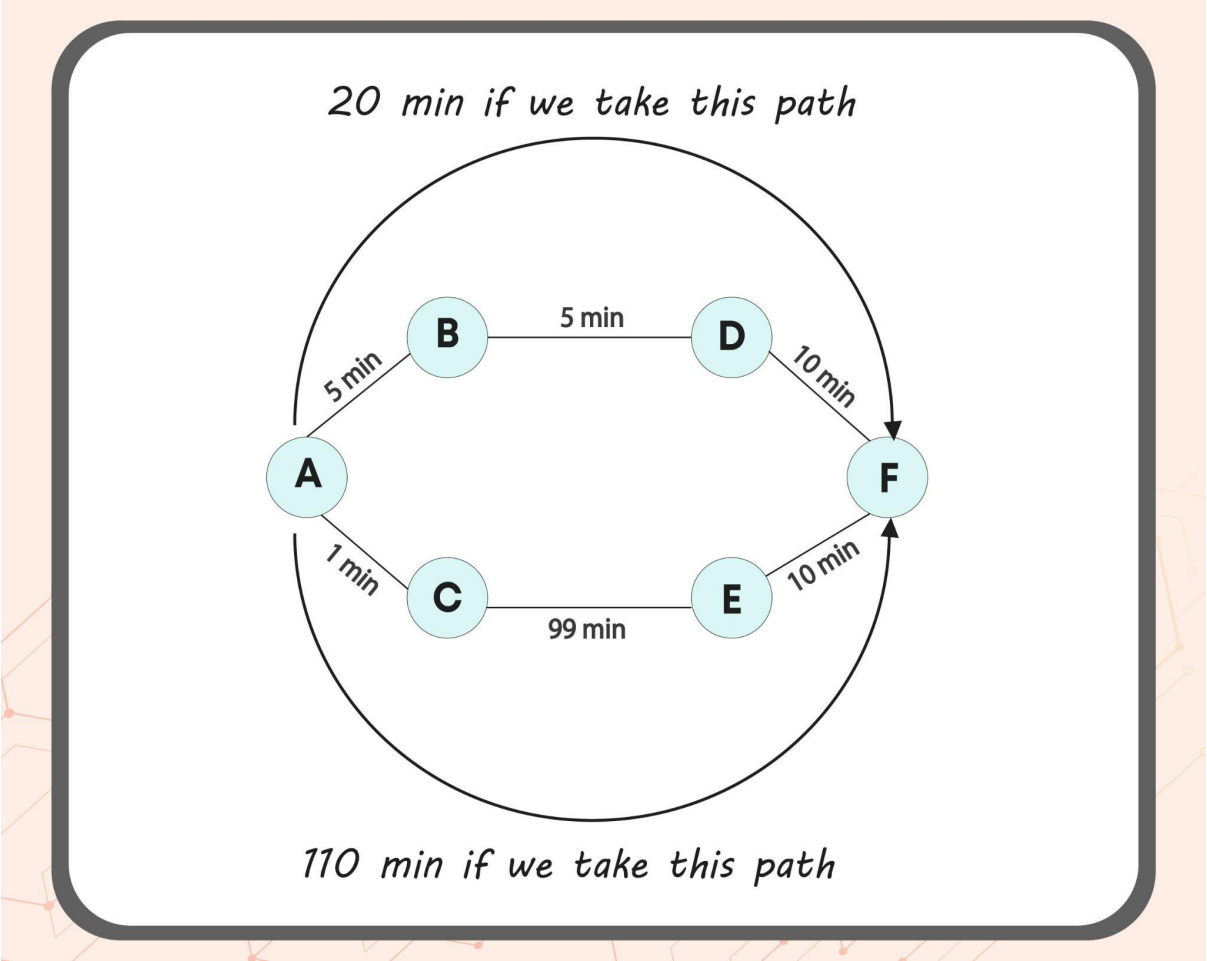
Greedy Problems

Introduction

Suppose we have 6 places A, B, C, D, E, F and we are given the time to reach from one destination to another, and we have to tell the shortest time required to reach from A to F.



Now, if we take the path A → B → D → F that will be our DP solution since we know that it is the overall shorter path, but if we take path A → C → E → F, then it will be greedy approach since we only know and judge our decision on the basis of path A → B and path A → C.

Greedy solution is not giving us the right answer in this problem but it might be a useful approach in others.

Activity Selection

Problem Statement: We are given the starting time and the finishing time of activities and we have to tell how many activities can be completed in a given time interval. We can not do more than one activity simultaneously.

