Find your way here

CSC 250: Foundations of Computer Science I

Fall 2023 - Lecture 6

Amitabha Dey

Department of Computer Science
University of North Carolina at Greensboro

Greedy Programming
Characteristics

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result. The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.

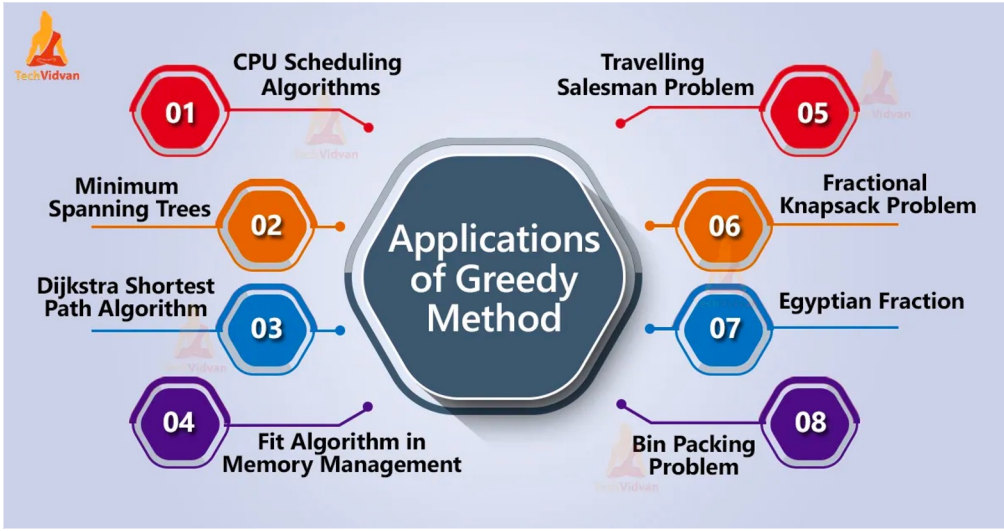
This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.
We can determine if the algorithm can be used with any problem if the problem has the following properties:

1. Greedy Choice Property

If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.

2. Optimal Substructure

If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach. This property is called optimal substructure.



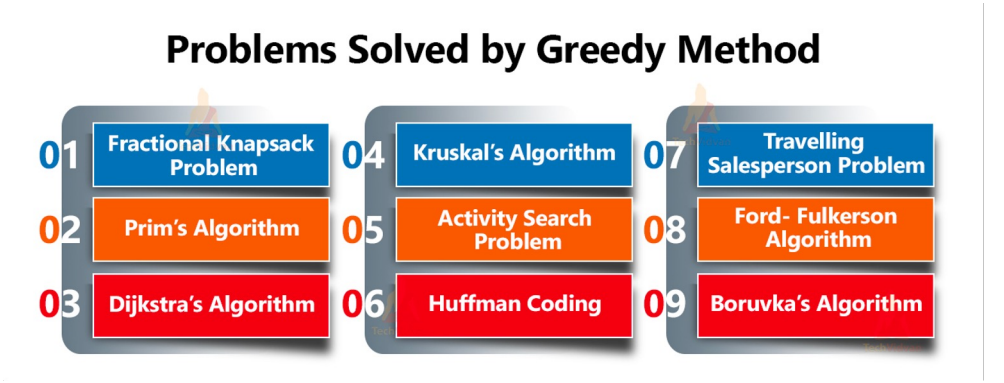


Image credit - https://techvidvan.com/

Greedy Programming
Template

- Identify an optimal substructure or subproblem in the problem.
- Then, determine what the solution will include (for example, the largest sum, the shortest path, etc.).
- Create some sort of iterative way to go through all of the subproblems and
- Build a solution.

Problem Statement: Find the best route to reach the destination city from the given starting point using a greedy method.

