
4: Sort $B[L_B...R_B]$
5: $m_A = \frac{L_A+R_A}{2}$, $m_B = \frac{L_B+R_B}{2}$.
6: $sum_A = 0, sum_B = 0$
7: **for** i from L_A to m_A **do**
8: $sum_A += i^3$
9: **for** i from m_B to R_B **do**
10: $sum_B += i^5$
11: **return** $sum_A + sum_B + f(A, L_A, m_A - 1, B, m_B + 1, R_B)$



Let $T(n)$ be the running time of this algorithm. Write out the equation for $T(n)$.

10 $T(n) = T(\frac{n}{2}) + n + 2n\lg n$ (I assume running time of function Sort is best one : $n \lg n$)

2. (*5 points*) Determine the running time of $T(n)$ in terms of big-O notation. (Big-O is sufficient; you don't need to determine θ).

Ans:

I use theorem of master to determine the big-O

because $T(n) = T(\frac{n}{2}) + n + 2n\lg n$

$= T(\frac{n}{2}) + n(1 + 2\lg n)$

5

set. $1 + 2\lg n = n^x \quad (x > 0)$,

so. $T(n) = T(\frac{n}{2}) + n^{1+x}$

because. $a = 1, \ b = 2, \ d = 1+x. \Rightarrow \log_b a = \log_2 1 = 0 < d = 1+x$

so, big-O is $O(n + 2n\lg n)$

Much more. because. $n + 2n\lg n < 3n\lg n$ if $n > 1$. $c = 3$.

so. big-O is $O(n\lg n)$

3. (*10 points*) Suppose that the arrays A and B in problem 1 consist of only integers between 0 and 200. Describe briefly how we can reduce the running time algorithm f.

Ans:

First, we could find out that running time of f is equal to running time of function Sort(). So, our aim is to reduce the running time of function Sort().

Second, because the scale of input of Sort() is limited and which is integers only, ↦ So, we could use algorithm of counting Sort to implement this Sort() func. Briefly the concept of counting Sort is that does not use comparison but use extended array to help remember temp position, which means saving time by using more spaces.

Finally, because R.T. of Sort() is $O(n)$ via counting Sort, so, we could reduce the running time of f to $O(n)$.

10

4. (*10 points*) Continuing on the previous problem, determine the running time of f in θ notation. Write out $T(n)$ and solve it using either Master's theorem or substitution.

Ans:

Based on the answer of previous problem (Q3), we know that,

$$T(n) = T\left(\frac{n}{2}\right) + n + 2n$$

$$= T\left(\frac{n}{2}\right) + 3n.$$

10

I use theorem of Master to determine

because $a = 1$, $b = 2$, $d = 1$,

so, $\log_b a = \log_2 1 = 0 < d = 1$

therefore, the running time is $\theta(n)$.

Max Sum

5. (*10 points*) In this problem, you need to find out the maximum-sum of a substring of a string. The input is a string $s_1, \cdots, s_n$. Each s_i is a number (positive or negative).

For example, the input can be $5, 15, -30, 10, -5, 40, 10$.

There are several substrings. For example, $5, 15, -30$ (sum=-10); or $-30, 10$ (sum=-20), or $40, 10$ (sum=50). The substring with maximum sum is $10, -5, 40, 10$ with sum=55.

Design an $\theta(n)$-dynamic programming algorithm (written in pseudocode) to find the maximum sum of all substrings.

Hint: First, define $Sum[i]$ to be the maximum sum of any substring, which must end at index i. Then, find the equation for Sum.

Your answer:

step-1: I define the sum[i] below as the mention of Hint:

$Sum[1] = S_1$

$$Sum[i] = \begin{cases} sum[i-1] + S_i & \text{if } sum[i-1] > 0 \\ S_i & \text{if } sum[i-1] \le 0. \end{cases}$$

step-2:

I implement the algorithm in pseudocode as below via a func "subMax(int)".

```
int subMax(int i){
    if (i==1)
        return S[1];
    else{
        if sum[i-1] >0 {
            sum[i] = sum[i-1] + S[i];
        else
            sum[i] = S[i];
        return subMax(i-1);
    }
}
```

6. (*20 points*) Given a directed graph G and two vertices s and t of G, find all paths from s to t using backtracking.

A few things to be think about. (1) Store the current path in an array called $path[\cdot]$; (2) what is the stopping condition?; (3) what does "promising" mean?; (4) how do you extend the current path?; (5) what does $path[1], path[2], \cdots$ mean?

Ans:

Here is an algorithm that will output all paths from s to t using backtracking.

First, I define some data structures.

