Page :
Date: / /

NAME - RADHIKA SAINI
SECTION - D
ROLL NUMBER - 31

Tutorial - 7

Sol (1) Greedy algorithm Paradigm :- Greedy is an algorithmic paradigm that builds up a solution piece by piece always choosing the next piece that offers the most obvious and immediate benefits. So the problems where choosing locally optimal also leads to Global solution are best fit for Greedy

→ Greedy algorithms are simply intuitive algorithms used for optimization (either maximized or minimized) problems. This algorithms makes the best choices at every step and attempts to find the optimal way to solve the whole problem.

Sol (2) (i) Activity Selection :-
→ Time Complexity :- $O(n \log n)$ (if input activities may not be stored)
→ $O(n)$ times (when input activities are stored)

(ii) Job sequencing :- → Time Complexity :- $O(n \log(n))$
Space Complexity :- $O(n)$

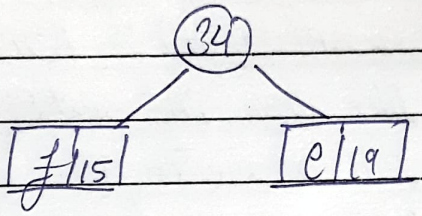
(iii) Fractional Knapsack :- → Time Complexity :- $O(n \log n)$
Space Complexity :- $O(1)$

Page :
Date: / /

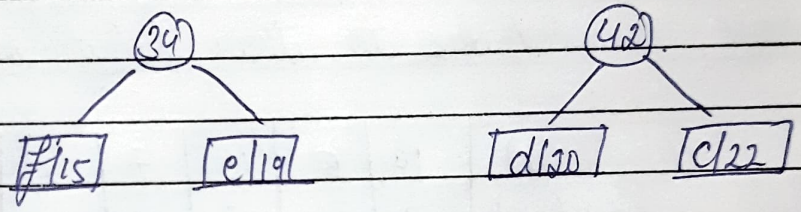
(iv) Huffman Coding :- Time Complexity :- $O(n \log n)$
Space Complexity :- $O(n)$

Sol (3) :- a = 45 c = 22 e = 19
b = 23 d = 20 f = 15

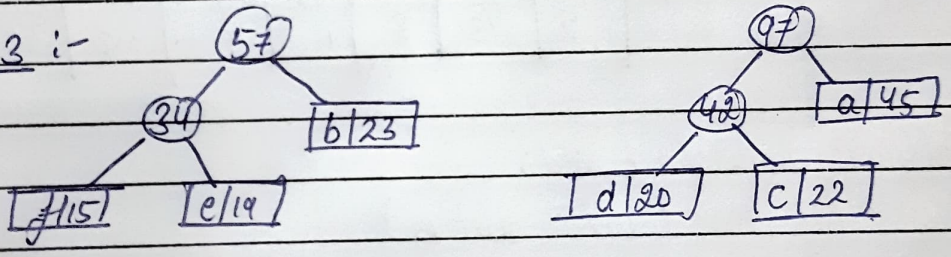
Step 1 :-



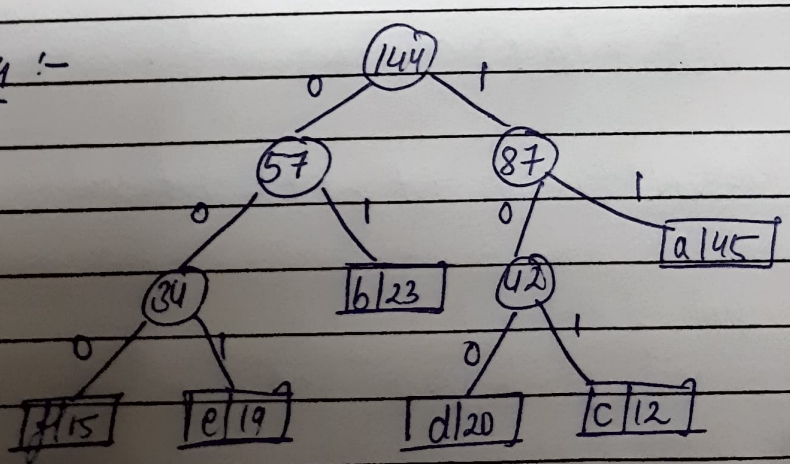
Step 2 :-



Step 3 :-



Step 4 :-



a = 11 c = 101 e = 001
b = 01 d = 100 f = 000

Page :
Date: / /

Sol(4) Priority queue is used for building the Huffman tree such that nodes which the lowest frequency have the highest priority. A Min Heap data structure can be used to implement the functionality of a priority queue.

→ Application of Huffman Encoding :-

- Huffman encoding is widely used in compression formats like GZIP, PKZIP (WinZip) & BZIP2
- Multimedia codes like JPEG, PNG, and MP3 uses Huffman encoding
- Huffman encoding still dominates the compression industry since arithmetic and range coding schemes are avoided due to their patent issues

Sol(5) →

Weight (W)	10	5	15	7	6	18	3
Weight (w)	2	3	5	1	1	4	1
W/w	5	5/3	8	1	6	4.5	3

```
using namespace std;
int max(int a, int b)
{
    return (a>b) ? a:b;
}
int knapsack(int w, int wt[], int val[], int n)
{
    int i, w;
    vector<vector<int>> k(n+1, vector<int>(w+1));
    for (i=0; i<=n; i++)
    {   for(w=0; w<=w; w++)
        {
            if (i==0 || w==0)
                k[i][w]=0;
```

```
        else if (wt[i-1] > j)
            K[i][w] = max [val[i-1] + K[i-1]
                                [w - wt[i-1]], K[i-1]]
        else
            K[i][w] = K[i-1][w];
        }
    }
    return K[n][w];
}
int main()
{
    int val = {10, 5, 15, 7, 6, 18, 3}
    int wt[] = {2, 3, 5, 7, 14, 1}
    int w = 15;
    int n = size of (val) / size of (val[0]);
    Cout << knapsack (w, wt, val, n);
    return 0;
}
```

Ans (6) Greedy choice property :- In greedy algorithm, we may whatever choice seems best at the movement and then solve the subproblems arising after the choice is made. The choice made by a greedy Algorithm may depends on choices so far, but it cannot depend on any future choices or on the solutions to subproblems.

