Sample I/O: Input:

Matrix 1

2	2	3	1
1	4	1	2
2	3	1	1
1	3	1	2

Matrix 2

2	1	2	1
3	1	2	1
3	2	1	1
1	4	3	2

Output:

20	14	14	9
19	15	17	10
17	11	14	8
16	14	15	9

Op verified 28/2/24

DRY RUN

$$A = \left[\begin{array}{cc|cc} 2 & 2 & 3 & 1 \\ 1 & 4 & 1 & 2 \\ \hline 2 & 3 & 1 & 1 \\ 1 & 3 & 1 & 2 \end{array}\right] \quad B = \left[\begin{array}{cc|cc} 2 & 1 & 2 & 1 \\ 3 & 1 & 2 & 1 \\ \hline 3 & 2 & 1 & 1 \\ 1 & 4 & 3 & 2 \end{array}\right]$$

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = (A_{21} + A_{22}) B_{11}$

$R = A_{11} (B_{12} - B_{22})$

$S = A_{22} (B_{21} - B_{11})$

$T = (A_{11} + A_{12}) B_{22}$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (A_{12} - A_{22})(B_{21} + B_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

$$\therefore C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Exp. 5. Strassen Matrix Multiplication

Aim:

To implement and analyse the complexit of strassen Matrix multiplication.

Algorithm:

(i) Divide both matrices A and B into four equal sized submatrices.

(ii) Calculate intermediary matrices using addition and substraction of submatrices.

(iii) Recursively compute the product of these intermediary matrices.

(iv) compute the Final result matrix using these intermediary matrices.

pseudocode:

```
// Here A and B are the input Matrices
   and c is the output matrix and n
   represent the size.

function strassen(A,B){
   if size(A) == 1
      return A*B
   new size = size(A)/2
   A11, A12, A21, A22 = submatrices(A)
   B11, B12, B21, B22 = submatrices(B)
   // Divides the matrices A and B into
    four submatrices

   P1 = strassen(A11, B12 - B22)
   P2 = strassen(A11 + A12, B22)
   P3 = strassen(A21 + A22, B11)
   P4 = strassen(A22, B21 - B11)
   P5 = strassen(A11 + A22, B11 + B22)
   P6 = strassen(A12 - A22, B21 + B22)
   P7 = strassen(A11 - A21, B11 + B12)
```

Complexity Analysis.

)Time complexity:

The strassen multiplication consisit of dividing the matrix into 4 submatrix and performing seven multiplication recursively along with addition and substration. Hence the Reccurence relation becomes.

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

Solving using master theorem we get,

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

$$= O(n^{2.81})$$

(ii) space complexity:

The space complexity is $O(n^2)$ as a new matrix is used to store the result of the multiplication.

However it can be optimized to $O(n^{\log_2 7})$.

// Calculate four submatrices of the result matrix using the products.

$C11 = P5 + P4 - P2 + P6$

$C12 = P1 + P2$

$C21 = P3 + P4$

$C22 = P5 + P1 - P3 + P7$

return combineMatrices (C11, C12, C21, C22)

// combine the four submatrices to get the final result.

Result.

Hence, strassen multiplication was implemented and its complexities were analysed.

Exp. 6.
a)

Finding Minimum and maximum of an array.

Aim: To find the smallest and largest entry in an array using divide and conquer

Algorithm:

1) Get user input for an array with n elements.
2) If the array has 2 items, return the pair of (min,max) using simple arithmetic comparisons.
3) In the case of larger array, calculate the middle index of the array compute the pair say left pair calling recurssion of array from low to mid. compute rightpair for the remaining half.
4) compare the respective entries of leftpair and rightpair and determine the final pair (min,max).

Pseudocode:

```
int* minMaxArray ( int arr[], int low, int high)
    int* minMax = new int[2];
    if (low == high){
        minMax[0] = arr[low];
        minMax[1] = arr[low];
    } else if ( low == high-1){
        if (arr[low] < arr[high]) {
            minMax[0] = arr[low];
            minMax[1] = arr[high];
        } else {
            minMax[0] = arr[high];
            minMax[1] = arr[low];
        }
    }
```

Time complexity. O(n)

$$T(n) = \begin{cases} 2T(n/2)+c & n>2 \\ 1 & n=2 \\ 1 & n=1 \end{cases}$$

after solving we get,

$T(n) = \frac{3}{2}n - 2$

$\approx O(n)$

Using master theorem.

