

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

1	Algorithms	4
1.1	Swap	4
1.1.1	Temporary Variable Swap	4
1.1.2	Arithmetic Swap	4
1.1.3	Bitwise XOR Swap	5
1.1.4	Multiplication/Division Swap	7
1.1.5	Parallel Assignment Swap	8
1.2	Sum of Arithmetic Series	8
1.3	Search	9
1.3.1	Linear Search	10
1.3.2	Binary Search	11
1.4	Min and Max	13
1.5	Sort	14
1.5.1	Bubble Sort	14
1.5.2	Selection Sort	15
1.5.3	Insertion Sort	16
1.5.4	Merge Sort	17
1.5.5	Quick Sort	18
1.5.6	Heap Sort	21
1.5.7	Tim Sort	22
2	Protocols	26
2.1	IP	26
2.1.1	IPv4	26
2.1.2	IPv6	28
2.1.3	Public	29
2.1.4	Private	29
2.1.5	Static	29
2.1.6	Dynamic	30
2.2	TCP	31
2.3	TLS	31
2.4	UDP	31
2.5	DNS	31
3	API	32

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 practice.

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

```
1 function temporaryVariableSwap(a, b) {  
2     int tmp = a;  
3     a = b;  
4     b = tmp;  
5 }
```

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 subtractions.

Steps

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

Implementation

```
1 function arithmeticSwap(a, b) {  
2   a = a + b;  
3   b = a - b;  
4   a = a - b;  
5  
6   // this also works  
7   a = a - b;  
8   b = a + b;  
9   a = b - a;  
10 }
```

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 swapped 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

```
1 function xorSwap(a, b) {  
2   a = a ^ b;  
3   b = a ^ b;  
4   a = a ^ b;  
5 }
```

Example

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

First step, the result is stored in a :

$$\begin{aligned} a &= 101 \\ b &= 111 \end{aligned}$$

$a = 010$

Second step, the result is stored in b :

$$\begin{aligned} a &= 010 \\ b &= 111 \end{aligned}$$

$b = 101$

Third step, the result is stored in a again:

$$\begin{aligned} a &= 010 \\ b &= 101 \end{aligned}$$

$a = 111$

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

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

```
1 function multiplicationDivisionSwap(a, b) {  
2     a = a * b;  
3     b = a / b;  
4     a = a / b;  
5 }
```

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

```

1 function parallelAssignmentSwap(a, b) {
2   // JavaScript, via array destructuring (unpacking)
3   [a, b] = [b, a]
4
5   // Python, tuple under the hood
6   // Ruby, Go
7   a, b = b, a
8 }

```

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 given by the expression:

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

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:

$$a_f = a_i + (n - 1)d$$

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

By isolating n , the formula ends up like this:

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

Implementation

```

1 function sumArithmeticSeries(n, ai, af) {
2   return n * (ai + af) / 2;
3 }
4
5 // if n is unknown
6 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 each 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 occurrence of x in the array, or the invalid index **-1** if it doesn't exist.

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

```
1 function linearSearch(arr, n, x) {  
2     for (int i = 0; i < n; i++)  
3         if (arr[i] == x)  
4             return i;  
5     return -1;  
6 }  
7  
8 int arr = [2, 5, 65, 12, 84];  
9 int n = arr.length;  
10 int x = 65;  
11 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.

Disadvantages

- 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** is 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

```
1 function iterativeBinarySearch(arr, n, x) {
2   int low = 0; // first index of arr
3   int high = n - 1; // last index of arr
4   int mid; // middle element
5   while (low <= high) {
6     mid = floor(low + (high - low) / 2);
7
8     // if x is present at mid
9     if (arr[mid] == x)
10      return mid;
11
12    // if x greater, ignore left half
13    if (arr[mid] < x)
14      low = mid + 1;
15
16    // else x smaller, ignore right half
17    else
18      high = mid - 1;
19  }
20  return -1;
21 }
22
23 int arr = [2, 5, 8, 21, 55, 78, 93];
24 int n = arr.length;
25 int x = 78;
26 int index = iterativeBinarySearch(arr, n, x);
```

