SKIT, Jaipur

AOA Assignment

Name : Prachi Munot

Roll No: 18ESKIT304

Q1 Recurrence relation is $T(n) = T(n/2) + 1$ where $T(n)$ is time required for binary search in an array of size n.

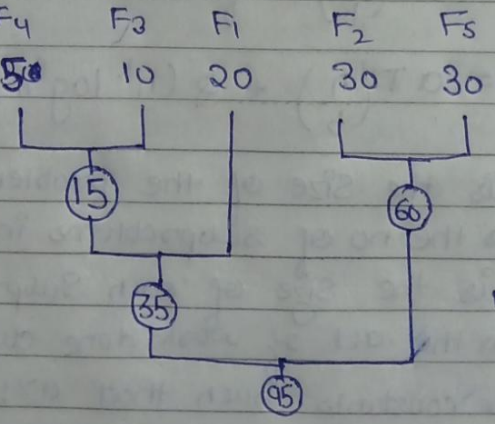
$$T(n) = T\left(\frac{n}{2^K}\right) + 1 + \dots + 1$$

Since $T(1) = 1$ where $n = 2^K$ $T(n) = T(1) + K = 1 + \log_2(n)$

$$\log_2(n) \le 1 + \log_2(n) \le 2\log_2(n) \quad \forall\, n \ge 2$$

$$T(n) = \theta(\log_2(n))$$

Q2 $n = 5$



no of moves needed

95+60+35+15 = 205

Q3 $F_3 < F_2 < F_4 < F_1$

$n\log n < n^{(3/2)} < n^{(\log n)} < 2^n$

Q4 Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller. Dynamic programming is used where we have problems which can be divided into similar sub-problems, so that their result can be used. mostly these are used for optimization.

Q5

Master's theorem

This theorem is also known as Cook Book theorem.

It is used for particular form of recurrence relations

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$= aT\left(\frac{n}{b}\right) + \Theta(n^k \log^p n)$$

where n is the size of the problem

a is the no of subproblems in the recursion

n/b is the size of each subproblem.

$f(n)$ is the act of work done outside the recursive calls.

and a & b are constants such that $a \geq 1$ and $b > 1$

→ $f(n)$ is asymptotically +ve functions.

→ $k \geq 0$ & p is real Number

It states that -

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k \log^p n)$$

Q6
Ans $len[i, j] = len[i-1, j-1] + 1$
if $i, j > 0$ & $a_i = b_j$

Q7
Ans Dynamic programming is the Algorithm design technique. used in chained matrix multiplication algorithm.

Q8
Ans Strassen in 1969 which gives an overview that how we can find the multiplication of two 2*2 dimension. matrix by brute force algorithm
For multiplying the two 2*2 dimension matrices Strassen's used some formula in which there are seven multiplication and eighteen addition, subtraction and in Brute force algorithm there is eight multiplication and four addition. The utility of Strassen's formula is shown by its asymptotic superiority when order n of matrix reaches infinity. Let us consider two matrices A and B having n * n dimension where n is power of 2. It can be observed that we can contain four n/2 * n/2 Submatrices from A, B and their product C. C is resultant matrix of A and B.
Procedure of Strassen matrix multiplication.
→ Divide a matrix of order of 2 * 2 recursively till we get the matrix of 2 * 2

→ Use the previous set of formulas to carry out 2×2 matrix multiplication

→ In this eight multiplication and four additions, subtraction are performed

→ Combine the result of two matrixes to find the final product or final matrix.

Algorithm.

begin

if n = threshold then compute

C = a * b is a Conventional matrix.

Else

Partition a into four sub matrices a11, a12, a21, a22

Partition b into four sub matrices b11, b12, b21, b22

Strassen (n/2, a11 + a22, b11 + b22, d1)

Strassen (n/2, a21 + a22, b11, d2)

Strassen (n/2, a11, b12 - b22, d3)

Strassen (n/2, a22, b21 - b11, d4)

Strassen (n/2, a11 + a12, b22, d5)

Strassen (n/2, a21 - a11, b11 + b22, d6)

Strassen (n/2, a12 - a22, b21 + b22, d7)

C = d1 + d4 - d5 + d7 d3 + d5

d2 + d4 d1 + d3 - d2 - d6

end if

return (C)

end.

9

Ans. The Complexity of an algorithm is a function describing the efficiency of the algorithm in terms of amount of data the algorithm must process.

There are two main Complexity measures of the efficiency of an Algorithm.

* Time Complexity
* Space Complexity

Time Complexity : It is a function describing the amount of time an algorithm takes in terms of amount of the amount of input to the Algorithm. "Time" can mean the number of memory access performed, the number of comparisons between integers, the Number of times some inner loop is executed. or some other Natural unit related to amount of real time the Algorithm will take

Space Complexity : It is a function describing the amount of memory (space) an algorithm takes in terms of amount of input to the algorithm. We use natural but fixed length units to measure this. It sometime ignored because the space used is minimal and obvious but Sometime it becomes as important an issue as time.

