**Activity Selection Problem | Greedy Algo-1**

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. 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](https://www.geeksforgeeks.org/fractional-knapsack-problem/) problem can be solved using Greedy, but [0-1 Knapsack](https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/) cannot be solved using Greedy.  
Following are some standard algorithms that are Greedy algorithms.   
**1)**[**Kruskal’s Minimum Spanning Tree (MST)**](https://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/)**:** In Kruskal’s algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn’t cause a cycle in the MST constructed so far.   
**2)**[**Prim’s Minimum Spanning Tree**](https://www.geeksforgeeks.org/prims-algorithm-using-priority_queue-stl/)**:** In Prim’s algorithm also, we create a MST by picking edges one by one. We maintain two sets: a set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.   
**3)**[**Dijkstra’s Shortest Path**](https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/)**:**Dijkstra’s algorithm is very similar to Prim’s algorithm. The shortest-path tree is built up, edge by edge. We maintain two sets: a set of the vertices already included in the tree and the set of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.   
**4)**[**Huffman Coding**](https://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/)**:** Huffman Coding is a loss-less compression technique. It assigns variable-length bit codes to different characters. The Greedy Choice is to assign the least bit length code to the most frequent character.  
The greedy algorithms are sometimes also used to get an approximation for Hard optimization problems. For example, [Traveling Salesman Problem](https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/) is an NP-Hard problem. A Greedy choice for this problem is to pick the nearest unvisited city from the current city at every step. These solutions don’t always produce the best optimal solution but can be used to get an approximately optimal solution.  
Let us consider the [Activity Selection problem](http://en.wikipedia.org/wiki/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:

**Example 1 :** Consider the following 3 activities sorted by

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 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[] ]

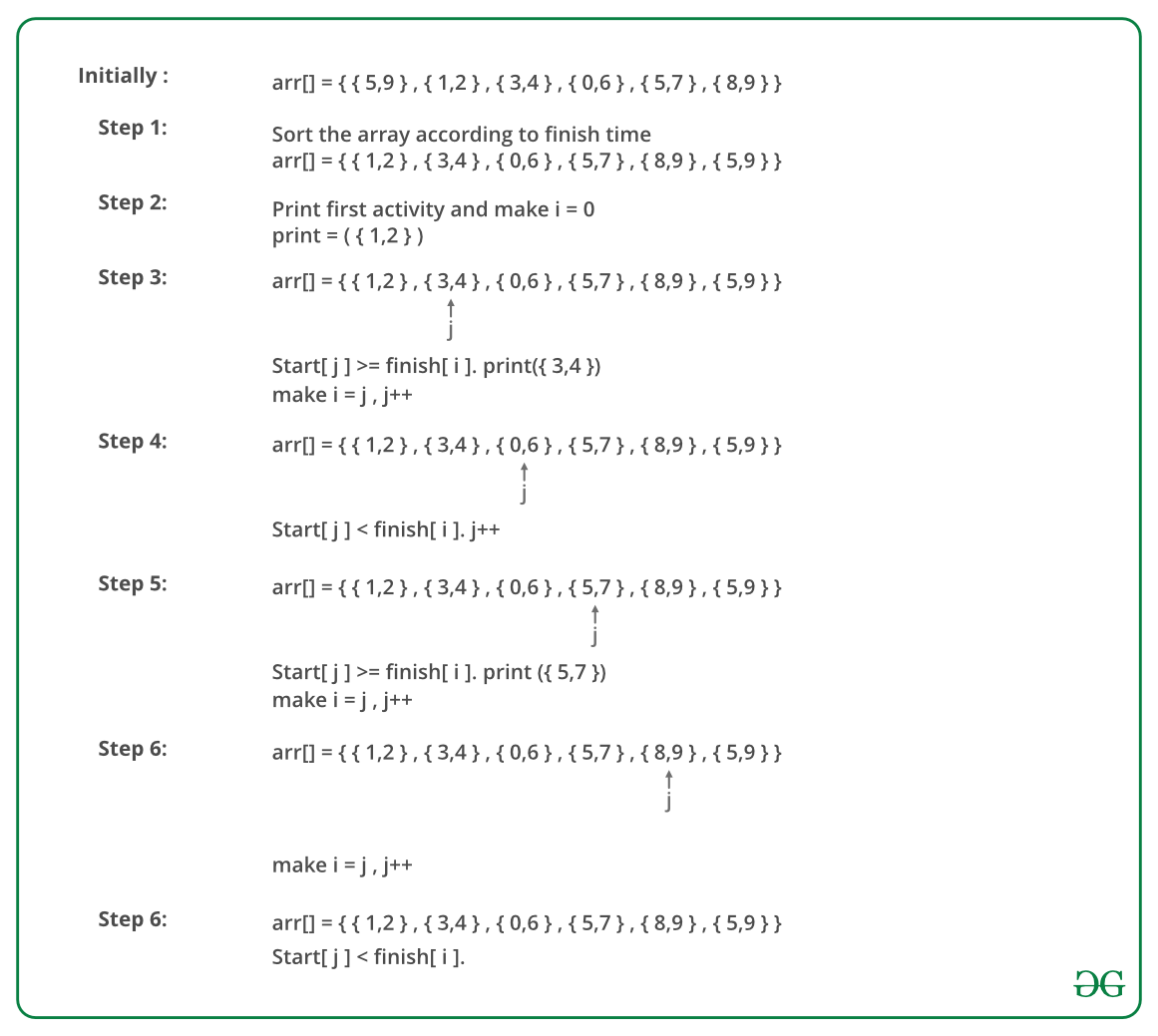
The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity. We can sort the activities according to their finishing time so that we always consider the 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.  
In the following C implementation, it is assumed that the activities are already sorted according to their finish time.