Recursive

```

1 function recursiveBinarySearch(arr, low, high, x) {
2   if (high >= low) {
3     mid = floor(low + (high - low) / 2);
4
5     // if x is present at mid
6     if (arr[mid] == x)
7       return mid;
8
9     // if x smaller, ignore right half
10    if (arr[mid] > x)
11      return recursiveBinarySearch(arr, low, mid - 1, x);
12
13    // else x larger, ignore left half
14    return recursiveBinarySearch(arr, mid + 1, high, x);
15  }
16  return -1;
17 }
18
19 int arr = [2, 5, 8, 21, 55, 78, 93];
20 int n = arr.length;
21 int x = 78;
22 int index = recursiveBinarySearch(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 $O(\log n)$.

Iterative is faster and more secure than recursive. Recursive may cause stack overflow if recursion depth is too large (for very 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 Min and Max

Search for maximum and minimum element in an array or list using [linear search](#).

Steps

- Traverse the array from the first index until the last comparing the current value with the next, and store the maximum value in a variable.

Implementation

```
1 // function for maximum value
2 function max(arr, n) {
3     int max = 0;
4     for (int i = 0; i < n; i++)
5         if (arr[i] > max)
6             max = arr[i];
7     return max;
8 }
9
10 // function for minimum value
11 function min(arr, n) {
12     int min = 0;
13     for (int i = 0; i < n; i++)
14         if (arr[i] < min)
15             min = arr[i];
16     return min;
17 }
18
19 // main
20 int arr = [1, 56, -23, 101, 34, 6, -11];
21 int n = arr.length;
22 // max
23 max(arr, n);
24 // min
25 min(arr, n);
```

Time and Space Complexity

Time Complexity: $O(n)$.

Space Complexity: $O(1)$.

Advantages and Disadvantages

Advantages:

- Best for small arrays.

Disadvantages:

- Inefficient for big arrays.

1.5 Sort

1.5.1 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 its average and worst-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

```
1 function bubbleSort(arr, n) {  
2     bool swapped;  
3     for (i = 0; i < n - 1; i++) {  
4         swapped = false;  
5         for (j = 0; j < n - 1; j++) {  
6             if (arr[j] > arr[j + 1]) {  
7                 // swap values  
8                 swap(arr[j], arr[j + 1]);  
9                 swapped = true;  
10            }  
11        }  
12        // if no elements were swapped break  
13        if (swapped == false)  
14            break;  
15    }  
16 }
```

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.

1.5.2 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

```
1 function selectionSort(arr, n) {  
2   for (int i = 0; i < n - 1; i++) {  
3     // current position  
4     int min_idx = i;  
5     // iterate through unsorted portion  
6     for (int j = i + 1; j < n; j++) {  
7       // update min_idx if smaller element is found  
8       if (arr[j] < arr[min_idx]) {  
9         min_idx = j;  
10      }  
11    }  
12    // move minimum element to correct position  
13    swap(arr[i], arr[min_idx]);  
14  }  
15 }
```

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](#).

1.5.3 Insertion Sort

Works by iteratively inserting each element of an unsorted list into its 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 its correct position.
- Repeat until the entire array is sorted.

Implementation

```
1 function insertionSort(arr, n) {  
2   for (int i = 1; i < n; ++i) {  
3     int key = arr[i];  
4     int j = i - 1;  
5     /* move elements greater than key to one position  
6        ahead of their current position */  
7     while (j >= 0 && arr[j] > key) {  
8       arr[j + 1] = arr[j];  
9       j = j - 1;  
10    }  
11    arr[j + 1] = key;  
12  }  
13 }
```

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](#).

1.5.4 Merge Sort

Merge Sort works by recursively dividing the input array into two halves, recursively sorting the two halves and finally merging them back together to obtain the sorted array.

Steps

- Divide the array recursively into two halves until it can no more be divided.
- Each subarray is sorted individually using the merge sort algorithm.
- The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Implementation

