What is an Algorithm?

A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.
An algorithm is a procedure or formula for solving a problem, based on conducting a sequence of specified actions

- **Finiteness -** A finite sequence of steps
- **Definiteness -** Each step should be clearly and precisely stated
- **Input -** There should be some input presented to it
- **Termination/output -** It should terminate and produce and output
- **Correctness -** The output produced should be correct for the given input
- **Effectiveness** - All operations to be performed must be sufficiently basic that they can be done exactly and in finite length.

Expressing Algorithm
An algorithm may be expressed in a number of ways, including:

- **natural language**: usually verbose and ambiguous
- **flow charts**: avoid most (if not all) issues of ambiguity; difficult to modify w/o specialized tools; largely standardized
- **pseudo-code**: also avoids most issues of ambiguity; vaguely resembles common elements of programming languages; no particular agreement on syntax
- **programming language**: tend to require expressing low-level details that are not necessary for a high-level understanding

Algorithm: It's an organized logical sequence of the actions or the approach towards a particular problem. A programmer implements an algorithm to solve a problem. Algorithms are expressed using natural verbal but somewhat technical annotations.
Pseudo code: It's simply an implementation of an algorithm in the form of annotations and informative text written in plain English. It has no syntax like any of the programming language and thus can't be compiled or interpreted by the computer.

Advantages of Pseudocode

- Improves the readability of any approach. It's one of the best approaches to start implementation of an algorithm.
- Acts as a bridge between the program and the algorithm or flowchart. Also works as a rough documentation, so the program of one developer can be understood easily when a pseudo code is written out. In industries, the approach of documentation is essential. And that's where a pseudo-code proves vital.
- The main goal of a pseudo code is to explain what exactly each line of a program should do, hence making the code construction phase easier for the programmer.
-

Disadvantages of Pseudocode

- Pseudocode does not provide a visual representation of the logic of programming.
- There are no proper format for writing the for pseudocode.
- In Pseudocode their is extra need of maintain documentation.
- In Pseudocode their is no proper standard very company follow their own standard for writing the pseudocode.

Features of a good algorithm (Wiki)

- **Precision**: a good algorithm must have a certain outlined steps. The steps should be exact enough, and not varying.
- **Uniqueness**: each step taken in the algorithm should give a definite result as stated by the writer of the algorithm. The results should not fluctuate by any means.
- **Feasibility**: the algorithm should be possible and practicable in real life. It should not be abstract or imaginary.
- **Input**: a good algorithm must be able to accept a set of defined input.
- **Output**: a good algorithm should be able to produce results as output, preferably solutions.
- **Finiteness**: the algorithm should have a stop after a certain number of instructions.
- **Generality**: the algorithm must apply to a set of defined inputs.

Another good resource is available in below link
https://courses.cs.vt.edu/cs2104/Fall14/notes/T16_Algorithms.pdf

Parameters that define how efficient an algorithm is ?

The simplest approach could be to write a code for that algorithm and log the time taken by the program to execute. But, there are certain drawbacks with this.

1. Dependency on hardware and software - It is very difficult to compare the running time of two algorithms, unless they have both been implemented using the same software and hardware.
2. Need to implement and run the algorithm on a specific hardware and software, which might not be always possible.
3. Experiments can only be done only on a limited set of inputs, Therefore, care should be taken to ensure that these are representative. Input should cover different possibilities as well. Like different set of inputs, different order of input, different size of input, etc.

Two notions that can be used to identify complexity of an algorithm.

1. **Time complexity** - Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm
2. **Space complexity** - Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

Source : https://www.cs.utexas.edu/users/djimenez/utsa/cs1723/lecture2.html

What is Data Structure ?
A data structure is a particular way to organize data in a computer, so it can be used effectively.
A data structure is a data organization, management, and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data

values, the relationships among them, and the functions or operations that can be applied to the data.

Properties for analytic framework for evaluation algorithms

1. Takes into account all possible inputs for all algorithm
2. Allow us to evaluate the relative efficiency of any two algorithm in a manner that is independent of the hardware or software environment.
3. Can be performed on a high-level description of an algorithm without having to implement it or run experiments on it.

Algorithm arrayMax(A, n):

    Input: An array A storing $n \geq 1$ integers.

    Output: The maximum element in A.

$$
\begin{array}{l}
\mathit{currentMax} \leftarrow A[0] \\
\textbf{for } i \leftarrow 1 \textbf{ to } n-1 \textbf{ do} \\
\quad \textbf{if } \mathit{currentMax} < A[i] \textbf{ then} \\
\quad\quad \mathit{currentMax} \leftarrow A[i] \\
\textbf{return } \mathit{currentMax}
\end{array}
$$

Analytical statement for above algorithm is

"Algorithm arrayMax runs in time proportional to N"

If we were to perform experiments, then the actual running time of arrayMax on any input of size n never exceeds c.n, where c is a constant which depends on the software and hardware environment.

Given two algorithms A and B, and A runs in time proportional to n and B runs in time proportional to N^2, we will prefer A to B, since the function n grows at a smaller rate than the function m^2.

Suppose there are two machines, A and B with c1 and c2 as 1000 and 5. Let's support on machine A we are using algorithm with complexity n and for B as complexity n^2

Then clearly

1000n > 5 * n^2 when n < 200 (say)

So even through complexity looks to be higher for machine B, but running time is better than machine A.

As the size of n increase, time taken by machine B will grow significantly, but for machine A it will grow linearly. So, running time is depending on size of input, and it is always better to choose algorithm which has less growth rate depending on N instead of actual running time. This helps in better scalability of system.

Primitive Operation

Any operation which is basic and cannot be subdivided into multiple operation is called primitive operation.
For ex:

1. **a < b + c**

There are two primitive operation

- b + c (p)
- a < p

2. **a < b + c * d**

There are three primitive operation

- c * d (p)
- b + p (sum)
- a < sum

3. **Lets take below algorithm and find primitive operations**

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.

  currentMax $\leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if *currentMax* $< A[i]$ **then**
      currentMax $\leftarrow A[i]$
  return *currentMax*

This algorithm can have a pseudocode as below

Operations	**Primitive operations**
currentMax = A[0]	**1**
i = 1	**1**
While (i < n)	**1, repeated n times**
If currentmax < A[i]	**1, repeated n times**
currentMax = A[i]	**1, repeated n times**
I = i + 1	**1, repeated n times**
Return currentMax	**1**

So total primitive operation will be < **4n + 3.** As above, operations are assumed for worst case.

