Backend

Arland Barrera

September 6, 2025

You see here kid? You gotta just go for it; don’t think about what comes after or what came before. You just gotta bend your knees, take a deep breath, and jump. And you might think; what if I fall? Well, what if you don’t? what if you fly?

Contents

Algorithms

1.1 XOR Swap

Given two values **a** and **b**, they can be swaped without the need of an temporary variable using the **xor**, exclusive or, operator. This works by changing the bits of the values. The caret symbol '^' is the most common operator for the XOR operation in many programming languages like c, c++, Java and Javascript .The process is the following:

Listing 1.1: XOR swap

```
function xorSwap(a, b) {
  if (a == b) return;
  a = a ^ b;
  b = a ^ b;
  a = a ^ b;
}
```

XOR only returns true (*1*) if the compared values are in an *or* state, otherwise returns false (*0*).

There are three steps that involve the operation between **a** and **b** using **xor**. In the first step the result is stored in a, then in b in the second and lastly in the third in a again.

This method is used in low level languages such as assembly.

Example:

a = 5 and b = 7, in binary a = 101 and b = 111.

First step, the result is stored in a:

$$a = 101$$
$$b = 111$$

$$\boxed{a = 010}$$

Second step, the result is stored in b:

$$a = 010$$
$$b = 111$$

$$\boxed{b = 101}$$

Third step, the result is stored in a again:

$$a = 010$$
$$b = 101$$

$$\boxed{a = 111}$$

a = 111 and b = 101, in other terms a = 7 and b = 5. The values have been swaped.

1.2 Sum of Arithmetic Series

The sum of an arithmetic series is is given by the expression:

$$\boxed{S_n = \frac{n}{2}\left(a_i + a_f\right)}$$

Computers process multiplications faster, so the division $n/2$ can be replaced by the product $n * 0.5$.

Elements:

- S_n = sum of series.
- n = number of terms.
- a_i = initial term.
- a_f = final term.

To find n, the formula for the n^{th} term of an arithmetic progression can be used:

$$\boxed{a_f = a_i + (n-1)d}$$

Where d is the step, the standard difference between each term.

By isolating n, the formula ends up like this:

$$\boxed{n = \left(\frac{a_f - a_i}{d}\right) + 1}$$

The algorithm has the following structure:

```
function sumArithmeticSeries(n, ai, af) {
  return n * (ai + af) / 2;
}

// if n is unknown
n = ((af - ai) /d) + 1;
```

This algorithm is $O(1)$, constant.

Example:

The sum of odd numbers (1, 3, 5, 7, ...) between 1 and 100. The range of odd values is 1-99, the first term is 1 and the last is 99. The step between eaxh term is 2, with n can be found.

$$
\begin{aligned}
n &= \left(\frac{99-1}{2}\right) + 1 \\
n &= \left(\frac{98}{2}\right) + 1 \\
n &= 49 + 1 \\
n &= 50
\end{aligned}
$$

The sum of the arithmetic series with $n = 50$ is:

$$
\begin{aligned}
S_{50} &= 50 * 0.5\,(1 + 99) \\
S_{50} &= 25 * 100 \\
S_{50} &= 2500
\end{aligned}
$$

1.3 Search

Given an array **arr**[] of **n** integers, and an integer element **x**, find whether element **x** is **present** in the array. Return the **index** of the first occurence of **x** in the array, or the invalid index **-1** if it doesn't exists.

1.3.1 Linear Search

Linear search iterates over all the elements of the array one by one and checks if the current element is equal to the target element. It is also known as **sequential search**.

1.3.1.1 Steps

- The search space begins in the first element.
- Compare the current element of the search space with a **key**.
- The search space proceeds with the next element.
- The process continues until the **key** is found or the total search space is exhausted.

1.3.1.2 Implementation

```
function linearSearch(arr, n, x) {
  for (int i = 0; i < n; i++)
    if (arr[i] == x)
      return i;
  return -1;
}
```

```

int arr = [2, 5, 65, 12, 84];
int n = arr.length;
int x = 65;
int index  = linearSearch(arr, n, x);
```

1.3.1.3 Time and Space Complexity

Time Complexity

- **Best case:** the key might be at the first index. The best case is complexity *O(1)*.
- **Worst case:** the key might be at the last index. The worst case is complexity *O(n)*, where n is the size of the array.
- **Average case:** *O(n)*.