```
1 function merge(arr, left, mid, right) {
2     const int n1 = mid - left + 1;
3     const int n2 = right - mid;
4
5     // tmp arrays
6     const int L[n1], R[n2];
7
8     // copy data to tmp arrays L[] and R[]
9     for (int i = 0; i < n1; i++)
10        L[i] = arr[left + i];
11    for (int j = 0; j < n2; j++)
12        R[j] = arr[mid + 1 + j];
13
14    int i = 0, j = 0;
15    int k = left;
16
17    // merge tmp arrays back into arr[left...right]
18    while (i < n1 && j < n2) {
19        if (L[i] <= R[j]) {
20            arr[k] = L[i];
21            i++;
22        } else {
23            arr[k] = R[j];
24            j++;
25        }
26        k++;
27    }
28
29    // copy remaining elements of L[], if there are any
30    while (i < n1) {
31        arr[k] = L[i];
32        i++;
33        k++;
34    }
35
36    // copy remaining elements of R[], if there are any
37    while (j < n2) {
38        arr[k] = R[j];
39        j++;
40    }
```

```

40     k++;
41 }
42 }
43
44 function mergeSort(arr, left, right) {
45     if (left >= right)
46         return;
47
48     const int mid = floor(left + (right - left) / 2);
49     mergeSort(arr, left, mid);
50     mergeSort(arr, mid + 1, right);
51     merge(arr, left, mid, right);
52 }
53
54 // main code
55 const int arr = [38, 27, 43, 10];
56 mergeSort(arr, 0, arr.length - 1);

```

Time and Space Complexity

Time Complexity

- **Best case:** $O(n \log n)$, already sorted or nearly sorted.
- **Average case:** $O(n \log n)$, randomly ordered.
- **Worst case:** $O(n \log n)$, sorted in reverse order.

Space Complexity: $O(n)$, temporary array used during merging.

Advantages and Disadvantages

Advantages:

- Guaranteed worst-case performance of $O(n \log n)$.
- The divide-and-conquer is simple to implement.
- Independently merge subarrays which makes it suitable for parallel processing.

Disadvantages:

- Additional space complexity to store merged subarrays.
- Slower than QuickSort in general.

1.5.5 Quick Sort

Quick Sort picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

Steps

- Select an element as the pivot. The choice of pivot can vary (first, last, random, median).

- Re arrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on it's left, and all elements grater than the pivot will be on it's right. The pivot is then i nit's correct position, and the index of the pivot is obtained.
- Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
- The recursion stops when there is only one element left in the subarray, as a single element is already sorted.

Choice of pivot

- **First or last:** the problem with this approach is it ends up in the worst case when array is already sorted.
- **Random:** This is a preferred approach beacuse it does not have a pattern for wich the worst case happens.
- **Median:** In terms of time complexity is the ideal approach as median can be found in linear time and the partition function will always divide the input array into two halves. It takes more time on average as median finding has high constants.

Partiton Algorithm

- **Naive Partition:** Create copy of the array. First put all smaller elements and then all greater. Finally the temporary array is copied back to the original array. This requires $O(n)$ extra space.
- **Lomuto Partition:** Keep track of the index of smaller elements and keep swapping. It is simple.
- **Hoare Partition:** The fastest of all. The array is traversed from both sided and keep swapping grater element on the left with smaller on right while the array is not partitioned.

Implementation

```

1 // partition function (Lomuto)
2 function patition(arr, low, high) {
3     // random pivot
4     randpivot = random(low, high)
5     // swap random pivot and last pivot
6     swap(arr, randompivot, high)
7     // pivot, always pick last
8     int pivot = arr[high];
9
10    // idx of smaller element
11    int i = low - 1;
12
13    // traverse arr[low...high] move all smaller elements to left side
14    for (int j = low; j <= high - 1; j++) {
15        if (arr[j] < pivot) {
16            i++;
17            // swap function
18            swap(arr, i, j);
19        }
20    }
21 }
```