A = {1, 3, 2, 5}

recursiveMax(A, 4)
= max(recursiveMax(A, 3), A[3])
= max(max(recursiveMax(A, 2), A[2]), A[3])
= max(max(max(recursiveMax(A, 1), A[1]), A[2]), A[3])
= max(max(max(A[0], A[1]), A[2]), A[3])
= max(max(max(1, 3), 2), 5)
= max(max(3, 2), 5)
= max(3, 5)
= 5

max(a, b)

if a > b
  return a
else
  return b

Algorithm recursiveMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

if $n = 1$ **then** 1
  return $A[0]$ 2
return $\max\{$recursiveMax$(A, n-1),\ A[n-1]\}$ T(n-1) + 6

Algorithm 1.4: Algorithm recursiveMax.

return: 1, max: 2, operation (n-1): 2, recursive call: T(n-1), indexing A[n-1]: 1

T(n) = no. of primitive operations required to compute recursiveMax(A, n)

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ T(n-1) + 7 & \text{otherwise} \end{cases}$$

T(n) = T(n-1) + 7
= T(n-2) + 7 + 7
= ....
= T(1) + 7(n-1)
= 3 + 7(n-1)
= 7n - 4

Algorithm recursiveMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

if $n = 1$ **then**

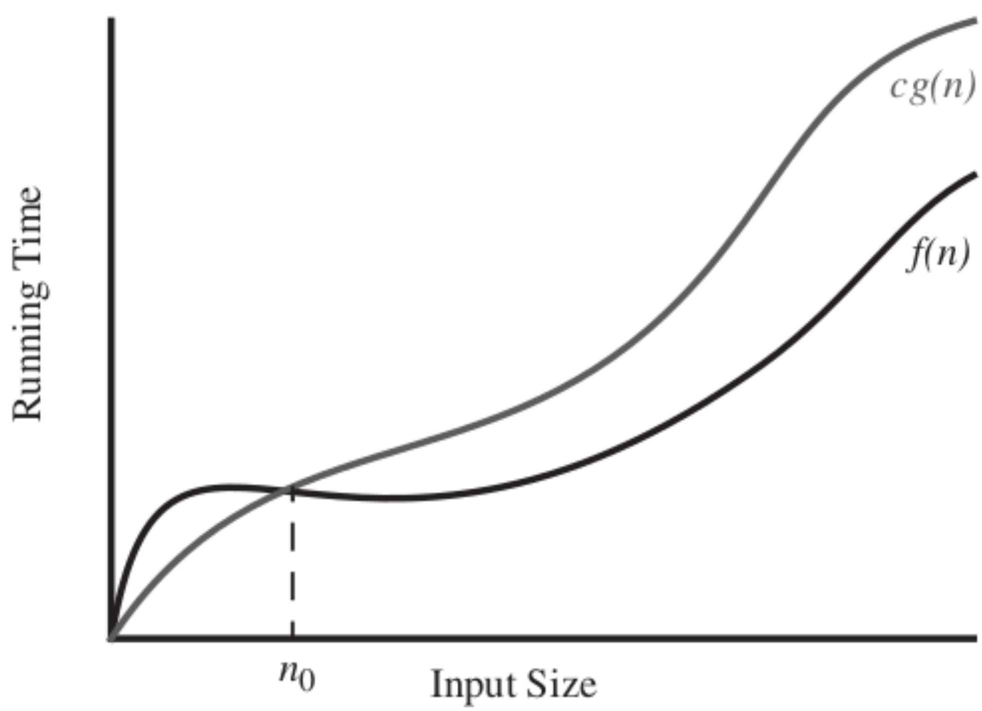
  return $A[0]$

return $\max\{\text{recursiveMax}(A, n-1),\ A[n-1]\}$

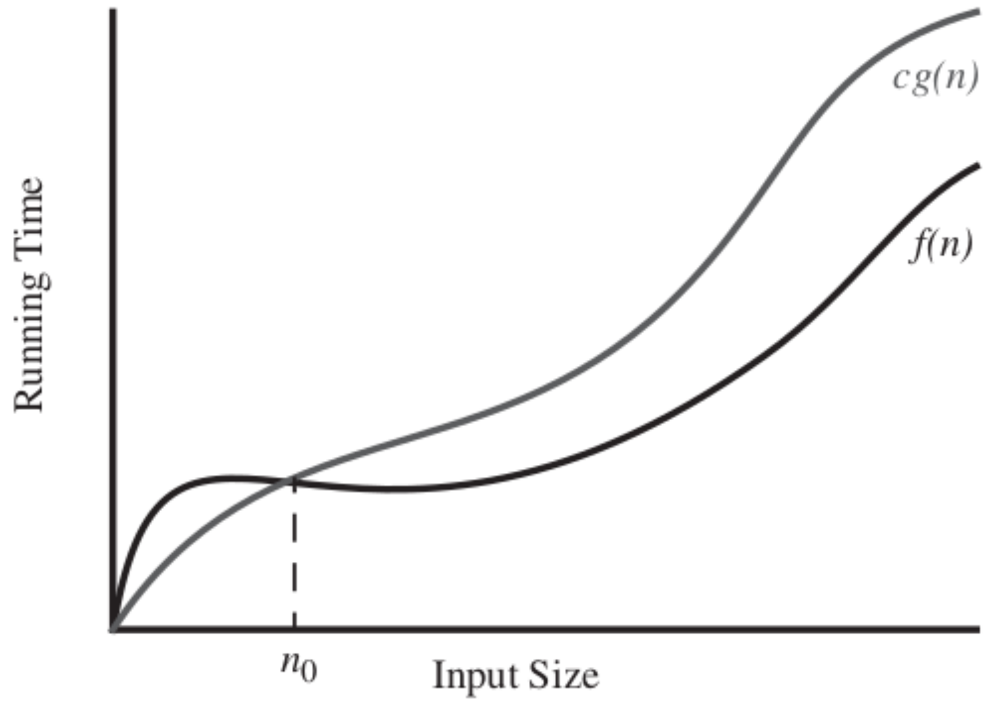
Algorithm 1.4: Algorithm recursiveMax.

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ T(n-1) + 7 & \text{otherwise} \end{cases}$$

$$T(n) = 7(n-1) + 3 = 7n - 4$$



The function $f(n)$ is $O(g(n))$, for $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.



The function $f(n)$ is $O(g(n))$, for $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

Let f(n) and g(n) be functions mapping non-negative integers to real numbers.

We say "f(n) is O(g(n))", or "f(n) is order of g(n)", if there exists a real constant c > 0 and an integer constant n0 >= 1 such that f(n) <= c.g(n) for every integer n >= n0.

Example 1.1: f(n) = 7n - 2 is O(n).

Proof: We need a real constant c > 0 and an integer constant n0 >= 1 such that
(7n-2) <= c.n for every integer n >= n0. One possible choice is c = 7, and n0 = 1.