Java code:

**ow 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 exist 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](https://www.geeksforgeeks.org/sort-c-stl/)). Once we have activities sorted, we apply the same algorithm.  
Below image is an illustration of the above approach: 



Java code:

**Time Complexity:** It takes O(n log n) time if input activities may not be sorted. It takes O(n) time when it is given that input activities are always sorted.  
**Using STL we can solve it as follows:**

**Huffman Coding | Greedy Algo-3**

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding 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](http://en.wikipedia.org/wiki/Prefix_code), means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.   
Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.  
See [this](http://en.wikipedia.org/wiki/Huffman_coding#Applications)for applications of Huffman Coding.   
There are mainly two 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***  
Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.  
   Let us understand the algorithm with an example:

character Frequency

a 5

b 9

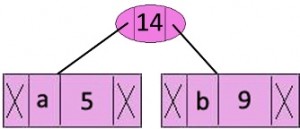
c 12

d 13

e 16

f 45

**Step 1.** Build a min heap that contains 6 nodes where each node represents root of a tree with single node.  
**Step 2** Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14. 



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character Frequency

c 12

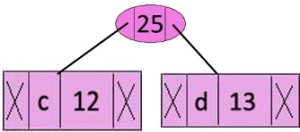
d 13

Internal Node 14

e 16

f 45

**Step 3:** Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

character Frequency

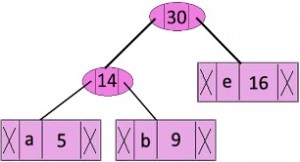
Internal Node 14

e 16

Internal Node 25

f 45

**Step 4:** Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



Now min heap contains 3 nodes.

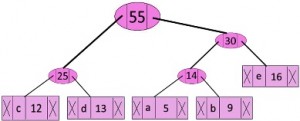
character Frequency

Internal Node 25

Internal Node 30

f 45

**Step 5:** Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



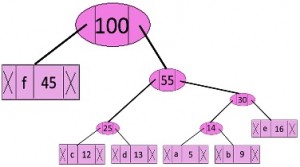
Now min heap contains 2 nodes.

character Frequency

f 45

Internal Node 55

**Step 6:** Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



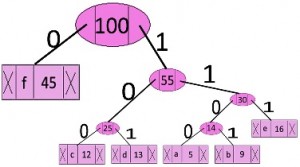
Now min heap contains only one node.

character Frequency

Internal Node 100

Since the heap contains only one node, the algorithm stops here.