$a=2$
$b=2$
$K=0$
$p=0$

$\log_b(a) = 1 > K$

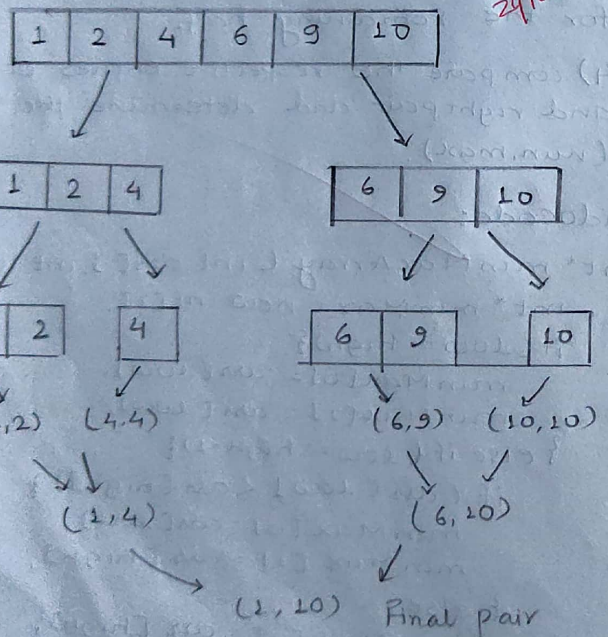
so, $T(n) = \Theta(n^1) = \Theta(n)$

Space complexity O(logn)

I/O: Input: -1 2 4 6 9 10
output: -1 10

o/p verified on 29/2/24

DRY RUN




```
else {
    int max, min;
    int mid = low + (high - low)/2;
    int * leftAns = minMaxArray(arr, low, mid);
    int * rightAns = minMaxArray(arr, mid+1, high);

    if (leftAns[0] < rightAns[0]){
        minMax[0] = leftAns[0]; }
    else {
        minMax[0] = rightAns[0]; }
    if (leftAns[1] > rightAns[1]) {
        minMax[1] = leftAns[1]; }
    else {
        minMax[1] = rightAns[1]; } }

    return minMax; }
```

Result:

Hence the smallest and largest item of an array with n items was found with $\Theta(\log n)$ complexity.

b) Finding Convex Hull problem solution

Aim: For a given set of points, return the co-ordinates of the points that satisfy convex hull criteria.

Algorithm:

1. Get the user input for n points (x,y) pairs
2. Sort the input vectors containing points in ascending order on the basis of their x-axis.
3. Next well divide the points into two halves S1 and S2. The set of S1 contains points to the left of median S2 has the ones to the right of median.
4. Find the convex hulls for the set S1 and S2 let say C1 and C2 respectively.
5. Merge C1 and C2 such that overall convex hull is C. Return C.

Pseudocode.

```
point[] findHullDC(int n, point S[]){
    if (n>1){
    int h = floor(n/2);
    m = n-h;
    point LH[], RH[]; // left and Right hulls
    LH = findHullDC(h, S[1...h]);
    RH = findHullDC(m, S[h+1...n]);
    return mergeHulls(LH.size(), RH.size(),
                                    LH, RH);
    }
     else{
        return S;
     }
}
```

Time complexity Analysis:

Worst case: $O(n \log n)$
Average case: $O(n \log n)$
Best case: $O(n \log n)$
Space complexity: $O(n)$

Recurrence Relation

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

$a = b = 2$
$k = 1$
$p = 0$

$\because \log_b(a) = 1 = 1 = k$

So, $p > -1$

$$O(n^k \log_b^{p+1} n)$$

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

Result:

Hence given a set of points in 2D space, we were able to find the set of convex hull points using aforementioned algorithm.

Exp. 7(a) HUFFMAN CODING

Aim: To form binary codes for characters in a string using variable length encoding.

Algorithm:

Step 1: Start the program

Step 2: Get user input for the characters and their occurence frequency in the ciphertext.

Step 3: create a leaf node for each unique characters and build a min heap of all leaf nodes. Min heap is used as priority queue. Value of frequency field is used to compare two nodes in min heap.

Step 4: Extract 2 nodes with minimum frequency from the min heap.

Step 5: Create new internal node with a frequency same equal to the sum of two node freq. Make the first extracted node as its left child & the other extracted node as its right child.

Step 6: Repeat 3,4,5 until the heap contains only 1 node. Remaining node is the root node and the tree is complete.

Step 7. Traverse the tree formed from the root maintaining an auxiliary array.

Step 8: For each charaters, perform 7 to store the code.

Step 9: End the program.