1>. type_of_vertex path[]; // saving path from s to t

2>. type_of_vertex W[][];
// saving each edges' value from vertex
// i to j in w[i][j].
// if there is not a edge between i and j, then w[i][j] == 0;

and, the main function is below,

```
void allPath(i){
  if promising(i){
    if (path[i] == t)
      print path[] through
        1 to i; // output one result
    else{
      for(all vertices x of G
        such that(x≠s && x≠t))
      { set path[i+1] = x;
        allPath(i+1);
      }
    }
  }
}
```

Third, the sub function of promising is below,

```
boolean promising(i){
  int k;
  if((i>1) && (!w[path[i-1]][path[i]]))
    return false;
  for(k=1 to i-1)
    if(path[i] == path[k])
      return false;
  if(i>n)   // Assume there are n
    return false; // vertices in G.
  return true;
}
```

Finally, inital path[] and call main function as below.

```
path[1] = s;
allPath(1);
```

20

10

7. (*10 points*) Use the definition of Ω to show that $\log n! \in \Omega(n \log n)$.

Ans:

1st, Based on the definition of Ω, I will determine that $\log n! \geq c \cdot n \log n$. if $n \geq 1$, $c = 1$, for $\log n! \in \Omega(n \log n)$.

Second, because that,

$\underbrace{\frac{n}{2} \cdot \frac{n}{2} \cdots \frac{n}{2}}_{n \text{ times}} < 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n = n!$

$\Downarrow$

$(\frac{n}{2})^n < n!$

so, $\log (\frac{n}{2})^n < \log n!$

$\Downarrow$

$n \log (\frac{n}{2}) < \log n!$

$\Downarrow$

$n \log n - n \log 2 < \log n!$

Set $b \cdot n \log n = c \cdot (n \cdot \log n - n \cdot \log 2)$, if b and c is constant number

so, $\log n! > b \cdot n \log n$

$\Rightarrow \log n! \in \Omega(n \log n)$

8. (*15 points*) Explain very briefly (1-2 sentences) why under some circumstances $\theta(\log n)$ is the fastest you can search for an element in an array A.

Ans:

Assume there are n items in array A. We need level n times for comparison, Use decision tree to image this situation, we need n node in that tree and depth of tree is the result of search time,

Because $\theta(\log n)$ is the best (smallest) of that tree's depth, So, we could consider $\theta(\log n)$ is the fastest answer for a search action

3

Suppose I store each element x of A in another array B at location $j = x \mod n^2$, where *mod* is the remainder function; e.g. $5 \mod 3 = 2$. (some programming language uses the operator %) Now, to see if a number y is in A, I look at $B[y \mod n^2]$. If y is at this entry of array B, return True; else, return False. On average, searching for y takes only constant time, $\theta(1)$.

Explain what assumption is broken that allows us to search faster than $\theta(\log n)$. (You only need 1 sentence, maximum 2, to answer this).

Ans:

Assumption is that value of each item x is distributing close to linear.

9. (*10 points*) We know that finding a shortest tour in a graph (optimization-TSP) is NP-complete. In this problem, you will use optimization-TSP to solve decision-Hamiltonian circuit. It is very important to remember that Hamiltonian-Circuit takes as input an unweighted graph, returns Yes or No. TSP takes as input a weighted graphs and returns a tour. Specifically:

1. You may assume to have an algorithm called **TSP** that takes as input a *weighted* graph G^*, and **TSP**(G^*) returns a minimum tour p that visits each vertex once (returning to the first vertex). You may also assume to have a function *cost*, where $cost(p')$ computes the cost for any tour p'.

2. Decision-Hamiltonian circuit takes as input a (unweighted) graph G and returns Yes if G has a tour that visits each vertex once (returning to the first vertex); and returns No, if not.

10

Write your transformation/reduction (that uses TSP to solve Hamiltonian circuit) out explicitly in pseudocode below.

Algorithm 2 Decision-HamiltonianCircuit(G)

Ans:

Here I will explain the concept brefly and show some definitions first. and then show the code in details.

step-1:

Set int W[][] to save each edge of that unweighted graph G. of HAM.

And $W[i][j] = \begin{cases} 1 & \text{if edge}\{v_i, v_j\} \text{ is available} \\ 2 & \text{otherwise.} \end{cases}$

Being reason of this setting is that,

if HAM return Yes which means there is a tour that include only and all kind of edges that W[i][j] == 1.

So, we could judge the answer of HAM by whether sum value of all edges is bigger than n, n is the number of vertices in G. If it is, HAM will return No, otherwise return Yes.

Step-2: The actual code is below,

```
Boolean DecisionHamiltonianCircuit(G){
    inital W[][] of G as I have
    defined in step-1;
    // unweighted → weighted.
  p=TSP(G); // because G is weighted now
    if (cost(p) > n) // Assume n is
        return No; // number of
    else
        return Yes; // vertices.
}
```