***Steps to print codes from Huffman Tree:***  
Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

character code-word

f 0

c 100

d 101

a 1100

b 1101

e 111

**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

[Recommended: Please solve it on “***PRACTICE*** ” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/job-sequencing-problem/0)

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.   
This is a standard [Greedy Algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/)problem.

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)Find a time slot i, such that slot is empty and i < deadline and i is greatest.Put the job in   
this slot and mark this slot filled.   
b)If no such i exists, then ignore the job.*

**Greedy Algorithm to find Minimum number of Coins**

Given a value V, if we want to make a change for V Rs, and we have an infinite supply of each of the denominations in Indian currency, i.e., we have an infinite supply of { 1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, what is the minimum number of coins and/or notes needed to make the change?

**Examples:**

**Input:** V = 70

**Output:** 2

We need a 50 Rs note and a 20 Rs note.

**Input:** V = 121

**Output:** 3

We need a 100 Rs note, a 20 Rs note and a 1 Rs coin.

[Recommended: Please solve it on “***PRACTICE***” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/-minimum-number-of-coins4426/1/)

**Solution:**Greedy Approach.  
**Approach:** A common intuition would be to take coins with greater value first. This can reduce the total number of coins needed. Start from the largest possible denomination and keep adding denominations while the remaining value is greater than 0.

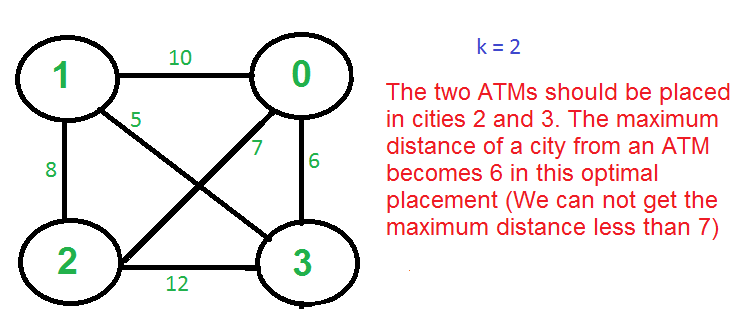
**Algorithm:**

1. Sort the array of coins in decreasing order.
2. Initialize result as empty.
3. Find the largest denomination that is smaller than current amount.
4. Add found denomination to result. Subtract value of found denomination from amount.
5. If amount becomes 0, then print result.
6. Else repeat steps 3 and 4 for new value of V.

**K Centers Problem | Set 1 (Greedy Approximate Algorithm)**

Given n cities and distances between every pair of cities, select k cities to place warehouses (or ATMs or Cloud Server) such that the maximum distance of a city to a warehouse (or ATM or Cloud Server) is minimized.

For example consider the following four cities, 0, 1, 2, and 3, and distances between them, how to do place 2 ATMs among these 4 cities so that the maximum distance of a city to an ATM is minimized. 



There is no polynomial-time solution available for this problem as the problem is a known NP-Hard problem. There is a polynomial-time Greedy approximate algorithm, the greedy algorithm provides a solution that is never worse than twice the optimal solution. The greedy solution works only if the distances between cities follow [Triangular Inequality](http://en.wikipedia.org/wiki/Triangle_inequality) (The distance between two points is always smaller than the sum of distances through a third point).

**The 2-Approximate Greedy Algorithm:**   
1) Choose the first center arbitrarily.   
2) Choose remaining k-1 centers using the following criteria.   
       Let c1, c2, c3, … ci be the already chosen centers. Choose   
       (i+1)’th center by picking the city which is farthest from already   
       selected centers, i.e, the point p which has following value as maximum   
                 Min[dist(p, c1), dist(p, c2), dist(p, c3), …. dist(p, ci)] 

greedyAlgo

**Example (k = 3 in the above-shown Graph)**   
a) Let the first arbitrarily picked vertex be 0.   
b) The next vertex is 1 because 1 is the farthest vertex from 0.   
c) Remaining cities are 2 and 3. Calculate their distances from already selected centers (0 and 1). The greedy algorithm basically calculates the following values.   
        Minimum of all distanced from 2 to already considered centers   
        Min[dist(2, 0), dist(2, 1)] = Min[7, 8] = 7   
        Minimum of all distanced from 3 to already considered centers   
        Min[dist(3, 0), dist(3, 1)] = Min[6, 5] = 5   
        After computing the above values, city 2 is picked as the value corresponding to 2 is maximum.

