# Introduction

## What Is Algorithm?

Let us consider the problem of preparing an omelet dish. To do that , we follow the steps given below:

1. Get the frying pan.
2. Get the oil.
   1. Do we have oil?
3. If yes, put it in the pan.
4. If no, do we want to buy oil?
5. If yes, then go out and buy.
6. If no, we can terminate.
7. Turn on the stove, etc...

What we are doing is are providing a step-by-step procedure for solving it.

Formal definition of an algorithm: **An algorithm is the step-by-step unambiguous instructions to solve a given problem.**

## What Is Analysis of Algorithms?

Multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort, etc.). Algorithm analysis helps us **determine which algorithm is most efficient in terms of time and memory consumed**.

## How to Compare Algorithms?

Do you think following measures are good to compare algorithms?

* ~~Execution times~~? Not a good measure as they are specific to a particular computer.
* ~~Number of statements executed~~? Not a good measure as it varies with the programming language and the style of the individual programmer.

Ideal solution? Let us assume that we express the running time of a given algorithm as a function of the input size n (f(n)) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc.

## What Is Rate of Growth?

The rate at which the **running time increases as a function of input** is called *rate of growth*.

Below is the list of growth rates you will come across in the following chapters:

|  |  |  |
| --- | --- | --- |
| **Time Complexity** | **Name** | **Example** |
| 1 | Constant | Adding an element to the front of a linked list |
| logn | Logarithmic | Finding an element in a sorted array |
| n | Linear | Finding an element in a unsorted array |
| lognlogn |  |  |
| nlogn | Linear Logarithmic | Sorting n items by ‘Divide and Conquer’ |
| n2 | Quadratic | Shortest path between 2 nodes in a graph |
| n3 | Cubic | Matrix Multiplication |
| 2n | Exponential | The Towers of Hanoi problem |

## How Many Types of Analysis?

There are three types of analysis:

1. **Worst case**

* Defines the input for which the algorithm takes **the longest time to complete**.

1. **Best case**

* Defines the input for which the algorithm takes **the fastest time to complete**.

1. **Average case**

* Provides a prediction about the running time of the algorithm.
* Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
* Assumes that the input is random.

Lower Bound <= Average Time <= Upper Bound

## Notation

### Big-O Notation

This notation gives the tight upper bound of the given algorithm and we represent it as f(n) = O(g(n)).

For example, if f(n) = n4 + 100n2 + 10n + 50 is the given algorithm, then g(n) is n4 => O(n4).

**O(1)**

Time complexity of a function (or set of statements) is considered as O(1) if it doesn’t contain loop, recursion and call to any other non-constant time function. For example:

// set of non-recursive and non-loop statements

A loop or recursion that runs a constant number of times is also considered as O(1). For example:

// Here c is a constant

for (int i = 1; i <= c; i++) {

    // some O(1) expressions

}

**O(n)**

Time complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example:

// Here c is a positive integer constant

for (int i = 1; i <= n; i += c) {

    // some O(1) expressions

}

**O(nc)**

Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example – the following sample loops have O(n2) time complexity:

for (int i = 1; i <= n; i += c) {

    for (int j = 1; j <=n; j += c) {

        // some O(1) expressions

    }

}

**O(logn)**

Time complexity of a loop is considered as O(logn) if the loop variables is divided / multiplied by a constant amount. For example:

for (int i = 1; i <= n; i \*= c) {

    // some O(1) expressions

}

**O(loglogn)**

Time complexity of a loop is considered as O(loglogn) if the loop variables is reduced / increased exponentially by a constant amount. For example:

// Here c is a constant greater than 1

for (int i = 2; i <= n; i = pow(i, c)) {

    // some O(1) expressions

}

### Omega-Ω Notation

This notation gives the tight lower bound of the given algorithm and we represent it as f(n) = Ω(g(n)).

For example, if f(n) = 100n2 + 10n + 50 is the given algorithm, then g(n) is n2 => Ω(n2).

### Theta-Θ Notation

This notation decides whether the upper and lower bounds of a given algorithm are the same.

IMPORTANT NOTE:

In the remaining chapters, we generally **focus on the upper bound (O) because knowing the lower bound (Ω) of an algorithm is of no practical importance**, and we use the Θ notation if the upper bound (O) and lower bound (Ω) are the same.

# Supported Tools

## Visualization

Data Structures and Algorithms Visualization Tool: <https://csvistool.com/>

Tree Drawing and Visualization Tool: <https://tree-visualizer.netlify.app/>

Graph Drawing Tool: <https://csacademy.com/app/graph_editor/>

# Recursion

<https://www.youtube.com/watch?v=ngCos392W4w&ab_channel=Reducible>

## One Function Call

Example:

#include <iostream>

using namespace std;

void func(int n) {

if (n > 2) {

func(n - 1);

}

cout << n << endl;

}

int main() {

func(5);

return 0;

}

Output:

|  |  |
| --- | --- |
|  | 2  3  4  5 |

## Multiple Function Calls

Example:

#include <iostream>

using namespace std;

void func(int n) {

if (n > 2) {

func(n - 1);

func(n - 2);

func(n - 3);

}

cout << n << endl;

}

int main() {

func(5);

return 0;

}

Output:

|  |  |
| --- | --- |
|  | 2  1  0  3  2  1  4  2  1  0  3  2  5 |

Explanation:

<https://www.youtube.com/watch?v=0nKIr3kAt-k&ab_channel=GinaSprint> (from 4:18 to 12:35)

<https://www.youtube.com/watch?v=B3U6LExgevE&ab_channel=BytebyByte>

# Backtracking

# Dynamic Programming (DB)

<https://www.geeksforgeeks.org/dynamic-programming/>

<https://www.youtube.com/watch?v=aPQY__2H3tE&list=PLpXOY-RxVRTM_-Lvss2ezy1lVl6VUrzW2&index=3&ab_channel=Reducible>

## What Is DP?

DP is an algorithmic technique that solves complex problems by breaking them down into simpler subproblems. By solving each subproblem only once and storing the results, it avoids redundant computations, leading to more efficient solutions for a wide range of problems.