```

22 // move pivot after smaller elements and returns it's position
23 swap(arr, i + 1, high);
24 return i + 1;
25 }
26
27 // quicksort function
28 function quickSort(arr, low, high) {
29     if (low < high) {
30         // pi -> partition return index of pivot
31         int pi = partition(arr, low, high);
32
33         // recursion calls for smaller elements and grater or equal elements
34         quickSort(arr, low, pi - 1);
35         quickSort(arr, pi + 1, high);
36     }
37 }
38
39 // main
40 int arr = [10, 7, 8, 9, 1, 5];
41 int n = arr.length;
42 // call quickSort
43 quickSort(arr, 0, n - 1);

```

Time and Space Complexity

Time Complexity

- **Best case:** $O(n \log n)$, pivot element divides the array into two equal halves.
- **Average case:** $O(n \log n)$, pivot divides the array into two parts, but not necessarily equal.
- **Worst case:** $O(n^2)$, smaller or largest element is always chosen as the pivot (e.g. sorted arrays).

Space Complexity

- **Best case:** $O(\log n)$, as a result of balanced partitioning leading to a balanced recursion tree requiring a call stack of size $O(\log n)$.
- **Worst case:** $O(n)$, due to inbalanced partitioning leading to a skewed recursion tree requiring a call stack of size $O(n)$.

Advantages and Disadvantages

Advantages:

- Efficient on large data sets.
- Requires a small amount of memory to function.
- Cache friendly as it works on the same array.
- Fastest general purpose algorithm for large data sets when stability is required.

Disadvantages:

- Worst case time complexity of $O(n^2)$, occurs when pivot is chosen poorly.

- Inefficient for small data sets.
- Not stable, if two elements have the same key their relative order will not be preserved.

1.5.6 Heap Sort

Heap sort is based on **Binary Heap Data Structure**. It can be seen as an optimization over [selection sort](#) where the max (or min) element is first found and swapped with the last (or first). In Heap Sort, the use of Binary Heap can quickly find and move the max element in $O(\log n)$ instead of $O(n)$ and hence achieve the $O(n \log n)$ time complexity.

Heapsort algorithm has limited uses because [quick sort](#) is better in practice. Nevertheless, the **Heap Data Structure** itself is enormously used.

Steps

- Convert the array into a **max heap** using **heapify**. This happens in-place.
- One by one delete root node of the Max-heap and replace it with the last node and **heapify**. Repeat this process while size of heap is greater than 1.

Implementation

```

1 // heapify a subtree rooted with node i
2 function heapify(arr, n, i) {
3     // initialize largest as root
4     int largest = i;
5
6     // left index = 2*i + 1
7     int left = 2 * i + 1;
8
9     // right index = 2*i + 2
10    int right = 2 * i + 2;
11
12    // if left child greater than root
13    if (left < n && arr[left] > arr[largest])
14        largest = left;
15
16    // if right child greater than largest so far
17    if (right < n && arr[right] > arr[largest])
18        largest = right;
19
20    // if largest is not root
21    if (largest != i) {
22        // swap function
23        swap(arr[i], arr[largest]);
24        // recursively heapify the affected sub-tree
25        heapify(arr, n, largest);
26    }
27 }
28
29 // function for heap sort
30 function heapSort(arr, n) {
31     // build heap (rearrange array)

```

```

32     for (int i = floor(n / 2) - 1; i >= 0; i--)
33         heapify(arr, n, i);
34
35     // one by one extract element from heap
36     for (int i = n - 1; i > 0; i--) {
37         // swap function
38         swap(arr[0], arr[i]);
39         // call max heapify on the reduced heap
40         heapify(arr, i, 0);
41     }
42 }
43
44 // main
45 int arr = [9, 4, 3, 8, 10, 2, 5];
46 int n = arr.length;
47 heapSort(arr, n);

```

Time and Space Complexity

Time Complexity: $O(n \log n)$, in all cases.

Space Complexity: $O(\log n)$, due to recursive call stack.

Advantages and Disadvantages

Advantages:

- Efficient for large datasets due to time complexity.
- Simple to use.

Disadvantages:

- Costly, as the constants are higher compared to other (e.g. [merge sort](#)).
- Unstable, it might rearrange the relative order of elements.
- Inefficient because of high constants in the time complexity.

1.5.7 Tim Sort

Is a hybrid sorting algorithm derived from [merge sort](#) and [insertion sort](#). It was designed to perform well on many kinds of real-world data. Tim Sort is the default sorting algorithm used by Python's `sorted()` and `list.sort()` functions.

The main idea behind Tim Sort is to exploit the existing order in the data to minimize the number of comparisons and swaps. It achieves this by dividing the array into small subarrays called runs, which are already sorted, and then merging these runs using a modified merge sort algorithm.

Steps

- Define the size of the run. Minimum run size of 32.
- Divide the array into runs. Use the insertion sort to sort the small subsequences (runs) within the array.

- Merge the runs using a modified merge sort algorithm.
- Adjust the run size. After each merge operation, double the size of the run until it exceeds the length of the array. In small arrays this can be ignored.
- Continue merging until the array is sorted.

Implementation

