

TUTORIAL → 7

① Greedy algorithm is an algo paradigm that builds up a solution piece by piece always choosing the next piece that offers the most obvious & immediate benefit. Some problem where choosing totally optimal also leads to global solution are best fit for greedy.

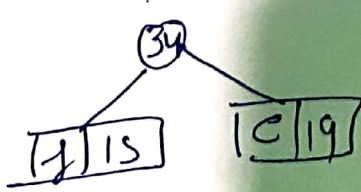
These are simple ~~Intensive~~ algo used for optimization either maximized or minimized problems. This algo makes best choice at every step & attempts to find ~~the~~ the optimal way to solve the whole problem.

② Activity selection	Time complexity	Space complexity
	$O(nd \log n)$ if input activities may not be sorted $O(n)$ when if p is sorted	$O(1)$ Constant. No extra space is used
Job sequencing	$O(n \log(n))$	$O(n)$
Fractional knapsack	$O(n \log n)$	$O(1)$ $O(1)$
Huffman coding	$O(n \log n)$	$O(n)$

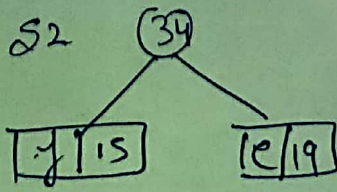
$a=45 \quad b=23 \quad c=22 \quad d=20 \quad e=19 \quad f=15$

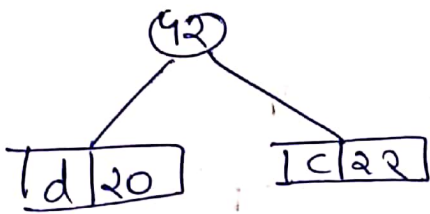
③

S1:

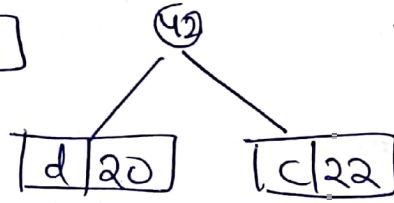
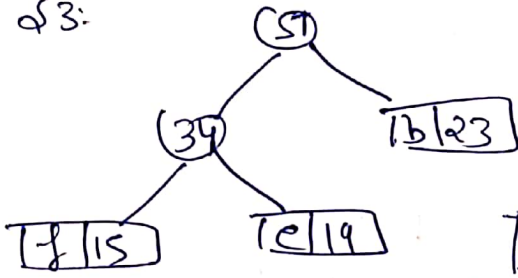


S2

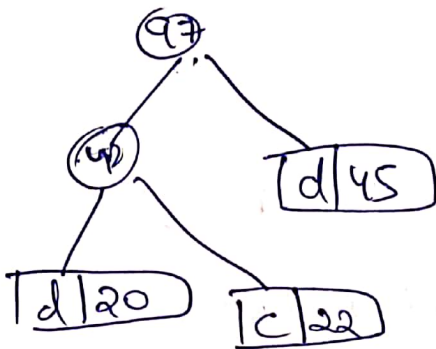
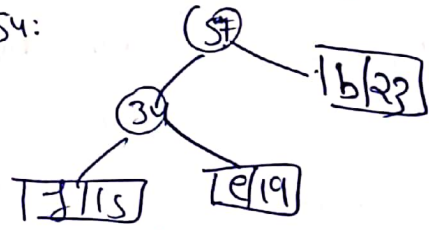




Q3:

