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 Swap

1.1.1 Temporary Variable Swap

Uses a temporary variable to swap two values. Compilers optimize this method very well. Fastest and safest in precice.

Steps

- Assign a value **a** to a temporary variable.
- Assign a value **b** to **a**.
- Assign the value of the temporary variable to **b**.

Implementation

```
function temporaryVariableSwap(a, b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

Time and Space Complexity

Time Complexity: *O(1)*, 3 assignments.

Space Complexity: *O(1)*, 1 extra variable.

Advantages and Disadvantages

Advantages:

- Works for all data types.
- Compilers optimize it very well.

Disadvantages:

- Needs one extra variable (negligible cost).

1.1.2 Arithmetic Swap

Series of additions and substractions.

Steps

- Assign to **a** the sum of **a** and **b**.
- Assign to **b** the substraction of **a** and **b**.
- Assign to **a** the substraction of **a** and **b**.

Implementation

```
function arithmeticSwap(a, b) {
  a = a + b;
  b = a - b;
  a = a - b;

  // this also works
  a = a - b;
  b = a + b;
  a = b - a;
}
```

Time and Space Complexity

Time Complexity: *O(1)*, 3 arithmetic operations.

Space Complexity: *O(1)*, no extra variable.

Advantages and Disadvantages

Advantages:

- No need for an extra variable.

Disadvantages:

- Risk of overflow.
- Only works on numeric types.

1.1.3 Bitwise 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 .

XOR only returns true (*1*) if the compared values are in an *or* state, otherwise returns false (*0*). This method is used in low level languages such as assembly.

Steps

- Assign to **a** the xor operation of **a** and **b**.
- Assign to **b** the xor operation of **a** and **b**.
- Assign to **a** the xor operation of **a** and **b**.

Implementation

Listing 1.1: XOR swap

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

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.

Time and Space Complexity

Time Complexity: *O(1)*, 3 bitwise XOR operations.

Space Complexity: *O(1)*, no extra variable.

Advantages and Disadvantages

Advantages:

- No extra memory.
- Works well for integers.

Disadvantages:

- Only works on integers types
- Fails if **a** and **b** point to the same memory location.

1.1.4 Multiplication/Division Swap

Series of multiplications and divisions.

Steps

- Assign to **a** the product of **a** and **b**.
- Assign to **b** the division of **a** and **b**.
- Assign to **a** the division of **a** and **b**.

Implementation

```
function multiplicationDivisionSwap(a, b) {
  a = a * b;
  b = a / b;
  a = a / b;
}
```

Time and Space Complexity

Time Complexity: *O(1)*, 2 multiplications and 2 divisions

Space Complexity: *O(1)*, no extra variable.

Advantages and Disadvantages

Advantages:

- No need for an extra variable.

Disadvantages:

- Risk of overflow.
- Only works with non-zero integers.
- Division is slower than addition or XOR.
- Practically never used.

1.1.5 Parallel Assignment Swap

Internally creates an array, then unpacks it. Due to clarity is preferred in high level languages like Python and JavaScript. Even though it looks simultaneous, it's really a two-step process of evaluate and assign.

Steps

- Assign to **a** and **b** the opposite values.

Implementation

```
function parallelAssignmentSwap(a, b) {
  // JavaScript, via array destructuring (unpacking)
  [a, b] = [b, a]

  // Python, tuple under the hood
  // Ruby, Go
  a, b = b, a
}
```

Time and Space Complexity

Time Complexity: *O(1)*, array/object creation + assignment.

Space Complexity: *O(1)*, temporary array of size 2.

Advantages and Disadvantages

Advantages:

- Concise and safe.

Disadvantages:

- Slight overhead from creating temporary object (often negligible).

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**.

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.

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);
```

Time and Space Complexity

Time Complexity

- **Best case:** the key might be at the first index. The best case is complexity *O(1)*.
- **Average case:** *O(n)*.
- **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.

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

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.

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.

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);
```

Time and Space Complexity

Time Complexity

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

Space Complexity: *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).

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.

1.4 Sort

1.4.1 Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as it's average and wort-case time complexity are quite high.

Steps

- Checks if the current element is larger than the adjacent element and swaps them. This happens for every element.
- This is done in multiple passes.

Implementation

```
function bubbleSort(arr, n) {
  bool swapped;
  for (i = 0; i < n - 1; i++) {
    swapped = false;
    for (j = 0; j < n - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        // swap values
        swap(arr[j], arr[j + 1]);
        swapped = true;
      }
    }
    // if no elements were swapped break
    if (swapped == false)
      break;
  }
}
```

Time and Space Complexity

Time Complexity:

- **Best case:** *O(n)*.
- **Average case:** $O(n^2)$.
- **Worst case:** $O(n^2)$.

Space Complexity: *O(1)*.

Advantages and Disadvantages

Advantages:

- Easy to implement.

Disadvantages:

- It is slow for large data sets.

- it has almost no or limited real world applications.

1.4.2 Selection Sort

Selection Sort works by repeatedly selecting the smallest (or largest) element from unsorted portion and swapping it with the first unsorted element.

Steps

- Find the smallest element and swap it with the first element.
- Find the smallest among the remaining elements (or second smallest) and swap it with the second element.
- This is done until all the elements are moved to their correct positions.

Implementation

```
function selectionSort(arr, n) {
  for (int i = 0; i < n - 1; i++) {
    // current position
    int min_idx = i;
    // iterate through unsorted portion
    for (int j = i + 1; j < n; j++) {
      // update min_idx if smaller element is found
      if (arr[j] < arr[min_idx]) {
        min_idx = j;
      }
    }
    // move minimum element to correct position
    swap(arr[i], arr[min_idx]);
  }
}
```

Time and Space Complexity

Time Complexity: $O(n^2)$, as there are two nested loops.

Space Complexity: $O(1)$.

Advantages and Disadvantages

Advantages:

- Easy to implement.
- Requires less number of swaps (or memory writes) compared to many other standard algorithms.

Disadvantages:

- The time complexity of $O(n^2)$ makes it slower compared to others like **Quick Sort** or **Merge Sort**.

1.4.3 Insertion Sort

Works by iteratively inserting each element of an unsorted list into it's correct position in a sorted portion of the list. It is like sorting playing cards in your hands.

Steps

- It starts with the second element of the array as the first element is assumed to be sorted.
- Compare the second element with the first element if the second element is smaller then swap them.
- Move to the third element, compare it with the first two elements, and put it in it's correct position.
- Repeat until the entire array is sorted.

Implementation

```
function insertionSort(arr, n) {
  for (int i = 1; i < n; ++i) {
    int key = arr[i];
    int j = i - 1;
    /* move elements greater than key to one position
       ahead of their current position */
    while (j >= 0 && arr[j] > key) {
      arr[j + 1] = arr[j];
      j = j - 1;
    }
    arr[j + 1] = key;
  }
}
```

Time and Space Complexity

Time Complexity

- **Best case:** *O(n)*.
- **Average case:** $O(n^2)$.
- **Worst case:** $O(n^2)$.

Space Complexity: *O(1)*, it is a space-efficient sorting algorithm.

Advantages and Disadvantages

Advantages:

- Efficient for small and nearly sorted lists.
- Space-efficient algorithm.

Disadvantages:

- Inefficient for large lists.
- For most cases not as efficient as others like **Merge Sort** or **Quick 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

API

An **API** (*Application Programming Interface*) is a set of rules and protocols that allows different software applications to communicate with each other, share data, and use each other's features or functionalities.