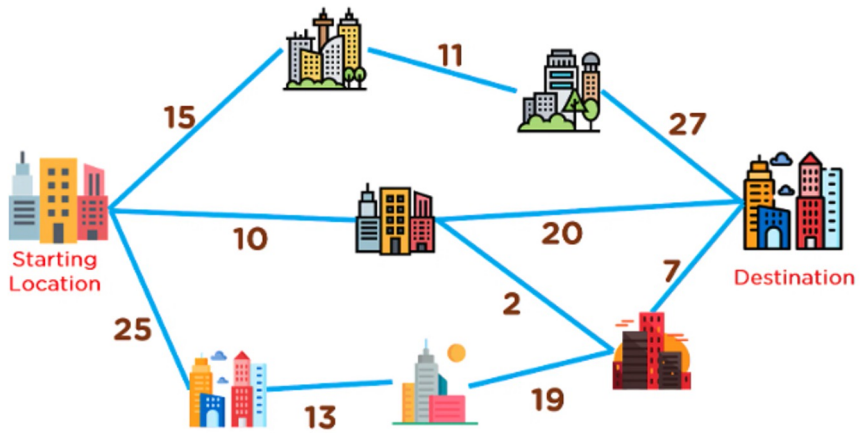


Image credit - https://www.simplilearn.com/

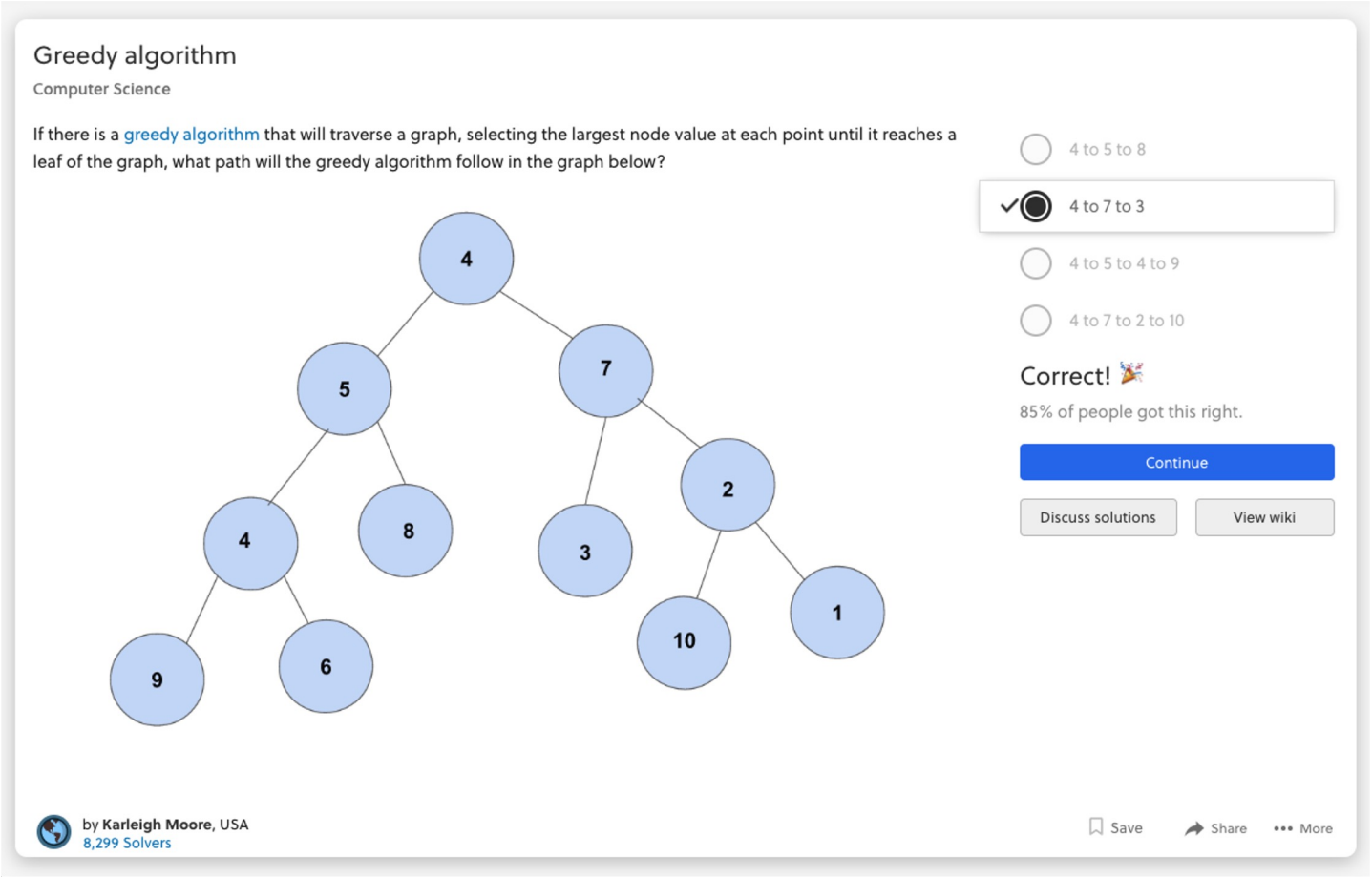


Image credit - https://brilliant.org/

Greedy Programming

How a greedy algorithm may fail to achieve the optimal solution

Starting from A, a greedy algorithm that tries to find the maximum by following the greatest slope will find the local maximum at "m", oblivious to the global maximum at "M".

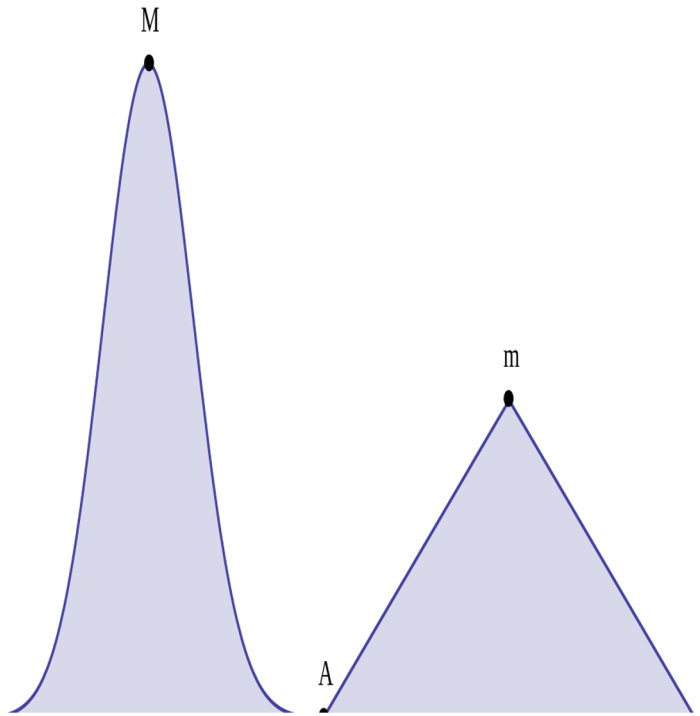


Image credit - https://en.wikipedia.org/

Greedy heuristics are known to produce suboptimal results on many problems, and so natural questions are:

- For which problems do greedy algorithms perform optimally?
- For which problems do greedy algorithms guarantee an approximately optimal solution?
- For which problems are the greedy algorithm guaranteed *not* to produce an optimal solution?

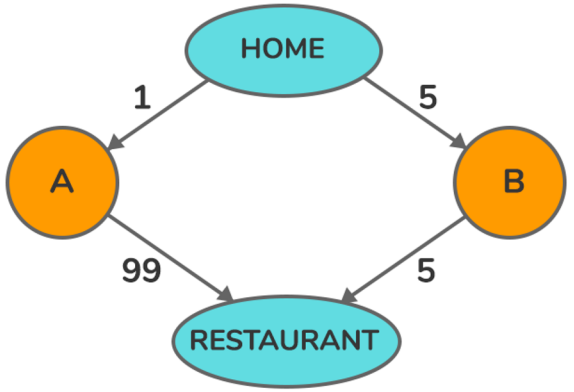


Image credit - https://www.interviewbit.com/

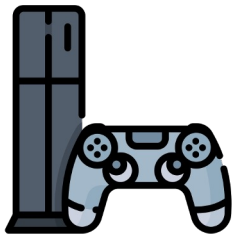
Limitations of Greedy Programming

Here, we will look at one form of the knapsack problem. The knapsack problem involves deciding which subset of items you should take from a set of items if you want to optimize some value: perhaps the worth of the items, the size of the items, or the ratio of worth to size.