```

1  int MIN_MERGE = 32;
2
3  // minimum range of run
4  function minRunLength(n) {
5      int r = 0;
6      while (n >= MIN_MERGE) {
7          r |= (n & 1);
8          n >>= 1;
9      }
10     return n + r;
11 }
12
13 // insertion sort
14 function insertionSort(arr, left, right) {
15     for (int i = left + 1; i <= right; i++) {
16         int tmp = arr[i];
17         int j = i - 1;
18         while (j >= left && arr[j] > tmp) {
19             arr[j + 1] = arr[j];
20             j--;
21         }
22         arr[j + 1] = tmp;
23     }
24 }
25
26 // merge sorted runs
27 function merge(arr, low, mid, high) {
28     // original array is broken in two parts left and right
29     int n1 = mid - low + 1;
30     int n2 = high - mid;
31
32     int left = [n1];
33     int right = [n2];
34
35     for (int i = 0; i < n1; i++)
36         left[i] = arr[low + i];
37
38     for (int j = 0; j < n2; j++)
39         right[j] = arr[mid + 1 + j];
40
41     int i = 0;
42     int j = 0;
43     int k = 0;
44
45     while (i < n1 && j < n2) {
46         if (left[i] <= right[j]) {

```

```

47     arr[k] = left[i];
48     i++;
49 } else {
50     arr[k] = right[j];
51     j++;
52 }
53 }
54
55 // copy remaining elements of left, if any
56 while (i < n1) {
57     arr[k] = left[i];
58     k++;
59     i++;
60 }
61
62 // copy remaining elements of right, if any
63 while (j < n2) {
64     arr[k] = right[j];
65     k++;
66     j++;
67 }
68 }
69
70 // Timsort to sort arr[0...n-1]
71 function timSort(arr, n) {
72     int minRun = minRunLength(MIN_MERGE);
73
74     // sort individual subarrays of size MIN_MERGE
75     for (int i = 0; i < n; i += minRun)
76         insertionSort(arr, i, min((i + MIN_MERGE - 1), (n - 1)));
77
78     // start merging from size MIN_MERGE
79     for (int size = minRun; size < n; size = 2 * size) {
80         // pick starting point of left sub array
81         for (int left = 0; ; left < n; left += 2 * size) {
82             // find ending point left sub array
83             int mid = left + size - 1;
84             int right = min((left + 2 * size - 1), (n - 1));
85             if (mid < right)
86                 merge(arr, left, mid, right);
87         }
88     }
89 }
90
91 // main
92 int arr = [-2, 7, 15, -14, 0, -13, 5, 8, -14, 12];
93 int n = arr.length;
94 // call function timSort
95 timSort(arr, n);

```

Time and Space Complexity

Time Complexity

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

Space Complexity: $O(n)$.

Advantages and Disadvantages

Advantages:

- Well suited for general purpose.
- It is stable.

Disadvantages:

- It requires extra space.

Protocols

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

Once packets arrive at their destination, they are handled differently depending on which transport protocol is used in combination with IP. A transport protocol dictates the way data is sent and received, the most common are TCP and UDP.

2.1 IP

IP stands for *Internet Protocol*, a set of rules and a unique numerical label, called an IP Address, assigned to every device on a network, such as the internet. IP addresses allow computers and other devices to send and receive data by identifying their specific location and ensuring that data packets reach the correct destination, enabling communication between different networks.

An IP address helps devices to find whatever data or content is located to allow for retrieval. Common tasks for an IP address include both the identification of a host or a network, or identifying the location of a device. An IP address is not random, its creation has the basis of math. With the mathematical assignment of an IP address, the unique identification to make a connection to a destination can be made.

1. **Data packets:** Data sent over the internet is broken down into smaller pieces called packets.
2. **IP information:** Each packet is given an IP address, acting as the "electronic return address" for these packets.
3. **Routing:** Routers read this IP information and direct the packets to the right place, allowing machines to communicate with each other even if they are on different networks.

IP addresses can be classified by:

Classification method	Types
Version or standards	IPv4, IPv6
Function	Public, Private
Assignment	Static, Dynamic

2.1.1 IPv4

The Internet Protocol version 4 (IPv4) address is the older version. It is one of the core protocols of the standards-based methods used to interconnect the internet and other networks. It was deployed on the *Atlantic Packet Satellite Network* (SATNET), which was a satellite network, in 1982. It is still used to route most internet traffic despite the existence of IPv6.

