Algorithms

```
Module Code: ELEE1147

Module Name: Programming for Engineers

Credits: 15

Module Leader: Seb Blair BEng(H) PGCAP MIET MIHEEM FHEA
```

Download as a PDF

O Notation

- **Big-O Notation (O-notation):**
  - Represents the upper bound of the running time of an algorithm.
  - Shows the worst-case complexity of an algorithm.
- **Omega Notation (Ω-notation):**
  - Represents the lower bound of the running time of an algorithm.
  - Provides the best case complexity of an algorithm.
- **Theta Notation (Θ-notation):**
  - Theta notation encloses the function from above and below.
  - Used for analysing the average-case complexity of an algorithm.

Why Big O?

Importance:

- Efficient algorithms are crucial in computer science and programming.
- Big O helps in quantifying and comparing algorithm efficiency.
- Allows for better decision-making in algorithm selection.

Analysing Algorithm Complexity

Factors Affecting Complexity:

- **Time Complexity:**
  - How the runtime of an algorithm increases with the input size.
- **Space Complexity:**
  - How the memory requirements of an algorithm scale with the input size.

Time Complexity

Time complexity represents the amount of time an algorithm takes to complete as a function of the input size.

- Constant Time $\Longrightarrow O(1)$
- Logarithmic Time $\Longrightarrow O(log\,n)$
- Linear Time $\Longrightarrow O(n)$
- Log-linear Time $\Longrightarrow O(n\;log\,n)$
- Quadratic Time $\Longrightarrow O(n^2)$
- ...

Time Complexity Metrics

Big O Notation	n	$n\,log\,n$	n^2	n^3	2^n	$n!$
n = 10	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
n = 30	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 18 min	10^{25} years
n = 100	< 1 sec	< 1 sec	< 1 sec	1s	10^{17} years	Very Long Time
n = 1000	< 1 sec	< 1 sec	1 sec	18 min	Very Long Time	Very Long Time
n = 10,000	< 1 sec	< 1 sec	2 min	12 days	Very Long Time	Very Long Time
n = 100,000	< 1 sec	2 sec	3 hours	32 years	Very Long Time	Very Long Time
n = 1,000,000	1 sec	20 sec	12 days	31,710 years	Very Long Time	Very Long Time

Space Complexity

Space complexity represents the amount of memory space an algorithm requires as a function of the input size.

- Constant Space $\implies O(1)$
- Linear Space $\implies O(n)$
- Log-linear Space $\implies O(n\ log\ n)$
- Quadratic Space $\implies O(n^2)$
- ...

Recogonising Algorithms Complexity

- Constant runtime is represented by $O(1)$
- linear growth is $O(n)$
- logarithmic growth is $O(log\ n)$
- log-linear growth is $O(n\ log\ n)$
- quadratic growth is $O(n^2)$
- exponential growth is $O(2^n)$
- factorial growth is $O(n!)$