10

Ans Total Computing time for Simple matrix multiplication using recurrence.

$$T(n) = \begin{cases} b & \text{for } n < 2 \\ 8T\left(\frac{n}{2}\right) + an^2 & \text{for } n \geq 2. \end{cases}$$

where a = count

Their recurrence can be solved to obtain $T(n) = O(n^3)$ using master method.

Strassen's matrix multiplication algorithm uses only 7 multiplication and 18 additional subtractions the recurrence for this will be

$$T(n) = \begin{cases} b & , n < 2 \\ 7T\left(\frac{n}{2}\right) + 18n^2 & n \geq 2 \end{cases}$$

Solution is $O(n^{\log_2 7}) = O(n^{2.81})$.

11

Sol

length (x) = m = 6

length (y) = n = 10

Two Tables of [0 ... M, 0 ... n] & 0 = [1 ... n ... n]

i \ j	0	e	b	d	s	e	g	h	f	s	a
0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0	0	0	1
b	0	0	1	1	1	1	1	1	1	1	1
d	0	0	1	2	2	2	2	2	2	2	2
g	0	0	1	2	2	2	3	3	3	3	3
f	0	0	1	2	2	2	3	3	4	4	4
s	0	0	1	2	3	3	3	3	4	5	5

i \ j	1	2	3	4	5	6	7	8	9	10
1	↑	↑	↑	↑	↑	↑	↑	↑	↑	↖
2	↑	↖	←	←	←	←	←	←	←	↑
3	↑	↑	↖	←	←	←	←	←	←	←
4	↑	↑	↑	↑	↑	↖	←	←	←	←
5	↑	↑	↑	↑	↑	↑	↑	↖	←	←
6	↑	↑	↑	↖	↖	↑	↑	↑	↖	←

The LCS is of length 3

Start from D[6,10] = ← o/p LCS (D, A, 6, 9)

(D, A, 6, 10)

D[6,9] = ↖ Print a_6 = S, call (D, A, 5, 8)

D[5,8] = ↑ Print a_5 = f, call [D, A, 4, 7)

D[4,7] = ← call (D, A, 4, 6]

D[4,6] = ↖ Print a_4 = g, call (D, A, 3, 5)

D[3,5] = ← call [D, A, 3, 4]

D[3,4] = ← call [D, A, 3, 3)

D[3,3] = ↖ Print a_3 = d, call [D, A, 2, 2)

D[2,2] = ↖ Print a_2 = b, call [D, A, 1, 1)

D[1,1] = ↑ call (D, A, 0, 1) exit.

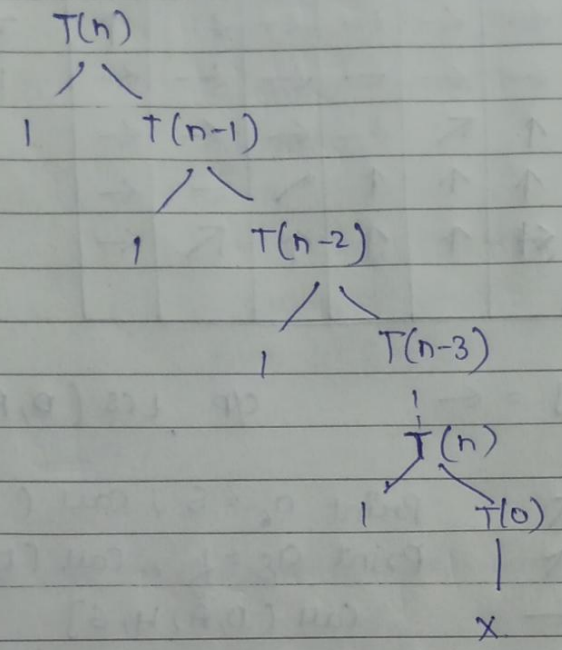
LCS = 'bdgfs'

12

$$T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-1)+1 & n>0 \end{cases}$$

Recurrence Tree method

Tracing Tree



Total time taken
$1+1+1+\ldots 1$ (n times)
$T(n) = \frac{n}{2}(1+1)$
$= O(n)$.

QB

	Parameter	Backtracking	Branch & Bound
1	Approach.	Find all possible Sol^n's available to a problem when made bad choice, undoes the last choice & backs it up.	Solve optimisation problems when has a better optimal Sol^n that the pre-sol leads to it abandons that pre-Sol^n.
2	Searching	It searches the state space tree until it has found a Sol^n for the problem.	It completely searches state space tree to get a optimal Solution.