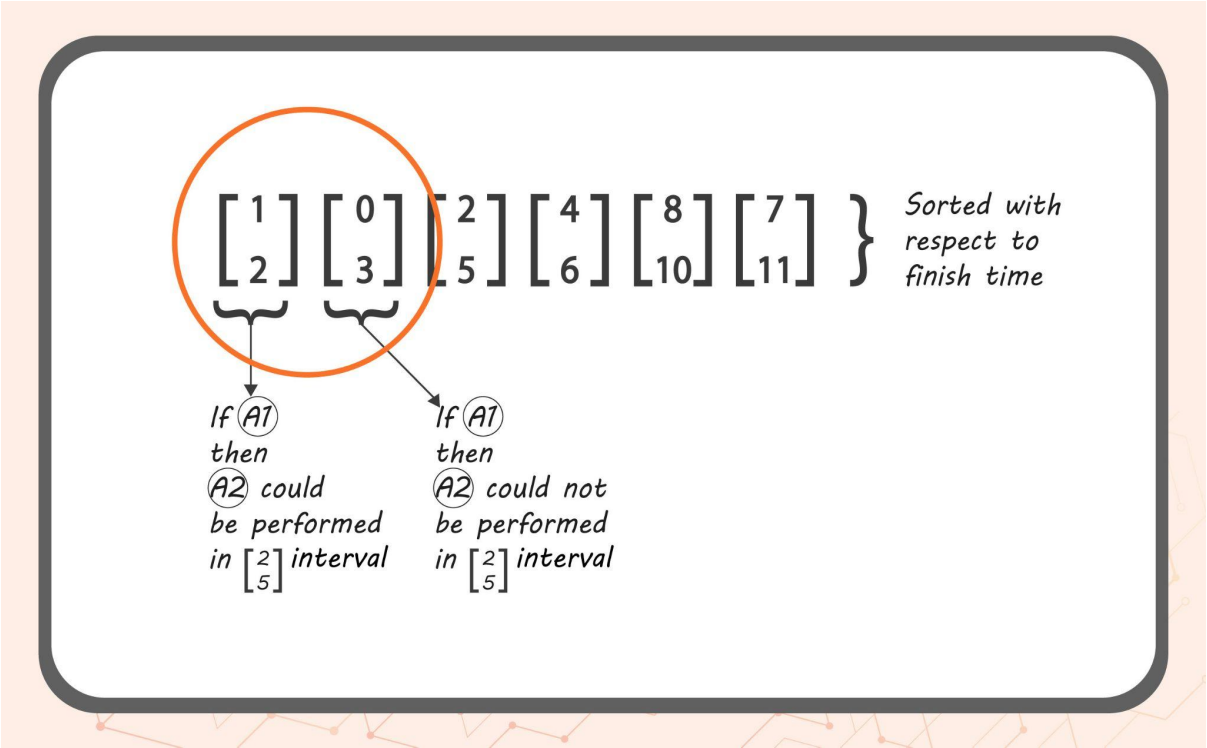
Example:

Given time interval to perform the activities: **0 to 11**.

Output: **3** (because the first activity will be performed in the time interval **1 to 2**, the second activity will be performed in the time interval **2 to 5** and the third activity will be performed in interval **8 to 10**.

	A1	A2	A3	A4	A5	A6
1 Start	[0	1	2	4	7	8]
A finish	[3	2	5	6	11	10]

The greedy approach to solving this problem starts by thinking to finish an activity as soon as possible and start with a new one. So we sort these two arrays on the basis of their finishing time or rather than saying that we sort the two arrays, we can say that we make a structure that has both the start time and finish time and we sort that on the basis of the finish time.



Now, A1 could be finished early if we do it in [1, 2] time interval which leaves us the option to start A2 in [2, 5] time interval.

So we say that greedy technique is applied when we start from an optimal answer from the first step and continue to look for the optimal answers in the next steps also.

The code to this solution is left for the reader as an exercise.

Minimum Absolute Difference in Array

Problem Statement: We are given an array and we have to tell the minimum absolute difference between two elements.