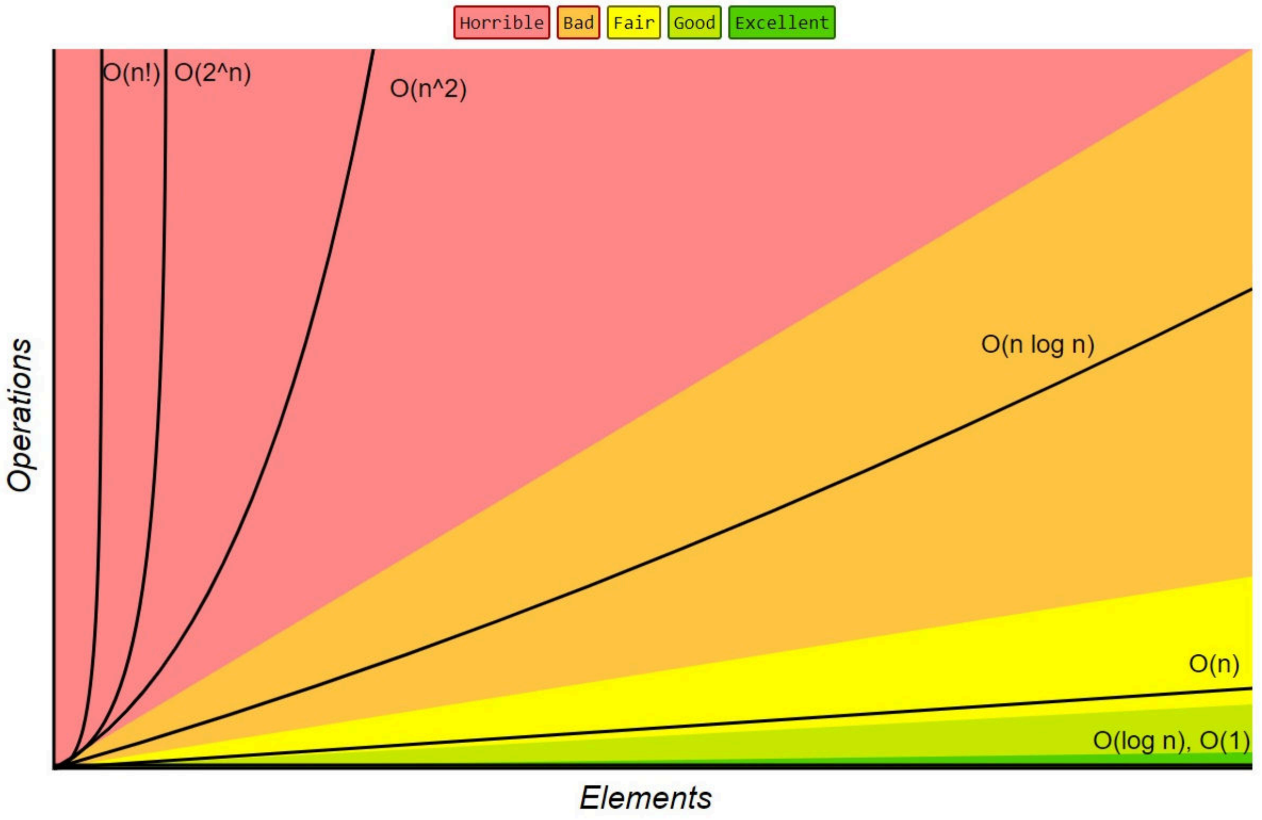


Table of Big O

Big O Notation	Relationship with 'n'	Description	Assumption
O(1)	Constant	The algorithm's runtime is constant regardless of the input size.	The algorithm performs a single operation.
O(log n)	Logarithmic	The algorithm's runtime grows logarithmically as the input size increases.	The algorithm divides the input in half at each step (e.g., binary search).
O(n)	Linear	The algorithm's runtime grows linearly with the input size.	The algorithm iterates through the input once.
O(n log n)	Linearithmic	The algorithm's runtime grows in between linear and logarithmic as the input size increases.	Typically seen in efficient sorting algorithms like merge sort or quicksort.
O(n^2)	Quadratic	The algorithm's runtime grows quadratically with the input size.	The algorithm has nested iterations over the input (e.g., nested loops).
O(n^3)	Cubic	The algorithm's runtime grows cubically with the input size.	The algorithm has triple nested iterations over the input.
O(2^n)	Exponential	The algorithm's runtime grows exponentially with the input size.	The algorithm performs exhaustive search or generates all subsets of the input.

Real-world Applications

- Choosing the right data structures and algorithms for software development.
- Optimizing database queries.

	Seach in a table	**Seach in an index**
Seach Algorithm	Linear Scan	Binary Scan
Complexity	$O(N)$	$O\ (Log\ N)$

- Designing efficient algorithms for large-scale data processing.

Examples of Big O Notation

- Linear Search $\implies O(n)$
- Binary Search $\implies O(log\ n)$
- Bubble Sort $\implies O(n^2)$
- Merge Sort $\implies O(n\ log\ n)$
- ...

Linear Search Example, $O(n)$:

- Searching for a value and its index
- Unordered List, Small Data Sets, Linked Lists.

```c
#include <stdio.h>

int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i;  // Target found
        }
    }
    return -1;  // Target not found
}
```

```c
int main() {
    int arr[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 4;

    int result = linearSearch(arr, size, target);

    if (result != -1) {
        printf("Target %d found at index %d\n", target, result);
    } else {
        printf("Target %d not found\n", target);
    }

    return 0;
}
```

Second example of $O(n)$, finding Max:

```
// Linear complexity: O(n)
int FindMaxElement(int[] array)
{
  int max = int.MinValue;
  for (int i = 1; i < array.Length; i++)
  {
    if (array[i] > max)
    {
      max = array[i];
    }
  }
  return max;
}
```