In this problem, we will assume that we can either take an item or leave it (we cannot take a fractional part of an item). We will also assume that there is only one of each item. Our knapsack has a fixed size, and we want to optimize the worth of the items we take, so we must choose the items we take with care.

Our **knapsack** can hold at most **25 units of space**.

Which items do we choose to optimize for price?



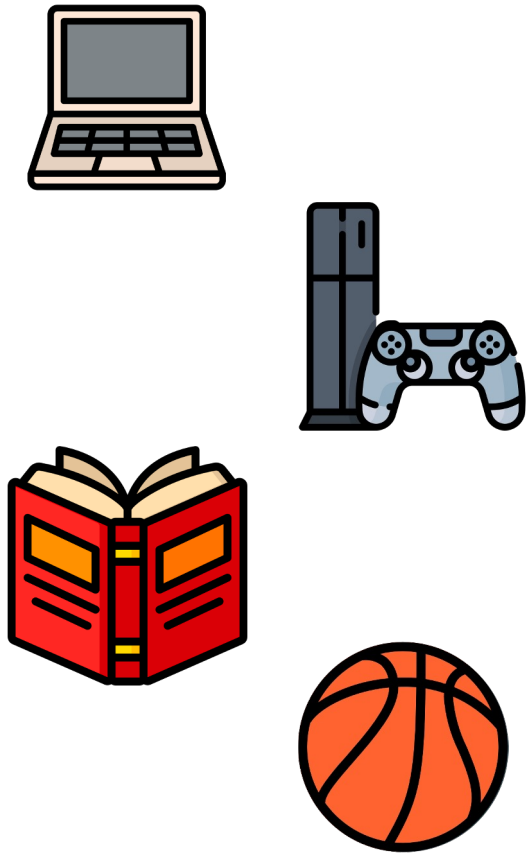
Item	Size	Price
Laptop	22	12
PlayStation	10	9
Textbook	9	9
Basketball	7	6

Image credit - https://brilliant.org/

Limitations of Greedy Programming

There are two greedy algorithms we could propose to solve this. One has a rule that selects the item with the largest price at each step, and the other has a rule that selects the smallest sized item at each step.

- **Largest-price Algorithm**: At the first step, we take the laptop. We gain 12 units of worth, but can now only carry 25−22=3 units of additional space in the knapsack. Since no items that remain will fit into the bag, we can only take the laptop and have a total of 12 units of worth.
- **Smallest-sized-item Algorithm**: At the first step, we will take the smallest-sized item: the basketball. This gives us 6 units of worth, and leaves us with 25−7=18 units of space in our bag. Next, we select the next smallest item, the textbook. This gives us a total of 6+9=15 units of worth, and leaves us with 18−9=9 units of space. Since no remaining items are 9 units of space or less, we can take no more items.



Item	Size	Price
Laptop	22	12
PlayStation	10	9
Textbook	9	9
Basketball	7	6

Image credit - https://brilliant.org/

The greedy algorithms yield solutions that give us 12 units of worth and 15 units of worth.
But neither of these are the optimal solution.

Limitations of Greedy Programming

There are two gree
the item with the la
sized item at each

- **Largest-pric**
  worth, but ca
  no items tha
  12 units of w
- **Smallest-siz**
  the basketba
  in our bag. N
  6+9=15 units
  items are 9 u

Taking the **Textbook** and the **PlayStation** yields 9+9=18 units of worth and takes up 10+9=19 units of space. This is the optimal answer, and we can see that a greedy algorithm will not solve the knapsack problem since the greedy choice and optimal substructure properties do not hold.

	Size	Price
	22	12
n	10	9
	9	9
l	7	6

Image credit - https://brilliant.org/

The greedy algorithms yield solutions that give us 12 units of worth and 15 units of worth.
But neither of these are the optimal solution.

Greedy Programming
Coin Change Problem

The famous coin change problem is a classic example of using greedy algorithms. According to the coin change problem, we are given a set of coins of various denominations.

Our task is to use these coins to form a sum of money using the minimum (or optimal) number of coins. Also, we can assume that a particular denomination has an infinite number of coins. In other words, we can use a particular denomination as many times as we want.

As an example, if we have to achieve a sum of 93, we need a minimum of 5 coins: {**50, 20, 20, 2, 1**}

As you can see, at each step we use the highest possible coin from the denominations.

At last, we are able to reach the value of 93 just by using 5 coins.

