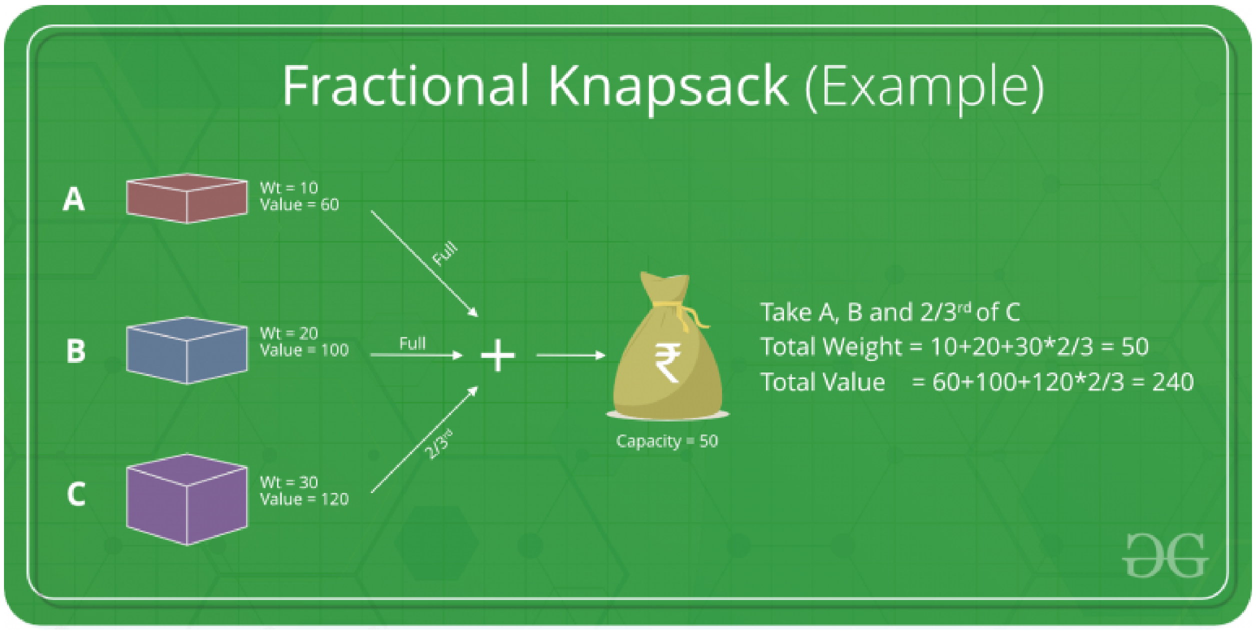
TOPIC - GREEDY ALGORITHMS

Introduction:

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solutions are best fit for Greedy.

For example consider the Fractional Knapsack Problem. The local optimal strategy is to choose the item that has maximum value vs weight ratio. This strategy also leads to a global optimal solution because we are allowed to take fractions of an item. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property: *At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.*



If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming. But Greedy algorithms cannot always be applied. For example, the Fractional Knapsack problem (See this) can be solved using Greedy, but 0-1 Knapsack cannot be solved using Greedy.

Activity Selection Problems:

Let us consider the Activity Selection problem as our first example of Greedy algorithms. Following is the problem statement.

You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

```
Example 1 : Consider the following 3 activities sorted by finish
time.

     start[]  =  {10, 12, 20};

     finish[] =  {20, 25, 30};

A person can perform at most two activities. The maximum set of
activities that can be executed is {0, 2} [ These are indexes in
start[] and finish[] ]

Example 2 : Consider the following 6 activities sorted by finish
time.

     start[]  =  {1, 3, 0, 5, 8, 5};

     finish[] =  {2, 4, 6, 7, 9, 9};

A person can perform at most four activities. The maximum set of
activities that can be executed is {0, 1, 3, 4} [ These are
indexes in start[] and finish[] ]
```

The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.

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 the remaining activities in the sorted array.

   a. If the start time of this activity is greater than or equal to the finish time of the previously selected activity then select this activity and print it.

How does Greedy Choice work for Activities sorted according to finish time?

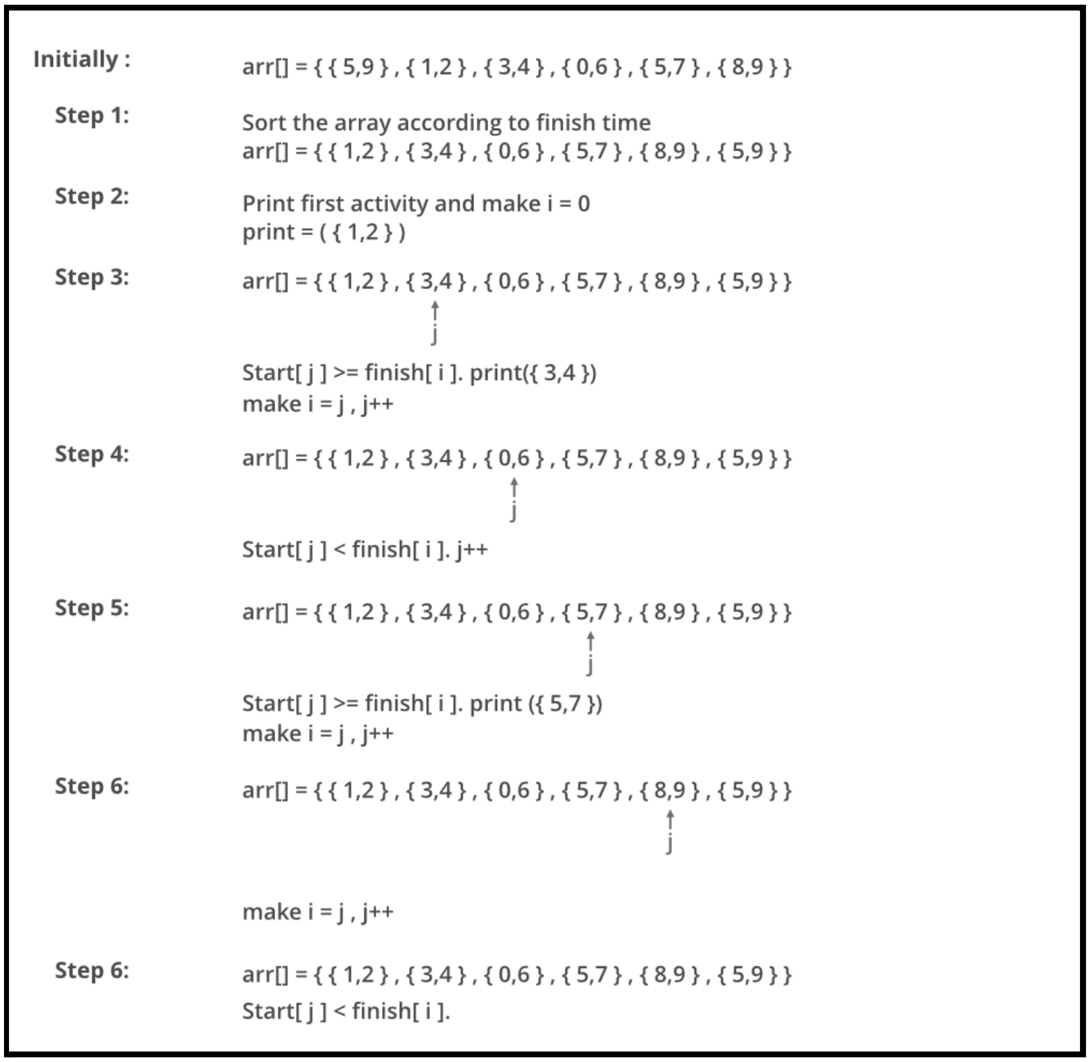
Let the given set of activities be S = {1, 2, 3, …n} and activities are sorted by finish time. The greedy choice is to always pick activity 1. How come activity 1 always provides one of the optimal solutions? We can prove it by showing that if there is another solution B with the first activity other than 1, then there is also a solution A of the same size with activity 1 as the first activity. Let the first activity selected by B be k, then there always exists A = {B – {k}} U {1}.

(Note that the activities in B are independent and k has the smallest finishing time among all. Since k is not 1, finish(k) >= finish(1)).

How to implement when given activities are not sorted?

We create a structure/class for activities. We sort all activities by finish time (Refer sort in C++ STL). Once we have activities sorted, we apply the same algorithm.

Below image is an illustration of the above approach:



```
Initially : arr[] = { { 5,9 } , { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } }

Step 1:     Sort the array according to finish time
            arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }

Step 2:     Print first activity and make i = 0
            print = ( { 1,2 } )

Step 3:     arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }
                                   ↑
                                   j
            Start[ j ] >= finish[ i ]. print({ 3,4 })
            make i = j , j++

Step 4:     arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }
                                             ↑
                                             j
            Start[ j ] < finish[ i ]. j++

Step 5:     arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }
                                                       ↑
                                                       j
            Start[ j ] >= finish[ i ]. print ({ 5,7 })
            make i = j , j++

Step 6:     arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }
                                                                 ↑
                                                                 j

            make i = j , j++

Step 6:     arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }
            Start[ j ] < finish[ i ].
```

Time Complexity: It takes O(nlog(n)) time if input activities may not be sorted. It takes O(n) time when it is given that input activities are always sorted.

For implementation refer this

Greedy Algorithm for Egyptian Fraction:

Every positive fraction can be represented as a sum of unique unit fractions. ***A fraction is a unit fraction if the numerator is 1 and the denominator is a positive integer, for example 1/3 is a unit fraction. Such a representation is called Egyptian Fraction as it was used by ancient Egyptians.***

Following are few examples:

```
Egyptian Fraction Representation of 2/3 is 1/2 + 1/6

Egyptian Fraction Representation of 6/14 is 1/3 + 1/11 + 1/231

Egyptian Fraction Representation of 12/13 is 1/2 + 1/3 + 1/12 + 1/156
```

We can generate Egyptian Fractions using Greedy Algorithm. For a given number of the form 'nr/dr' where dr > nr, first find the greatest possible unit fraction, then recur for the remaining part. For example, consider 6/14, we first find ceiling of 14/6, i.e., 3. So the first unit fraction becomes 1/3, then recur for (6/14 – 1/3) i.e., 4/42.

The Greedy algorithm works because a fraction is always reduced to a form where the denominator is greater than the numerator and the numerator doesn't divide the denominator. For such reduced forms, the highlighted recursive call is made for reduced numerator. So the recursive calls keep on reducing the numerator till it reaches 1.

For implementation refer this

Job Sequencing Problem:

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. How to maximize total profit if only one job can be scheduled at a time.

Examples:

```
Input: Four Jobs with following deadlines and profits

JobID    Deadline    Profit
  a         4          20
  b         1          10
  c         1          40
  d         1          30

Output: Following is maximum profit sequence of jobs

c, a

Input:  Five Jobs with following deadlines and profits

JobID   Deadline    Profit
  a         2         100
  b         1         19
  c         2         27
  d         1         25
  e         3         15

Output: Following is maximum profit sequence of jobs

c, a, e
```

A Simple Solution is to generate all subsets of a given set of jobs and check individual subsets for the feasibility of jobs in that subset. Keep track of maximum profit among all feasible subsets. The time complexity of this solution is exponential.

Following is the algorithm.

1) Sort all jobs in decreasing order of profit.

2) Iterate on jobs in decreasing order of profit.For each job , do the following :

  a. ***For each job find an empty time slot from deadline to 0. If found empty slot put the job in the slot and mark this slot filled.***

Time Complexity: The Time Complexity of the above solution is O(n^2). It can be optimized using Disjoint Set Data Structure. Please refer to the below post for details.

For implementation refer this

Job Sequencing Problem – Loss Minimization

We are given N jobs numbered 1 to N. For each activity, let $\mathbf{T_i}$ denote the number of days required to complete the job. For each day of delay before starting to work for job i, a loss of $\mathbf{L_i}$ is incurred.

We are required to find a sequence to complete the jobs so that overall loss is minimized. We can only work on one job at a time.

If multiple such solutions are possible, then we are required to give the lexicographically least permutation (i.e earliest in dictionary order).

Examples:

```
Input : L = {3, 1, 2, 4} and
```

```
        T = {4, 1000, 2, 5}
```

Output : `3, 4, 1, 2`

Explanation: We should first complete job 3, then jobs 4, 1, 2 respectively.

Input : `L = {1, 2, 3, 5, 6}`

```
        T = {2, 4, 1, 3, 2}
```

Output : `3, 5, 4, 1, 2`

Explanation: We should complete jobs 3, 5, 4, 1 and then 2 in this order.

Let us consider two extreme cases and we shall deduce the general case solution from them.

1. All jobs take the same time to finish, i.e T_i = k for all i. Since all jobs take the same time to finish we should first select jobs which have large Loss (L_i). We should select jobs which have the highest losses and finish them as early as possible.
   Thus this is a greedy algorithm. Sort the jobs in descending order based on L_i only.

