Greedy

→ Greedy Algorithms are mainly used to optimization Problem.

08 Optimization Problem means maximizing (or) minimizing something.

eg shortest path Problem, longest path Problem.

09

→ Greedy is an algorithmic paradigm that builds up a solution piece by piece, always

10 choosing the next piece that offers that most obvious and immediate benefit.
So the problem where choosing locally optimal also leads to the global optimal

11 solution are best fit for greedy.

12 → At every step, we can make a choice that looks best at that moment, and we get the optimal solution of the complete problem.

01

eg Consider infinite supply of the following value coins 10 5 2 1.

02 If someone asks for an amount, how will you give this amount using minimum coins?

Implementation

```
def minCoins(coins, amount):
    coins.sort(reverse=True)
04  res = 0
    for i in coins:
05      if i <= amount:
            c = amount//i
06          res += c
            amount -= c*i
07      if amount == 0:
            break
    return res
```

amount = 57
coins = [5, 10, 2, 1]
After sorting
coins = [10, 5, 2, 1]
res = 0
n = 10:
c = 5, res = 5, amount = 7
n = 5:
c = 1, res = 6, amount = 2
n = 2:
c = 1, res = 7, amount = 0

→ Greedy algorithm may not work always.
Consider coins [] = {18, 1, 10}, amount = 20.
greedy will give - 18, 1, 1, but better approach is 10, 10
3 coins 2 coins

OCTOBER
22
TUESDAY
'24
43rd Week
296-070

S	M	T	W	T	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

OCT 2024

→ General structure of greedy Algorithms

```
def getOptimal(coin):        → sorting a/q to desired output.
    res = 0
    while (all items are not considered):
        i = selectANItem()    // taking current coin
        if (feasible(i)):
            res = res + i
    return res
```

Applications

(a) Finding optimal solution.

- Activity selection
- Fractional knapsack
- Job sequencing
- Huffman coding
- Prim's Algorithm.
- Kruskal's Algorithm.
- Dijkstra's Algorithm.

(b) Finding close to optimal solutions for NP Hard problem like Travelling Salesman's Problem.

Activity selection problem

→ We are given a set of activities, Every activities is represented as a pair.
→ First element of pair is starttime. The second element of pair is endtime.
→ We are also given a single machine which can do only one task at a time. machine
→ We need to maximize the number of activities that we can perform on a single machine.

i/p = {(2,3), (1,4), (5,8), (6,10)}
o/p = 2

i/p = {(1,3), (2,4), (3,8), (10,11)}
o/p = 3

→ No two activities can overlap eachother.

Steps

1. Sort the activities according to their finishing time.
2. Select the first activity from the sorted array and print it.
3. Do the following for remaining activities. [initialize]
   a. If current activity overlaps with the last picked activity in the solution, ignore the current activity.
   b. Else add the current activity to the solution.

Implementation

```
def activities(arr):
    n = len(arr)
    arr.sort(key = lambda x: x[1])
    Prev = 0
    res = 1
    for curr in range(1,n):
        if arr[curr][0] >= arr[Prev][1]:
            res += 1
            Prev = curr
    return res
```

TC = O(nlogn) → If input is not sorted.
O(n) → If input is sorted.
Auxilary Space → O(1)

Fractional Knapsack Problem

→ Given the weight and value of n items, put these items in a knapsack (bag) of capacity W to get the maximum total value in the knapsack.
↔ In knapsack (fractional) we can break items for maximizing the total value of knapsack.

i/p:

	I_1	I_2	I_3
weight	50	20	30
value	600	500	400

Knapsack capicity = 70

o/p: 1140

$I_3 + I_2 + \frac{I_1 \times 20}{50}$

i/p:

	I_1	I_2	I_3
weight	10	5	20
value	200	50	100

Knapsack capacity : 15

o/p = 250

OCTOBER
24 '24
43rd Week
298-068
THURSDAY

S	M	T	W	T	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

OCT 2024

Follow the given steps to solve the Problem using the above approach:

- Calculate the ratio (Value/weight) for each item.
- Sort all the item in decreasing order of the ratio.
- Initialize res=0 , curr_cap = given_cap.
- Do the following for every item 'i' in sorted order.
- If the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result.

```
else if (I.weight <= curr_cap):
        curr_cap -= I.weight
        res += I.value
```

- else add the current item as much as we can and break out of loop.

```
return res.
```

Implementation

```
def knapsack (W, arr):
    arr = sort (key = lambda x: (x.value / x.weight), reverse = True)
    res = 0.0
    for item in arr:
        if item.weight <= w:
            w -= item.weight
            res += item.value
        else:
            res += item.value * w / item.weight
            break
    return res
```

{ TC = O(nlogn) , SC = O(N) }

It is as if the money tied up in the small bag, getting snatched away

Job sequencing with deadline.