Auxiliary Space: *O(1)* as except for the variable to iterate through the list, no other variable is used.

1.3.1.4 Advantages and Disadvantages

Advantages

- Works on sorted or unsorted arrays. It can be used on arrays of any data type.
- Does not require any additional memory.
- Well-suited for small datasets.

Dissdvantages

- Time complexity *O(n)*, which is slow for large datasets.

1.3.2 Binary Search

Binary search operates on a sorted or monotonic search space, repeatedly dividing it into halves to find a target value or optimal answer.

1.3.2.1 Steps

- The search space is divided into two halves by **finding the middle index "mid"**.
- Compare the middle element of the search space with a **key**.
- if the **key** if found at middle element, the process is terminated.
- if the **key** is not found at the middle element, choose which half will be used as the next search space.
  - if the **key** is **smaller** than the **middle element**, then the **left** side is used for the next search.
  - if the **key** is **larger** than the **middle element**, then the **right** side is used for the next search.
- The process continues until the **key** is found or the total search space is exhausted.

1.3.2.2 Implementation

This algorithm can be **iterative** or **recursive**.

Iterative

```
function iterativeBinarySearch(arr, n, x) {
  int low = 0; // first index of arr
  int high = n - 1; // last index of arr
  int mid; // middle element
  while (low <= high) {
    mid = low + (high - low) / 2;

    // if x is present at mid
    if (arr[mid] == x)
      return mid;

    // if x greater, ignore left half
    if (arr[mid] < x)
      low = mid + 1;

    // else x smaller, ignore right half
    else
      high = mid - 1;
  }
  return -1;
}

int arr = [2, 5, 8, 21, 55, 78, 93];
int n = arr.length;
int x = 78;
int index = iterativeBinarySearch(arr, n, x);
```

Recursive

```
function recursiveBinarySearch(arr, low, high, x) {
  if (high >= low) {
    mid = low + (high - low) / 2;

    // if x is present at mid
    if (arr[mid] == x)
      return mid;

    // if x smaller, ignore right half
    if (arr[mid] > x)
      return recursiveBinarySearch(arr, low, mid - 1, x);

    // else x larger, ignore left half
    return recursiveBinarySearch(arr, mid + 1, high, x);
  }
  return -1;
}

int arr = [2, 5, 8, 21, 55, 78, 93];
int n = arr.length;
int x = 78;
int index = iterativeBinarySearch(arr, 0, n -1, x);
```

1.3.2.3 Time and Space Complexity

Time Complexity

- **Best case:** *O(1)*.
- **Worst case:** *O(log n)*.
- **Average case:** *O(log n)*.

Auxiliary Space: *O(1)*, if recursive call stack is considered then the auxiliary space will be *O(lon n)*.

Iterative is faster and more secure than recursive. Recursive may cause stack overflow if recursion depth is too large (for vey big arrays).

1.3.2.4 Advantages and Disadvantages

Advantages:

- Fast search for large datasets.
- Efficient on sorted arrays..
- Low memory use, only three extra variables (low, high, mid).

Disadvantages:

- Requires sorted data.
- Less efficient on small arrays.

1.3.3 Hash Table Search

1.4 Sort

1.4.1 Bubble Sort

1.4.2 Selection Sort

1.4.3 Insertion Sort

1.4.4 Quick Sort

1.4.5 Merge Sort

1.4.6 Tim Sort

Protocols

A protocol is a set of rules for data communication in a netwotk. It allows devices to send and receive data in packets. Data packets are adressed, routed and sent across networks.

2.1 IP

IP stands for *Internet Protocol*.

There are two main versions of the Internet Protocol: the older, but still widely used, **IPv4** and the more recent and scalable **IPv6**.

An IP Adress is a unique numerical label assigned to every device, such as a computer, phone or server; that is connected to a computer network or the internet.

When information is sent online, it is broken into smaller pieces calles data packets. The IP adress acts as the "electronic return adress" for these packets, ensuring they arrive at the correct destination.

IP adresses can be public or private.

2.1.1 Public IP Adress

2.1.2 Private IP Adress

192.186.0.0

2.2 TCP

TCP stands for *Transmission Control Protocol*.

It establishes a connection between the sender and receiver before any data is sent, maintaining this connection until communication is complete.

2.3 TLS

2.4 UDP

2.5 DNS