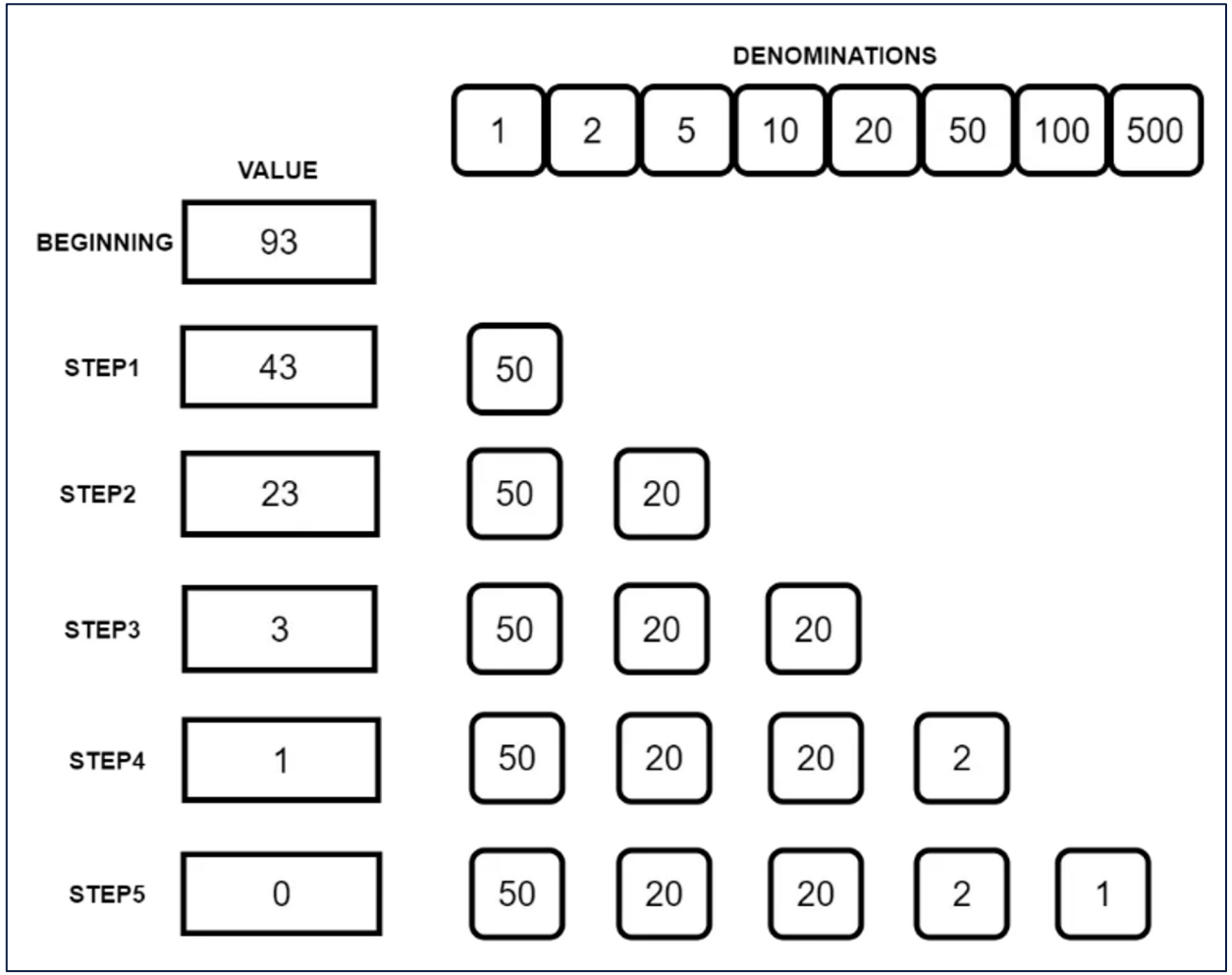


Image credit - https://progressivecoder.com/

Greedy Programming
Coin Change Problem - Issues

While the coin change problem can be solved using Greedy algorithm, there are scenarios in which it does not produce an optimal result.

For example, consider the below denominations.
{**1, 5, 6, 9**}

Now, using these denominations, if we have to reach a sum of 11, the greedy algorithm will provide the answer to the right.

Here, accordingly to the Greedy algorithm, we will end up the denomination 9, 1, 1 i.e. 3 coins to reach the value of 11.

However, if you look closely, there is a more optimal solution. And that is by using the denominations 5 & 6. Using them, we can reach 11 with only 2 coins.

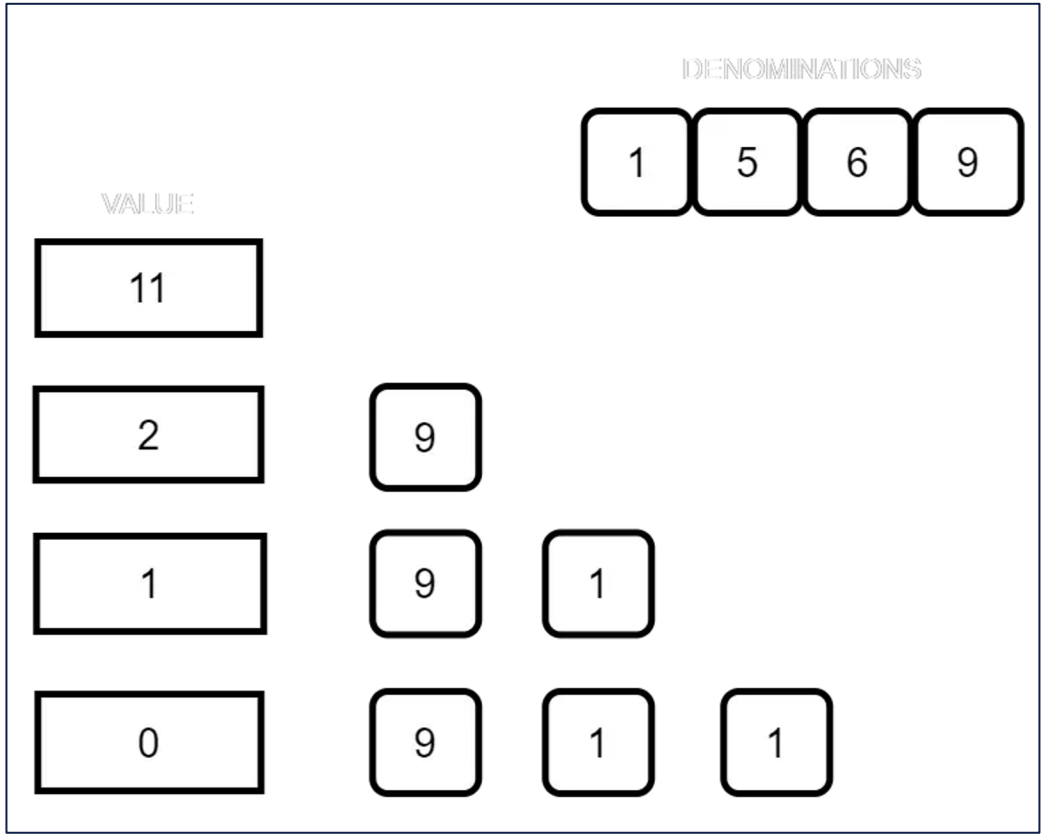


Image credit - https://progressivecoder.com/

Greedy Programming
Fractional Knapsack Problem

Let's say that the maximum capacity of the knapsack is 17, and there are three items available. The first item is gold, the second item is silver, and third item is wood.