	Parameter	Backtracking	Branch & bound.
3	Traversal	Traverses the state space tree by DFS.	Traverses the tree, in any manner DFS or BFS.
4	Function	Involves feasibility function	Involves a bounding function.
5	Problems	Solve decision problem	Solve optimization problem
6	efficiency	More efficient	less efficient.
7	Application	N-Queen problem, Sum of subset	Knapsack Problem, Travelling Salesman problem.

Q14.

Soln Jobs to be arranged in increasing order of their profits

	J_7	J_4	J_2	J_6	J_1	J_3	J_5
P_i	10	20	30	40	50	90	
D_i	4	2	6	1	3	5	

0 —J_1— 1 —J_2— 2 —J_3— 3 —J_4— 4 —J_5— 5 —J_6— 6

Jobs	Slot assigned	Soln (seq)	Profit
J_1	[0,1]	J_1	40
J_2	[0,1] [1,2]	J_1 J_2	40+20 = 60
J_3	[0,1] [1,2] [2,3]	J_1 J_2 J_3	60+50 = 110
J_4	[0,1] [1,2] [2,3] [3,4]	J_1 J_2 J_3 J_4	110+10 = 120
J_5	[0,1] [1,2] [2,3] [3,4] [4,5]	J_1 J_2 J_3 J_4 J_5	120+90 = 210
J_6	[0,1] [1,2] [2,3] [3,4] [4,5] [5,6]	J_1 J_2 J_3 J_4 J_5 J_6	210+30 = 240