Sample I/O:

Char arr:	A	B	C	D	E	F
resp Freq:	5	9	12	13	16	45

Output:

~~char | Huffman code~~

~~c 100~~
~~B 100~~
~~D 101~~
~~A 11~~

char	Huffman code
F	0
C	100
D	101
A	1100
B	1101
E	111

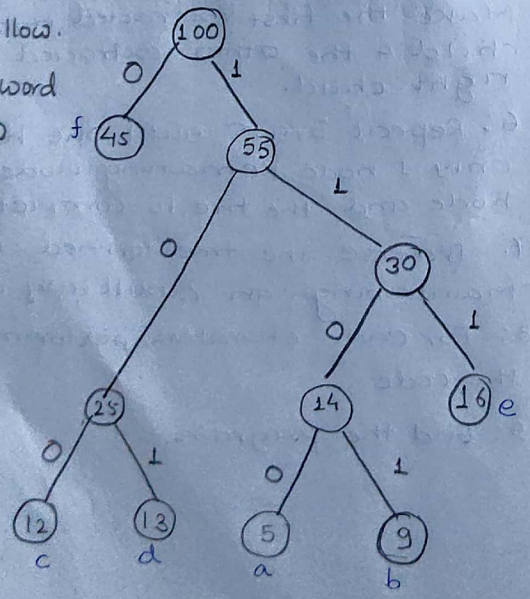
o/p verified Selvam 29/2/24

DRY RUN.

For the above same Input the min heap tree can be constructed as,

codes are as follow.

char	code-word
a	1100
b	1101
c	100
d	101
e	111
f	0



pseudocode:

```
// to build the Huffman Tree
function buildHuffmanTree (characters, frequencies):
    create min-heap // A min heap to store nodes sorted by frequencies
    for (each character, frequency in characters, frequencies) do
        create new_node
        insert new_node into min-heap
    while (min_heap has more than one node) do
        create new_node // a new internal node
        new_node.left_child = node_with_lowest_frequency
        new_node.right_child = node_with_lowest_frequency
        new_node.frequency = sum_of_left_and_right_child_frequencies
        insert new_node into min-heap.
    return the remaining node in min-heap.
}

// to generate Huffman codes from the Huffman Tree.

function generateHuffmanCodes (Huffman_tree_node, current_code) is
    if (huffman_tree_node is a leaf node)
        character = huffman_tree_node.character
        code = current_code
        Store character and code in a dictionary
    else
        generateHuffmanCodes (huffman_tree_node.left_child, current_code+"0")
        generateHuffmanCodes (huffman_tree_node.right_child, current_code+"1")
```

Result:

Hence we can encode the characters using Huffman encoding with optimal time complexity and disambiguity.

Time complexity: O(nlogn)

→ Building Frequency Table:

This step takes O(n) time because each symbol needs to be processed once.

→ Creating the Huffman Tree:

Extracting minimum frequency takes O(Ulogn) since there are n elements.

⇒ O(nlogn)

→ Constructing Huffman codes:

This takes O(n) time since each character is visited once.

$$T(n) = O(n+n\log n+n) = O(n\log n)$$

Space complexity.

The algorithm needs to store frequency table and huffman tree for n elements so space complexity is O(n)

Exp. 7.b. KNAPSACK PROBLEM

AIM: For given max weight capacity, item weight and profit, obtain the combination that yeilds highest profit.

Algorithm;

Step 1: Start the program

Step 2: Create a structure to store each item's wt as well as profit in that weight.

step 3: Obtain user input for max capacity, no of items their wt and respective profits.

step 4: Sort the array of items based on their P/w ratio. Use custom compare function to achieve that in descending order.

step 5: Traverse through the sorted array for each entry perform 6,7

Step 6: If current item's wt is <= remaining capacity, decrement capacity with the weight and increment profit.

Step 7: If not fully accupiable increment profit by $\frac{\text{remaining object}}{\text{item weight}}$ x item profit.