- Weight of gold is 10, the weight of silver is 6, and the weight of wood is 2
- the value (profit) of gold is 40, the value (profit) of silver is 30, and the value (profit) of wood is 6.
- Ratio of gold= value/weight = 40/10 = 4
- Ratio of silver = value/weight=30/6 = 5
- Ratio of wood = value/weight = 6/2 = 3
- Arranging the ratios in descending: 5, 4, 3.
- The largest ratio is 5 and we match it to the corresponding weight “6”. It points to silver.

maximum capacity = 17

sum of weights for gold and silver = 6 + 10 = 16

$$\text{Ratio of wood that can fill the bag} = \frac{\text{maximum capacity - sum of gold and silver}}{\text{weight of wood}}$$

$$\text{Ratio of wood that can fill the bag} = \frac{17 - 16}{2} = \frac{1}{2}$$

Weight of wood added = weight of wood * Ratio of wood that can fill the bag

$$\text{Total sum of items in the bag} = 6 + 10 + 2 * \frac{1}{2} = 17$$

$$\text{Total Profit} = 40 + 30 + 6 * \frac{1}{2} = 73$$

Image credit - https://www.freecodecamp.org/

Greedy Programming
Fractional Knapsack Problem

- We put silver in the knapsack first and compare it to the maximum weight which is 17. 6 is less than 17 so we have to add another item. Going back to the ratios, the second largest is “4” and it corresponds to the weight of “10” which points to gold.
- Now, we put gold in the knapsack, add the weight of the silver and gold, and compare it with the knapsack weight. (6 + 10 = 16). Checking it against the maximum weight, we see that it is less. So we can take another item. We go back to the list of ratios and take the 3rd largest which is “3” and it corresponds to “2” which points to wood.

maximum capacity = 17

sum of weights for gold and silver = 6 + 10 = 16

$$\text{Ratio of wood that can fill the bag} = \frac{\text{maximum capacity - sum of gold and silver}}{\text{weight of wood}}$$

$$\text{Ratio of wood that can fill the bag} = \frac{17 - 16}{2} = \frac{1}{2}$$

Weight of wood added = weight of wood * Ratio of wood that can fill the bag

$$\text{Total sum of items in the bag} = 6 + 10 + 2 * \frac{1}{2} = 17$$

$$\text{Total Profit} = 40 + 30 + 6 * \frac{1}{2} = 73$$

Image credit - https://www.freecodecamp.org/

Greedy Programming
Fractional Knapsack Problem

Knapsack size - 17

Item	Weight	Value	Value / Weight
Gold	10	40	40 / 10 = 4
Silver	6	30	30 / 6 = 5
Wood	2	6	6 / 2 = 3

Sort **v/w** in descending order.

Value / Weight	Weight	Remaining Capacity	Value
5	6	17 - 6 = 11	5 x 6 = 30
4	10	11 - 10 = 1	4 x 10 = 40
3	2	1 - 1 = 0	3 x 1 = 3

Total Profit = 30 + 40 + 3 = 73

Greedy Programming
Fractional Knapsack Problem

- When we add wood in the knapsack, the total weight is (6 +10+2 = 18) but that is greater than our maximum weight which is 17. We take out the wood from the knapsack and we are left with gold and silver. The total sum of the two is 16 and the maximum capacity is 17. So we need a weight of 1 to make it equal. Now we apply condition 2 discussed above to find the fraction of wood to fit in the knapsack.

Now the knapsack is filled.

maximum capacity = 17

sum of weights for gold and silver = 6 + 10 = 16

$$\text{Ratio of wood that can fill the bag} = \frac{\text{maximum capacity - sum of gold and silver}}{\text{weight of wood}}$$

$$\text{Ratio of wood that can fill the bag} = \frac{17 - 16}{2} = \frac{1}{2}$$

Weight of wood added = weight of wood * Ratio of wood that can fill the bag

$$\text{Total sum of items in the bag} = 6 + 10 + 2 * \frac{1}{2} = 17$$

$$\text{Total Profit} = 40 + 30 + 6 * \frac{1}{2} = 73$$

Image credit - https://www.freecodecamp.org/

Greedy Programming

Finding the Path With Maximum Reward

Suppose we have a robot that is placed at cell `(0, 0)` of an `m * n` grid.

The robot has to navigate the grid and reach its goal position, while collecting a reward from each cell it passes through.

The aim of navigation is to follow a path that maximizes the reward through the grid. The only legal moves allowed are an "**up**" move and a "**right**" move.

The greedy algorithm for maximizing reward in a path starts simply--with us taking a step in a direction which maximizes reward.

It doesn't keep track of any other path. The algorithm only follows a specific direction, which is the `local best` direction. The pseudo-code for the algorithm is provided here.