Explanation:

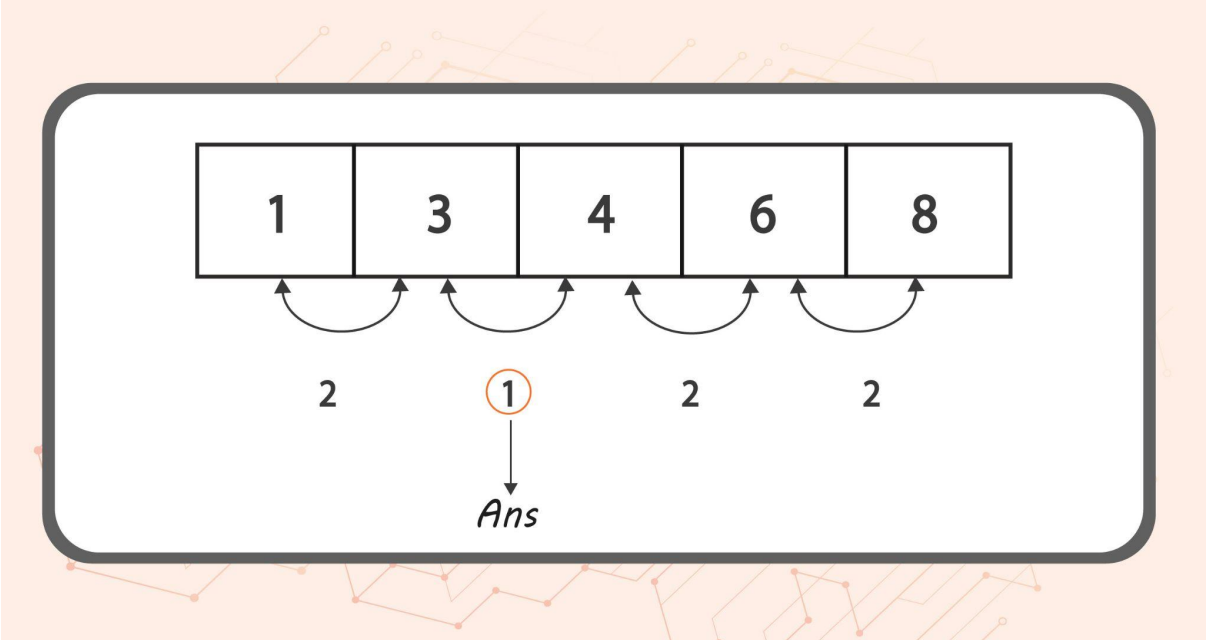
For example, **Arr = [1, 3, 2, 5, 4, 11, 7]**. Now, the difference between 1 and 3 is 2 and the difference between 3 and 2 is 1. So in this example the minimum absolute difference is 1.

Another example is, **Arr = [1, 4, 6, 3, 8]** then also the answer will be 1 i.e., the difference between 3 and 4.

Basic Approach: The basic approach will take $O(N^2)$ time as we will have to generate all the differences of elements by traversing using two loops.

Greedy Approach: If we sort this array then we can think of the greedy technique. After sorting, the array becomes **Arr = [1, 3, 4, 6, 8]**. Now if we start traversing the elements

we need not use two loops because we just have to check the minimum difference in the consecutive elements only.



The code to this solution is left for the reader as an exercise.

Fractional Knapsack

Problem Statement: This problem is just like the original knapsack problem in which we were given two arrays: one with weights of the items and the other with the values of these items and we were given a maximum weight that we can put in the knapsack. Now in this we can take the fractional weight of an item by adding only a fraction of the item.

Explanation:

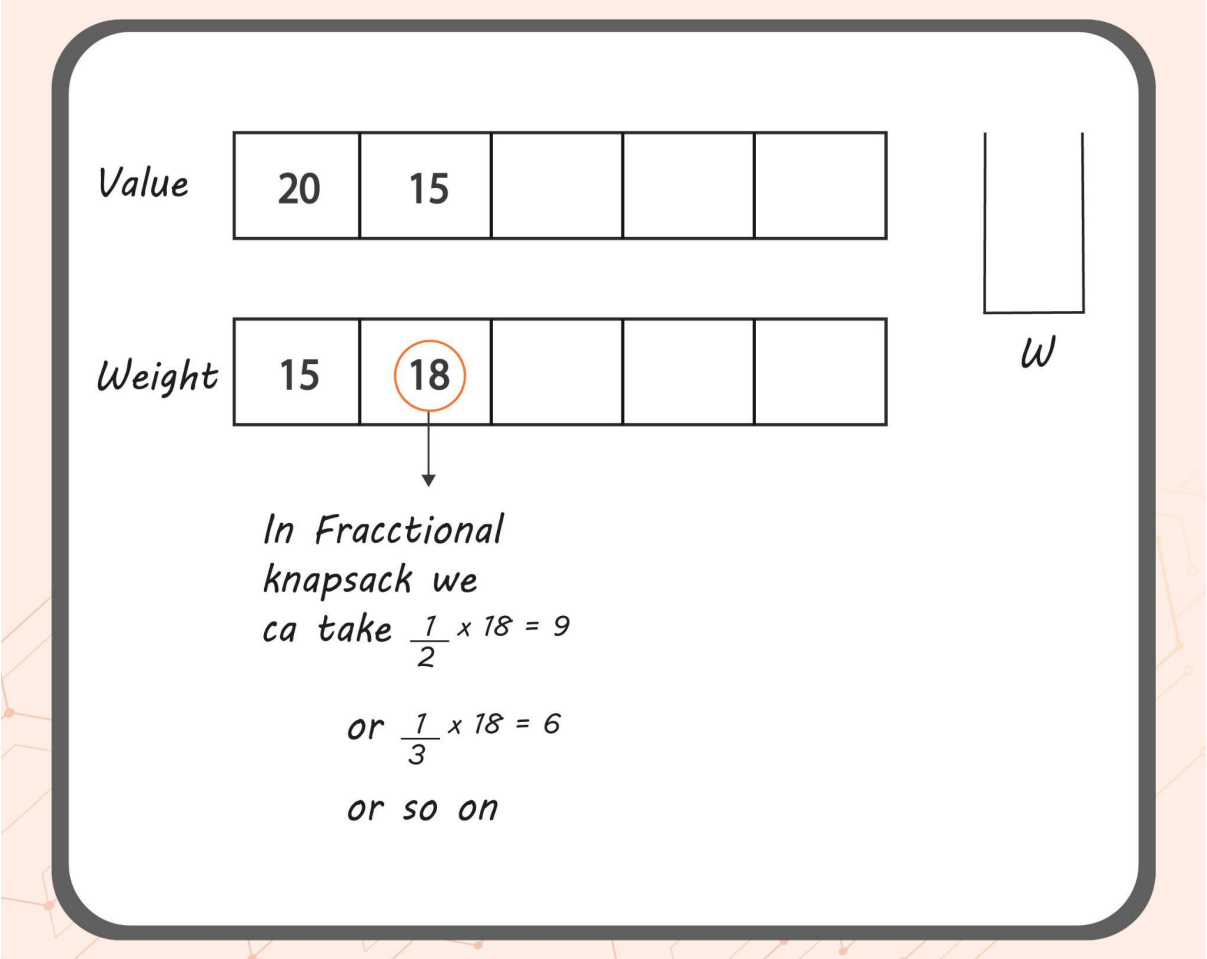
Example 1:

Let us consider the different items that need to be put in the knapsack as dry fruits like Cashews, Raisins, Almonds in quantities 6 kg, 3 kg, and 4 kg respectively and we have a knapsack with weight capacity as 5 kg. The relative prices of these are in the order Cashew > Raisins > Almonds. Now, as we can include fractional weight of items, hence, we will obviously want to fill our knapsack with the items with maximum relative price i.e. Cashews.

Now, let's think for a second that what if these items were not dry fruits and were statues, then we would not have the option to take a part of the total weight of the item as breaking a statue into parts will result in the loss of its value and we could not have chosen the 6 kg statue with the maximum value because our knapsack's capacity is only 5 kg. Therefore, if we have statues as items, then it becomes a case of 0-1 Knapsack.

But since dry fruits can be easily divided and measured according to the weight, that means we now have the option to take only 5 kg out of 6 kg Cashews and still maximize the value of items in the knapsack.

Let's take a look at the image below with another example to understand fractional knapsack.



So, in this problem, we will have to make an array that contains the value per weight of the items and sort that in descending order and then start filling the knapsack accordingly. Therefore we are applying a greedy approach by choosing the items that have the maximum value per weight.

Let's look at the dry run of another example:

Weight of Knapsack = 60 kg

ITEM	WEIGHT	VALUE	VALUE / WEIGHT
I1	5	30	6

I2	10	40	4
I3	15	45	3
I4	22	77	3.5
I5	25	90	3.6

Now we sort all the items in decreasing order of their value per weight.