## How Does DP Work?

* **Identify subproblems**: Divide the main problem into smaller, independent subproblems.
* **Store solutions**: Solve each subproblem and store the solution in a table or array.
* **Build up solutions**: Use the stored solutions to build up the solution to the main problem.
* **Avoid redundancy**: By storing solutions, DP ensures that each subproblem is solved only once, reducing computation time.

## Examples of DP

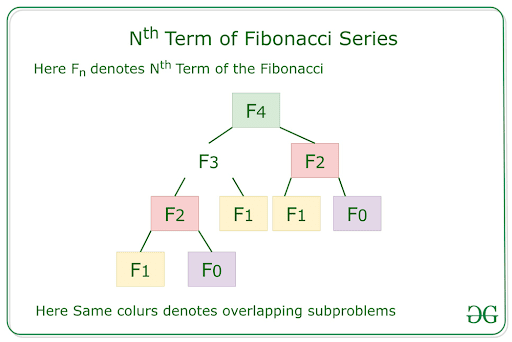
**Example 1:**Consider the problem of finding the Fibonacci sequence:

***Fibonacci sequence:****0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …*

**Brute Force Approach:**

To find the nth Fibonacci number using a brute force approach, you would simply add the **(n-1)th**and**(n-2)th** Fibonacci numbers. This would work, but it would be inefficient for large values of **n**, as it would require calculating all the previous Fibonacci numbers.

**Dynamic Programming Approach:**



Fibonacci series using DP:

* **Subproblems:**F(0), F(1), F(2), F(3), …
* **Store solutions:** Create a table to store the values of F(n) as they are calculated.
* **Build up Solutions:** For F(n), look up F(n-1) and F(n-2) in the table and add them.
* **Avoid redundancy:**The table ensures that each subproblem (e.g., F(2)) is solved only once.

By using DP, we can efficiently calculate the Fibonacci sequence without having to recompute subproblems.

## When to Use DP?

DP is an optimization technique used when solving problems that consists of the following characteristics:

1. **Optimal substructure**: Optimal substructure means that we combine the optimal results of subproblems to achieve the optimal result of the bigger problem.

*Example:*

*Consider the problem of finding the minimum cost path in a weighted graph from a source node to a destination node. We can break this problem down into smaller subproblems:*

*Find the minimum cost path from the source node to each intermediate node.*

*Find the minimum cost path from each intermediate node to the destination node.*

*The solution to the larger problem (finding the minimum cost path from the source node to the destination node) can be constructed from the solutions to these smaller subproblems.*

2. **Overlapping subproblems**: The same subproblems are solved repeatedly in different parts of the problem.

*Example:*

*Consider the problem of computing the Fibonacci series. To compute the Fibonacci number at index n, we need to compute the Fibonacci numbers at indices n-1 and n-2. This means that the subproblem of computing the Fibonacci number at index n-1 is used twice in the solution to the larger problem of computing the Fibonacci number at index n.*

## DP Algorithms

* Fibonacci Sequence: Calculates the nth Fibonacci number.
* Longest Common Subsequence (LCS): Finds the longest common subsequence between two strings.
* Shortest Path in a Graph: Finds the shortest path between two nodes in a graph.
* Knapsack Problem: Determines the maximum value of items that can be placed in a knapsack with a given capacity.
* Matrix Chain Multiplication: Optimizes the order of matrix multiplication to minimize the number of operations.

# Greedy

## What Is Greedy?

Greedy algorithms are a **class of algorithms that make locally optimal choices at each step with the hope of finding a global optimum solution**.

### It operates on the principle of “taking the best option now” without considering the long-term consequences.

## How Does Greedy Work?

* Define the problem: Clearly state the problem to be solved and the objective to be optimized.
* Identify the greedy choice: Determine the locally optimal choice at each step based on the current state.
* Make the greedy choice: Select the greedy choice and update the current state.
* Repeat: Continue making greedy choices until a solution is reached.

## Examples of Greedy

## When To Use Greedy?

Below are some applications of Greedy Algorithm:

* Assigning tasks to resources to minimize waiting time or maximize efficiency.
* Selecting the most valuable items to fit into a knapsack with limited capacity.
* Dividing an image into regions with similar characteristics.
* Reducing the size of data by removing redundant information.

Below are some disadvantages of greedy algorithm:

* Greedy algorithms may not always find the best possible solution.
* The order in which the elements are considered can significantly impact the outcome.
* Greedy algorithms focus on local optimizations and may miss better solutions that require considering a broader context.
* Greedy algorithms are not applicable to problems where the greedy choice does not lead to an optimal solution.