2. All jobs have the same penalty. Since all jobs have the same penalty we will do those jobs first which will take less amount of time to finish. This will minimize the total delay, and hence also the total loss incurred.
   This is also a greedy algorithm. Sort the jobs in ascending order based on T_i. Or we can also sort in descending order of 1/ T_i.

3. From the above cases, we can easily see that we should sort the jobs not on the basis of L_i or T_i alone. Instead, we should sort the jobs according to the ratio L_i/ T_i, in descending order.

We can get the lexicographically smallest permutation of jobs if we perform a stable sort on the jobs. An example of a stable sort is merge sort.
To get the most accurate result avoid dividing Li by Ti. Instead, compare the two ratios like fractions. To compare a/b and c/d, compare ad and bc.

Time Complexity: O(N log N)

Space Complexity: O(N)

For implementation refer this

Job Selection Problem – Loss Minimization Strategy

Here we have a slightly different problem than the previous one. Problem statement :

We are given a sequence of N goods of production numbered from 1 to N. Each good has a volume denoted by (Vi). The constraint is that once a good has been completed its volume starts decaying at a fixed percentage (P) per day. All goods decay at the same rate and further each good takes one day to complete.

We are required to find the order in which the goods should be produced so that the overall volume of goods is maximized.

Example :

```
Input: 4, 2, 151, 15, 1, 52, 12 and P = 10%
Output: 222.503
```

What do we need?

On each day we make a selection from among the goods that are yet to be produced. Thus, all we need is a local selection criterion, which when applied to select the jobs will give us the optimum result.

Approach:

Since the total volume that can be obtained from all goods is constant, if we minimize the losses we'll have the optimal solution.

Now consider any good having percentage decay per day P and volume V,
Loss after Day 1: PV
Loss after Day 2: PV + P(1-P)V or V(2P-P^2)
Loss after Day 3: V(2P-P^2) + P(1 – 2P + P^2)V or V(3P – 3P^2 + P^3)

As the day increases the losses too increase. So the trick would be to ensure that the goods are not kept idle after production. Further, since we are required to produce at least one job per day, we should perform low volume jobs, and then perform the high volume jobs. This strategy works due to two factors :

1. High Volume goods are not kept idle after production
2. As the volume decreases the loss per day decreases, so for low volume goods the losses become negligible after a few days.

So in order to obtain the optimum solution we produce the larger volume goods later on. For the first day select the good with the least volume and produce it. Remove the produced goods from the list of goods. For the next day repeat the same. Keep repeating while there are goods left to be produced.

When calculating the total volume at the end of production, keep in mind the goods produced on day i, will have $(1\text{-}P)^{N\text{-}i}$ times its volume left.

Algorithm :

```
Step 1: Add all the goods to a min heap
Step 2: Repeat following steps while Queue is not empty
        Extract the good  at the head of the heap
        Print the good
        Remove the good from the heap
        [END OF LOOP]
Step 4: End
```

For implementation refer this

Policeman catch thieves

Given an array of size n that has the following specifications:

1. ***Each element in the array contains either a policeman or a thief.***
2. ***Each policeman can catch only one thief.***
3. ***A policeman cannot catch a thief who is more than K units away from the policeman.***

We need to find the maximum number of thieves that can be caught.

Example :

```
Input : arr[] = {'P', 'T', 'T', 'P', 'T'},
            k = 1.
Output : 2.
Here maximum 2 thieves can be caught, first
policeman catches first thief and second police-
man can catch either second or third thief.

Input : arr[] = {'T', 'T', 'P', 'P', 'T', 'P'},
            k = 2.
Output : 3.

Input : arr[] = {'P', 'T', 'P', 'T', 'T', 'P'},
            k = 3.
Output : 3.
```

A **brute force** approach would be to check all feasible sets of combinations of police and thief and return the maximum size set among them. Its time complexity is exponential and it can be optimized if we apply greedy.

Thinking irrespective of the policeman and focusing on the allotment works :

1. Get the lowest index of policeman p and thief t. Make an allotment if |p-t| <= k and increment to the next p and t found.
2. Otherwise increment min(p, t) to the next p or t found.
3. Repeat above two steps until the next p and t are found.
4. Return the number of allotments made.

We can use vectors to store the indices of police and thieves in the array and process them.
For implementation refer this

Minimum Swaps for Bracket Balancing

You are given a string of 2N characters consisting of N '[' brackets and N ']' brackets. A string is considered balanced if it can be represented in the for S2[S1] where S1 and S2 are balanced strings. We can make an unbalanced string balanced by swapping adjacent characters. Calculate the minimum number of swaps necessary to make a string balanced.