IPv4 is currently assigned to all computers. An IPv4 address uses **32-bit** binary numbers to form a unique IP address. It takes the format of four sets of numbers, each within ranges from **0** to **255** and represents an eight-digit binary number, separated by a period point.

The **full range** of IP addresses can go from **0.0.0.0** to **255.255.255.255**. The **total combination** of IP addresses is $2^{32} = 4\ 294\ 967\ 296$ (four billion, two hundred ninety-four million, nine hundred sixty-seven thousand, two hundred ninety-six)

Some IP addresses are reserved for networks that carry a specific purpose on the TCP/IP. Four of these IP address classes include:

- **0.0.0.0:** In IPv4 is also known as the **default network**. It is the non-routable meta address that designates an invalid, non-applicable, or unknown network target.
- **127.0.0.1:** Is known as the **loopback address**, also known as **localhost**, which a computer uses to identify itself regardless of whether it has been assigned an IP address.
- **169.254.0.1 to 169.254.254.254:** A range of addresses that are automatically assigned if a computer is unsuccessful in an attempt to receive an address from the DHCP.
- **255.255.255.255:** An address dedicated to messages that need to be sent to every computer on a network or **broadcasted across a network**.

Further reserved IP addresses are for what is known as **subnet classes**. A **subnet** IP is a logically smaller division within a larger IP network, created through a process called subnetting. Subnetting divides a large network into smaller, more manageable segments, or subnets, which allows for more efficient data routing, improved network performance and better security.

Each subnet is assigned a unique range of IP addresses, and a **subnet mask** is used to identify which part of an IP address belongs to the network and which part belongs to the host device within that subnet.

Subnetting

1. **IP Address Structure:** An IP address is logically divided into two parts: a network number (or routing prefix) and a host identifier.
2. **Subnet Mask:** A subnet mask is used to determine these two parts of an IP address. It's a number with a specific bit pattern, where the bits set to '1' indicate the **network portion**, and the bits set to '0' indicate the **host portion**.
3. **Logical Division:** By changing some of the bits from the host portion to the network portion, an administrator can effectively create smaller subnets from a single, larger network.

For example **255.255.0.0** in binary is **11111111.11111111.00000000.00000000**, '1' refers to the network part and '0' to the host part.

The router on a TCP/IP network can be configured to ensure it recognizes subnets, then route the traffic onto the appropriate network. IP addresses are reserved for the following subnets:

Address Class	Range	Default Subnet Mask	Number of Networks	Hosts per Network
A	1.0.0.0 126.255.255.255	255.0.0.0	$2^7 = 128$	$2^{24} - 2 = 16\,777\,214$
B	128.0.0.0 191.255.255.255	255.255.0.0	$2^{14} = 16\,384$	$2^{16} - 2 = 65\,534$
C	192.0.0.0 223.255.255.255	255.255.255.0	$2^{21} = 2\,097\,152$	$2^8 - 2 = 254$
D	224.0.0.0 239.255.255.255	Multicast		
E	240.0.0.0 254.255.255.255	Experimental		

Class A addresses **127.0.0.0** to **127.255.255.255** cannot be used and are reserved for loopback testing.

Fundamental networking formulas are based on the number of bits allocated for network and host parts in the subnet mask.

- **Number of subnets:** total subnets = 2^n , where n is the number of bits borrowed from the original host part to create more network bits in subnetting.
- **Number of usable hosts:** usable hosts = $2^h - 2$, where h is the number of bits used for hosts in the subnet mask. The subtraction of 2 accounts for the network and broadcast address, which cannot be assigned to hosts.

Key elements

- **Total bits for IPv4,** $T = 32$.
- **Network bits** = n .
- **Number of subnets** = 2^n .
- **Host bits,** $h = T - n$.
- **Usable hosts** = $2^h - 2$.

2.1.2 IPv6

IPv4 has not been able to cope with the massive explosion in the quantity and range of devices beyond simply mobile phones, desktop computers and laptops. The original IP address format was not able to handle the number of IP addresses being created.