Page :
Date: / /

→ Fractional knapsack

eg:- Robbery

- Want to rob a house and have a knapsack which hold 'W' pounds of stuff.

→ Want to fill the knapsack with the most profitable items

In fractional knapsack:- can take a fraction of an item

Let j be the item with maximum v_i/w_i then there exists an optimal solution in which you take as much of item j as possible

→ Suppose that there exists an optimal solution in you didn't take as much of item j as possible.

→ If the knapsack is not full, add to more of item j and you have a higher value solution

→ we thus assume that knapsack is full

→ There must exist some item $k \neq j$ with $v_k/w_k < v_j/w_j$ that is in the knapsack

→ we can therefore take a piece of k, with ε weight, out of the knapsack and put a piece of j with ε weight in

→ This increases the knapsack value

→ Huffman Coding

Suppose that we have a 100,000 character data file that we wish to store. The file contain only 6 character, appearing with the following freq.

a	b	c	d	e	f
45	13	12	16	9	5

→ we would like to find a binary code that encodes the file using a few bits as possible
→ we can encode using two scheme
  - fixed - length code
  - variable - length code
→ a code will be a set of code

Vol (7)

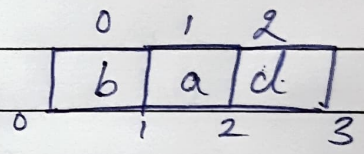
Start time	1	2	0	6	9	10	No of maximum activities = 3
End time	3	5	7	8	11	12	

```
#include <bits/stdc++.h>
using namespace std;
struct Activity
{
    int start, finish;
};
bool activityCompare(Activity s1, Activity s2)
{
    return (s1.finish < s2.finish);
}
void printMaxActivities(Activity arr[], int n)
{
    sort(arr, arr+n, activityCompare);
    cout << "Following activities are selected";
    int i=0;
    cout << "(" << arr[i].start << ", " << arr[i].finish <<
    for(int j=1; j<n; j++)
    {
        if(arr[j].start >= arr[i].finish)
        {
            cout << ", " << arr[j].start << ", " << arr[j];
```

```
        finish << ")";
        i = j;
    }
  }
}
int main()
{
    Activity arr[] = {{1,3}, {2,5}, {0,7}, {6,8}, {9,11}, {10,12}};
    int n = sizeof(arr) / sizeof(arr[0]);
    printMaxActivity(arr, n);
    return 0;
}
```

Sol (8) :-

	Profit	deadline
a	20	2
b	15	2
c	10	1 x
d	5	3
e	1	3 x



total people = 3

Profit = 20+15+5 = 40

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
bool compare(pair<int, int> a, pair<int, int> b)
{
    return a.first > b.first;
}
```

```
int main()
{
    Vector <pair <int, int> job;
    int n, profit, deadline;
    int >>n;
    for (int i=0 ; i<n ; i++)
    {
        Cin >> profit>> deadline;
        job.push_back (make_pair (profit, deadline));
    }
    sort (job.begin(), job.end(), compare);
    int maxEndTime =0;
    for (int i=0 ; i<n ; i++)
    {
        if (job[i].second > maxEndTime)
            maxEndTime = job[i].second ;
    }
    int fill (maxEndTime);
    int count =0, maxProfit =0;
    for (i=0 ; i< maxEndTime ; i++)
    {
        fill[i] = -1 ;
    }
    for (int i=0 ; i<n ; i++)
    {
        int g = job[i].second - 1;
        while (g>=0 && fill[g] != -1)
        {
            g--;
        }
```

```
if (j >= 0 && ful[j] == -1);
    ful[j] = i;
    count++;
    maxProfit += job[i].first;
}
{
cout << count << " " << maxProfit << endl;
}
```

Ans(9) Disadvantages of Greedy Approach

→ It is not suitable for problems where a solution is required for every subproblem the greedy strategy can be wrong in most can even that to a non optimal solution

Eg :- i) Dijkstra's Algorithm fails to find sp for path with negative graphs

ii) We can't break objects in the knapsack problem, the soln that we obtain when using a greedy strategy can be pretty bad to we can always built an input to the problem that makes greedy algo. fails badly.

(iii) We can use greedy approach the problem by always going to the nearest possible city. We select any of the cities as the first one.

(iv) We can build a disposition of cities in a way that the greedy strategy, finds the worst possible solution.

Page :
Date: / /

Ans (16) We can optimize the approach use to solve the job sequencing problem by using priority queue (max. heap)

Algorithm :-

→ Start the job based on their deadlines
→ Iterate from the end and calculate the available slots between every 2 consecutive deadline, include the profit, deadlines and job ID of the job in max heap
→ While the slot are available and there are job left in the max heap include the job ID with maximum profit and deadline in result
→ Sort the result array based on their deadline.