**Divide and Conquer Algorithm**

A **divide and conquer algorithm** is a strategy of solving a large problem by

1. breaking the problem into smaller sub-problems
2. solving the sub-problems, and
3. combining them to get the desired output.

To use the divide and conquer algorithm, **recursion** is used.

## How Divide and Conquer Algorithms Work?

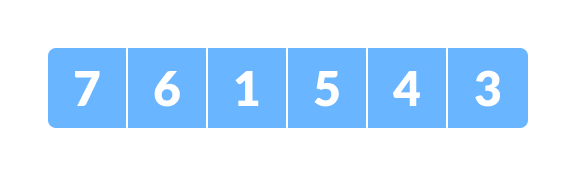
Here are the steps involved:

1. **Divide**: Divide the given problem into sub-problems using recursion.
2. **Conquer**: Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
3. **Combine:** Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

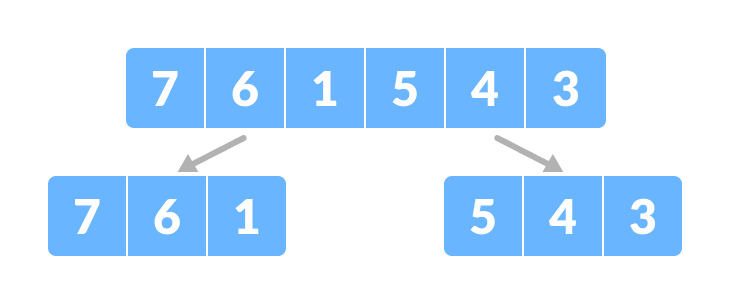
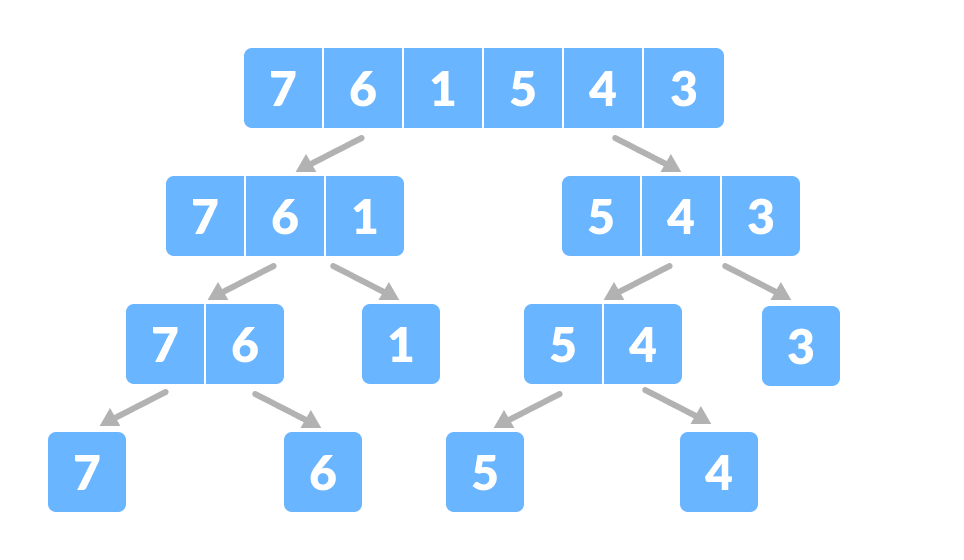
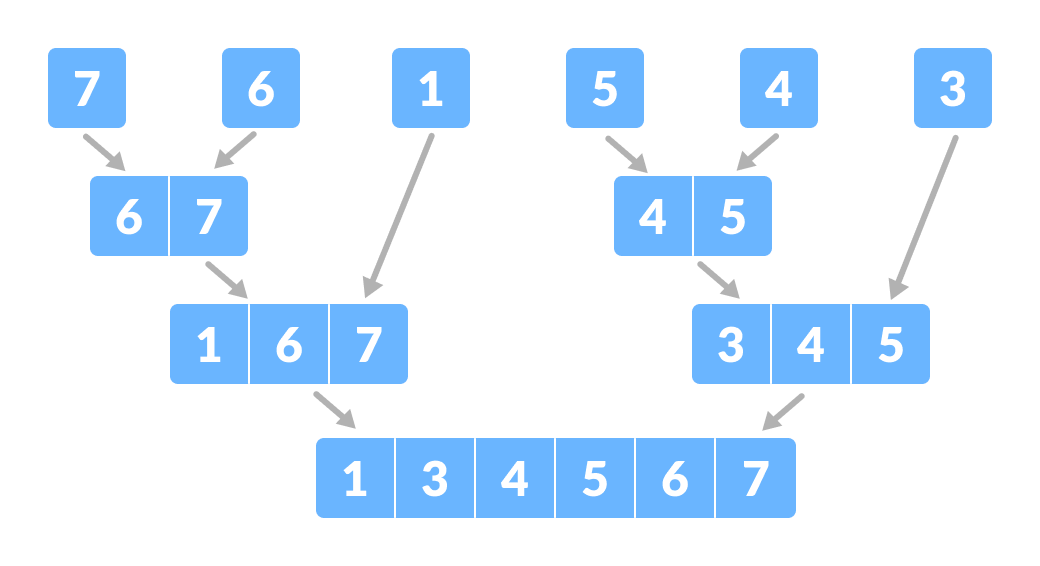
Let us understand this concept with the help of an example.

