Tutorial - 7

Date: / /

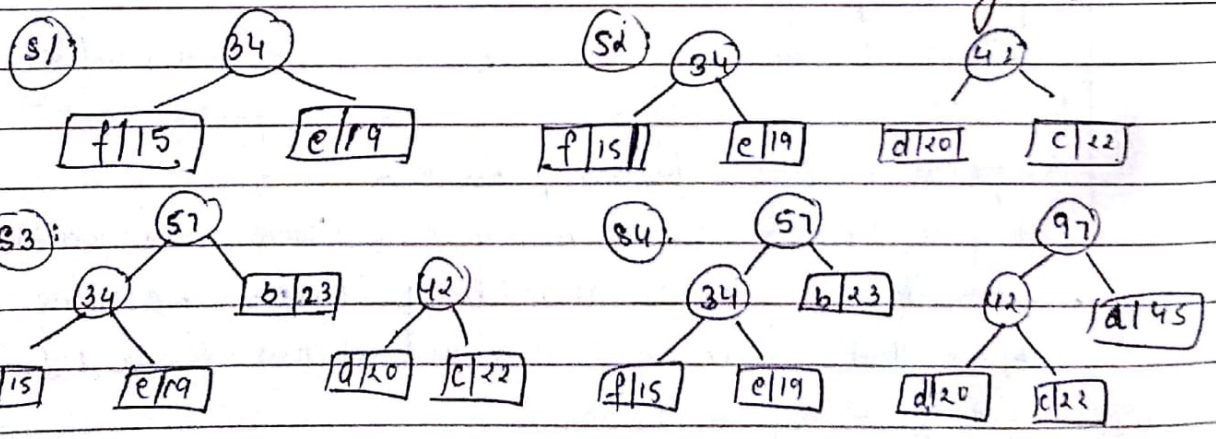
1- Greedy algorithm paradigm: Greedy is an algo. paradigm that builds up a solution piece by piece always choosing the next piece that offers the most obvious & immediate benefit. So the problems where choosing locally optimal also leads to global solution are best fit for Greedy.

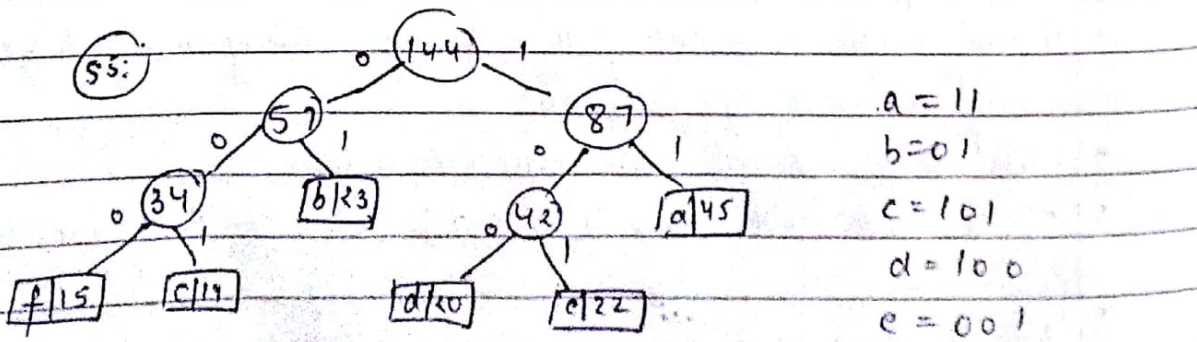
→ These are simple simple intuitive algo's used for optimization (either maximized or minimized) problems. This algo. makes the best choice at every step & attempts to find the optimal way to solve the whole problem.

2-

	Time complexity	Space Complexity
Activity selection	O(n log n) if input activities may not be sorted O(n) when i/p is sorted	O(1) constant. No extra space is used.
Job sequencing	O(n log(n))	O(n)
fractional knapsack	O(n log n)	O(1)
Huffman coding	O(n log n)	O(n)

3- a = 45 b = 22 c = 22 d = 20 e = 19 f = 15





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

Date: / /

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.

Applications 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 newer arithmetic and range coding schemes are avoided due to their patent issues.

6- 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 it 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 roba house and have a knapsack which holds '3' pounds of stuff. Want to fill the knapsack with the most profitable items. 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 items j, as possible
- Suppose that there exists an optimal solution if you didn't take as much of item j as possible.
- If the knapsack is not full, add some more of item j, & you have a higher value of solution.
- We thus assume that knapsack is full.
- There must exist some item $k \neq j$ with $\frac{v_k}{w_k} < \frac{v_j}{w_j}$ i.e, 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

Date: / /

a piece of j with E 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 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 scheme ① fixed-length code ② variable-length code.
- A code will be a set of code words.

7-

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

No. of max. activities = 3

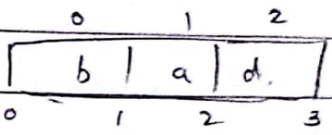
```
#include <bits/stdc++.h>
using namespace std;
Struct Activity {
    int start, finish;
};
bool activityCompare(Activity s1, Activity s2){
    return(s1.finish<s2.finish);}
void printMaxActivity(Activity arr[], int n){
    sort(arr, arr+n, activityCompare);
    cout<<"Following activities are selected";
    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(){
```

Date: / /

Activity ar[] = { {1,3}, {2,5}, {0,7} {6,8}, {9,11}, {10,12} };
int n = sizeof(ar) / sizeof(ar[0]);
PrintmaxActivity(ar, n);
return 0;
}

8-

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



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];
```

Date: / /

```
int count = 0, maxProfit = 0;
for(int i = 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;
```

9- Disadvantage of greedy approach :

It is not suitable for problems where a solution is required for every sub problem the greedy strategy can be wrong, in worst case even lead to a non-optimal solution.

Eg: (i) Dijikstra's algorithm facts to find or fails with negative graphs

(ii) We can't breaks 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 algo. fail badly

(iii) Another example is the travelling salesman problem. Given a list of cities & the distance btw 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 solution.
- We have seen that a greedy strategy could lead us to disaster. But there are problem in which such an approach can approx. the optimal solution quite well.

Date: / /

10-

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 b/w every 2 consecutive deadlines. Include the profit, deadline & jobID 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.