I1	I2	I5	I4	I3

We are ready to start filling the knapsack

Remaining Knapsack Weight	Items in Knapsack	Cost
60	NIL	0
55	I1	30
45	I1, I2	70
20	I1, I2, I5	160

Now, the remaining knapsack weight is 20kg but I4 weighs 22kg. Therefore we add only 20 kg of I4 which makes the total cost of our knapsack to be 160 + (20/22)*77 = 230.

Min No Of Swaps

Problem Statement: You are given a string ‘S’ of length ‘N’, an array A of length ‘M’ (consisting of lowercase letters). and a positive integer ‘K’. You can take a character from 'A' and change any character in 'S' with this character. The task is to minimize the number of swaps required (between ‘S’ and ‘A’) to make the string ‘S’ ‘k’-periodic.

Approach: Frequency counting and Hashing

Use a 2-dimensional array ‘frequency[K][26]’, where frequency[k][j] stores the frequency of characters ‘S[i]’ which lies in the series of index ‘k’ in K-periodic string, where ‘j’ = ‘i’ % ‘K’ for all 0 <= ‘i’ < N. And swap all the characters except the character which occurred maximum times in the series of ‘k-th’ index.

Algorithm :

- Let ‘N’, ‘M’, ‘K’ be the length of string, length of a character array, and k-period value.
- Let ‘S’ be the given string of length ‘N’ of the string.
- Let ‘arr‘ be the given array of small letters of length ‘M’.
- Maintain a ‘flag[26]‘ array, where index 0 represents ‘a’,1 represents ‘b’, and so on. Initialize it to false
- Now loop for ‘i’ through ‘arr’ from 0 to ‘m’ - 1.
    - Mark flag[arr[i] - ’a’] = 1.
- Initialize the frequency[K][26] with 0.
- Now loop through string ‘S’ from 0 to ‘N’.
    - Increment the value of frequency[i % K][S[i] - ’a’] by 1
- Let ‘ans’ store the answer(minimum number of swaps) initialized to 0.
- Now loop for ‘i’ from 0 to K - 1.
    - Maintain a variable ‘mx’ initialized to 0 which stores the maximum frequency of character in series of ‘i-th’ index from string
    - Maintain 'totalCount' initialized to 0 which counts the total number of characters in the ‘k-th’ index.
    - Loop for j from 0 to 25.
        - Update totalCount+=frequency[i][j]
        - If frequency[i][j] is greater than current maximum i.e ‘mx’ and flag[j]==1(character is present in given ‘arr’)
    - update the ‘mx’ value as 'mx' = frequency[i][j]
    - Update the ‘ans’ as ‘ans’ += ('totalCount' - ‘mx’)
- Finally return the answer ‘ans’.

Time Complexity:

O(max(N, M)), Where ‘N’ and ‘M’ are sizes of a given string and array.

The major task done in this algorithm is calculating the frequency of characters of string ‘S’ that takes O(N) time. Initializing the flag that takes O(M) and in the last loop, we are updating ‘mx’ for each k, that takes O(K * 26). Hence the Overall time complexity = O(N) + O(M) + O(K) = O(max(N, M)) as 1 <= 'K' < 'N'.

Space Complexity:

O(N), where 'N' is the length of string ‘S’.

The major space we are using is flag array, which is O(26) = O(1), and 2D frequency array which is O(K * 26) = O(K) hence, the overall time complexity is O(1) + O(K) = O(N) , where '1' <= ‘K’ < ‘N’.

Maximum Total Value

Link: COCI '13 Contest 1 #4 Lopov - DMOJ: Modern Online Judge

Problem Statement:

The difficult economic situation in the country and reductions in government agricultural subsidy funding have caused Mirko to change his career again, this time to a thief. His first professional endeavour is a jewellery store heist.

The store contains **N** pieces of jewellery, and each piece has some mass $\mathbf{M_i}$ and value $\mathbf{V_i}$. Mirko has K bags to store his loot, and each bag can hold some maximum mass $\mathbf{C_i}$. He plans to store all his loot in these bags, but at most one jewellery piece in each bag, in order to reduce the likelihood of damage during the escape.

Find the maximum total jewellery value that Mirko can "liberate".

Explanation:

- So we have N pieces of jewelry and each piece has some mass and some value. Now in a bag first we want to store that piece of jewelry that has the maximum value according to the greedy approach.
- So sort the array on the basis of value in descending order.
- Find out the bag which has weight just greater than or equal to the jewelry you want to put. (lower_bound)
- There can be bags which have the same value so use Multiset to store the bags.
- Also Multiset stores in a sorted order like a BST so operations are optimized.