Here, we will sort an array using the divide and conquer approach (ie. [merge sort](https://www.programiz.com/dsa/merge-sort)).

1. Let the given array be:



Array for merge sort

1. **Divide** the array into two halves.
2. Divide the array into two subparts  
   Again, divide each subpart recursively into two halves until you get individual elements.
3. Divide the array into smaller subparts
4. Now, combine the individual elements in a sorted manner. Here, **conquer** and **combine** steps go side by side.Combine the subparts

## Time Complexity

The complexity of the divide and conquer algorithm is calculated using the [master theorem](https://www.programiz.com/dsa/master-theorem).

T(n) = aT(n/b) + f(n),

where,

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed to have the same size.

f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

Let us take an example to find the time complexity of a recursive problem.

For a merge sort, the equation can be written as:

T(n) = aT(n/b) + f(n)

= 2T(n/2) + O(n)

Where,

a = 2 (each time, a problem is divided into 2 subproblems)

n/b = n/2 (size of each sub problem is half of the input)

f(n) = time taken to divide the problem and merging the subproblems

T(n/2) = O(n log n) (To understand this, please refer to the master theorem.)

Now, T(n) = 2T(n log n) + O(n)

≈ O(n log n)

## Divide and Conquer Vs Dynamic approach

The divide and conquer approach divides a problem into smaller subproblems; these subproblems are further solved recursively. The result of each subproblem is not stored for future reference, whereas, in a dynamic approach, the result of each subproblem is stored for future reference.

Use the divide and conquer approach when the same subproblem is not solved multiple times. Use the dynamic approach when the result of a subproblem is to be used multiple times in the future.

Let us understand this with an example. Suppose we are trying to find the Fibonacci series. Then,

**Divide and Conquer approach:**

fib(n)

If n < 2, return 1

Else , return f(n - 1) + f(n -2)

**Dynamic approach:**

mem = []

fib(n)

If n in mem: return mem[n]

else,

If n < 2, f = 1

else , f = f(n - 1) + f(n -2)

mem[n] = f

return f

In a dynamic approach, mem stores the result of each subproblem.

## Advantages of Divide and Conquer Algorithm

* The complexity for the multiplication of two matrices using the naive method is O(n3), whereas using the divide and conquer approach (i.e. Strassen's matrix multiplication) is O(n2.8074). This approach also simplifies other problems, such as the Tower of Hanoi.
* This approach is suitable for multiprocessing systems.
* It makes efficient use of memory caches.

# Greedy Algorithm

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.

However, 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.

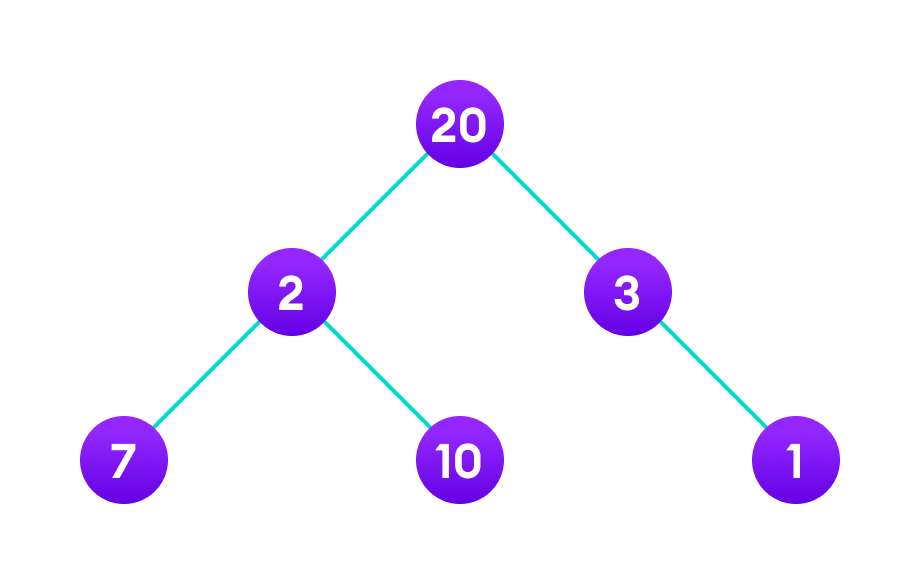
## Advantages of Greedy Approach

* The algorithm is **easier to describe**.
* This algorithm can **perform better** than other algorithms (but, not in all cases).