Given an array of Jobs where every Job has a deadline and associated profit if the Job is finished before the deadline. It is also given that every Job takes a single unit of time, so the minimum possible deadline for any Job is 1.

Maximize the total profit if only one Job can be scheduled at a time.

Rules

(i) One unit Per Job.

(ii) Only one Job can be assigned at time.

(iii) Time start with zero.

Steps

- Sort all Jobs in decreasing order of Profit.
- Iterate on Jobs in decreasing order of Profit. For each Jobs do these
  - find a time slot i, such that slot is empty and i < deadline and i is the greatest. Put the Job in this slot and mark this slot filled
  - If no such i exist, then ignore the Job.

```
def JobSequencing(arr, t):
    n = len(arr)
    for i in range(n):
        for j in range(n-1-i):
            if arr[j][2] < arr[j+1][2]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    result = [False] * t
    job = ['-1'] * t
    for i in range(len(arr)):
        for j in range(min(t-1, arr[i][1]-1), -1, -1):
            if result[j] is False: result[j] = True
                job[j] = arr[i][0] break
    print(job)
```

{ TC = O(n²) ; SC = O(n) }

OCTOBER
26
'24
43rd Week
300-066
SATURDAY

S	M	T	W	T	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

OCT 2024

Huffman Coding

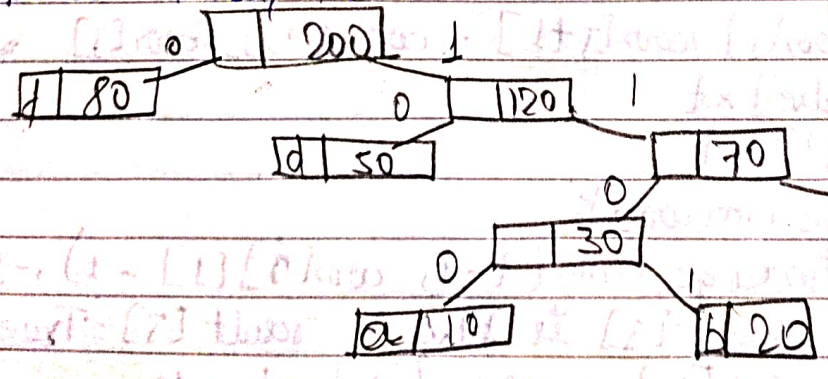
→ Huffman coding is a lossless data compression algorithm.
→ The idea is to assign variable-length codes to input characters, length of the assigned codes are based on the frequencies of corresponing characters.
→ The most frequent character gets the smallest code and the least frequent character gets the largest code.
→ The variable length codes assigned to input characters are Prefix codes, means the codes (bit sequence) are assigned in such a way that the code assigned to one character is not the prefix of code to any other character. This is how huffman coding makes sure that there is not ambiguity when decoding the generated bit stream.

There are mainly 2 major parts in huffman coding:—
1. Build a huffman Tree from input characters.
2. Traverse the huffman Tree and assign codes to characters.

Steps to build huffman tree:—
1. Create a leaf node for each unique character and build a min heap of all leaf nodes. [Every input character is a leaf]
2. Extract two nodes with the minimum frequency from the min heap. [2 minimum]
3. Every left child edge is labelled as 0 and right edge as 1.
4. Every root to leaf path represent huffman code of the leaf.

1. Build a binary tree

i/p: ['a', 'd', 'b', 'e', 'f']
[10, 50, 20, 40, 80]



(2) Traverse the binary tree and print the codes

08 for 0 a = 1100
d = 10 b = 1101
09 a = 1100 e = 111

10 Time complexity would be o(nlogn) [See Code in chatGPT]

11 Applications

12 (1) They are used for transmitting fax & text.
(2) " " " " Conventional compression format like PKZIP, GZIP...
01 (3) multimedia codes like JPEGs, PNGs, MP3 use huffman encoding.

02

03

04

05

06

07