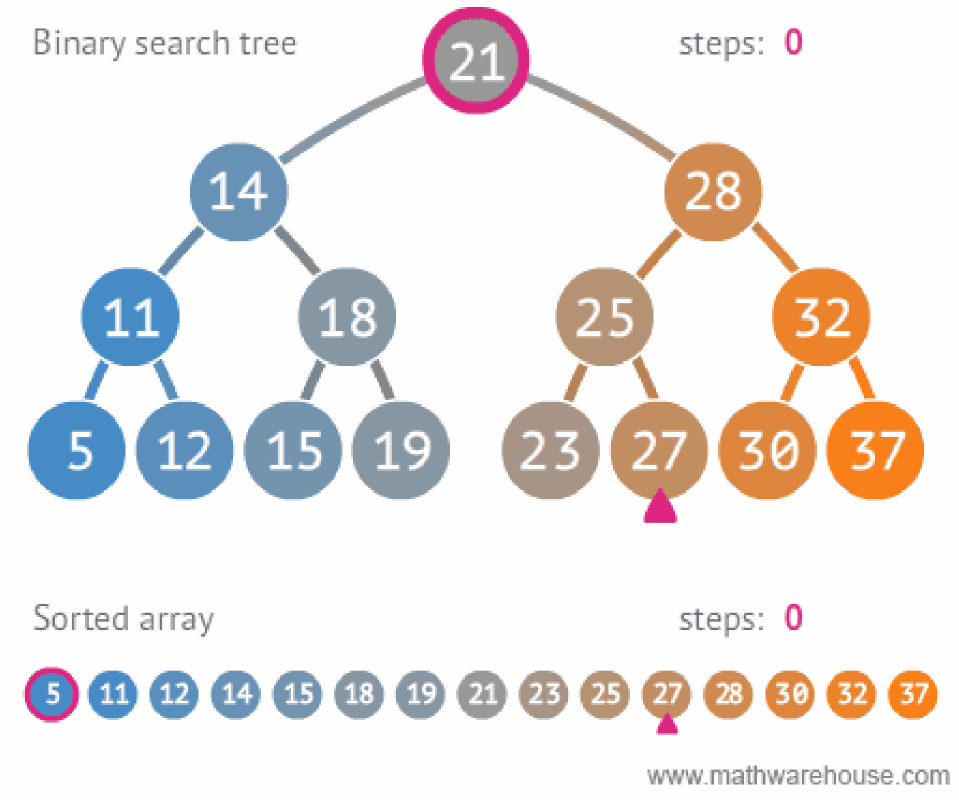
- The algorithm's time complexity is linearly dependent on the size of the input (each additional element in the array results in one more iteration through the loop)
- it is denoted as $O(n)$, where n is the length of the array. This makes it an efficient linear time algorithm for finding the maximum element in an array.