Corollary: The running time of arrayMax is O(n).

Example 1.3: $20n^3 + 10n \log n + 5$ *is* $O(n^3)$.

Proof: $20n^3 + 10n \log n + 5 \leq 35n^3$, *for* $n \geq 1$.

Note - If f(n) is a polynomial function of degree k, then f(n) will always be O(n^k).

Example 1.4: f(n) = 3.log(n) + log(log(n)) is O(log n).
Proof: We can choose c=4 and n0 = 2.

Example 1.5: 2^{100} *is* $O(1)$.

Proof: $2^{100} \leq 2^{100} \cdot 1$, *for* $n \geq 1$. *Note that variable* n *does not appear in the inequality, since we are dealing with constant-valued functions.* ■

We say "f(n) is Ω(g(n))", or "f(n) is big-Omega of g(n)", if there exists a real constant $c > 0$ and an integer constant $n0 >= 1$ such that $f(n) >= c.g(n)$ for every integer $n >= n0$.

Example 1.9: $3\log n + \log\log n$ *is* $\Omega(\log n)$.

Proof: $3\log n + \log\log n \geq 3\log n$, *for* $n \geq 2$.

We say "f(n) is Θ(g(n))", or "f(n) is big-Theta of g(n)",
if there exists real constants $c1, c2 > 0$ and an integer constant $n0 >= 1$
such that $f(n) <= c1.g(n)$ and $f(n) >= c2.g(n)$ for every integer $n >= n0$.

Example 1.10: $3\log n + \log\log n$ *is* $\Theta(\log n)$.

$f(n) = 1000\ n$

$g(n) = 2\ n^2$

Find smallest n where:
$g(n) >= f(n)$

$2.n^2 >= 1000\ n$
$n >= 500$

n	f(n)	g(n)	f(n) / f(n-1)	g(n) / g(n-1)
1	1000	2	-	-
2	2000	8	2	4
3	3000	18	1.5	2.25
4	4000	32	1.33	1.78
5	5000	50	1.2	1.5625

....

Scalability

1) $f(n)$ is $O(g(n))$ => $g(n)$ is $\Omega(f(n))$

2) $f(n)$ is $\Omega(g(n))$ => $g(n)$ is $O(f(n))$

3) $f(n)$ is $\Theta(g(n))$ <=> $f(n)$ is $O(g(n))$ AND $f(n)$ is $\Omega(g(n))$

To prove: $f(n)$ is $\Theta(g(n))$ <=> $g(n)$ is $\Theta(f(n))$

$f(n)$ is $\Theta(g(n))$ => $f(n)$ is $O(g(n))$ AND $f(n)$ is $\Omega(g(n))$ [apply rule 3]
=> $g(n)$ is $\Omega(f(n))$ AND $g(n)$ is $O(f(n))$ [apply rules 1 and 2]
=> $g(n)$ is $\Theta(f(n))$ [apply rule 3]

$f(n) = 2.n^2 + 4.n^3$
$= d(n) + e(n)$
$= O(n^2 + n^3)$
$= O(g(n))$

$g(n) = O(n^3)$

Thus, $f(n) = O(n^3)$

$d(n) = 2.n^2$ is $O(n^2)$

$e(n) = 4.n^3$ is $O(n^3)$

$g(n) = n^2 + n^3$
Now, $g(n)$ is a polynomial of degree 3,
$g(n)$ is $O(n^3)$.

$f(n) = 3.\log(n) + \log(\log(n))$
$= d(n) + e(n)$
$= O(\log(n) + \log(n))$
$= O(\ 2.\log(n)\ )$
$= O(\ b(n)\ )$

$b(n) = 2.\log(n)$ is $O(\log(n))$

Thus, $f(n)$ is $O(\log(n))$

$d(n) = 3.\log(n)$ is $O(\log(n))$

$e(n) = \log(\log(n))$ is $O(\log(n))$

So, $e(n)$ is $O(\log(n))$

$2n^5$ is $O(n^5)$

$n^5 + 3$ is $O(n^5)$

Input: { 1, 4, 7, 2, 5, 3 }

Output: { 1, 2, 3, 4, 5, 7 }

Bubble Sort --

Initial : 1 4 7 2 5 3

Pass 1: 1 4 2 5 3 7 (swaps = 3)

Pass 2: 1 2 4 3 5 7 (swaps = 2)

Pass 3: 1 2 3 4 5 7 (swaps = 1)

Pass 4: 1 2 3 4 5 7 (swaps = 0)

```
BubbleSort( int[] A, int n )
  Input: An array A containing n >= 1 integers
  Output: The sorted version of the array A

for i = 1 to (n-1)
{
     for j = 0 to (n-2)
          if A[j] > A[j+1]
          {
                 // swap A[j] with A[j+1]
                 tmp <- A[j]
                 A[j] <- A[j+1]
                 A[j+1] <- tmp
           }
}
return A
```

Complexity (best and worst case):

(c*(n-1)) * (n-1) = c*(n-1)^2
= O(n^2)

c = 1 + 3 + 2 + 3 + 2 + 2 = 13

Input: {cat, mat, bat, ant}

Output: {ant, bat, cat, mat}

Input: { 7, 5, 4, 3, 2, 1 }

Output: { 1, 2, 3, 4, 5, 7 }

Bubble Sort --

Initial : 7 5 4 3 2 1

Pass 1: 5 4 3 2 1 7 (swaps = 5)

Pass 2: 4 3 2 1 5 7 (swaps = 4)

Pass 3: 3 2 1 4 5 7 (swaps = 3)

Pass 4: 2 1 3 4 5 7 (swaps = 2)

Pass 5: 1 2 3 4 5 7 (swaps = 1)

```
BubbleSortOptimized( int[] A, int n )
  Input: An array A containing n >= 1 integers
  Output: The sorted version of the array A

for i = 1 to (n-1)
{
     swaps = 0
     for j = 0 to (n-1-i)
          if A[j] > A[j+1]
          {
                 // swap A[j] with A[j+1]
                 tmp <- A[j]
                 A[j] <- A[j+1]
                 A[j+1] <- tmp
                 swaps <- swaps + 1
           }
      if swaps == 0:
          break
}
return A
```

Worst case complexity:
c * [(n-1) + (n-2) + (n-3) + ... + 1]
= c * [(n-1)*n/2]
= O(n^2)

Best case complexity:
c*(n-1)
= O(n)

Input: { 1, 4, 7, 5, 2, 3 }

Output: { 1, 2, 3, 4, 5, 7 }

Inversions:
(4, 2)
(4, 3)
(7, 5)
(7, 2)
(7, 3)
(5, 2)
(5, 3)

Selection Sort --

```
Initial :  1  4  7  5  2  3
Pass 1:    1  4  3  5  2  7
Pass 2:    1  4  3  2  5  7
Pass 3:    1  2  3  4  5  7
Pass 4:    1  2  3  4  5  7
Pass 5:    1  2  3  4  5  7
```

```
SelectionSort( int[] A, int n )
  Input: An array A containing n >= 1 integers
  Output: The sorted version of the array A

for i = 1 to (n-1)
{
     currentMax = A[0]
     maxIndex = 0
     for j = 1 to (n-i)
     {
            if A[j] > currentMax
            {
                  currentMax = A[j]
                  maxIndex = j
            }
     }
     // swap A[maxIndex] with A[n-i]
     tmp <- A[maxIndex]
     A[maxIndex] <- A[n-i]
     A[n-i] <- tmp
}
return A
```

Complexity (best and worst case):

$$c * [(n-1) + (n-2) + (n-3) + \ldots + 1]$$
$$= c * [(n-1)*n/2]$$
$$= O(n^2)$$

Input: { 7, 5, 4, 3, 2, 1 }

Output: { 1, 2, 3, 4, 5, 7 }

Selection Sort --

```
Initial :  7  5  4  3  2  1
Pass 1:    1  5  4  3  2  7
Pass 2:    1  2  4  3  5  7
Pass 3:    1  2  3  4  5  7
Pass 4:    1  2  3  4  5  7
Pass 5:    1  2  3  4  5  7
```

```
SelectionSortOptimized( int[] A, int n )
  Input: An array A containing n >= 1 integers
  Output: The sorted version of the array A

for i = 1 to (n-1)
{
     inversions = 0
     for j = 0 to (n-1-i)
         if A[j] > A[j+1]
                inversions <- inversions + 1
      if inversions == 0:
         break

     currentMax = A[0]
     maxIndex = 0
     for j = 1 to (n-i)
     {
            if A[j] > currentMax
            {
                  currentMax = A[j]
                  maxIndex = j
            }
     }
     // swap A[maxIndex] with A[n-i]
     tmp <- A[maxIndex]
     A[maxIndex] <- A[n-i]
     A[n-i] <- tmp
}
return A
```

Worst case complexity = $O(n^2)$
Best case complexity = $O(n)$

Value:	0	1	2	3	4	5	6	7	8	9
Frequency:	0	1	1	2	0	0	1	2	1	1

A = { 6, 1, 8, 3, 7, 2, 3, 9, 7}

S = { 1, 2, 3, 3, 6, 7, 7, 8, 9 }

Time complexity:

O(R) + O(n) + O(n+R)
= O(2n + 2R)
= O(n + R)
= O(max(n, R))

F[i] = 0 -> O(1)
F[i] > 0 -> O(F[i])

R * O(1) + Sum_i { F[i] } = O(n+R)

R * O(1) + O(n) = O(n+R)

```
CountSort( int[] A, int n, int a, int b )
   // Input: An array of integers of length n,
             where the values are in [a, b]
   // Output: The sorted version of A

   R = b-a+1
   int F[R]
   for i = 0 to (R-1)
       F[i] = 0

   for i = 0 to (n-1)
       tmp = A[i] - a
       F[tmp] = F[tmp] + 1

   int S[n]
   k = 0
   for i = 0 to (R-1)
       freq = F[i]
       for j = 1 to freq
           S[k] = i + a
           k = k+1
   return S
```

For Binary Search, the time complexity is given by the following recurrence:

T(n) = O(1) + T(n/2) => T(n) = O(log n)

For Merge Sort, the time complexity is given by the following recurrence:

T(n) = 2.T(n/2) + O(n) => T(n) = O(n.log(n))

Solving Recurrence Relations --

1) Substitution method: Guess a solution, and then check whether it is correct.

Eg. - Let us guess the solution for Binary Search as T(n) = O(log n), which means that we must have T(n) <= c.(log n) for large enough n (for all n >= n0).

T(n) = O(1) + T(n/2)
<= O(1) + O(log(n/2))
<= c1 + c2.(log(n/2))
= c1 + c2.(log(n) - log(2))
= c1 + c2.log(n) - c2
= c2.log(n) - (c2 - c1)
<= c2.log(n)
= O(log n)

2) Recurrence Tree method: Figure out the solution by studying the recurrence tree.

3) Master's Theorem method:

It is a direct method to get solutions for recurrences of the form T(n) = a.T(n/b) + O(n^c), where a>=1 and b>1. Then, the following three cases are used to obtain the solution directly -

(i) If c < log_b(a), then T(n) = Θ(n^(log_b(a)))
Eg. - For the recurrence T(n) = 16.T(n/4) + O(n), we have: a=16, b=4, c=1
Therefore, 1 = c < log_b(a) = log4(16) = 2, and so we have: T(n) = Θ(n^2)

(ii) If c = log_b(a), then T(n) = Θ(n^c. log(n))
Eg. - For binary search recurrence T(n) = 1.T(n/2) + O(1), we have: a=1, b=2, c=0
Therefore, c = log_b(a) = log2(1) = 0, and so we have: T(n) = Θ(n^0. log(n))
Eg. - For merge sort recurrence T(n) = 2.T(n/2) + O(n), we have: a=2, b=2, c=1
Therefore, c = log_b(a) = log2(2) = 1, and so we have: T(n) = Θ(n^1. log(n))

(iii) If c > log_b(a), then T(n) = Θ(n^c)
Eg. - For the recurrence T(n) = 2.T(n/4) + O(n^2), we have: a=
Therefore, 2 = c > log_b(a) = log4(2) = 0.5, and so we have: T(n) = Θ(n^2)

T(n) = 2.T(n/2) + O(n)
= 2.(2.T(n/4) + O(n/2)) + O(n)
= 2.(2.(2.T(n/8) + O(n/4)) + O(n/2)) + O(n)
....
....
= 2^k.T(n/(2^k))) + (O(n) + 2.O(n/2) + 4.O(n/4) + ... + (2^k).O(n/(2^k)))
= n.T(1) + (O(n) + O(n) + ..(k times).. + O(n))
= n + k.O(n)
= n + O(n).log(n) = O(n. log(n))

```
LinearSearch(int[] A, int n, int key)
    pos = -1
    for i = 0 to (n-1)
          if (A[i] == key)
          {
                pos = i
                break
           }
    if pos == -1
           print "Not found!"
    else
           print "Found!"
    return pos
```

A = { 1, 7, 4, 3, 9, 6, 2 }

key = 5 ---> Not found - failed!

key = 4 ---> Found, at index 2!