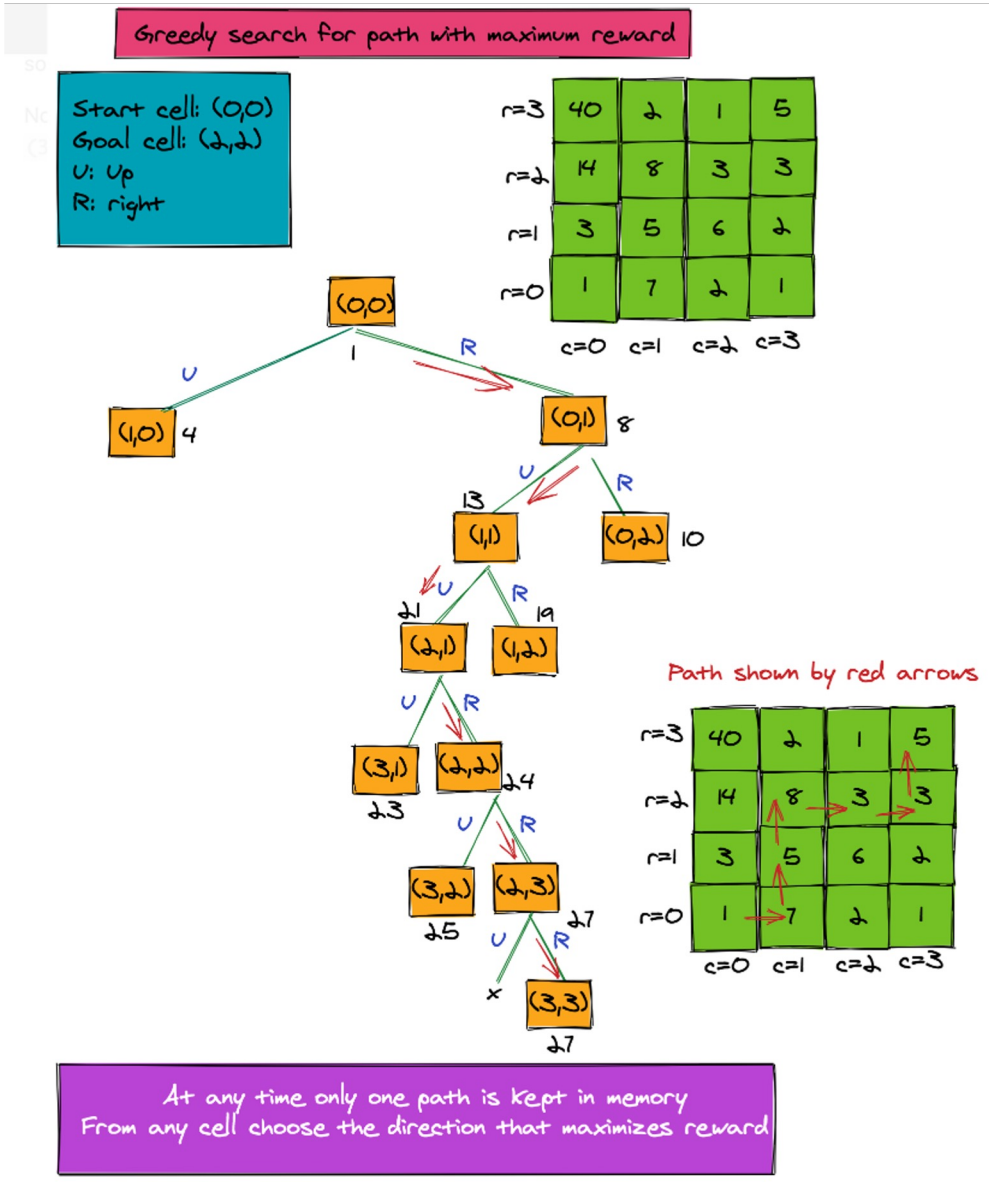


Image credit - https://algodaily.com/

Greedy Programming
Buy and Sell Stocks

Problem Statement: You are given array prices [] which contain the price of a stock on some days. You have to choose one day for buying the stock and a different one for selling it. Return the days on which you buy and sell the stock.

Approach towards Solution: In this question, you need not know anything about stocks, the approach we're providing is easy enough to understand even for beginners. Now, it's quite obvious that when you want to purchase anything, you want to pay the least price you can. Similarly, while selling the same stuff, you'll want to increase your profits by selling it at a larger price. Thus, you're greedy to increase your profits.

Example:

Input: {7,1,5,3,6,4}

Output: 5

Explanation: As we've mentioned in the approach, first you'll find the least element, which is 1, here. Now, the buyingDay becomes 1. After that, we search for the maximum value of stock in the days following 1. So, the sellingDay becomes day having 6 value. Now, we find the difference between both and get 5 as our output. Take a look at the below illustration.

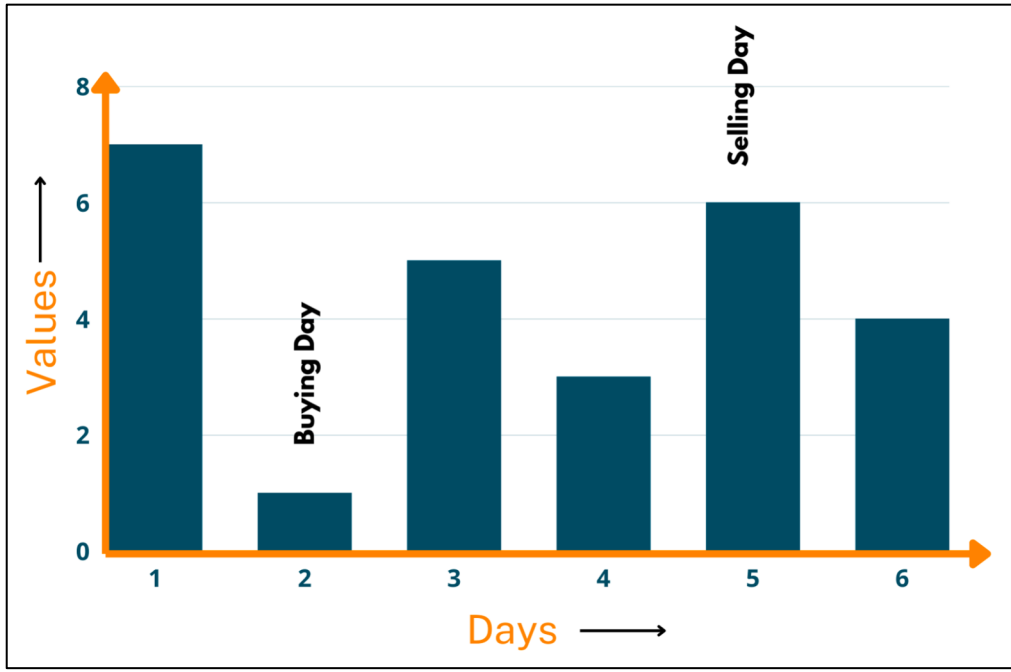


Image credit - https://favtutor.com/

Greedy Programming
Activity Selection Problem

This problem contains a set of activities or tasks that need to be completed. Each one has a start and finish time. The algorithm finds the maximum number of activities that can be done in a given time without them overlapping.