Step 8: Return the final value of profit.

step 9: End the program.

pseudocode:

```
struct Item{
    int profit;
    int weight;
    Item(int p, int w){
        this->profit = p;
        this->weight = w;
    }
};
```

Sample I/O.

n = 7
w = 15

array representing weights

2	3	5	7	1	4	1

array representing profits

10	5	15	7	6	18	3

Output:

The max profit obtainable = 55.3333

DRY RUN

	①	②	③	④	⑤	⑥	⑦
p =	10	5	15	7	6	18	3
w =	2	3	5	7	1	4	1
p/w =	5	1.67	3	1	6	4.5	3

o/p verified

Sorted p/w = 6 5 4.5 3 3 1.67 1

⑤ ① ⑥ ③ ⑦ ② ④

6	5	4.5	3	3	1.67	1
Item 5	Item 1	Item 6	Item 3	Item 7	Item 2	Item 4
15 - 1 = 14	14 - 2 = 12	12 - 4 = 8	8 - 5 = 3	3 - 1 = 2	2 - 2/3 x 3 = 0	not used

Final selection

[1 2/3 1 0 1 1 1]

item 1 ... item n

Constraint satisfaction:-

1 x 2 + 2/3 x 3 + 1 x 5 + 0 + 1 x 1 + 1 x 4 ≤ 1 x 1

= 15 ≤ 15 (True)

profit maximization.

= 1 x 10 + 2/3 x 5 + 1 x 15 + 0 + 1 x 6 + 1 x 18 + 1 x 3

= 55.3333 (Max)

```
bool compare fun( Item a, Item b) {
    double r1 = (double) a.profit / a.weight;
    double r2 = (double) b.profit / b.weight;
    return r1 > r2;
}

double knapsack ( Item arr[], int n, int w){
    sort (arr, arr+n, compare func);
    double val = 0;
    for (int i=0; i<n; i++){
        if (arr[i].weight <= w) {
            w -= arr[i].weight;
            val += arr[i].profit;
        }else {
            val += arr[i].profit * (double)(w/
            break;                          arr[i].weig
        }
    }
    return val;
}
```

Result:

Hence the profit could be maximised in the given constraints using knapsack problem solution in o(nlogn) time.

Time complexity:

→ If the input array is **sorted** based on the P/w ratio. We only iterate through the array once so the time taken is O(n)

→ Sorting takes O(nlogn) so T(n) = O(nlogn) for **unsorted** input array

→ **Space complexity**

Space complexity is O(n) since the space is used to store p/w values.

0/1 Knapsack

Algorithm:

(i) Start the program

(ii) Initialize a base case that checks whether the total no of items and base wt. capacity is zero or not.

(iii) If the weight of the nth item is greater than 'w', then the Nth item cannot be included.

else,

(iv) consider the max of the value of Nth item plus maximum value obtained by remaining N-1 items and remaining weight i.e. wt - wt of nth item and maximum value obtained by N-1 items and w weight. by excluding the Nth item.

(v) stop the program.

Pseudocode.

```
int Knapsack( int w, int wt[], int val[], int n)
{
  if (w==0 || n==0)
      return 0;
  if ( wt[n-1] > wt) {
    return Knapsack(w, wt, val, n-1);
  else
    return max(
        val[n-1] + Knapsack( w - wt[n-1], wt,
                             val, n-1),
        Knapsack( w, wt, val, n-1));
  }
```

RESULT:

Hence 0/1 Knapsack was implemented using recursion method.

Sample I/O

Input:

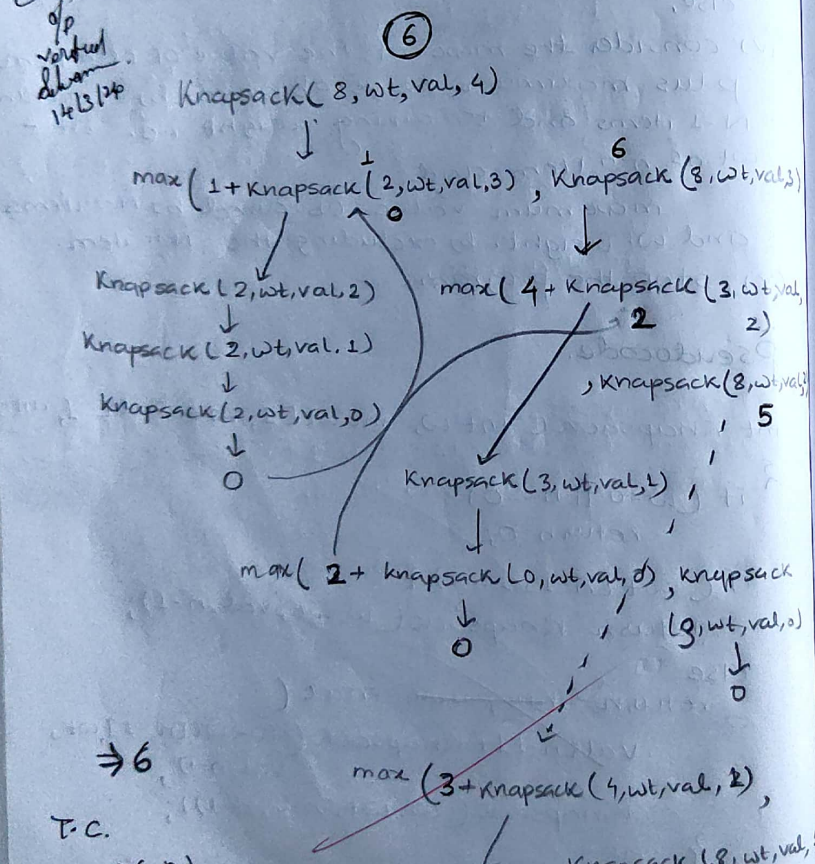
Weight = 3 4 5 6

profit = 2 3 4 1

Capacity = 8

Output:

Maximum profit using 0/1 knapsack is 6



T.C.

$= O(2^n)$

as fxn is called recursively for each item in the knapsack and at each call it makes two recursive call

S.C.

O(n)

Stack required for recursion

Longest Common Subsequence

Aim:

To find the length of the longest common subsequence given two strings S1 and S2. and analyses the time and space complexities

Algorithm:

(i) Start the program

(ii) Check whether one of the string is empty or not if so return 0.

(iii) Check if the last char of the strings are same if so add 1 to the recursive call of the fxn with reduced size in both string.

(iv) else return max of the function called recursively on two strings reduced in size by 1 seperately.

(v) End the program.

pseudocode:

```
int lcs (string X, string Y, int m, int n){
    if(m==0 || n==0) return 0;
    if (X[m-1] == Y[n-1]) {
     return 1+lcs(X,Y,m-1,n-1);}
    else return max( lcs(X, Y, m-1, n),
                     lcs(X, Y, m, n-1));
    }
```

Result:

Hence the length of the longest common subsequence was found using recussion.

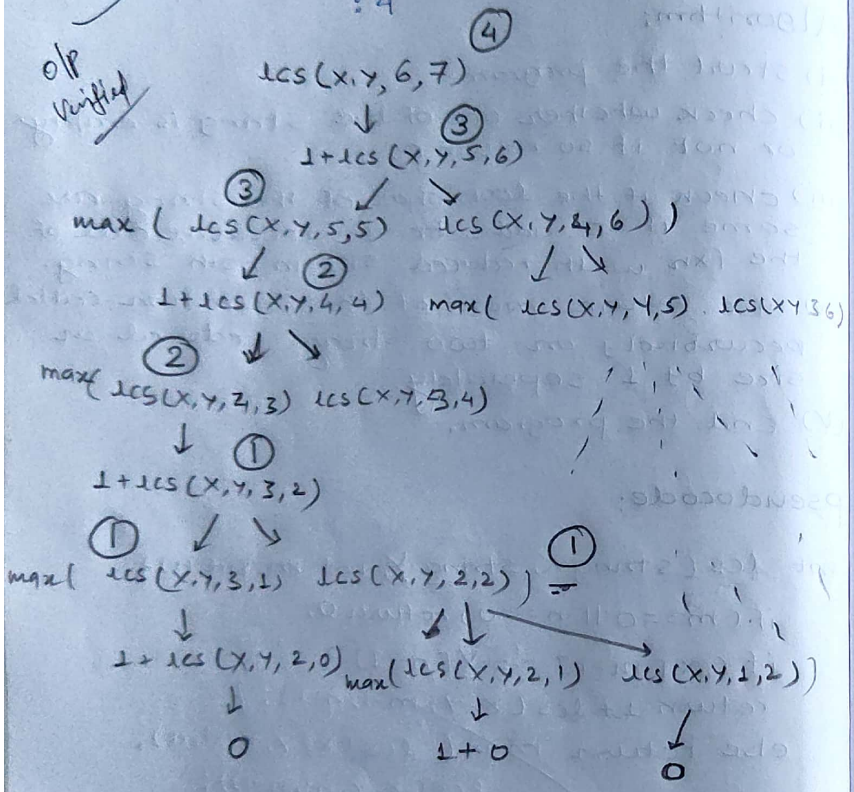
Sample Input/ output

Enter the first string: "aggtab" (0 1 2 3 4 5)

Enter the second string: "gxtxayb" (0 1 2 3 4 5 6)

The length of largest common subsequence is : 4

o/p verified



T.C.

$O(2^{m+n})$, the function explores two recursive call for each pair of char in the worst case scenario.

S.C.

$O(\max(m,n))$ → space used by the recursive calls ⇒ order of max depth of recursive calls.