## Drawback of Greedy Approach

As mentioned earlier, the greedy algorithm doesn't always produce the optimal solution. This is the major disadvantage of the algorithm

For example, suppose we want to find the longest path in the graph below from root to leaf. Let's use the greedy algorithm here.

Apply greedy approach to this tree to find the longest route

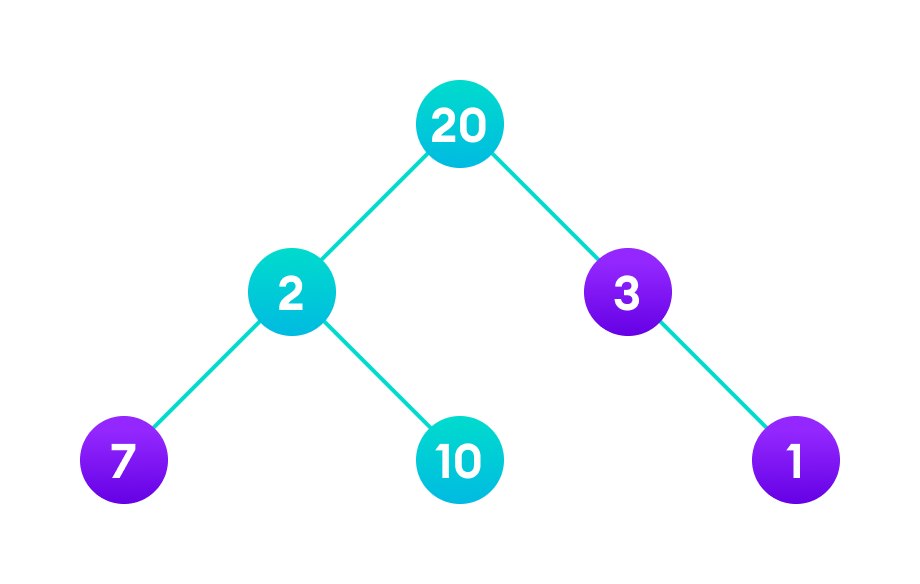
**Greedy Approach**

1. Let's start with the root node **20**. The weight of the right child is **3** and the weight of the left child is **2**.

2. Our problem is to find the largest path. And, the optimal solution at the moment is **3**. So, the greedy algorithm will choose **3**.

3. Finally the weight of an only child of **3** is **1**. This gives us our final result 20 + 3 + 1 = 24.

However, it is not the optimal solution. There is another path that carries more weight (20 + 2 + 10 = 32) as shown in the image below.

Longest path

Therefore, greedy algorithms do not always give an optimal/feasible solution.

## Greedy Algorithm

1. To begin with, the solution set (containing answers) is empty.
2. At each step, an item is added to the solution set until a solution is reached.
3. If the solution set is feasible, the current item is kept.
4. Else, the item is rejected and never considered again.

Let's now use this algorithm to solve a problem.

## Example - Greedy Approach

Problem: You have to make a change of an amount using the smallest possible number of coins.

Amount: $18

Available coins are

$5 coin

$2 coin

$1 coin

There is no limit to the number of each coin you can use.

**Solution:**

1. Create an empty solution-set = { }. Available coins are {5, 2, 1}.
2. We are supposed to find the sum = 18. Let's start with sum = 0.
3. Always select the coin with the largest value (i.e. 5) until the sum > 18. (When we select the largest value at each step, we hope to reach the destination faster. This concept is called **greedy choice property**.)
4. In the first iteration, solution-set = {5} and sum = 5.
5. In the second iteration, solution-set = {5, 5} and sum = 10.
6. In the third iteration, solution-set = {5, 5, 5} and sum = 15.
7. In the fourth iteration, solution-set = {5, 5, 5, 2} and sum = 17. (We cannot select 5 here because if we do so, sum = 20 which is greater than 18. So, we select the 2nd largest item which is 2.)
8. Similarly, in the fifth iteration, select 1. Now sum = 18 and solution-set = {5, 5, 5, 2, 1}.

## Different Types of Greedy Algorithm

* [Selection Sort](https://www.programiz.com/dsa/selection-sort)
* [Knapsack Problem](https://en.wikipedia.org/wiki/Knapsack_problem)
* [Minimum Spanning Tree](https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree)
* [Single-Source Shortest Path Problem](https://en.wikipedia.org/wiki/Shortest_path_problem)
* Job Scheduling Problem
* [Prim's Minimal Spanning Tree Algorithm](https://www.programiz.com/dsa/prim-algorithm)
* [Kruskal's Minimal Spanning Tree Algorithm](https://www.programiz.com/dsa/kruskal-algorithm)
* [Dijkstra's Minimal Spanning Tree Algorithm](https://www.programiz.com/dsa/dijkstra-algorithm)
* [Huffman Coding](https://www.programiz.com/dsa/huffman-coding)
* [Ford-Fulkerson Algorithm](https://www.programiz.com/dsa/ford-fulkerson-algorithm)