## Greedy Algorithms

* Fractional Knapsack: Optimizes the value of items that can be fractionally included in a knapsack with limited capacity.
* Dijkstra’s algorithm: Finds the shortest path from a source vertex to all other vertices in a weighted graph.
* Kruskal’s algorithm: Finds the minimum spanning tree of a weighted graph.
* Huffman coding: Compresses data by assigning shorter codes to more frequent symbols.

# Searching

## Linear Search – O(n)

**Problem**

Given an array arr[] of n elements, write a function to search a given element x in arr[].

**Examples**

Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}; x = 110;

Output: 6 -> Element x is at index 6

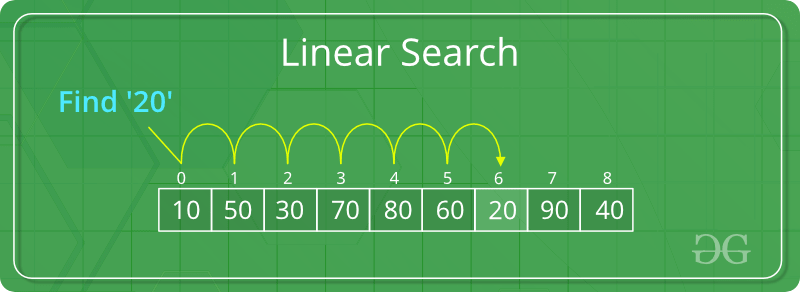
Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}; x = 175;

Output: -1 -> Element x is not in arr[].

**Algorithm**

Start from the leftmost element of arr[] and one by one compare x with each element of arr[]:

1. If x matches with an element, return the index.
2. If x doesn’t match with any of elements, return -1.



**Time Complexity**:O(n)

**Code**

<https://www.geeksforgeeks.org/linear-search/>

## Binary Search – O(1) or O(logn)

**Problem**

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

**Examples**

Input: arr[] = {10, 20, 30, 40, 50, 60}; x = 20;

Output: 1 -> Element x is at index 1

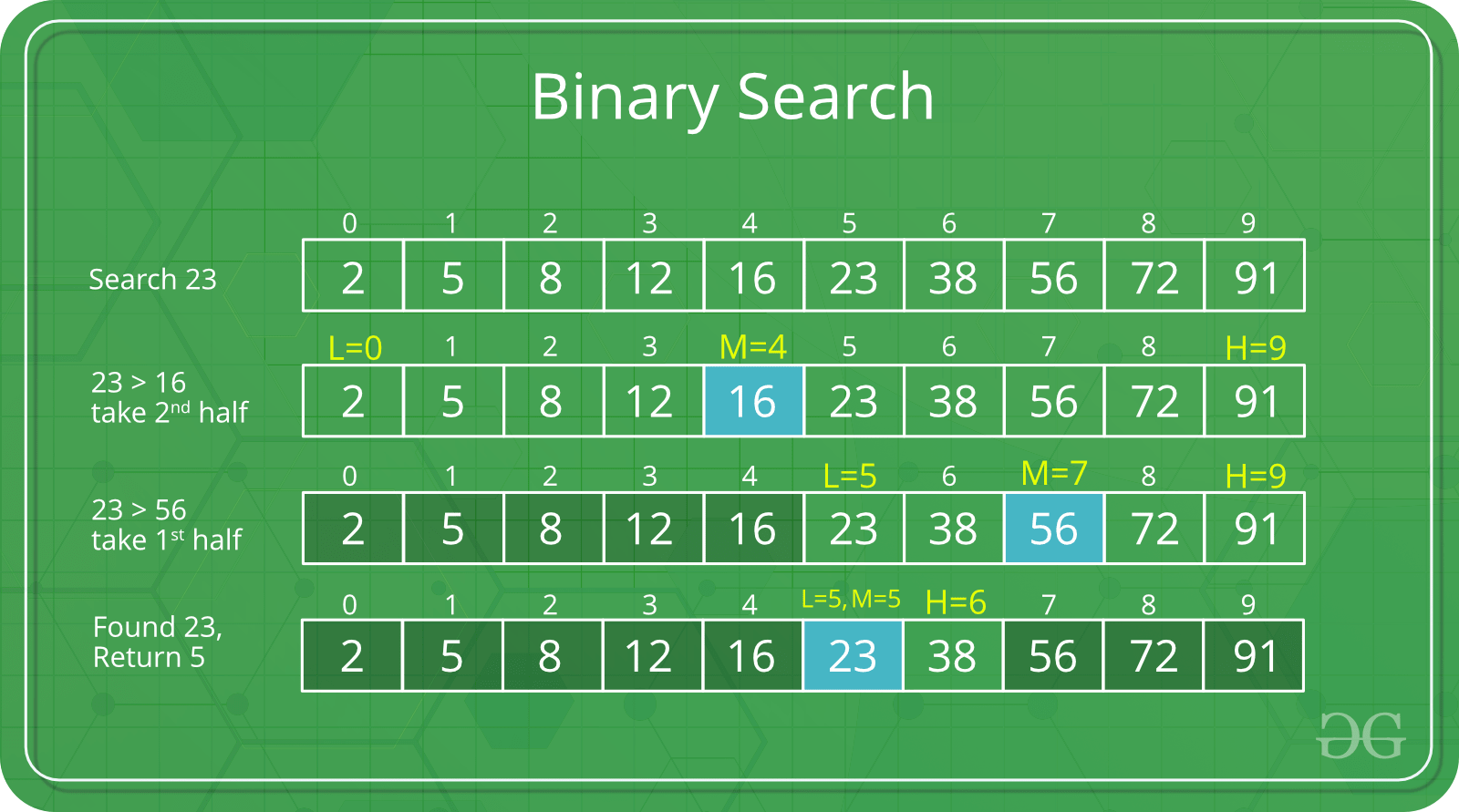
Input: arr[] = {10, 20, 30, 40, 50, 60}; x = 70;

Output: -1 -> Element x is not in arr[]

**Algorithm**

Compare x with the middle element in the array:

1. If x matches with middle element, return the mid index.
2. Else:
   1. If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
   2. Else (x is smaller) recur for the left half.



**Time Complexity**: O(1) if using iteration. Or O(logn) if using recursion.

**Code**

<https://www.geeksforgeeks.org/binary-search/>

## Ternary Search

## Jump Search – O(√n)

## Interpolation Search

## Exponential Search

## Fibonacci Search

## The Ubiquitous Binary Search

# Sorting

Visualization of the most famous sorting algorithms: <https://www.toptal.com/developers/sorting-algorithms>

## Selection Sort – O(n2)

**Problem**

Given an array arr[] of n elements, write a function to sort this array in an ascending order.

**Algorithm**

Repeatedly find the minimum element (considering ascending order) from unsorted part. Each time, the found element from the unsorted array is picked and moved to the sorted subarray (one by one, after each other).

The algorithm maintains two subarrays in a given array.

1. The subarray which is already sorted.
2. The remaining subarray which is unsorted.

Example: arr[] = { 64 25 12 11 }

// Find the minimum element in { 64 25 12 11 } and place it at the beginning of { 64 25 12 11 }:

**11** 25 12 64

// Find the minimum element in { 25 12 64 } and place it at the beginning of { 25 12 64 }:

11 **12** 25 64

// Find the minimum element in { 25 64 } place it at the beginning of { 25 64 }:

11 12 **25** 64

**Time Complexity**: O(n2) as there are two nested loops.

**Code**

<https://www.geeksforgeeks.org/selection-sort/>

#include <bits/stdc++.h>

using namespace std;

void selectionSort(int arr[], int n)

{

    int i, j, min\_idx;

    for (i = 0; i < n - 1; i++) {

        // Find the minimum element in unsorted array

        min\_idx = i;

        for (j = i + 1; j < n; j++) {

            if (arr[j] < arr[min\_idx])

                min\_idx = j;

        }

        // Swap the found minimum element with the first element

        if (min\_idx != i)

            swap(arr[min\_idx], arr[i]);

    }

}

void printArray(int arr[], int size)

{

    for (int i = 0; i < size; i++) {

        cout << arr[i] << " ";

    }

}

int main()

{

    int arr[] = { 64, 25, 12, 22, 11 };

    int n = sizeof(arr) / sizeof(arr[0]);

    selectionSort(arr, n);

    cout << "Sorted array: \n";

    printArray(arr, n);

    return 0;

}

## Bubble Sort – O(n2)

**Problem**

Given an array arr[] of n elements, write a function to sort this array in an ascending order.

**Algorithm**

Repeatedly swap adjacent elements in the array if they are in wrong order.

**Example**:

First pass:

{ **5 1** 4 2 8 } –> { **1 5** 4 2 8 }, compares the first two elements and swaps because 5 > 1.

{ 1 **5 4** 2 8 } –> { 1 **4 5** 2 8 }, swaps because 5 > 4

{ 1 4 **5 2** 8 } –> { 1 4 **2 5** 8 }, swaps because 5 > 2

{ 1 4 2 **5 8** } –> { 1 4 2 **5 8** }, because these elements are already in order, does not swap them.

Second pass:

{ **1 4** 2 5 8 } –> { **1 4** 2 5 8 }

{ 1 **4 2** 5 8 } –> { 1 **2 4** 5 8 }, swap because 4 > 2

{ 1 2 **4 5** 8 } –> { 1 2 **4 5** 8 }

{ 1 2 4 **5 8** } –> { 1 2 4 **5 8** }

Now, the array is already sorted, but our algorithm does not know if it is completed. It needs one more pass without any swap to know it is sorted.

Third pass:

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

**Time Complexity**: O(n2)

**Code**

<https://www.geeksforgeeks.org/bubble-sort/>

#include <bits/stdc++.h>

using namespace std;

void bubbleSort(int arr[], int n)

{

    int i, j;

    bool swapped;

    for (i = 0; i < n - 1; i++) {

        swapped = false;

        for (j = 0; j < n - i - 1; j++) {

            if (arr[j] > arr[j + 1]) {

                swap(arr[j], arr[j + 1]);

                swapped = true;

            }

        }

        if (swapped == false)

            break;

    }

}

void printArray(int arr[], int size)

{

    int i;

    for (i = 0; i < size; i++)

        cout << " " << arr[i];

}

int main()

{

    int arr[] = { 64, 34, 25, 12, 22, 11, 90 };

    int n = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, n);

    cout << "Sorted array: \n";

    printArray(arr, n);

    return 0;

}

## Quick Sort - O(nLogn)

**Problem**

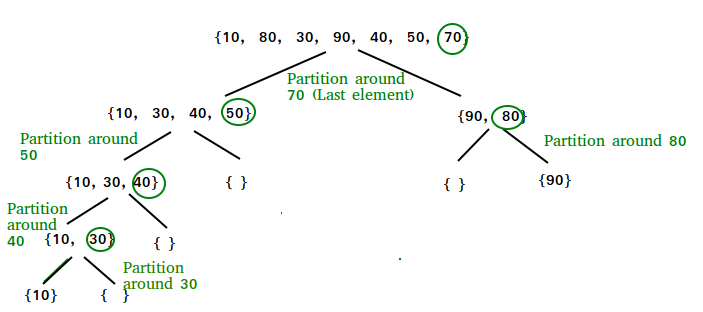
Given an array arr[] of n elements, write a function to sort this array in an ascending order.

**Algorithm**

Picks an element as pivot and partitions the array around that pivot. The pivot can be picked in different ways:

* Always pick first element as pivot.
* Always pick last element as pivot (illustration below).
* Pick a random element as pivot.
* Pick median as pivot.

The key process in quick sort is partition: put all smaller elements (than the pivot) before the pivot (if ascending order), and put all greater elements (than the pivot) after the pivot. All this should be done in linear time. Then place the pivot at its correct place.



For example:

arr[] = {10, 80, 40, 90, 30, 50, 70}

Index: 0 1 2 3 4 5 6

start = 0, end = 6, pivot = arr[end] = 70

Index of smaller element: i = -1

Traverse elements from j = start to end-1

j = 0 : Because arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 0, arr[] = {10, 80, 40, 90, 30, 50, 70} // No change as i and j are same

j = 1 : Because arr[j] > pivot, do nothing

j = 2 : Because arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 1, arr[] = {10, **40**, **80**, 90, 30, 50, 70} // We swapped 80 and 40

j = 3 : Because arr[j] > pivot, do nothing

j = 4 : Because arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2, arr[] = {10, 40, **30**, 90, **80**, 50, 70} // We swapped 80 and 30

j = 5 : Because arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i = 3, arr[] = {10, 40, 30, **50**, 80, **90**, 70} // We swapped 90 and 50

Loop ends because j = end-1.

We place pivot at correct position by swapping arr[i+1] and arr[end] (or pivot)

arr[] = {10, 40, 30, 50, **70**, 90, **80**} // We swapped 80 and 70

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

But we have not done sorting. For each sub-array {10, 40, 30, 50} and {90, 80}, we partition again (and might again and again …) until start > end.

The final result will be arr[] = {10, 30, 40, 50, 70, 80, 90}.

**Time Complexity**

Worst case: Θ(n2)

Best case: Θ(nLogn)

**Code**

<https://www.geeksforgeeks.org/quick-sort/>

#include <bits/stdc++.h>

using namespace std;

int partition(int arr[], int low, int high)

{

    // Choose the rightmost element as pivot

    int pivot = arr[high];

    // Index of smaller element and indicates the right position of pivot found so far

    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {

        // If current element is smaller than the pivot

        if (arr[j] < pivot) {

            i++;

            swap(arr[i], arr[j]);

        }

    }

    swap(arr[i + 1], pivot);

    return i + 1;

}

// low --> Starting index,

// high --> Ending index

void quickSort(int arr[], int low, int high)

{

    if (low < high) {

        // Partition the array and get the partitioning index

        int pi = partition(arr, low, high);

        // Recursively sort the sub-arrays

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

void printArray(int arr[], int size)

{

    int i;

    for (i = 0; i < size; i++)

        cout << " " << arr[i];

}

int main()

{

    int arr[] = { 10, 7, 8, 9, 1, 5 };

    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);

    cout << "Sorted array: " << endl;

    printArray(arr, n);

    return 0;

}

## Merge Sort - O(nLogn)

**Problem**

Given an array arr[] of n elements, write a function to sort this array in an ascending order.

**Algorithm**

Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

Example: arr[] = {38, 27, 43, 10}

Dividing step:

1st dividing: {38, 27, 43, 10} -> **{38, 27} and {43, 10}**

2nd dividing: {38, 27} -> **{38} and {27}**; {43, 10} -> **{43} and {10}**

Stop dividing because now no longer be divided more.

Merging step:

Sorted subarrays are merged together: {27, 38}, {10, 43}

Continuing mergeing until the sorted array is built from the smaller subarrays.: {10, 27, 38, 43}

**Time Complexity**: O(nLogn)

**Code**

<https://www.geeksforgeeks.org/merge-sort/>

#include <iostream>

using namespace std;

// Merges two subarrays of array[].

// First subarray is arr[begin..mid]

// Second subarray is arr[mid+1..end]

void merge(int array[], int left, int mid, int right)

{

    int subLeftArrLength = mid - left + 1;

    int subRightArrLength = right - mid;

    // Create temp arrays

    int\* leftArr = new int[subLeftArrLength];

    int\* rightArr = new int[subRightArrLength];

    // Copy data to temp arrays

    for (int i = 0; i < subLeftArrLength; i++)

        leftArr[i] = array[left + i];

    for (int j = 0; j < subRightArrLength; j++)

        rightArr[j] = array[mid + 1 + j];

    int subArr1Idx = 0;

    int subArr2Idx = 0;

    int mergedArrIdx = left;

    // Sort and merge the temp arrays back into array

    while (subArr1Idx < subLeftArrLength && subArr2Idx < subRightArrLength) {

        if (leftArr[subArr1Idx] <= rightArr[subArr2Idx]) {

            array[mergedArrIdx] = leftArr[subArr1Idx];

            subArr1Idx++;

        }

        else {

            array[mergedArrIdx] = rightArr[subArr2Idx];

            subArr2Idx++;

        }

        mergedArrIdx++;

    }

    // Copy the remaining elements of left[], if there are any

    while (subArr1Idx < subLeftArrLength) {

        array[mergedArrIdx] = leftArr[subArr1Idx];

        subArr1Idx++;

        mergedArrIdx++;

    }

    // Copy the remaining elements of right[], if there are any

    while (subArr2Idx < subRightArrLength) {

        array[mergedArrIdx] = rightArr[subArr2Idx];

        subArr2Idx++;

        mergedArrIdx++;

    }

    delete[] leftArr;

    delete[] rightArr;

}

void mergeSort(int array[], int left, int right)

{

    if (left >= right)

        return;

    // calculate the middle index of the array being divided

    //  (right - left): length of the subarray

    //  ((right - left) / 2): middle index within this subarray

    int mid = left + ((right - left) / 2);

    mergeSort(array, left, mid);

    mergeSort(array, mid + 1, right);

    merge(array, left, mid, right);

}

void printArray(int arr[], int size)

{

    for (int i = 0; i < size; i++)

        cout << arr[i] << " ";

}

int main()

{

    int arr[] = { 12, 11, 13, 5, 6, 7 };

    int n = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, n - 1);

    cout << "Sorted array is:\n";

    printArray(arr, n);

    return 0;

}

## Heap Sort - O(nLogn)

**Problem**

Given an array arr[] of n elements, write a function to sort this array in an ascending order.

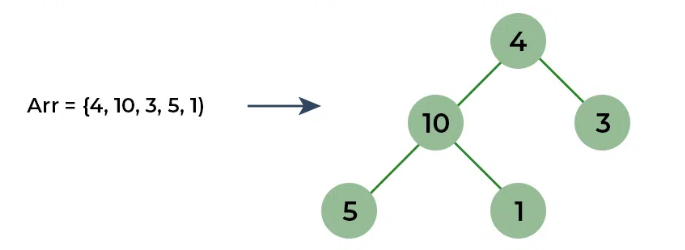
**Algorithm**

Steps:

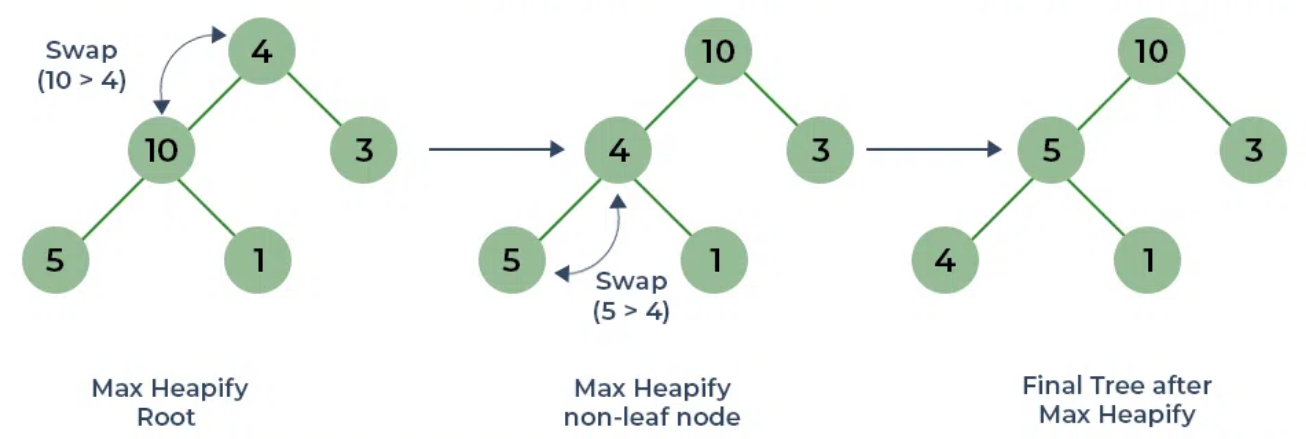
1. **Build heap**: Build a max heap from the input array.
   * Starting from the last non-leaf node, heapify each node in reverse order. Heapify is the process of adjusting the heap to maintain the max heap property.
   * Heapify compares the node with its children and swaps the node with the larger child if necessary. This process is recursively applied to the affected child until the max heap property is satisfied.
2. **Heapify and extract**: After building the max heap, the largest element (at the root) is in the correct position. We swap it with the last element of the heap (which is the last element of the array) and reduce the size of the heap by 1.
   * Swap the root with the last element of the heap.
   * Reduce the heap size by 1.
   * Heapify the root element to restore the max heap property.
3. **Repeat**: Repeat step 2 until the heap size is 1. This process moves the largest elements to the end of the array in ascending order.
4. **Sorted array**: The array is now sorted in ascending order.

Example:

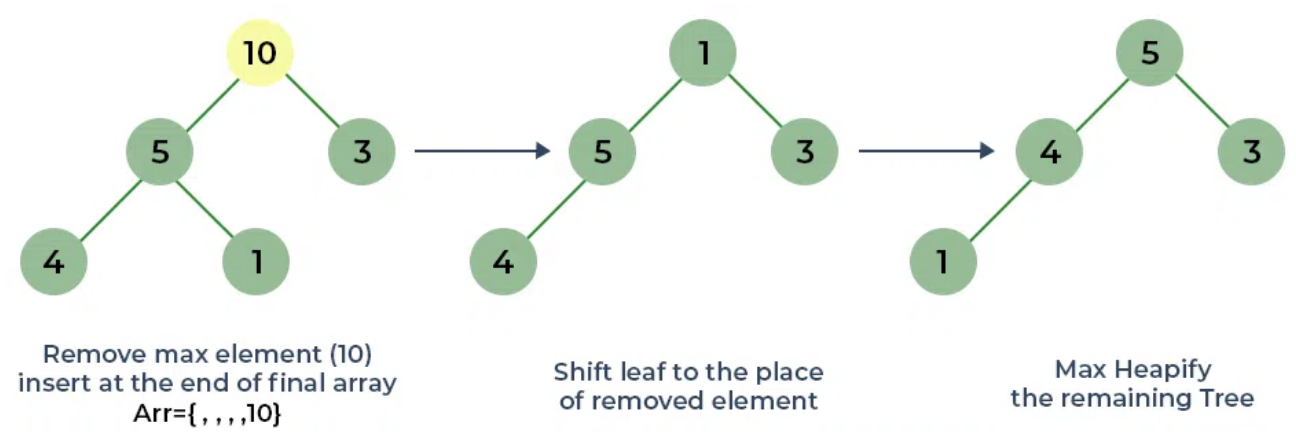
1. Initial array: {4, 10, 3, 5, 1}
2. Build a binary tree:



1. Build max heap:
   * Convert the array into a max heap: [10, 5, 3, 4, 1]



1. Heapify and extract:
   * Swap the root (10) with the last element (1): [**1**, 5, 3, 4, **10**]
   * Reduce the heap size by 1.
   * Heapify the root (1) to restore the max heap property: [5, 4, 3, 1, 10]



1. Heapify and extract:
   * Swap the root (5) with the last element (1): [1, 4, 3, 5, 10]
   * Reduce the heap size by 1.
   * Heapify the root (1) to restore the max heap property: [4, 1, 3, 5, 10]
2. Heapify and extract:
   * Swap the root (4) with the last element (3): [3, 1, 4, 5, 10]
   * Reduce the heap size by 1.
   * Heapify the root (3) to restore the max heap property: [1, 3, 4, 5, 10]
3. Heapify and extract:
   * Swap the root (1) with the last element (1): [1, 3, 4, 5, 10]
   * Reduce the heap size by 1.
   * Heapify the root (1) to restore the max heap property: [3, 1, 4, 5, 10]
4. Heapify and extract:
   * Swap the root (3) with the last element (1): [1, 1, 4, 5, 10]
   * Reduce the heap size by 1.
   * Heapify the root (1) to restore the max heap property: [1, 1, 4, 5, 10]
5. The heap size is now 1, and the array is sorted: [1, 1, 4, 5, 10]

The final sorted array is [1, 1, 4, 5, 10].

**Time Complexity**

**Code**

# Page Replacement

In OS using paging for memory management, page replacement algorithms help **decide which page needed to be replaced** when there is a new page request, but there is not enough space in the main memory to allocate the new page.

Whenever a new page is referred and not present in memory, *page fault* occurs and OS replaces one of the existing pages with the new page.

Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

For details about paging technique, check Personal\Tutorials\Embedded Systems\Embedded System Tutorial.docx

## FIFO (First In First Out)

This is the simplest page replacement algorithm. In this algorithm, OS keeps track of all pages in the memory in a queue, the **oldest page is in the front of the queue, which will be selected for replaced**.

For example: Consider the page references 1, 3, 0, 3, 5, 6, 3 with 3 page frames:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 0 | 3 | 5 | 6 | 3 |
|  |  | 0 | 0 | 0 | 0 | 3 |
|  | 3 | 3 | 3 | 3 | 6 | 6 |
| 1 | 1 | 1 | 1 | 5 | 5 | 5 |
| Miss | Miss | Miss | Hit | Miss | Miss | Miss |

**Explanation:**

* Initially all slots are empty.
* When 1, 3, 0 came, they are allocated to the empty slots —> 3 Page Faults.
* When 3 comes, it is already in memory —> 0 Page Faults.
* When 5 comes, it is not available in  memory —> replaces the oldest page 1 —> 1 Page Fault.
* When 6 comes, it is not available in memory —> replaces the oldest page 3 —> 1 Page Fault.
* Finally, when 3 come, it is not available —> replaces the oldest page slot 0 —> 1 Page Fault.

**Algorithm:**

Input:

* Memory capacity (number of pages the memory can hold): N
* A string representing pages to be referred (stored in a list): L[i]

Output:

* Find number of page faults: PF
* Find the final queue which holding all pages after all operations: Q

Steps:

I - Start traversing the pages in a loop (the looping times = the number of requested pages).

1. If current page is present in list L, do nothing:

2. Else:

If list L has less pages than N:

a) Insert new page into list L.

b) Increase page fault PF by 1.

Else:

a) Remove the first page from queue Q as it was the first to be entered in the memory

b) Insert new page into list L.

d) Increase page fault PF by 1.

II - Return page fault PF.

III - Return queue Q.

Implementation (C++):

Personal\Tutorials\Data Structures - Algorithms\Code\PageReplacementAlgorithms

## LRU (Least Recently Used)

In this algorithm, **page which is least recently used will be replaced**.

For example, consider the page reference 7, 0, 1, 2, 0, 3, 0, 4, 2 with 4 page frames:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 |
|  |  |  | 2 | 2 | 2 | 2 | 2 | 2 |
|  |  | 1 | 1 | 1 | 1 | 1 | 4 | 4 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 7 | 7 | 7 | 7 | 3 | 3 | 3 | 3 |
| Miss | Miss | Miss | Miss | Hit | Miss | Hit | Miss | Hit |

**Explanation**:

* Initially all slots are empty
* When 7 0 1 2 are allocated to the empty slots —> 4 Page Faults.
* 0 is already their —> 0 Page Fault.
* When 3 came, it will take the place of 7 because it is least recently used —> 1 Page Fault.
* 0 is already in memory —> 0 Page Fault.
* 4 will takes place of 1 —> 1 Page Fault.
* 2 is already in memory —> 0 Page Fault.

**Algorithm:**

Input:

* Memory capacity (number of pages the memory can hold): N
* A string representing pages to be referred (stored in a list): L[i]

Output:

* Find number of page faults: PF
* Find the final list which holding all pages after all operations: Q

Steps:

I - Start traversing the pages in a loop (the looping times = the number of requested pages).

1. If current page is present in list L:

a) Increase page frequency by 1

2. Else:

If list L has less pages than N:

a) Insert new page into list L.

b) Increase page frequency by 1

c) Increase page fault PF by 1.

Else:

a) Find the page which is least recently used.

b) Remove this page.

c) Insert new page into list L.

d) Increase page fault PF by 1.

II - Return page fault PF.

III - Return list L.

Implementation (C++):

Personal\Tutorials\Data Structures - Algorithms\Code\PageReplacementAlgorithms

## LFU (Least Frequently Used)

In this algorithm, **page which is least recently used will be replaced**.

For example, consider the page reference 1, 2, 3, 4, 2, 1, 5 with 3 page frames:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 2 | 1 | 5 |
|  |  | 3 | 3 | 3 | 1 | 1 |
|  | 2 | 2 | 2 | 2 | 2 | 2 |
| 1 | 1 | 1 | 4 | 4 | 4 | 5 |
| Miss | Miss | Miss | Miss | Hit | Miss | Miss |

**Explanation**:

* Initially all slots are empty
* When 1 2 3 are allocated to the empty slots —> 3 Page Faults.
* When 4 cames —> replace page 1 --> 1 Page Fault.
* 2 is already in memory —> 0 Page Fault.
* When 1 cames —> replace page 3 (because compared to page 2, page 3 is the least frequently used) --> 1 Page Fault.
* When 5 cames —> replace page 4 (because compared to page 1 and 2, page 4 is the least frequently used) --> 1 Page Fault.

**Algorithm:**

Input:

* Memory capacity (number of pages the memory can hold): N
* A string representing pages to be referred (stored in a list): L[i]

Output:

* Find number of page faults: PF
* Find the final list which holding all pages after all operations: Q

Steps:

I - Start traversing the pages in a loop (the looping times = the number of requested pages).

1. If current page is present in list L:

a) Increase page frequency by 1

2. Else:

If list L has less pages than N:

a) Insert new page into list L.

b) Increase page frequency by 1.

c) Increase page fault PF by 1.

Else:

a) Find the page which is least frequently used.

b) Remove this page.

c) Insert new page into list L.

d) Increase page fault PF by 1.

II - Return page fault PF.

III - Return list L.

Implementation (C++):

Personal\Tutorials\Data Structures - Algorithms\Code\PageReplacementAlgorithms

## OPT (Optimal Page Replacement)

In this algorithm, pages are replaced which would **not be used for the longest duration of time in the future**. This is the best page replacement algorithm as it gives the least number of page faults.

# Others

## Checking Balancing of Brackets

#### Problem-1: Discuss how stacks can be used for checking balancing of brackets.

**Solution**

Stacks can be used to check whether the given expression has balanced brackets. This algorithm is very useful in compilers. Each time the parser reads one character at a time. If the character is an opening delimiter such as (, {, or [ - then it is written to the stack. When a closing delimiter is encountered like ), }, or ] - the stack is popped.

The opening and closing delimiters are then compared. If they match, the parsing of the string continues. If they do not match, the parser indicates that there is an error on the line.

A linear-time O(n) algorithm based on stack can be given as:

1. Create a stack.
2. while (end of input is not reached) {
3. If the character read is not a bracket to be balanced, ignore it.
4. Else if the character is an opening bracket like (, [, {, push it onto the stack.
5. Else if it is a closing symbol like ), ], }
6. If the stack is empty, report an error. Otherwise pop the stack.
   * 1. If the bracket popped is not the corresponding opening bracket, report an error.

}

1. At end of input, if the stack is not empty report an error.

Time Complexity: O(n). Space Complexity: O(n) [for stack]. Since we are scanning the input only once (using one loop).