Worst case time complexity: O(n)

Best case time complexity: O(1)

A = { 1, 3, 4, 6, 8, 9, 11 }
key = 4

Comparison 1: key < 6

A' = { 1, 3, 4 }
key = 4

Comparison 2: key > 3

A'' = { 4 }
key = 4

Comparison 3: key == 4

A = { 1, 3, 4, 6, 8, 9, 11 }
key = 8

Comparison 1: key > 6

A' = { 8, 9, 11 }
key = 8

Comparison 2: key < 9

A'' = { 8 }
key = 8

Comparison 3: key == 8

Algorithm BinarySearch$(A, k, \text{low}, \text{high})$:

Input: An ordered array, A, storing n items, whose keys are accessed with method key(i) and whose elements are accessed with method elem(i); a search key k; and integers low and high

Output: An element of A with key k and index between low and high, if such an element exists, and otherwise the special element ***null***

if low $>$ high **then**
    return ***null***
else
    mid $\leftarrow \lfloor(\text{low} + \text{high})/2\rfloor$
    if $k = $ key(mid) **then**
        return elem(mid)
    else if $k <$ key(mid) **then**
        return BinarySearch$(A, k, \text{low}, \text{mid} - 1)$
    else
        return BinarySearch$(A, k, \text{mid} + 1, \text{high})$

Worst case time complexity: O(log n)

A = { 1, 3, 4, 6, 8, 9, 11 }
key = 2

Comparison 1: key < 6

A' = { 1, 3, 4 }
key = 2

Comparison 2: key < 3

A'' = { 1 }
key = 2

Comparison 3: key > 1

A''' = {}
Return "Not found"

```
BinarySearch( A, key, low, high )

if (low > high)          // if A is an empty array
      return (-1)        // "Not found"

mid <-  ⌊ (low + high) / 2 ⌋

if key == A[mid]
      return mid

else if key < A[mid]
      BinarySearch( A, key, low, mid-1 )

else if key > A[mid]
      BinarySearch( A, key, mid+1, high )
```

low = 6
high = 7
mid = (6+7)/2 = 6

key < A[6]   =>
low = 6, high = 5

key > A[6]   =>
low = 7, high = 7

A = { 1, 3, 4, 6, 8, 9, 11 }
key = 2

low = 0
high = 7
mid = ⌊(0+7)/2⌋ = 3

Comparison 1: key < A[3] = 6

low = 0
high = mid-1 = 3-1 = 2
mid = ⌊(0+2)/2⌋ = 1

Comparison 2: key < A[1] = 3

low = 0
high = mid-1 = 1-1 = 0
mid = ⌊(0+0)/2⌋ = 0

Comparison 3: key > A[0] = 1

low = 0
high = mid-1 = 0-1 = -1

Since (low > high) => "Not found"

A = { 1, 3, 4, 6, 8, 9, 11 }
key = 8

low = 0
high = 7
mid = ⌊(0+7)/2⌋ = 3

Comparison 1: key > A[3] = 6

low = mid+1 = 3+1 = 4
high = 7
mid = ⌊(4+7)/2⌋ = 5

Comparison 2: key < A[5] = 9

low = 4
high = mid-1 = 5-1 = 4
mid = ⌊(4+4)/2⌋ = 4

Comparison 3: key == A[4] = 8

Return (mid) => "Found at index 4"

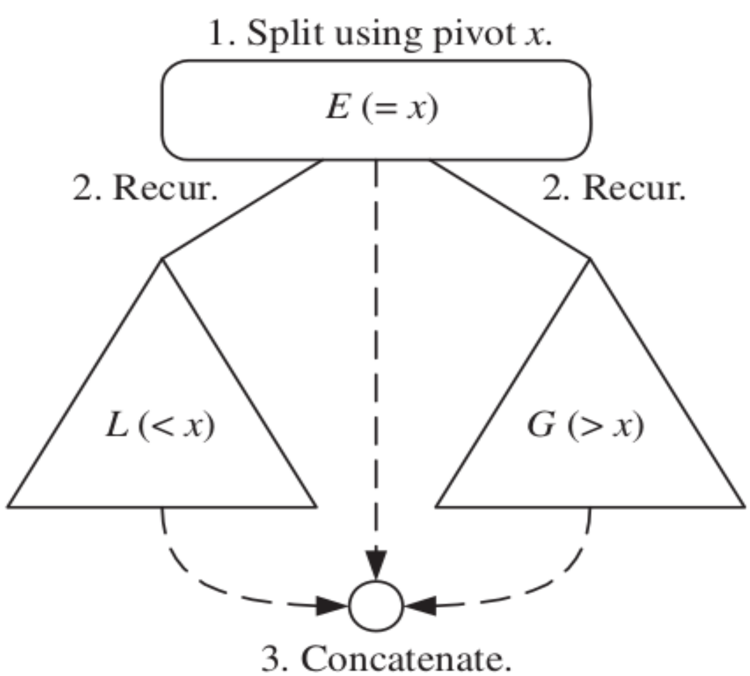
QUICK SORT - Steps:

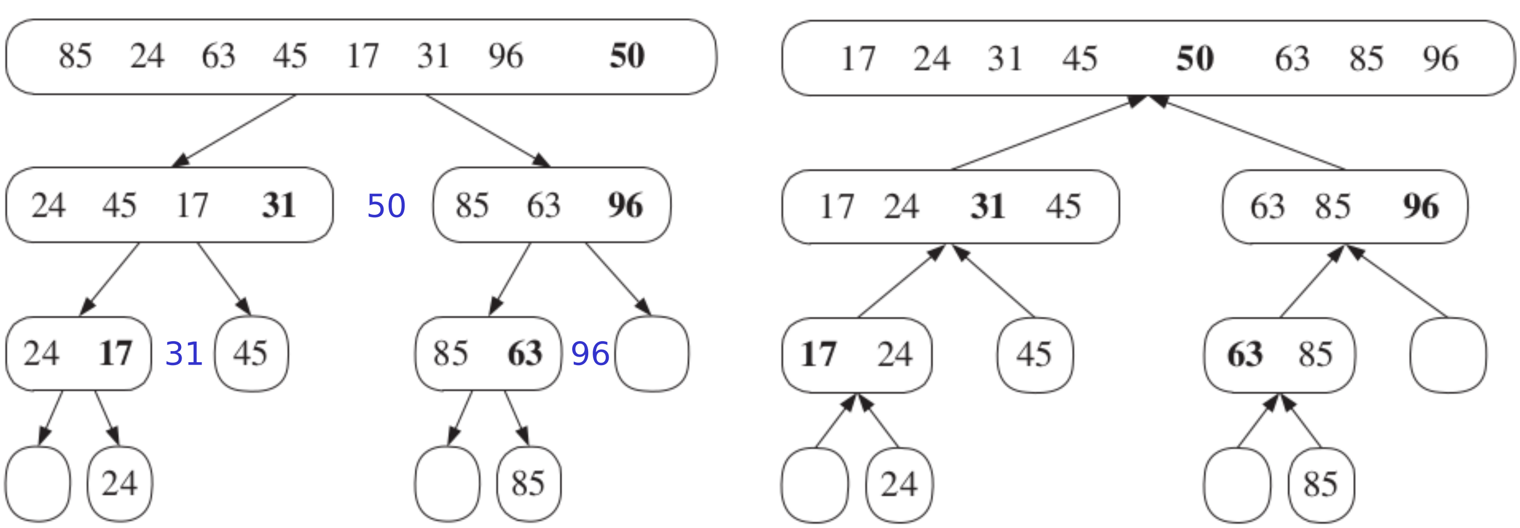
1. ***Divide:*** If S has at least two elements (nothing needs to be done if S has zero or one element), select a specific element x from S, which is called the ***pivot***. As is common practice, choose the pivot x to be the last element in S. Remove all the elements from S and put them into three sequences:

    - L, storing the elements in S less than x
    - E, storing the elements in S equal to x
    - G, storing the elements in S greater than x.

    (If the elements of S are all distinct, E holds just one element—the pivot.)

2. ***Recur:*** Recursively sort sequences L and G.
3. ***Conquer:*** Put the elements back into S in order by first inserting the elements of L, then those of E, and finally those of G.





{85 24 63 45 50 31 96 17}

L={} E={17} G={85 24 63 45 50 31 96}

{85 24 63 45 17 31 50 96}

L={85 24 63 45 31 50} E={96} G={}

```
partition (arr[], low, high)
{
   pivot = arr[high];
   i = (low - 1)
   for (j = low; j < high; j++)
   {
      // If current element is smaller than the pivot
      if (arr[j] < pivot)
      {
         i++;    // increment index of smaller element
         swap arr[i] and arr[j]
      }
   }
   swap arr[i+1] and arr[high])
   return (i + 1)
}
```

Initial: 85 24 63 45 17 31 96 50
low = 0, high = 7

Pivot: arr[7] = 50

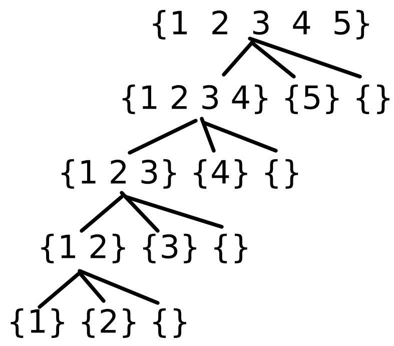
i = 3
j = 6

```
85  24  63  45  17  31  96  50
24  85  63  45  17  31  96  50
24  45  63  85  17  31  96  50
24  45  17  85  63  31  96  50
24  45  17  31  63  85  96  50
24  45  17  31  50  85  96  63
```

24 45 17 31 | 50 | 85 96 63

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)    // i.e. if arr[] is of length >= 2
   {
      pi = partition(arr, low, high);
      /* pi is partitioning index, i.e. index of pivot */

      quickSort(arr, low, pi - 1);    // Before pi, i.e. L
      quickSort(arr, pi + 1, high); // After pi, i.e. G
   }
}
```



Worst case time compexity:

(n-1) levels, and
O(n) operations at each level
(counted over all partition functions)
= O(n^2)

```
partition (arr[], low, high)
{
   pivot = arr[high];
   i = low
   for (j = low; j < high; j++)
   {
      // If current element is smaller than the pivot
      if (arr[j] < pivot)
      {
         swap arr[i] and arr[j]
         i++;    // increment index of smaller element
      }
   }
   swap arr[i] and arr[high])
   return (i)
}
```

Best case time complexity:

O(log(n)) levels, and
O(n) operations at each level
(counted over all partition functions)
= O(n log(n))

Average case time complexity:
O(n log(n))

Input: { 1, 4, 7, 5, 2, 3 }

Output: { 1, 2, 3, 4, 5, 7 }

Inversions:
(4, 2)
(4, 3)
(7, 5)
(7, 2)
(7, 3)
(5, 2)
(5, 3)

Insertion Sort --

```
Initial :  1  4  7  5  2  3
Pass 1:    1  4 | 7  5  2  3      4>1
Pass 2:    1  4  7 | 5  2  3      7>4
Pass 3:    1  4  5  7 | 2  3      5<7,  5>4
Pass 4:    1  2  4  5  7 | 3      2<7, 2<5, 2<4, 2>1
Pass 5:    1  2  3  4  5  7       3<7, 3<5, 3<4, 3>2
```

Pass 4: j=3, j=2, j=1, j=0
Elements at indices (j+1) to (i-1) to be shifted

```
1 4 5 7 2 3
1 4 5 _ 7 3      tmp <- 2
1 4 _ 5 7 3      tmp <- 2
1 _ 4 5 7 3      tmp <- 2
1 2 4 5 7 3
```

InsertionSort(int[] A, int n)
  Input: An array A containing n >= 1 integers
  Output: The sorted version of the array A

```
for i = 1 to (n-1)
{
     j <- i - 1
     while A[i] < A[j]
            j <- j - 1
            if j < 0
                break
     tmp <- A[i]
     // shift all elements > A[i] by 1 position
     k = i - 1
     while k >= j+1
     {
            A[k+1] <- A[k]
            k <- k - 1
     }
     // insert A[i] in position (j+1)
     A[j+1] <- tmp
}
return A
```

{ 4, 1, ... }

Worst case complexity = O(n^2)
Best case complexity = O(n)

Input: { 7, 5, 4, 3, 2, 1 }

Output: { 1, 2, 3, 4, 5, 7 }

Insertion Sort --

```
Initial :  7  5  4  3  2  1
Pass 1:    5  7 | 4  3  2  1      5<7
Pass 2:    4  5  7 | 3  2  1      4<7, 4<5
Pass 3:    3  4  5  7 | 2  1      3<7, 3<5, 3<4
Pass 4:    2  3  4  5  7 | 1
Pass 5:    1  2  3  4  5  7
```

InsertionSortOptimized(int[] A, int n)
  Input: An array A containing n >= 1 integers
  Output: The sorted version of the array A

```
for i = 1 to (n-1)
{
     inversions = 0
     for j = 0 to (n-1)
         if A[j] > A[j+1]
                inversions <- inversions + 1
      if inversions == 0:
         break

      j <- i - 1
     while A[i] < A[j]
            j <- j - 1
            if j < 0
                break
     tmp <- A[i]
     // shift all elements > A[i] by 1 position
     k = i - 1
     while k >= j+1
     {
            A[k+1] <- A[k]
            k <- k - 1
     }
     // insert A[i] in position (j+1)
     A[j+1] <- tmp
}
return A
```

Worst case complexity = O(n^2)
Best case complexity = O(n)

We say "f(n) is Ω(g(n))", or "f(n) is big-Omega of g(n)", if there exists a real constant c > 0 and an integer constant n0 >= 1 such that f(n) >= c.g(n) for every integer n >= n0.

Example 1.9: $3\log n + \log\log n$ *is* $\Omega(\log n)$.

Proof: $3\log n + \log\log n \geq 3\log n$, *for* $n \geq 2$.

We say "f(n) is Θ(g(n))", or "f(n) is big-Theta of g(n)",
if there exists real constants c1, c2 > 0 and an integer constant n0 >= 1
such that f(n) <= c1.g(n) and f(n) >= c2.g(n) for every integer n >= n0.

Example 1.10: $3\log n + \log\log n$ *is* $\Theta(\log n)$.

NB - 1) If f(n) is O(g(n)), then g(n) must be Ω(f(n))
2) If f(n) is Ω(g(n)), then g(n) must be O(f(n))
3) If f(n) is Θ(g(n)), then f(n) must be both O(g(n)) and Ω(g(n))
4) If f(n) is Θ(g(n)), then also g(n) must be Θ(f(n))

Theorem 1.7: *Let $d(n)$, $e(n)$, $f(n)$, and $g(n)$ be functions mapping nonnegative integers to nonnegative reals.*

1. *If $d(n)$ is $O(f(n))$, then $ad(n)$ is $O(f(n))$, for any constant $a > 0$.*
2. *If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n)+e(n)$ is $O(f(n)+g(n))$.*
3. *If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n)e(n)$ is $O(f(n)g(n))$.*
4. *If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$, then $d(n)$ is $O(g(n))$.*
5. *If $f(n)$ is a polynomial of degree d (that is, $f(n) = a_0 + a_1 n + \cdots + a_d n^d$), then $f(n)$ is $O(n^d)$.*
6. *n^x is $O(a^n)$ for any fixed $x > 0$ and $a > 1$.*
7. *$\log n^x$ is $O(\log n)$ for any fixed $x > 0$.*
8. *$\log^x n$ is $O(n^y)$ for any fixed constants $x > 0$ and $y > 0$.*

---

d(n) = 2n^2 is O(n^2)
e(n) = 4n^3 is O(n^3)
d(n) + e(n) = 2n^2 + 4n^3 is O(n^2 + n^3) = O(n^3)

f(n) = 10^100 . n
g(n) = 0.01 . n^2

efficient => polynomial running time => O(n^k) for some constant k > 0
not efficient => exponential running time => O(2^n)

f(n) = 3 . n^1001
g(n) = 2 . 2^n

It is considered poor taste to include constant factors and lower order terms in the big-Oh notation. For example, it is not fashionable to say that the function $2n^2$ is $O(4n^2 + 6n\log n)$, although this is completely correct. We should strive instead to describe the function in the big-Oh in ***simplest terms***.

logarithmic	**linear**	**quadratic**	**polynomial**	**exponential**
$O(\log n)$	$O(n)$	$O(n^2)$	$O(n^k)$ $(k \geq 1)$	$O(a^n)$ $(a > 1)$

A few words of caution about asymptotic notation are in order at this point. First, note that the use of the big-Oh and related notations can be somewhat misleading should the constant factors they "hide" be very large. For example, while it is true that the function $10^{100}n$ is $\Theta(n)$, if this is the running time of an algorithm being compared to one whose running time is $10n\log n$, we should prefer the $\Theta(n\log n)$-time algorithm, even though the linear-time algorithm is asymptotically faster. This preference is because the constant factor, 10^{100}, which is called "one googol," is believed by many astronomers to be an upper bound on the number of atoms in the observable universe. So we are unlikely to ever have a real-world problem that has this number as its input size. Thus, even when using the big-Oh notation, we should at least be somewhat mindful of the constant factors and lower order terms we are "hiding."

Some Functions Ordered by Growth Rate	Common Name
$\log n$	logarithmic
$\log^2 n$	polylogarithmic
$\sqrt{n}$	square root
n	linear
$n\log n$	linearithmic
n^2	quadratic
n^3	cubic
2^n	exponential

n	$\log n$	$\log^2 n$	$\sqrt{n}$	$n\log n$	n^2	n^3	2^n
4	2	4	2	8	16	64	16
16	4	16	4	64	256	4,096	65,536
64	6	36	8	384	4,096	262,144	1.84×10^{19}
256	8	64	16	2,048	65,536	16,777,216	1.15×10^{77}
1,024	10	100	32	10,240	1,048,576	1.07×10^{9}	1.79×10^{308}
4,096	12	144	64	49,152	16,777,216	6.87×10^{10}	10^{1233}
16,384	14	196	128	229,376	268,435,456	4.4×10^{12}	10^{4932}
65,536	16	256	256	1,048,576	4.29×10^{9}	2.81×10^{14}	10^{19728}
262,144	18	324	512	4,718,592	6.87×10^{10}	1.8×10^{16}	10^{78913}

We say that "f(n) is o(g(n))", or "f(n) is little-oh of g(n)", if for any constant c > 0, there exists a constant n0 > 0 such that f(n) <= c.g(n) for n >= n0.

We say that "f(n) is ω(g(n))", or "f(n) is little-omega of g(n)", if g(n) is o(f(n)).

Example 1.11: *The function $f(n) = 12n^2 + 6n$ is $o(n^3)$ and $\omega(n)$.*

Proof: *Let us first show that $f(n)$ is $o(n^3)$. Let $c > 0$ be any constant. If we take $n_0 = (12 + 6)/c = 18/c$, then $18 \leq cn$, for $n \geq n_0$. Thus, if $n \geq n_0$,*

$$f(n) = 12n^2 + 6n \leq 12n^2 + 6n^2 = 18n^2 \leq cn^3.$$

Thus, $f(n)$ is $o(n^3)$.

To show that $f(n)$ is $\omega(n)$, let $c > 0$ again be any constant. If we take $n_0 = c/12$, then, for $n \geq n_0$, $12n \geq c$. Thus, if $n \geq n_0$,

$$f(n) = 12n^2 + 6n \geq 12n^2 \geq cn.$$

Thus, $f(n)$ is $\omega(n)$. ■

For the reader familiar with limits, we note that $f(n)$ is $o(g(n))$ if and only if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0,$$

provided this limit exists. The main difference between the little-oh and big-Oh notions is that $f(n)$ is $O(g(n))$ if ***there exist*** constants $c > 0$ and $n_0 \geq 1$ such that $f(n) \leq cg(n)$, for $n \geq n_0$; whereas $f(n)$ is $o(g(n))$ if ***for all*** constants $c > 0$ there is a constant n_0 such that $f(n) \leq cg(n)$, for $n \geq n_0$. Intuitively, $f(n)$ is $o(g(n))$ if $f(n)$ becomes insignificant compared to $g(n)$ as n grows toward infinity. As previously mentioned, asymptotic notation is useful because it allows us to concentrate on the main factor determining a function's growth.

Example of 'polylogarithmic' function : 11(log n)^4 + 5(log n)^2 + 3(log n) + 2

An algorithm is a step-by-step procedure for performing some task in a finite amount of time.

Experimental studies on running times are useful, but they have some limitations:

- Experiments can be done only on a limited set of test inputs, and care must be taken to make sure these are representative.
- It is difficult to compare the efficiency of two algorithms unless experiments on their running times have been performed in the same hardware and software environments.
- It is necessary to implement and execute an algorithm in order to study its running time experimentally.

Thus, while experimentation has an important role to play in algorithm analysis, it alone is not sufficient. Therefore, in addition to experimentation, we desire an analytic framework that

- Takes into account all possible inputs
- Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent from the hardware and software environment
- Can be performed by studying a high-level description of the algorithm without actually implementing it or running experiments on it.

"Algorithm A runs in time proportional to n" =>
If we were to perform experiments, then we would find that the actual running time of algorithm A on any input of size n never exceeds c.n, where c is a constant that depends on the hardware and software environment used.

Given two algorithms A and B, where A runs in time proportional to n and B runs in time proportional to n^2 , we will prefer A to B, since the function n grows at a smaller rate than the function n^2 .

"Algorithm A runs in time proportional to n" =>
If we were to perform experiments, then we would find that the actual running time of algorithm A on any input of size n never exceeds c.n, where c is a constant that depends on the hardware and software environment used.

- A language for describing algorithms
- A computational model that algorithms execute within
- A metric for measuring algorithm running time
- An approach for characterizing running times, including those for recursive algorithms.

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.
  currentMax $\leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if *currentMax* $< A[i]$ **then**
      currentMax $\leftarrow A[i]$
  return *currentMax*

By inspecting the pseudocode, we can argue about the correctness of algorithm arrayMax with a simple argument. Variable *currentMax* starts out being equal to the first element of A. We claim that at the beginning of the ith iteration of the loop, *currentMax* is equal to the maximum of the first i elements in A. Since we compare *currentMax* to $A[i]$ in iteration i, if this claim is true before this iteration, it will be true after it for $i + 1$ (which is the next value of counter i). Thus, after $n - 1$ iterations, *currentMax* will equal the maximum element in A. As with this example, we want our pseudocode descriptions to always be detailed enough to fully justify the correctness of the algorithm they describe, while being simple enough for human readers to understand.

- Assigning a value to a variable
- Calling a method
- Performing an arithmetic operation
- Comparing two numbers
- Indexing into an array
- Following an object reference
- Returning from a method.

Specifically, a primitive operation corresponds to a low-level instruction with an execution time that depends on the hardware and software environment but is nevertheless constant. Instead of trying to determine the specific execution time of each primitive operation, we will simply ***count*** how many primitive operations are executed, and use this number t as a high-level estimate of the running time of the algorithm. This operation count will correlate to an actual running time in a specific hardware and software environment, for each primitive operation corresponds to a constant-time instruction, and there are only a fixed number of primitive operations. The implicit assumption in this approach is that the running times of different primitive operations will be fairly similar. Thus, the number, t, of primitive operations an algorithm performs will be proportional to the actual running time of that algorithm.

RAM (Random Access Machine) Model -
A computer is viewed simply as a CPU connected to a bank of memory cells.
Each memory cell stores a word, which can be a number, a string, or an address.
The term "random access" refers to the ability of the CPU to access an arbitrary memory location using just one single primitive operation.
We assume the CPU in the RAM model can perform any primitive operation in a constant number of steps, which do not depend on the size of the input.

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

currentMax $\leftarrow A[0]$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  if *currentMax* $< A[i]$ **then**
    currentMax $\leftarrow A[i]$
return *currentMax*

- Initializing the variable *currentMax* to $A[0]$ corresponds to two primitive operations (indexing into an array and assigning a value to a variable) and is executed only once at the beginning of the algorithm. Thus, it contributes two units to the count.
- At the beginning of the for loop, counter i is initialized to 1. This action corresponds to executing one primitive operation (assigning a value to a variable).
- Before entering the body of the for loop, condition $i < n$ is verified. This action corresponds to executing one primitive instruction (comparing two numbers). Since counter i starts at 1 and is incremented by 1 at the end of each iteration of the loop, the comparison $i < n$ is performed n times. Thus, it contributes n units to the count.
- The body of the for loop is executed $n - 1$ times (for values $1, 2, \ldots, n - 1$ of the counter). At each iteration, $A[i]$ is compared with *currentMax* (two primitive operations, indexing and comparing), $A[i]$ is possibly assigned to *currentMax* (two primitive operations, indexing and assigning), and the counter i is incremented (two primitive operations, summing and assigning). Hence, at each iteration of the loop, either four or six primitive operations are performed, depending on whether $A[i] \leq$ *currentMax* or $A[i] >$ *currentMax*. Therefore, the body of the loop contributes between $4(n - 1)$ and $6(n - 1)$ units to the count.
- Returning the value of variable *currentMax* corresponds to one primitive operation, and is executed only once.

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.

```
currentMax ← A[0]
for i ← 1 to n − 1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

To summarize, the number of primitive operations $t(n)$ executed by algorithm arrayMax is at least

$$2 + 1 + n + 4(n - 1) + 1 = 5n$$

and at most

$$2 + 1 + n + 6(n - 1) + 1 = 7n - 2.$$

The best case ($t(n) = 5n$) occurs when $A[0]$ is the maximum element, so that variable *currentMax* is never reassigned. The worst case ($t(n) = 7n - 2$) occurs when the elements are sorted in increasing order, so that variable *currentMax* is reassigned at each iteration of the for loop.

We will, for the remainder of this course, typically characterize running times in terms of the worst case. We say, for example, that algorithm arrayMax executes t(n) = 7n − 2 primitive operations in the worst case, meaning that the maximum number of primitive operations executed by the algorithm, taken over all inputs of size n, is 7n − 2.

This type of analysis is much easier than an average-case analysis, as it does not require probability theory; it just requires the ability to identify the worst-case input, which is often straightforward. In addition, taking a worst-case approach can actually lead to better algorithms. Making the standard of success that of having an algorithm perform well in the worst case necessarily requires that it perform well on every input.