To address this problem, IPv6 was introduced. This new standard operates a hexadecimal format, that means billions of unique IP addresses can now be created. As a result, the IPv4 system that could support up to 4.3 billion unique numbers has been replaced by an alternative that, theoretically, offers unlimited IP addresses.

This is because an IPv6 IP address consists of eight groups that contain four hexadecimal digits, which use 16 distinct symbols of 0 to 9 followed by A to F to represent values of 10 to 15. It uses a **128-bit** alphanumeric address written using a colon-separated hexadecimal notation, for example **2001:0db8:85a3:0000:0000:8a2e:0370:7334**. IPv6 also has a **shorthand notation**, for example **2001:db8:85a3::8a2e:370:7334**, this is typically used for convenience and to avoid visual clutter.

With 128-bit address space, it allows **340 undecillion** unique address space. IPv6 supports a theoretical maximum of **340 282 366 920 938 463 463 374 607 431 768 211 456**.

IPv6 is not inherently faster than IPv4 in terms of raw speed. However, it improves the efficiency of data transmission. Despite the differences, both protocols can run concurrently, allowing for seamless transitions in compatibility and support across various network infrastructures. Many modern network devices, including routers and switches, are equipped to handle both protocols.

The main reason for the little adoption of IPv6 is cost of maintenance. Manage a system that uses IPv4 and instead use IPv6 is very demanding. Though the adoption is slow and steady.

2.1.3 Public

Is a globally unique address used for devices to communicate on the internet, assign by the *Internet Service Provider (ISP)* to a home or router. Enables access to the internet and online services. It is global and unique across the internet, visible to anyone on the internet.

Example of this are a website's server or a router's IP address when accessing the internet. Outside private IP addresses, the rest are public.

Public IP addresses can be traced back to the ISP that can easily trace the geographical location. This might reveal the location very easily to advertisers, hackers, etc. For using the Internet anonymously, the IP address can be hidden by using different ways like VPN, Tor Browser, etc. But among different ways, VPN is the fastest and most secure way of using the Internet.

2.1.4 Private

Used for devices within a LAN (like a home or an office), is not accessible from the Internet, and is assigned by a router. This facilitates communication between devices on a private network, offering increased security by limiting external visibility. Examples are a computer's or smart TV's IP address when connected to a home Wi-Fi.

By using private IP addresses internally, organizations can avoid the need to obtain and manage large blocks of public IP addresses, reducing costs associated with Internet connectivity. Requires *Network Address Translation (NAT)*, which can cause overhead in terms of processing energy, latency, and complexity, in particular in big-scale deployments.

The range of private IP addresses are:

- **Class A:** 10.0.0.0 - 10.255.255.255, for large networks with many devices.
- **Class B:** 172.16.0.0 - 172.31.255.255, for medium-sized networks, such as schools or businesses.
- **Class C:** 192.168.0.0 - 192.168.255.255, for small networks, such as homes or small offices.

2.1.5 Static

A static IP address is a manually configured identifier assigned to a device that remains consistent and cannot change across multiple network sessions. Individuals do not typically need a static IP address, but businesses need them to host their own servers.

An IP address is considered to be static if the same IP address is assigned every time the user or device connects. Static IP addresses are particularly useful for enterprises that need to guarantee

server and websites uptime. They also offer reliable internet connections, quicker data exchanges, and more convenient remote access.

Advantages

- **Better online name resolution:** devices with static IP addresses can be reliably discovered and reached via their assigned hostnames and do not need to be tracked for changes. For this reason, components like *File Transfer Protocol (FTP)* servers and web servers use fixed addresses.
- **Anywhere, anytime access:** makes a device accessible anywhere in the world. Can make it quick and easy for people to locate and use shared devices, such as a printer on their network.
