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 Big O Notation

It is used to describe the time or space complexity of algorithms. **Big-O** is a way to express the **upper bound**, meaning the **worst case**, of an algorihtm's time or space complexity. Describes the asymptotic behavior (order of growth) of a function, not it's exact value.

The "O" in Big O stands for order, as in order of growth. The letter "O" was chosen by Paul Bachmann to stand for the german word "Ordnung", which means "order". In mathematics, the order of a function refers to how quickly it grows or declines.

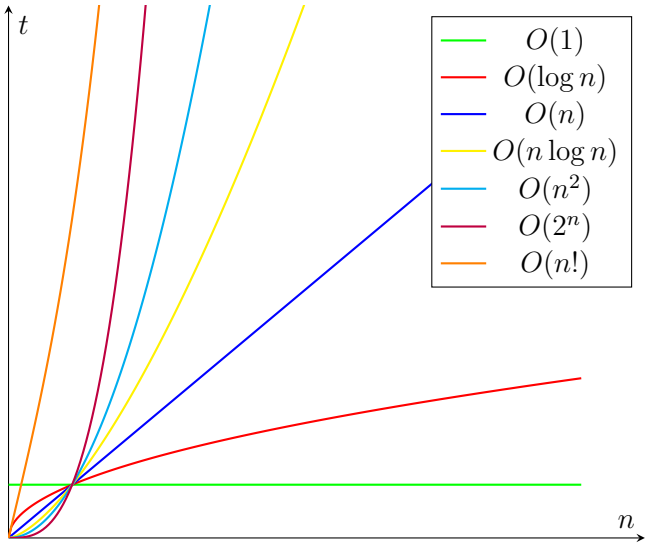
Big Omega notation (Ω) defines the **lower bound**, meaning the **best case**. **Big Theta** notation (Θ) describes the exact order of growth, this is the **tight bound** or **average case**. Big O is used most of the times.

Time complexity considers two factors, the **time** (t) it takes to complete the task and the **size** (n) of the input.

Fastest to slowest time complexity:

- **Constant: O(1)**. Time is independent of value of n. For example the sum of arithmetic series.
- **Logarithmic: O($\log n$)** Running time is proportional to the logarithm of n. In computing a plain 'log' refers to a logarithm of base 2 ($\log_2$). Base 10 ($\log_{10}$) is faster than base 2 ($\log_2$). An example of base 2 is binary search.
- **Linear: O(n)**. Grows linearly with the size of n. Single variable loops like linear search.
- **Quasilinear: O($n \log n$)**. Running time is proportional to n times the logarithm of n. Fastest sorting runtime (e.g. merge sort, heap sort).
- **Quadratic: O(n^2)**. Runnig time is porportional to the square of the input size, n times n. Traverse square matrix, bubble sort, selection sort, insertion sort.
- **Exponential: O(2^N)**. Running time doubles with each addition to n. Recursive fibonacci series, levels of a tree.
- **Factorial: O($n!$)**. Running time grows factorially with n. This is often seen in algorithms that generate all permutations of a set of data.

Plot of time complexity:



1.2 Swap

1.2.1 Temporary Variable Swap

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

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).

1.2.2 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.

1.2.3 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.

1.2.4 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.

1.2.5 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).

1.3 Sum of Arithmetic Series

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

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

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:

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

Implementation

```
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:

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

1.4 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.

1.4.1 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.

1.4.2 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 = floor(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 = floor(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.

1.5 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

```
// function for maximum value
function max(arr, n) {
  int max = 0;
  for (int i = 0; i < n; i++)
    if (arr[i] > max)
      max = arr[i];
  return max;
}

// function for minimum value
function min(arr, n) {
  int min = 0;
  for (int i = 0; i < n; i++)
    if (arr[i] < min)
      min = arr[i];
  return min;
}

// main
int arr = [1, 56, -23, 101, 34, 6, -11];
int n = arr.length;
// max
max(arr, n);
// min
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.

1.6 Sort

1.6.1 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.

1.6.2 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.

1.6.3 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.

1.6.4 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

```
function merge(arr, left, mid, right) {
  const int n1 = mid - left + 1;
  const int n2 = right - mid;

  // tmp arrays
  const int L[n1], R[n2];

  // copy data to tmp arrays L[] and R[]
  for (int i = 0; i < n1; i++)
    L[i] = arr[left + 1];
  for (int j = 0; j < n2; j++)
    R[i] = arr[mid + 1 + j];

  int i = 0, j = 0;
  int k = left;

  // merge tmp arrays back into arr[left...right]
  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = R[j];
      j++;
    }
    k++;
  }

  // copy remaining elements of L[], if there are any
```

```
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  // copy remaining elements of R[], if there are any
  while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
  }
}

function mergeSort(arr, left, right) {
  if (left >= right)
    return;

  const int mid = floor(left + (right - left) / 2);
  mergeSort(arr, left, mid);
  mergeSort(arr, mid + 1, right);
  merge(arr, left, mid, right);
}

// main code
const int arr = [38, 27, 43, 10];
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 wich makes it suitable for parallel processing.

Disadvantages:

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

1.6.5 Quick Sort

Quick Sort picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in it's 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

```
// partition function (Lomuto)
function patition(arr, low, high) {
  // random pivot
  randpivot = random(low, high)
  // swap random pivot and last pivot
  swap(arr, randompivot, high)
  // pivot, always pick last
  int pivot = arr[high];

  // idx of smaller element
  int i = low -1;

```

```
  // traverse arr[low...high] move all smaller elements to left side
  for (int j = low; j <= high - 1; j++) {
    if (arr[j] < pivot) {
      i++;
      // swap function
      swap(arr, i, j);
    }
  }

  // move pivot after smaller elements and returns it's position
  swap(arr, i + 1, high);
  return i + 1;
}

// quicksort function
function quickSort(arr, low, high) {
  if (low < high) {
    // pi -> partition return index of pivot
    int pi = partition(arr, low, high);

    // recursion calls for smaller elements and grater or equal elements
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}

// main
int arr = [10, 7, 8, 9, 1, 5];
int n = arr.length;
// call quickSort
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.

1.6.6 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

```
// heapify a subtree rooted with node i
function heapify(arr, n, i) {
  // initialize largest as root
  int largest = i;

  // left index = 2*i + 1
  int left = 2 * i + 1;

  // right index = 2*i + 2
  int right = 2 * i + 2;

  // if left child greater than root
  if (left < n && arr[left] > arr[largest])
    largest = l;

  // if right child greater than largest so far
  if (right < n && arr[right] > arr[largest])
    largest = right;

  // if largest is not root
  if (largest != i) {
```

```
    // swap function
    swap(arr[i], arr[largest]);
    // recursively heapify the affected sub-tree
    heapify(arr, n, largest);
  }
}

// function for heap sort
function heapSort(arr, n) {
  // build heap (rearrange array)
  for (int i = floor(n / 2) - 1; i >= 0; i--)
    heapify(arr, n, i);

  // one by one extract element from heap
  for (int i = n - 1; i > 0; i--) {
    // swap function
    swap(arr[0], arr[i]);
    // call max heapify on the reduced heap
    heapify(arr, i, 0);
  }
}

// main
int arr = [9, 4, 3, 8, 10, 2, 5];
int n = arr.length;
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.

1.6.7 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 Pyhton'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 algorihtm.

Steps

- Define the size og the run. Minimum run size of 32.
- Divide the array into runs. Use the insertion sort to 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 operatio, 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

```
int MIN_MERGE = 32;

// minimum range of run
function minRunLength(n) {
  int r = 0;
  while (n >= MIN_MERGE) {
    r |= (n & 1);
    n >>= 1;
  }
  return n + r;
}

// insertion sort
function insertionSort(arr, left, right) {
  for (int i = left + 1; i <= right; i++) {
    int tmp = arr[i];
    int j = i - 1;
    while (j >= left && arr[j] > tmp) {
      arr[j + 1] = arr[j];
      j--;
    }
    arr[j + 1] = tmp;
  }
}

// merge sorted runs
function merge(arr, low, mid, high) {
  // original array is broken in two parts left and right
  int n1 = mid - low + 1;
  int n2 = high - mid;

  int left = [n1];
  int right = [n2];

  for (int i = 0; i < n1; i++)
```

```
    left[i] = arr[low + i];

  for (int j = 0; j < n2; j++)
    right[j] = arr[mid + 1 + j];

  int i = 0;
  int j = 0;
  int k = 0;

  while (i < n1 && j < n2) {
    if (left[i] <= right[j]) {
      arr[k] = left[i];
      i++;
    } else {
      arr[k] = right[j];
      j++;
    }
  }

  // copy remaining elements of left, if any
  while (i < n1) {
    arr[k] = left[i];
    k++;
    i++;
  }

  // copy remaining elements of right, if any
  while (j < n2) {
    arr[k] = right[j];
    k++;
    j++;
  }
}

// Timsort to sort arr[0...n-1]
function timSort(arr, n) {
  int minRun = minRunLength(MIN_MERGE);

  // sort individual subarrays of size MIN_MERGE
  for (int i = 0; i < n; i += minRun)
    insertionSort(arr, i, min((i + MIN_MERGE - 1), (n - 1)));

  // start merging from size MIN_MERGE
  for (int size = minRun; size < n; size = 2 * size) {
    // pick starting point of left sub array
    for (int left = 0; ; left < n; left += 2 * size) {
      // find ending point left sub array
      int mid = left + size - 1;
      int right = min((left + 2 * size - 1), (n - 1));
      if (mid < right)
        merge(arr, left, mid, right);
    }
  }
}
```

```
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 netwotk. 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 wich 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 and an IP Adress, assigned to very device on a network, such as the internet. IP addresses allow commputers 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. Commom 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, it's 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 and 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 and eight-digit binary number, separated by a period point. Each group of numbers that are separated by periods is called an **octect**.

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 addresses classes innclude:

- **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 and 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**.

The **network address** or **network id** is a number that is assigned to a network. Every network has a unique address. This corresponds to the first address in a range and they end in 0, which can be expressed as **x.x.x.0**. For example **192.168.1.0**.

The **broadcast address** or **broadcast id** is a number that is assigned to broadcast within a network. Every broadcast has a unique address. This corresponds to the last address in a range and they end in the number that corresponds to the last available number in a range. The broadcast id equals the next subnet's network id minus 1. Or once the first broadcast id is known, add the number of possible hosts to get the subsequent broadcast id's for the following subnets.

The **host address** or **host id** is what is assigned to hosts within that network such as computers, servers, tablets, routers, phones, etc. Every host has a unique host address. These are assigned after the network address, the look like **x.x.x.1**, **x.x.x.2**, **x.x.x.3**, etc. For example **192.168.1.1**, **192.168.1.2**, **192.168.1.3** and so on.

Further reserved IP addresses are for what is known as **subnet classes**. A **subnet** IP is a logically smaller division within a lerger IP network, crated through a process called subnetting. Subnetting divides a large network into smaller, more managable segments, or subnets, wich 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.

2.1.1.1 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 indicate the **network**

portion, and the bits set to '**0**' indicate the **host portion**.

3. **Logical Divison:** 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^m, where m is the number of bits borrowed from the original host part to create more network bits in subnetting.
- **Number of possible hosts**: possible addresses in a subnet = 2^n, where n is the number of ceros in the subnet mask. This is the number of possible hosts. In other words, the number of words allocated for the host portion.
- **Number of usable hosts:** usable hosts = $2^n - 2$, where n is the number of bits used for hosts in the subnet mask. The substraction of 2 accounts for the first and last address in the range. The first address is reserved for the network id, which identifies the subnet itself. The last address is reserved for the broadcast address, used to send data to all devices in a subnet. These two cannot be assigned to hosts.

Key elements

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

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 ans 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 esaily 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 smat 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 wich 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 signifier assigned to a device that remains consistent and cannot change across multiple networks 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. The 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 thier 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 for and easy for people to locate and use shared devices, such as a printer on their network.
- **Reduced connection lapses:** connection lapses typically happens 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 statis IP addresses makes it easy for people to track a device the data they access or share. This could be a securiry 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 bby 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-intense online activities.
- **Remote access** users with dynamic IP addresses may have trouble accessing the Internet from devices outside their primary network.

2.2 DNS

The *Domain Name System* (**DNS**) is a hierarchical and distributed naming system that translates domain names into IP addresses. Humans access information online through **domain names** and web browsers interact through IP addresses. When a user types a domain like **www.example.com** into a browser, DNS ensures that the request reaches the correct server by resolving the domain to it's corresponding IP address. Without DNS, users would have to remember the numerical IP address of every website we want to visit, which is highly impractical. It is the **phonebook of the internet**.

A **DNS server** is a computer with a database containing the public IP addresses associated with the names of the websites an IP address brings a user to. The process of **DNS Lookup**, also called **DNS resolution**, involves converting a hostname (such as www.example.com) into a computer-friendly IP address, which computers use to locate and communicate with each other on the internet.

Reverse DNS Lookup is the proccess of mapping an IP address back into it's corresponding domain name. This is used for network and email security.

2.3 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.

High-level protocols that need to transmit data use TCP protocol. Examples include peer-to-peer sharing methods like *File Transfer Protocol* (**FTP**), *Secure Shell* (**SSH**) and *Telnet*. It is also used for web access through the *Hypertext Transfer Protocol* (**HTTP**.)

The two computers begin by establishing a connection vi an automated process called a **"handshake"**. Only once this handshake has been completed will one computer actually transfer data packets to the other. A three-way handshake is a three-message process that involves a **SYN** (synchronize) packet from the client to the server, a **SYN-ACK** (synchronize-acknowledge) response from the server, and finally, an **ACK** (acknowledge) packet from the client to the server.

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.

The two protocols are frequently used together and rely on each other for data to have destination and safely reach it, which is why the process is regularly referred to as TCP/IP.

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. It is used for time-sensitive applications like gaming, playing videos or *Domain Name System* (**DNS**) lookups.

UDP can cause data packets to get lost as they go from the source to the destination. It can also make it relatively easy for a hacker to execute a *distributed denial-of-service* (**DDoS**) attack.

The process behind UDP is fairly simple. A target computer is identified and the data packets, called **"datagrams"**, are sent to it. There is nothing to indicate the order in which the packets should arrive. There is also no process for checking if the datagrams reahced the destination.

As a result, the data may get delivered, and it may not. In addition, the order in which it arrives it not controlled, as it is in TCP, so the way the data appears at the final destination maw be glitchy, out of order or have blank spots. However, in a situation where there is no need to check fr errors or correct the data that has been sent, this may not pose a significant problem.

2.5 TLS

The *Transport Layer Security* (**TLS**) is a widely adopted security protocol designed to facilitate privacy and data security for communication over the Internet. A primary use case of TLS is

encrypting the communication between web applications and servers, such as web browsers loading a website. It was proposed by the *Internet Engineering Task Force* (**IETF**), and the first version of the protocol was published in 1999.

TLS evolved from a previous encryption protocol called *Secure Socket Layer* (**SSL**), which was developed by **Netscape**. **HTTPS** is an implementation of TLS encryption on top of the **HTTP** protocol.

There are three main components to what the TLS protocol accomplishes:

- **Encryption:** hides the data being transferred from third parties.
- **Authentication:** ensures that the parties exchanging information are who they claim to be.
- **Integrity:** verifies that the data has not forged or tampered with.

A TLS connection is initiated using a sequence known as the **TSL handshake**. During the TLS handshake, the user's device and the web server:

- Specify which version of TLS (1.0, 1.2, 1.3) they will use.
- Decide on which cipher suites they will use.
- Authenticate the identity of the server using the server's TLS certificate.
- Generate session keys for encrypting messages between them after the handshake is complete.

2.6 HTTP

The *Hypertext Transfer Protocol* (**HTTP**) is used for transmitting hypermedia documents, such as HTML. It was designed for communication between web browsers and web servers, but it can also be used for other purposes, such as machine-to-machine communication and programmataic access to APIs. Everything sent across HTTP is plain text.

Servers store web pages that are provided to the client's computer when a user accesses them. This communication between servers and clients creates a network, known as the World Wide Web, and HTTP is it's foundation.

HTTP follows a classical **client-server model**, with a client (browser) opening a connection to make a request, then waiting until it receives a response from the server (e.g. computer in the cloud). HTTP is a **stateless protocol**, meaning that the server does not keep any session data between two requests, although the later addition of **cookies** adds state to some client-server interactions.

Communication between clients and servers is done by **requests** and **responses**:

1. A client (browser) send an **HTTP request** to the web.
2. A web server receives the request.
3. The server runs an application to process the request.
4. The server returns an **HTTP response** (output) to the browser.
5. The client (browser) receives the response.

HTTP **request methods**, sometimes called **HTTP verbs**, indicate the purpose of the request and what is expected if the request is succesful. The most common methods are **GET** and **POST** for

retrieving and sending data to servers, respectively, but there are other methods which serve different purposes.

HTTP **response status codes** indicate whether a specific HTTP request has been succesfully completed. Responses are grouped in five classes:

1. **Informational:** 100 - 199.
2. **Succesful:** 200 - 299.
   - **200 OK:** the request succeded. The result of "success" depends on the HTTP method.
   - **201 Created:** a new resource was created. This is typically the responde sent after **POST** requests.
3. **Redirection:** 300 - 399.
4. **Client error:** 400 - 499.
   - **400 Bad Request:** the server cannot or will not process the request due to something that is perceived to be a client error.
   - **404 Not Found:** the server cannot find the requested resource. In the browser, this means the URL is not recognized.
5. **Server error:** 500 - 599.
   - **500 Internal Server Error:** the server has encountered a situation it does not know how to handle. This error is generic, indicating that the server cannot find a more appropiate *5xx* status code to respond with.
   - **503 Service Unavailable:** the server is not ready to handle request. Common cause are a server that is down for maintenance or that is overload.

The *Hypertext Transfer Protocol Secure* (**HTTPS**) is the secure version of HTTP, using TLS on top. Thecnically speaking, HTTPS is not a separate protocol from HTTP. It is simply using TLS/SSL encryption over the HTTP protocol.

2.7 FTP

The *File Transfer Protocol* (**FTP**) is a standard network protocol used for the transfer of files from one host to another over a TCP-based network, such as the Internet.

It works by opening two connections that link the computers trying to communicate with each other. One connection is designated for the commands and replies that get sent between the two clients, and the other channel handles the transfer of data. An **FTP server** can act as an online storage where files can be accessed and downloaded by multiple users.

Once the connection is established, the authorized users can perform the following operations:

- Upload files from the user's computer to the FTP server.
- Download files from the FTP server to the user's computer.
- Delete files on the FTP server.
- Change file permissions on the FTP server.

One of the main advantages of FTP is it's ability to perform large file size transfers. When sending a relatively small file, like a Word document, most methods will do, but with FTP, hundreds of gigabytes can be sent at once and still get a smooth transmision.

The *File Transfer Protocol Secure* (**FTPS**) is and extension of the standard FTP, that upgrades the connection with the implicit use of TLS/SSL. This is a stric form of FTP since the FTPS server will reject the connection it ti doesn't receive the TLS message from the client. Maintains a dual-channel architecture.

The *Secure File Transfer Protocol* (**SFTP**) is a secure version of FTP. It uses **SSH** for secure file transfers. It only uses one channel. It is not an FTP protocol, rather, it is an extension of the SSH for transferring, accessing and managing files. It is the preferred method of file distribution amongst server administrators and web developers due to the cryptographic protection it uses as a subest of the SSH protocol.

2.8 SSH

The *Secure Shell* (**SSH**) protocol is a method for secure remote login from one computer to another. It provides several alternative options for strong authentication, and it protects communications security and integrity with strong encryption. It is a secure alternative to the non-protected login protocols (such as **telnet**, **rlogin** and **rsh**) and insecure file transfer methods (such as FTP). SSH runs on top of the TCP/IP protocol suite.

When the SSH protocol became popular, Tatu Ylonen took it to the *Internet Engineering Task Force* (**IETF**) for standardization. It is now an internet protocol.

The protocol is used in corporate networks for:

- Providing secure access for users and automated processes.
- Interactive and automated file transfers.
- Issuing remote commands.
- Managing network infrastructure and other mission-critical system components.

The protocol works in the client-server model, which means that the connection is established by the SSH client connecting to the SSH server. The steps are the following:

1. Client initiates the connection by contacting server.
2. Sends server public key.
3. Negotiate parameters and open secure channel.
4. User login to server host operating system.

There are several options that can be used for user authentication. The most common ones are passwords and public key authentication. The public key authentication method is primarily used for automation and sometimes by system administrators for single sign-on.

The idea is to have a cryptographic key pair, public key and private key, and configure the public key on a server to authorize access and grant anyone who has a copy of the private access to the server. The keys used for authentication are called **SSH keys**.

SSH also allows for **tunneling**, or **port forwarding**, which is when data packets are able to cross networks that they would not otherwise be able to cross. Tunneling works by wrapping data packets with additional information, called headers, to change their destination

Linux and Mac operating systems come with SSH built in. Windows machines may need to have SSH client application installed.

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.