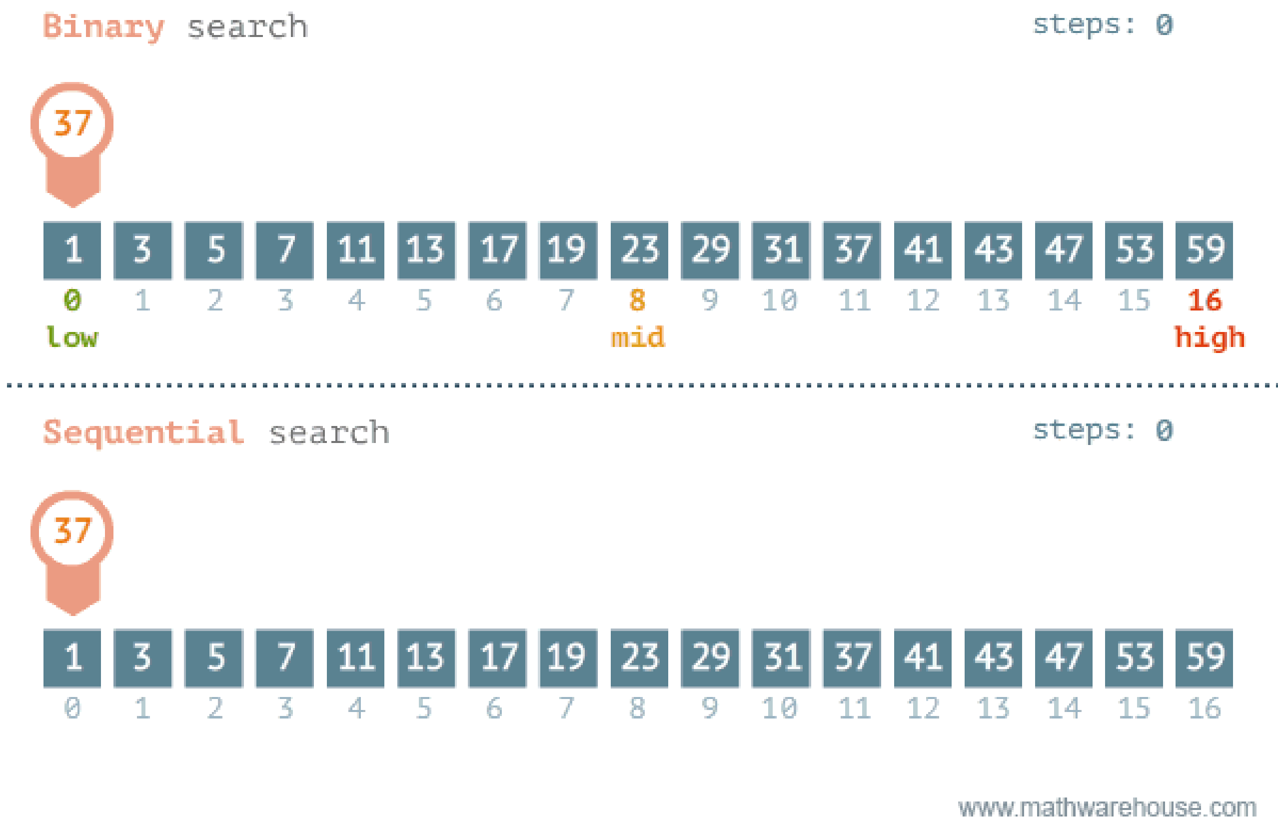
Binary search $O(log\ n)$ code example:

```c
#include <stdio.h>

int binarySearch(int arr[], int size, int target) {
    int low = 0, high = size - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == target) {
            return mid;  // Target found
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return -1;  // Target not found
}
```