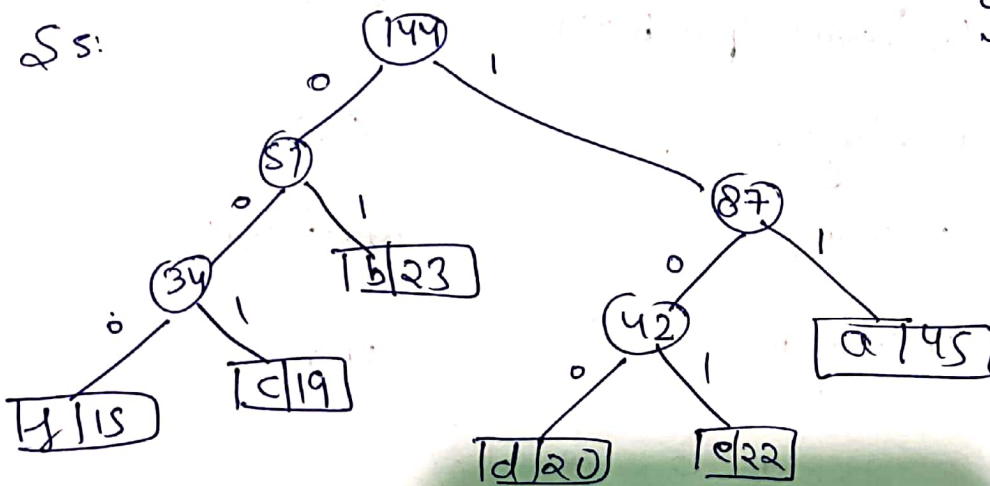


S4:



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

S5:



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, RZIP (Winzip)
- Multimedia codes like JPEG, PNG and MP3 uses Huffman encoding.
- Huffman encoding still dominates the compression ~~as~~ since newer arithmetic and range coding schemes are avoided ~~as~~

⑥ Greedy choice property:

In Greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblem using after the choice is made. The choice made by it may depend on choices so far, but it cannot depend on any further choices or on the solutions to subproblems.

→ Fractional Knapsack

Eg. Robbery

want to rob a house and have a knapsack which holds B 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 item j as possible.

• Suppose that there exists an optimal solution. 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 higher value of solution.

• We thus assume that knapsack is full.

• There must exist some $k \neq j$ 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 weight w_j out of the knapsack and put a piece of j with weight w_k in.

• This increases the knapsack value.

→ Huffman encoding

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 2 schemes: (1) fixed length code

(2) variable length code

• A code will be set of code words.

⑦

Start time	1	2	6	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() {
    Activity arr[] = { {1,3}, {2,5}, {6,7}, {6,8}, {9,11}, {10,12} };
```

```
int n = sizeof(arr)/sizeof(arr[0]);
```

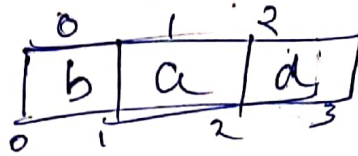
```
printMaxActivity(arr, n);
```

```
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 j = maxendtime;
int count = 0, maxprofit = 0;
```



```
for (int i=0; i < maxEndtime; i++)
```

```
    job[i] = -1;
```

```
for (int i=0; i < n; i++)
```

```
    int j = job[i].second - 1;
```

```
    while (j >= 0 && job[j] != -1)
```

```
        j--;
```

```
    if (j >= 0 && job[j] == -1)
```

```
    { job[j] = i;
```

```
      count++;
```

```
      maxProfit += job[i].first;
```

```
    }
```

```
    cout << count << " " << maxProfit << endl;
```

```
}
```

① 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. 0/1 Dijkstra's ~~algorithm~~ ^{algorithm} fails to find or fail with negative graphs.

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

(11) Another eg. is the travelling salesman problem. Given a list of cities & the distance b/w each pair of ~~city~~ cities, what is the 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 cities as the first one and apply that strategy.

- we built 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.

(10) We can optimize the approach used to solve the job sequencing problem by using priority queue (max. heap)

Algorithm:

- Sort the job based on their deadline.
- Iterate from the end and calculate the available slots b/w every 2 consecutive deadlines. Include the profit, deadline & jobID of i th job in the max heap.

- ~~is the~~ while the slot are available & there are job left in the max heap include the jobID with max. profit & deadline in result.
- Sort the result array based on their deadlines.