Approach to the Problem

- We have a list of activities. Each has a start time and finish time.
- First, we sort the activities and start time in ascending order using the finish time of each.
- Then we start by picking the first activity. We create a new list to store the selected activity.
- To choose the next activity, we compare the finish time of the last activity to the start time of the next activity. If the start time of the next activity is greater than the finish time of the last activity, it can be selected. If not we skip this and check the next one.
- This process is repeated until all activities are checked. The final solution is a list containing the activities that can be done.

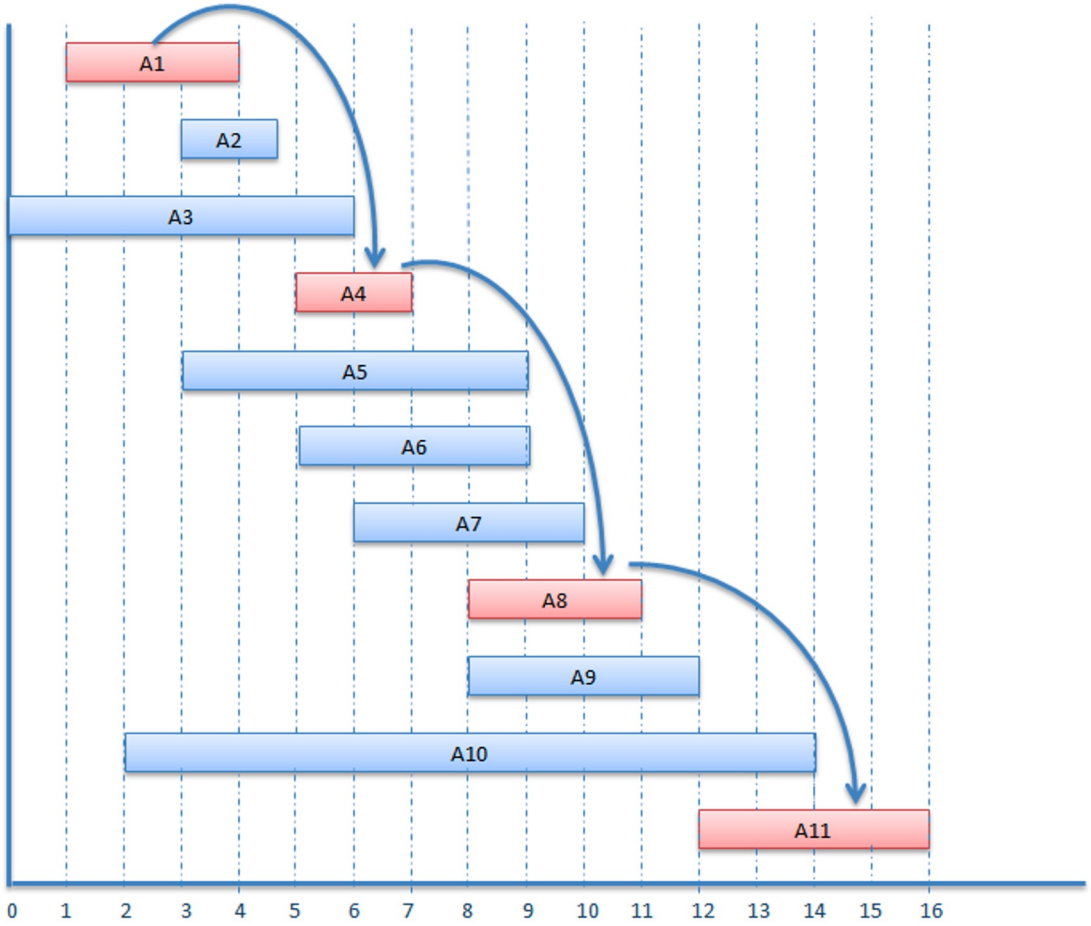


Image credit - https://scanftree.com/

Greedy Programming
Activity Selection Problem

The table below shows a list of activities as well as starting and finish times.

Starting Time	Finish Time	Task
2	5	Homework
6	10	Presentation
4	8	Term Paper
10	12	Volleyball practice
13	14	Biology lecture
7	15	Hangout

The first step is to sort the finish time in ascending order and arrange the activities with respect to the result.

Starting Time	Finish Time	Task
2	5	Homework
4	8	Term Paper
6	10	Presentation
10	12	Volleyball practice
13	14	Biology lecture
7	15	Hangout

Image credit - https://www.freecodecamp.org/

Greedy Programming
Activity Selection Problem

Starting Time	Finish Time	Task
2	5	Homework
4	8	Term Paper
6	10	Presentation
10	12	Volleyball practice
13	14	Biology lecture
7	15	Hangout

Image credit - https://www.freecodecamp.org/

Our final result is the list of selected activities that we can do without time overlapping: **{Homework, Presentation, Volleyball practice, Biology lecture}**.

After sorting the activities, we select the first activity and store it in the selected activity list. In our example, the first activity is “Homework”.

Moving to the next activity, we check the finish time of “Homework” (5) which was the last activity selected and the starting time of “Term paper” (4).

To pick an activity, **the starting time of the next activity must be greater than or equal to the finish time**.

(4) is less than (5), so we skip the activity and move to the next.

The next activity “Presentation” has a starting time of (6) and it is greater than the finish time (5) of “Homework”. So we select it and add it to our list of selected activities.

For the next activity, we do the same checking. Finish time of “Presentation” is (10), starting time of “Volleyball practice ” is (10). We see that the starting time is equal to the finish time which satisfies one of the conditions, so we select it and add it to our list of selected activities.

Continuing to the next activity, the finish time of “Volleyball” practice is (12) and the starting time of “Biology lecture” is (13). We see the starting time is greater than the finish time so we select it.