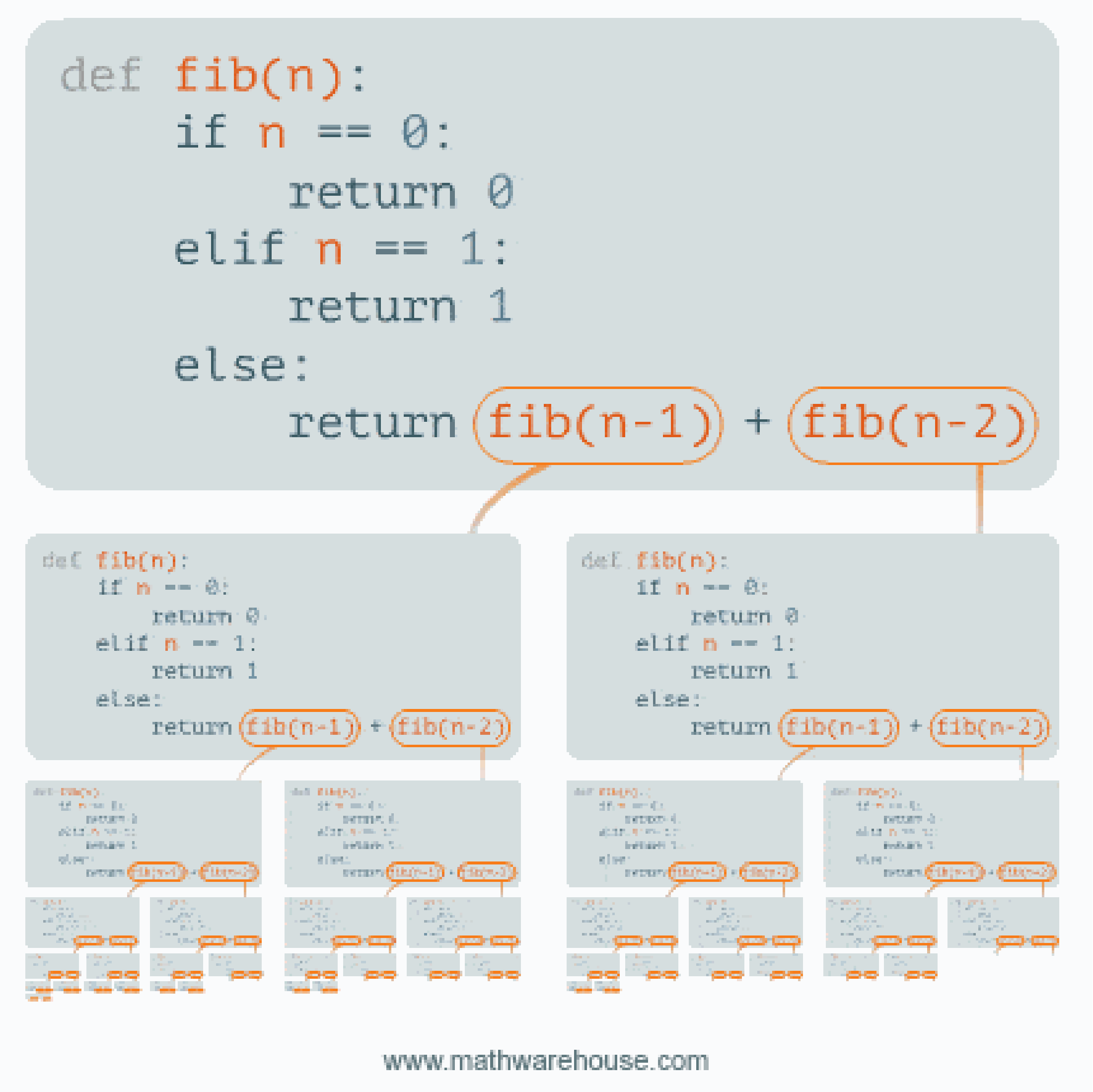
```c
int main() {
    int arr[] = {5,11,12,14,15,18,19,21,23,
                    27,25,28,30,32,37};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 27;

    int result = binarySearch(arr, size, target);

    if (result != -1) {
        printf("Target %d found at index %d\n",
                target, result);
    } else {
        printf("Target %d not found\n", target);
    }

    return 0;
}
```

Binary search $O(log\ n)$ and Linear(Seq) Search $O(n)$





Exponential growth is $O(2^n)$, Fibonacci:



Exponential growth is $O(2^n)$, Fibonacci:

```
// Exponential complexity: O(2^n)
long Fibonacci(int n)
{
    if (n == 0)
    {
        return 1;
    }
    else if (n == 1)
    {
        return 1;
    }
    else
    {
        return Fibonacci(n - 1) + Fibonacci(n - 2);
    }
}
```

- For each Fibonacci number, the algorithm makes two recursive calls: `Fibonacci(n - 1) and Fibonacci(n - 2)`. This leads to an exponential growth in the number of recursive calls as the input `n` increases.
- Similar to binary as this is base 2 too.
- An algorithm's performance can degrade rapidly as the input size increases.

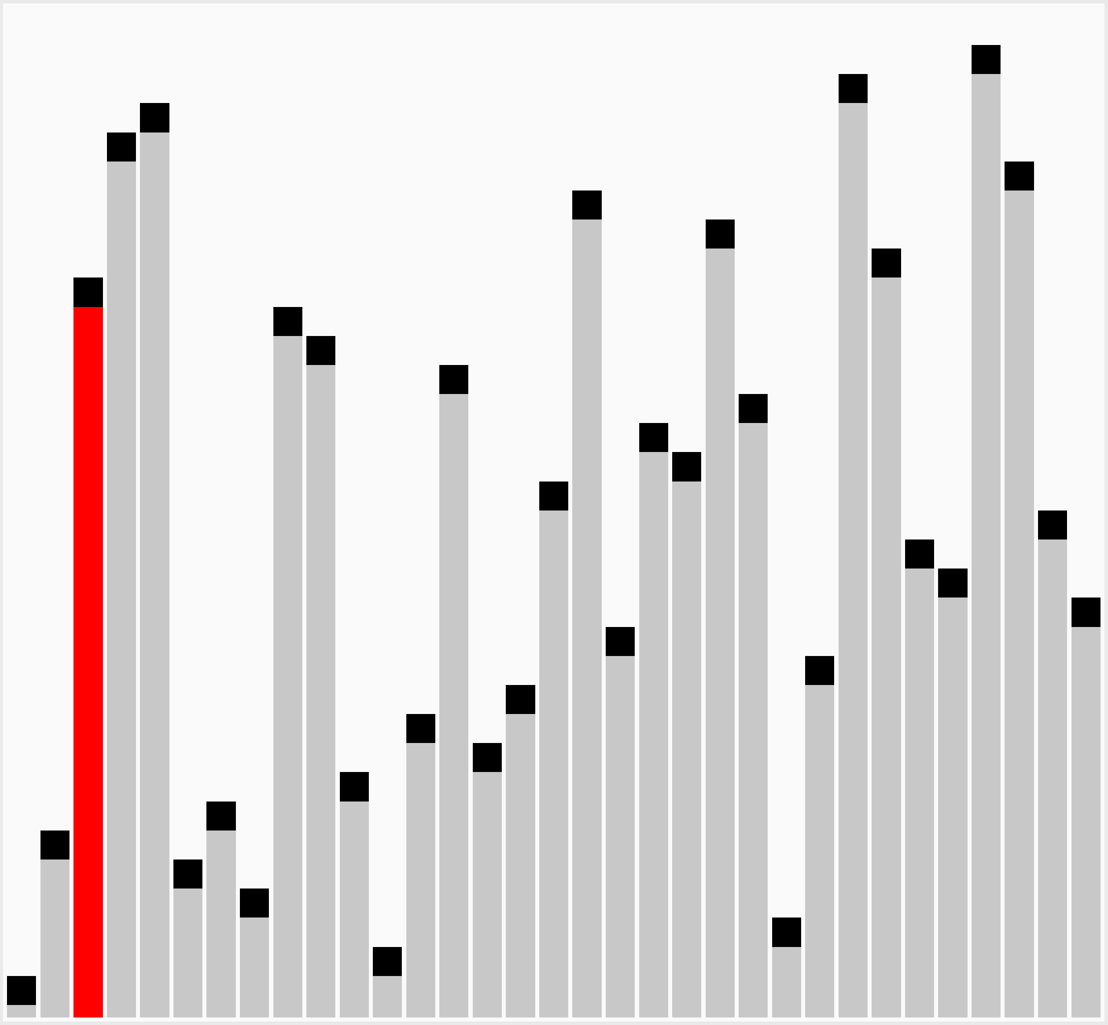
Greatest Common Demonator $O(log\ n)$

```c
// logarithmic growth is O(log n)
int GCD(int a, int b)
{
  if (b > a) // Ensure that a is greater than b
  {
    int temp = a;
    a = b;
    b = temp;
  }

  // Compute GCD using the Euclidean algorithm
  while (b > 0)
  {
    int r = a % b;
    a = b;
    b = r;
  }

  return a;
}
```

- The Euclidean algorithm has a time complexity of O(log N), where N is the larger of the two input integers.
- This is because, in each iteration of the algorithm, the size of the inputs is reduced by a factor of at least 2,
- which means that the number of iterations required is proportional to $log\ N$.

Bubble Sort $\Longrightarrow O(n^2)$:

```
for (c = 0 ; c < n - 1; c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (array[d] > array[d+1])
      {
        swap       = array[d];
        array[d]   = array[d+1];
        array[d+1] = swap;
      }
    }
  }
```



Quick Sort $\Longrightarrow O(n\ log\ n)$:

```
void quicksortMiddle(int arr[], int low, int high) {
  if (low < high) {
    // Selecting the middle element as the pivot
    int pivot = arr[(low + high) / 2];
    int i = low,j = high, temp;

    while (i <= j) {
      // Moving elements smaller than pivot to the left
        while (arr[i] < pivot) i++;
        // Moving elements greater than pivot to the right
        while (arr[j] > pivot) j--;

        if (i <= j) {
            temp = arr[i]; // Swapping elements
            arr[i] = arr[j];
            arr[j] = temp;
            i++;
            j--;
        }
    }
    // Recursively sort the two partitions
    if (low < j) quicksortMiddle(arr, low, j);
    if (i < high) quicksortMiddle(arr, i, high);
  }
}
```