Tutorial 7

Sol 1. Greedy algorithm paradigm: Greedy is an algorithm 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 lead to global solution are best fit for Greedy.

→ Greedy algorithm are simple instincture algorithm used for optimization (either maximized or minimized) problem. This algorithm makes the best choice at every step & 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 sorted }
: $O(n)$ { when input activities are sorted }
space complexity : $O(1)$

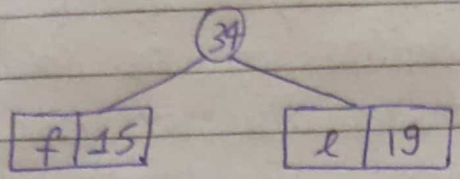
(ii) Job sequencing
time complexity : $O(n \log(n))$
space " : $O(n)$

Fractional knapsack :-
time complexity : $O(n \log n)$
space " : $O(1)$

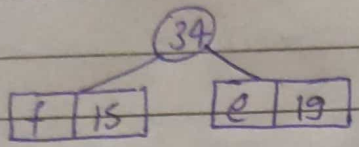
Huffman coding :
time complexity : $O(n \log n)$
space " : $O(n)$

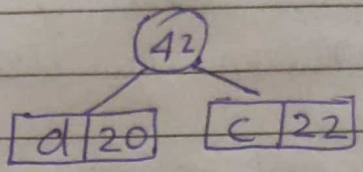
Sol 3. a = 45 c = 22 e = 19 f = 15
b = 23 d = 20

Step 1:

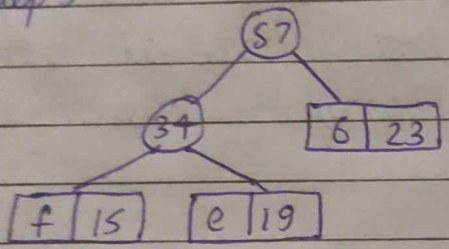


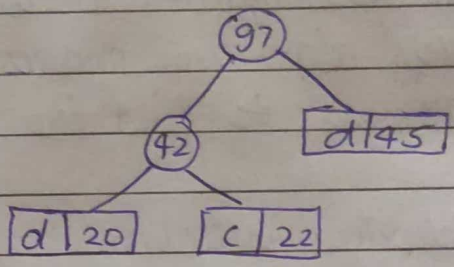
Step 2:



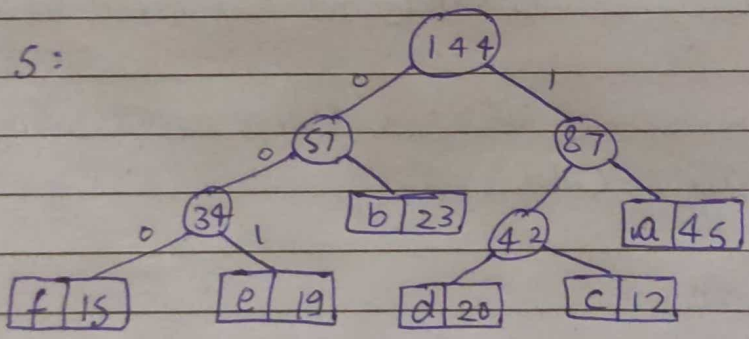


Step 3:





Step 5:



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

Sol 4. 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.

Application of Huffman Encoding:

- Huffman encoding is widely used in compression formats like GZIP, PKZIP & BZIP2.
- Multimedia codecs like JPEG, PNG & 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.

sol:

value (v)	10	5	15	7	6	18	3
weight (w)	2	3	5	7	1	4	1
V/W	5	2/3 = 0.67	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][w].

  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, 1, 4, 1};
    int w = 15;
    int n = size of (val) / size of val[0];
    cout << knapsack (w, wt, val, n);
    return 0;
}
```

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

Fractional knapsack

e.g. Robbery

- want to rob a house & have a knapsack which hold '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 sol^n in which you take as much of item j as possible

→ suppose there exist an optimal sol^n in you didn't take as much of item j as possible

→ if the knapsack is not full, add to me more of item j, & you have a higher value solution.

→ We thus assume that knapsack is full
→ There must exist some item k+j with $\frac{v_k}{w_k} < \frac{v_j}{w_j}$ that is in knapsack
→ We can 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 we have a 100,000 character data file that we wish to store. The file contains 6 characters, with following frequency.

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 2 schemes
→ fixed length code
→ variable length code.

Sol 7.

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

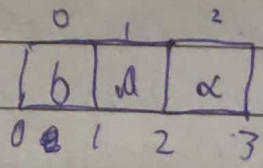
No. of maximum activities = 3

```
#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, activity compare);
cout<<" Following activities are selected";
int i = 0;
cout<<"(" <<arr[i].start<<", "<<arr[i].finish<<");
for(int j=i; 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}, {9,7}, {6,8}, {9,11} {10,12}}
    int n = sizeof(arr)/sizeof arr[0];
    PrintMaxActivity(arr, n);
    return 0;
}
```

Sol8.

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



total people = 3

profit = 20+15+5 = 40

```
# includ <iostream>
# include <vector>
# includ <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 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; 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;
}
```

Sol 9. Disadvantage of greedy approach

- It is not suitable for problem where a solution is required for every subproblem the greedy strategy can be wrong. in worst case even lead to a non-optimal sol^n.

E.g (i) Dijstra's algorithm fails to find as fails with negative graphs

(ii) we can't break objects in the knapsack problem, the sol^n that we obtain when using a greedy strategy can be pretty bad tho we can always build an input to the problem that makes greedy algo fail 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 sol^n.

Sol 10. 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.
- Start from the end & calculate the available time b/w every 2 consecutive deadline, include the profit, deadline & job ID of the job in max heap

→ While the slot are available & there are job left in the max heap, include the job ID with maximum profit & deadline in result.
→ Sort the result array based on their deadline.