For our last activity, the starting time for “Hangout” is (7) and the finishing time of our last activity “Biology lecture” is (14), 7 is less than 14, so we cannot select the activity. Since we are at the end of our activity list, the process ends.

Greedy Programming
Activity Selection Problem

A problem for which a greedy algorithm does work and gives the optimal solution is the `activity selection problem`.

The activity selection problem involves a set of `n` activities, each having a start and finish time. Some of these activities are overlapping. The objective is to select a maximum sized set of non-overlapping activities.

The following are the parameters of this problem:

1. `n` is the total number of intervals
2. Array `s` and `f` of size `n` to store the start and finish time of each activity
3. `s[j] < f[j]` for `j = 0..(n-1)`
4. The arrays are sorted according to finish time values (in `f` array) in ascending order

The algorithm first adds activity `0` to the selected pool. Next, it iteratively finds the first activity whose start time is greater than the finish time of the last added activity. It then adds it to the pool.

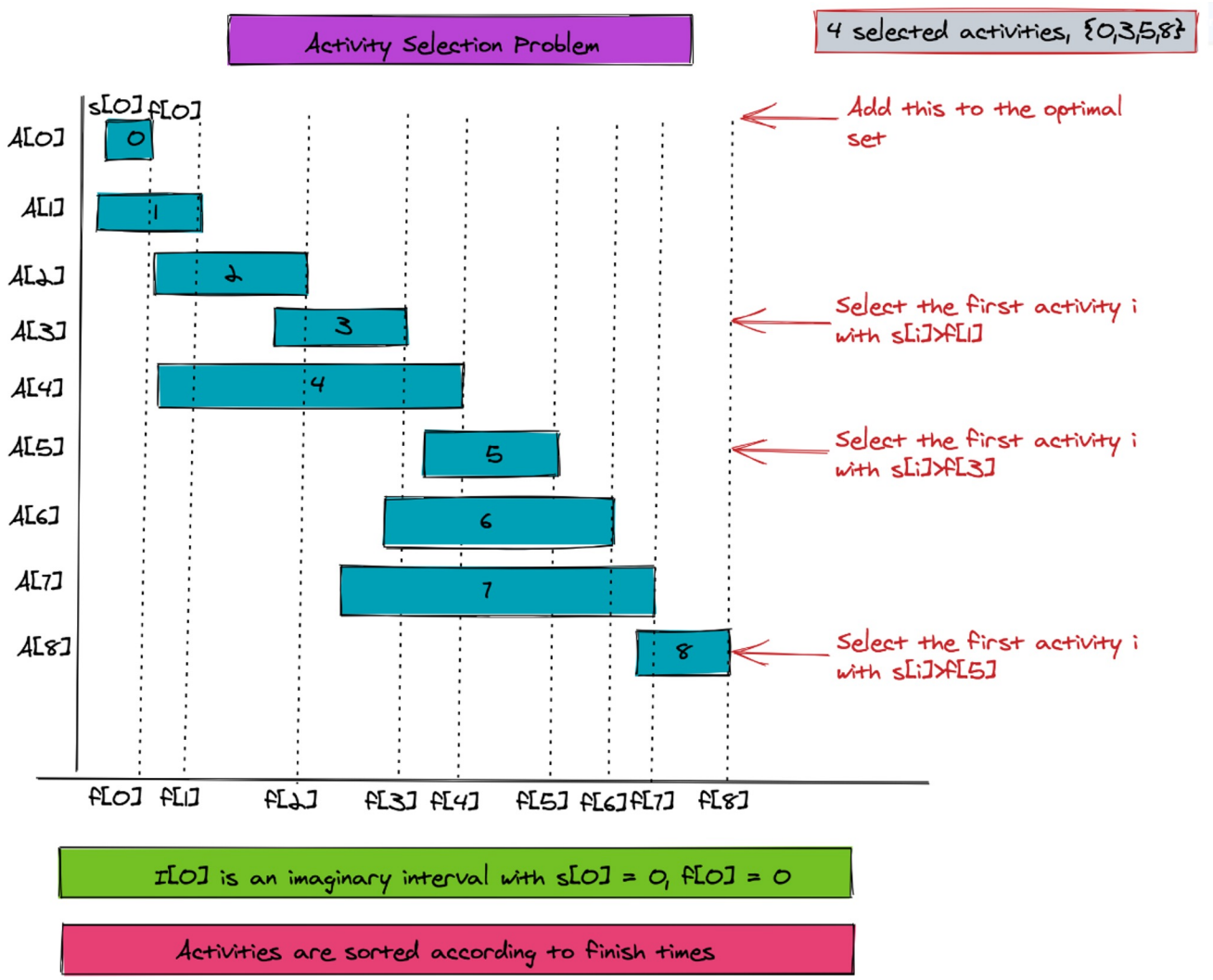


Image credit - https://algodaily.com/

Greedy Programming - Leetcode

#	Title	Acceptance	Difficulty	Frequency
11	Container With Most Water	54.1%	Medium	
44	Wildcard Matching	27.2%	Hard	
45	Jump Game II	40.1%	Medium	
55	Jump Game	38.7%	Medium	
122	Best Time to Buy and Sell Stock II	64.8%	Medium	
134	Gas Station	45.9%	Medium	
135	Candy	41.2%	Hard	
179	Largest Number	35.1%	Medium	
253	Meeting Rooms II	50.7%	Medium	
277	Find the Celebrity	46.8%	Medium	
280	Wiggle Sort	67.3%	Medium	
316	Remove Duplicate Letters	45.7%	Medium	
321	Create Maximum Number	29.4%	Hard	
330	Patching Array	40.7%	Hard	
334	Increasing Triplet Subsequence	41.4%	Medium	
358	Rearrange String k Distance Apart	38.0%	Hard	
376	Wiggle Subsequence	48.4%	Medium	
397	Integer Replacement	35.3%	Medium	

Image credit - https://leetcode.com/

Leetcode - Greedy Programming
https://leetcode.com/tag/greedy/