Tutorial - 7

Sol1:- Greedy algorithm parag paradigm : Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious & immediate benefit. So the problems where choosing locally optimal also leads to global solution are best fit for Greedy.

→ Greedy algorithms are simply simple instinctive algorithms used for optimization (either maximized or minimized) problems. This algorithm makes the best choice at every step & attempts to find the optimal way to solve the whole problem.

Sol2:- (i) Activity selection :-

→ time complexity :- $O(n \log n)$ { if inputs activities may not be sorted.

→ $O(n)$ times { when input activities are sorted.

→ space complexity :- $O(1)$ { No extra space is used.

(ii) job sequencing :-

→ time complexity :- $O(n \log(n))$

→ space complexity :- $O(n)$

(iii) Fractional knapsack :-

→ Time complexity :- $O(n \log n)$ { n is size of array.

→ space complexity :- $O(1)$

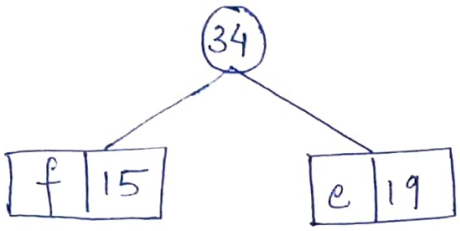
(iv) Huffman coding :-

→ Time complexity : $O(n \log n)$

→ space complexity :- $O(n)$

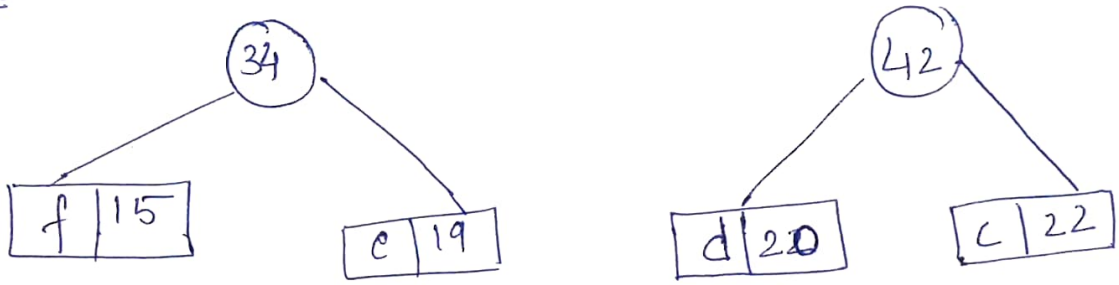
Sol3:- a = 45 c = 22 e = 19
b = 23 d = 20 f = 15

Step 1:-

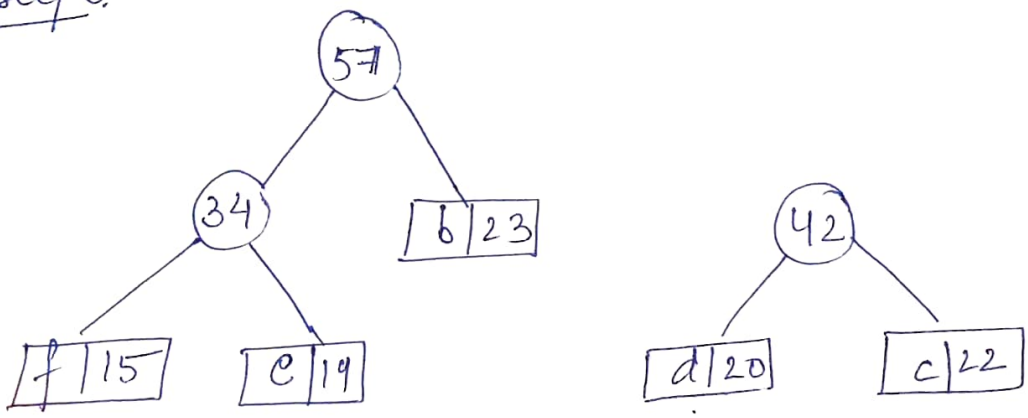


34

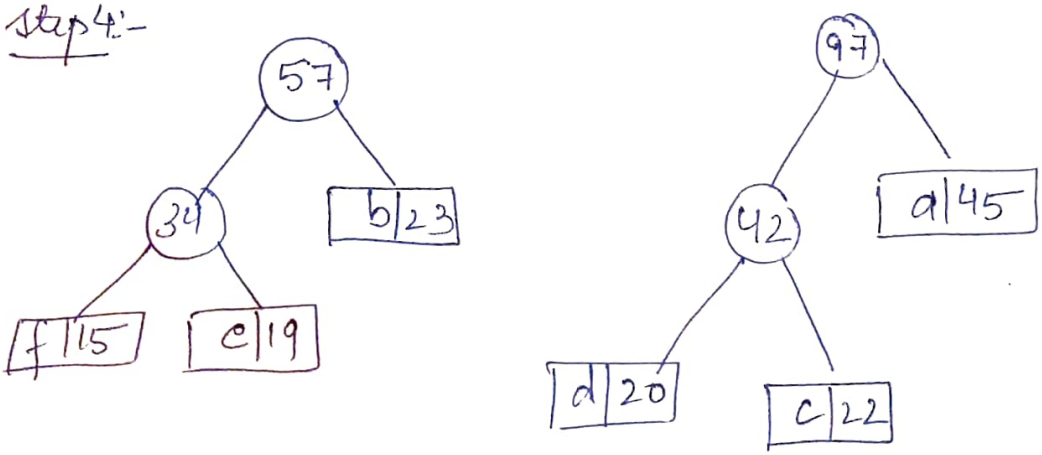
Step 2:-



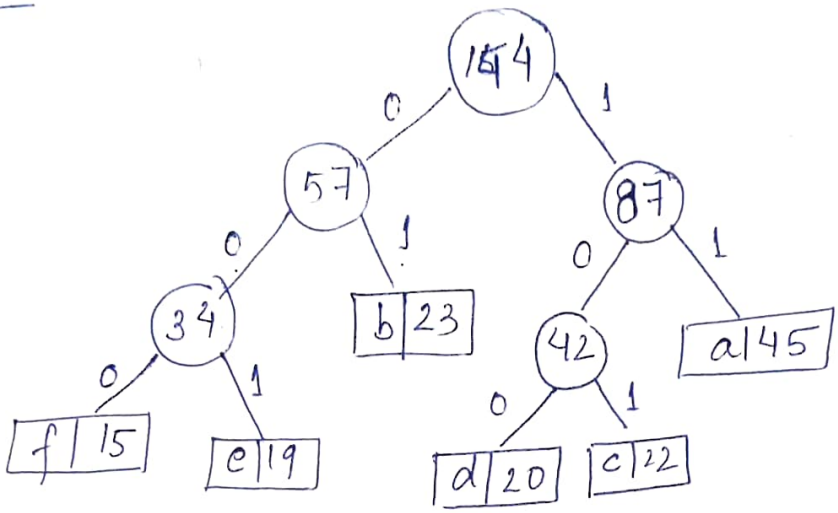
Step 3:-



Step 4:-



Step 5:-



a = 11 c = 101 e = 001

b = 01 d = 100 f = 000.

Sol4:- Priority queue is used for building the Huffman tree such that nodes with the lowest frequency have the highest priority. A Min Heap data structure can be used to implement the functionality of a priority queue.

→ Applications of Huffman Encoding -

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

Q15:-

value (v)	10	5	15	7	6	18	3
weight (w)	2	3	5	7	1	4	1
v/w	5	5/3 = 1.67	3	1	6	4.5	3

k = 15

```
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];
}
```

Tut-7

```
int main()
{
    int val[] = {10, 5, 15, 7, 6, 18, 3}
    int wt[] = {2, 3, 5, 7, 1, 4, 1}
    int W = 15;
    int n = sizeof(val) / sizeof(val[0]);
    cout<< knapsack(W, wt, val, n);
    return 0;
}
```

Q16:- Greedy choice property:- In greedy algorithm, we ma whatever choice seems best at the moment and then solve the subproblems arising after the choice is made The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems.

→ Fractional knapsack

Eg:- Robbery

- want to rob a house & have a knapsack which holds 'B' 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 some 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 $\frac{v_k}{w_k} < \frac{v_j}{w_j}$ that is in the knapsack.

→ we also must have that not all of j is in the knapsack

→ we can therefore take a piece of k, with ϵ weight, out of the knapsack, & 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 contains only 6 characters, 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 as few bits as possible.

→ we can encode using two schemes
- fixed-length code
- variable-length code.

→ a code will be a set of codewords

Ex 17:-

Start time	1	2	0	6	9	10
end time	3	5	7	8	11	12

No. of maximum activities = 3

```
#include <bits/stdc++.h>
using namespace std;

struct Activity
{ int start, finish;
};
```

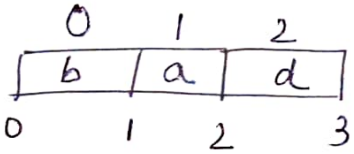
```
bool activityCompare (Activity s1, Activity s2)
{
    return (s1.finish < s2.finish)
}

void PrintMaxActivity (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;
    cin >> 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(int=0; i<maxEndTime; i++)
{
    fill[i] = -1;
}
for(int i=0; i<n; i++)
{
    int j = job[i].second - 1;
    while(j>=0 && fill[j]!=-1)
    {
        j--;
    }
    if(j>=0 && fill[j]==-1);
    {
        fill[j] = i;
        count++;
        maxProfit += job[i].first;
    }
}
cout << count << " " << maxProfit << endl;
}
```

Q19:- 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 worst case even lead to a non-optimal solution.

Eg: (i) Dijkstra's algorithm fails to find or fails too much negative graphs

(ii) We can't break objects in the knapsack problem, the solution that we obtain when using a greedy strategy can be pretty bad too. We can always build an input to the problem that makes greedy algorithm fail badly.

(iii) Another example is the Travelling Salesman Problem: Given a list of cities and the distances between each pair of cities, what is shortest possible route that visits each city exactly once & returns to the origin city?

- We can greedily approach the problem by always going to the nearest possible city. We select any of the cities as the first one & apply that strategy.
- We can build a disposition of the cities in a way that the greedy strategy finds the worst possible ~~tit~~ solution
- We have seen that a greedy strategy would lead us to disaster. But there are problems in which such an approach can approximate the optimal solution quite well.

Q110:- We can optimize the approach used 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 two consecutive deadlines. Include the profit, deadline, & job ID of ith job in the max heap.

→ While the slots are available & there are jobs left in the max heap, include the job ID with maximum profit & deadline in the result.

→ Sort the result array based on their deadlines.