Note that the greedy algorithm doesn’t give the best solution for k = 2 as this is just an approximate algorithm with a bound as twice optimal.

**Proof that the above greedy algorithm is 2 approximate.**   
Let OPT be the maximum distance of a city from a center in the Optimal solution. We need to show that the maximum distance obtained from the Greedy algorithm is 2\*OPT.   
The proof can be done using contradiction.   
a) Assume that the distance from the furthest point to all centers is > 2·OPT.   
b) This means that distances between all centers are also > 2·OPT.   
c) We have k + 1 points with distances > 2·OPT between every pair.   
d) Each point has a center of the optimal solution with distance <= OPT to it.   
e) There exists a pair of points with the same center X in the optimal solution (pigeonhole principle: k optimal centers, k+1 points)   
f) The distance between them is at most 2·OPT (triangle inequality) which is a contradiction.

**Minimum Number of Platforms Required for a Railway/Bus Station**

Given the arrival and departure times of all trains that reach a railway station, the task is to find the minimum number of platforms required for the railway station so that no train waits.   
We are given two arrays that represent the arrival and departure times of trains that stop.

**Examples:**

***Input****: arr[] = {9:00, 9:40, 9:50, 11:00, 15:00, 18:00}   
dep[] = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}****Output****: 3****Explanation:****There are at-most three trains at a time (time between 9:40 to 12:00)*

***Input****: arr[] = {9:00, 9:40}   
dep[] = {9:10, 12:00}****Output****: 1****Explanation:****Only one platform is needed.*

**Efficient Solution:**

* **Approach:** The idea is to consider all events in sorted order. Once the events are in sorted order, trace the number of trains at any time keeping track of trains that have arrived, but not departed.

For example, consider the above example.

arr[] = {9:00, 9:40, 9:50, 11:00, 15:00, 18:00}

dep[] = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}

**All events are sorted by time.**

Total platforms at any time can be obtained by

subtracting total departures from total arrivals

by that time.

**Time Event Type Total Platforms Needed**

**at this Time**

9:00 Arrival 1

9:10 Departure 0

9:40 Arrival 1

9:50 Arrival 2

11:00 Arrival 3

11:20 Departure 2

11:30 Departure 1

12:00 Departure 0

15:00 Arrival 1

18:00 Arrival 2

19:00 Departure 1

20:00 Departure 0

Minimum Platforms needed on railway station

= Maximum platforms needed at any time

= 3

**Note:**This approach assumes that trains are arriving and departing on the same date. 

**Algorithm:**

1. Sort the arrival and departure times of trains.
2. Create two pointers i=0, and j=0 and a variable to store *ans* and current count *plat*
3. Run a loop while i<n and j<n and compare the ith element of arrival array and jth element of departure array.
4. If the arrival time is less than or equal to departure then one more platform is needed so increase the count, i.e. plat++ and increment i
5. Else if the arrival time greater than departure then one less platform is needed so decrease the count, i.e. plat– and increment j
6. Update the ans, i.e ans = max(ans, plat).

**Implementation:** This doesn’t create a single sorted list of all events, rather it individually sorts arr[] and dep[] arrays, and then uses the [merge process of merge sort](http://geeksquiz.com/merge-sort/) to process them together as a single sorted array.

**Complexity Analysis:**

* **Time Complexity:** O(N \* log N).   
  One traversal O(n) of both the array is needed after sorting O(N \* log N), so the time complexity is O(N \* log N).
* **Space Complexity:**O(1).   
  As no extra space is required.

**Note:**The solution mentioned above uses O(n log n) time complexity and O(1) Space Complexity. There is one more approach to the problem which uses O(n) extra space and O(n) time to solve the problem: