

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	Sort	13
1.4.1	Bubble Sort	13
1.4.2	Selection Sort	14
1.4.3	Insertion Sort	15
1.4.4	Merge Sort	16
1.4.5	Quick Sort	17
1.4.6	Heap Sort	20
1.4.7	Tim Sort	21
2	Protocols	25
2.1	IP	25
2.1.1	Public IP Adress	25
2.1.2	Private IP Adress	25
2.2	TCP	25
2.3	TLS	25
2.4	UDP	25
2.5	DNS	25
3	API	26

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 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 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.

1.4.2 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

```
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**.

1.4.3 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**.

1.4.4 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.

1.4.5 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     // pivot, always pick last
4     int pivot = arr[high];
5
6     // idx of smaller element
7     int i = low - 1;
8
9     // traverse arr[low...high] move all smaller elements to left side
10    for (int j = low; j <= high - 1; j++) {
11        if (arr[j] < pivot) {
12            i++;
13            // swap function
14            swap(arr, i, j);
15        }
16    }
17
18    // move pivot after smaller elements and returns it's position
19    swap(arr, i + 1, high);
20    return i + 1;
21 }
```

```

22
23 // quicksort function
24 function quickSort(arr, low, high) {
25     if (low < high) {
26         // pi -> partition return index of pivot
27         int pi = partition(arr, low, high);
28
29         // recursion calls for smaller elements and grater or equal elements
30         quickSort(arr, low, pi - 1);
31         quickSort(arr, pi + 1, high);
32     }
33 }
34
35 // main
36 int arr = [10, 7, 8, 9, 1, 5];
37 int n = arr.length;
38 // call quickSort
39 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:** $O(n)$, due to imbalanced partitioning leading to a skewed recursion tree requiring a call stack of size $O(n)$.
- **Worst case:** $O(n)$, as a result of balanced partitioning leading to a balanced recursion tree requiring a call stack of size $O(\log n)$.

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.

1.4.6 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 Quicksort 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.

1.4.7 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.

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 Address 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 called data packets. The IP address acts as the "electronic return address" for these packets, ensuring they arrive at the correct destination.

IP addresses can be public or private.

2.1.1 Public IP Address

2.1.2 Private IP Address

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.