- **Reduced connection lapses:** connection lapses typically happen when devices are not recognized by a network. An IP address that never resets or adjusts is essential for device processing vast amounts of data.
- **Faster download and upload speed:** devices with static IP addresses enjoy higher access speeds, which is essential for heavy data users.
- **Accurate geolocation data:** provides access to precise geolocation data. More accurate data means businesses are better able to manage and log incidents in real time, as well as detect and remediate potential attacks before they cause damage to networks.

Disadvantages

- **Easy-to-track addresses:** the constant nature of static IP addresses makes it easy for people to track a device the data they access or share. This could be a security concern, giving cyber criminals a route into a machine and subsequently unauthorized access to corporate networks.
- **Post-breach difficulties:** static IP addresses increase the risk of a website being hacked. In the aftermath of a data breach, they also make it more difficult to change IP addresses, making the business more susceptible to ongoing issues.
- **Cost issues:** the cost is significantly high. Many Internet service and hosting providers require users to sign up for commercial accounts or pay one-time fees in order to assign a static IP on each of their devices and websites.

2.1.6 Dynamic

Internet Service Providers (ISPs) temporarily assign dynamic IP addresses via the *Dynamic Host Configuration Protocol (DHCP)* server. This means an IP address can change every time a user reboots their router or system, and when the user connects to their ISP server. When not in use, a dynamic IP address can be automatically assigned to another device. This makes dynamic IP addresses more suitable for home networks than large organizations.

An IP address is considered to be dynamic if it is pulled from a pool of available IP addresses by the router every time the user or device connects to the network. Dynamic IP addresses are better suited for home networks and personal Internet use.

Advantages

- **Cost reduction:** obtaining a dynamic IP address is typically automated, making it a more cost-efficient option.

- **Enhance security:** devices with dynamic IP addresses are more difficult to track, reducing the risk of attackers targeting business networks.
- **Improved configuration:** dynamic IP addresses are automatically configured by a DHCP server, which removes the need for users to do so manually.
- **Increased flexibility:** different devices can reuse addresses and are assigned a new IP address every time they join a network.

Disadvantages

- **Hosting problems:** the changing nature of dynamic IPs means users may encounter problems with the *Domain Name System* (**DNS**). This makes dynamic IP addresses less effective for hosting servers and websites and tracking geolocations.
- **Poor technical reliability:** dynamic IP addresses can result in frequent periods of downtime and connection dropout issues, which makes them ineffective for data-intensive online activities.
- **Remote access** users with dynamic IP addresses may have trouble accessing the Internet from devices outside their primary network.

2.2 TCP

The *Transmission Control Protocol* (**TCP**) is transport protocol, meaning it dictates the way data is sent and received. A TCP header is included in the data of each packet that uses TCP/IP. Before transmitting data, TCP opens a connection with the recipient. TCP ensures that all packets arrive in order once transmission begins. Via TCP, the recipient will acknowledge receiving each packet that arrives. Missing packets will be sent again if receipt is not acknowledge.

It establishes a connection between the sender and receiver before any data is sent, maintaining this connection until communication is complete. It is designed for reliability, not speed. Because TCP has to make sure all packets arrive in order, loading via TCP/IP can take longer if some packets are missing.

TCP and IP were originally designed to be used together, and these are often referred to as the TCP/IP suite. However, other transport protocols can be used with IP.

2.3 TLS

2.4 UDP

The *User Datagram Protocol* (**UDP**), is a widely used transport protocol. It is faster than TCP, but it is also less reliable. UDP does not make sure all packets are delivered and in order, and it does not establish a connection before beginning or receiving transmissions.

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.