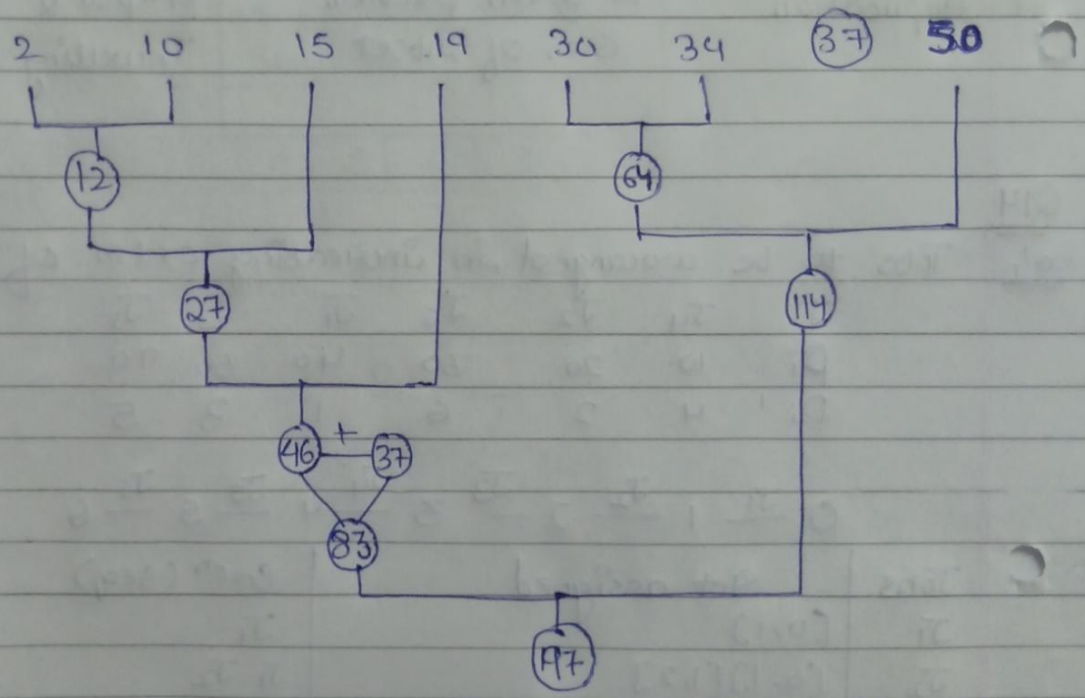
Total Profit = 240

15 For given file, first we will arrange files in increasing order of their size/lengths

$f_8 < f_1 < f_4 < f_2 < f_3 < f_7 < f_5 < f_6$

$2 < 10 < 15 < 19 < 30 < 34 < 37 < 50$

In this sequence we will select minimum two every time to merge



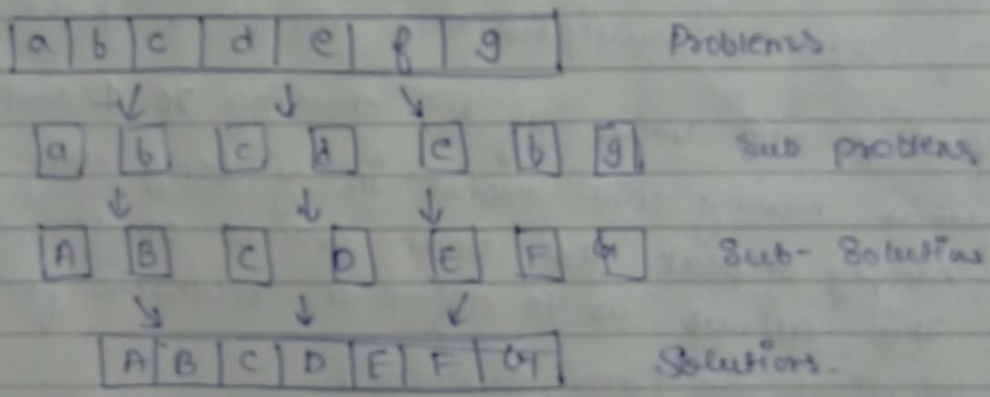
Total operations = $12 + 27 + 46 + 64 + 83 + 114 + 197 = 543$.

Same no of operations can be determined by

$$\sum_{i=1}^{8} (f_i d_i)$$

16 Divide and Conquer Algorithm can be divided into following three parts

1) Divide : This involves dividing the problem into some sub problem
2) Conquer : Sub problem by calling recursively until sub problem solved.
3) Combine : The sub problem solved so that we will get final optimal solution



The following computer algorithm are based on divide and conquer algorithm

1) Merge Sort
2) Quick Sort
3) Binary Search
4) Strassen's Matrix Multiplication
5) closest pair (points.

17 The terminal solution has the complexity of O(K)
The problem can be solved by squaring and multiplying

$p^K = p^x * p^y$, if $x+y = K$
or $p^K = (p^a)^b$, if $a+b = K$

for even value of K, choosing $a = 2$ and $b = K/2$, thus having $p^K = (p^2)^{(K/2)}$ will reduce the no of required multiplications almost in a half.

for an odd value of K, choosing $a = 2$ and $b = K/2$ thus having $p^K = (p^2)^{(K/2)}$ will reduce the no of required multiplications almost in half.

for an odd value of K, choosing ~~x = 1~~ and ~~y =~~ $K-1$ will result in y being even and we can simply repeat the same process as for the even case This allows us to define a recursive function.

```
functions pow (base, exponent)
    if exponent = 0
        return 1
    Else if exponent is even
        return pow (base * base, exponent/2)
    else
        return base * pow (base * base, (exponent - 1)/2)
    End if
```

This solution results in a complexity of O(log K).

Q18 Advantages

1 Insertion Sorts: The main advantage of the insertion Sort is its simplicity. It also exhibits a good performance when dealing with small list

2 Quick Sort: It is in-place since it uses only a small auxiliary stack.
It requires only n(logn) time to Sort n items.
It has an extremely short inner loop.

3 Merge Sort: It is quicker for larger lists because unlike insertion and bubble Sort. it doesn't go through the whole list several times.
It has consistent running time, carries out different bits with similar times in a stage.

4 Heap Sort: like insertion sort, the heap sort algorithm sorts in place.
- This algorithm is simple, fast and stable Sorting algorithm which can be used to sort large sets of data
doesn't require any extra buffer space.
It is useful in Software that requires reliable speed over optimal average runtime and has limited memory. to operate with the data. Thus, Systems with real time requirements and memory constraints benefit most from this algorithm.

Non-fractional Knapsack

19 Knapsack is basically means bag. A bag of given capacity we want to pack item in your luggage

In 0/1 Knapsack problem item cannot be broken into pieces which means theifs should take the item as a whole or leave it. that's why it is called 0/1 knapsack.

- Cannot take a fractional amount of an item taken or take an item more than once.
- It cannot be solved by the Greedy Approach because it is enable to fill the knapsack to capacity. because greedy approach doesn't ensure an optimal solution

Eg

The maximum weight the knapsack can hold is W is 11. There are five items to choose from. their weight and values are presented as.

Weight limit (i) :	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5$ $v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6$ $v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7$ $v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

The [i, j] entry here will be v[i, j] the best value obtainable using the first "i" rows of items if the maximum capacity were j We begin by initialization and first row

$$v[i, j] = \max \{v[i-1, j], v_i + v[i-1, j-w_i]$$

The maximum value of items in a Knapsack is 40 the bottom right entry The dynamic Programming approach can now be coded as following algorithm

Algo

```
1  for w=0 W
2   do V[0,w] ← 0
3   for i=0,n
4   do v[i,0] ← 0
5   for w=0,W
6     do if (wi ≤ w & vi + V[i-1, w-wi] > V[i-1, w])
7   then V[i,w] ← vi + V[i-1, w-wi]
8    else V[i, w] ← V[i-1, w]
```