Example:

```
Input  : []][][
Output : 2
First swap: Position 3 and 4
[][]][
Second swap: Position 5 and 6
[][][]

Input  : [[][]]
Output : 0
The string is already balanced.
```

We can solve this problem by using greedy strategies. If the first X characters form a balanced string, we can neglect these characters and continue on. If we encounter a ']' before the required '[', then we must start swapping elements to balance the string.

Approach :

- We can initially go through the string and store the positions of ‘[‘ in a vector say ‘**pos**‘
- Initialize ‘p’ to 0. We shall use p to traverse the vector ‘pos’.
- We’ll maintain a count of encountered ‘[‘ brackets. When we encounter a ‘[‘ we increase the count and increase ‘p’ by 1. When we encounter a ‘]’ we decrease the count.
- If the count ever goes negative, this means we must start swapping. The element pos[p] tells us the index of the next ‘[‘. We increase the sum by pos[p] – i, where i is the current index. We can swap the elements in the current index and pos[p] and reset the count to 0 and increment p so that it pos[p] indicates to the next ‘[‘.

Time Complexity : O(N)
Space Complexity : O(N)

Challenge ;) : Solve without storing the positions of ‘[‘

For implementation refer this

Fitting Shelves Problem

Given length of wall w and shelves of two lengths m and n, find the number of each type of shelf to be used and the remaining empty space in the optimal solution so that the empty space is minimum. The larger of the two shelves is cheaper so it is preferred. However cost is secondary and the first priority is to minimize empty space on the wall.

Example :

```
Input : w = 24 m = 3 n = 5
Output : 3 3 0
We use three units of both shelves
and 0 space is left.
3 * 3 + 3 * 5 = 24
So empty space  = 24 - 24 = 0
Another solution could have been 8 0 0
but since the larger shelf of length 5
is cheaper the former will be the answer.
```

```
Input : w = 29 m = 3 n = 9
Output : 0 3 2
0 * 3 + 3 * 9 = 27
29 - 27 = 2

Input : w = 24 m = 4 n = 7
Output : 6 0 0
6 * 4 + 0 * 7 = 24
24 - 24 = 0
```

Approach :
We will try all possible combinations of shelves that fit within the length of the wall.

To implement this approach along with the constraint that a larger shelf costs less than the smaller one, starting from 0, we increase no of larger type shelves till they can be fit. For each case we calculate the empty space and finally store that value which minimizes the empty space. if empty space is the same in two cases we prefer the one with more no of larger shelves.

For implementation refer this

Assign Mice to Holes

There are N Mice and N holes are placed in a straight line. Each hole can accommodate only 1 mouse. A mouse can stay at his position, move one step right from x to x + 1, or move one step left from x to x -1. Any of these moves consumes 1 minute. Assign mice to holes so that the time when the last mouse gets inside a hole is minimized.

Example:

```
Input : positions of mice are:
          4 -4 2
        positions of holes are:
          4 0 5
Output :  4
Assign mouse at position x = 4 to hole at
position x = 4 : Time taken is 0 minutes
Assign mouse at position x=-4 to hole at
position x = 0 : Time taken is 4 minutes
Assign mouse at position x=2 to hole at
position x = 5 : Time taken is 3 minutes
After 4 minutes all of the mice are in the holes.
Since, there is no combination possible where
the last mouse's time is less than 4,
answer = 4.
```

```
Input :  positions of mice are:
        -10, -79, -79, 67, 93, -85, -28, -94
          positions of holes are:
         -2, 9, 69, 25, -31, 23, 50, 78
Output : 102
```

Approach :

We will put every mouse to its nearest hole to minimize the time.

How to do this?

This can be done by sorting the positions of mice and holes. This allows us to put the ith mice to the corresponding hole in the holes list. We can then find the maximum difference between the mice and corresponding hole position.

Proof of correctness :

Let i1 < i2 be the positions of two mice and let j1 < j2 be the positions of two holes. It suffices to show via case analysis that

```
max(|i1-j1|, |i2-j2|) <= max(|i1-j2|, |i2-j1|),
   where '|a - b|' represent absolute value of (a - b)
```

For implementation refer this

Additional Resources:

- https://www.geeksforgeeks.org/job-sequencing-using-disjoint-set-union/
- https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/
- https://www.geeksforgeeks.org/efficient-huffman-coding-for-sorted-input-greedy-algo-4/
- https://www.geeksforgeeks.org/huffman-decoding/