Design and Analysis of Algorithms

Dina El-Manakhly, Ph. D.

dina_almnakhly@science.suez.edu.eg

Lecture 4(30/10/2022)

Some standard algorithms that follow Divide and Conquer algorithm

- ❑ Binary Search
- ❑ Merge Sort
- ❑ $\boxed{\text{Quick Sort}}$
- ❑ Closest Pair of Points
- ❑ Strassen’s Algorithm (matrix multiplication)
- ❑ Finding maximum and minimum

Quick Sort Algorithm

❑ Definition

Problem Definition: Given an array A=(a_1, a_2,…, a_n) of n elements. Sorting the array is rearrangement the elements of the array such that $a_i \leq a_{i+1}$, $1 \leq i \leq n-1$.

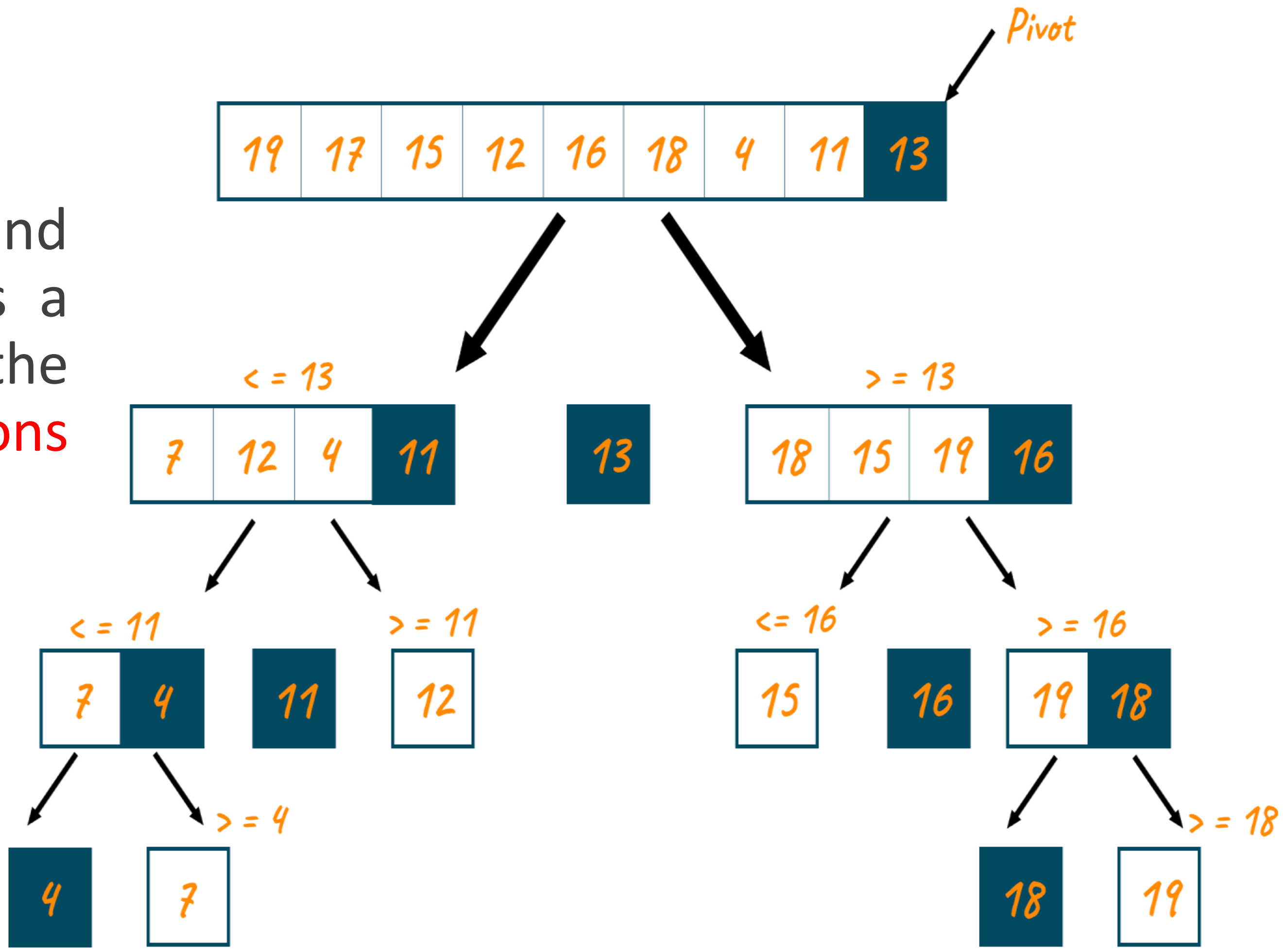
Example 1: Given A=(20,4,10,17,6,7,2)

Goal A=(2,4,6,7,10,17,20)

Quick Sort Algorithm

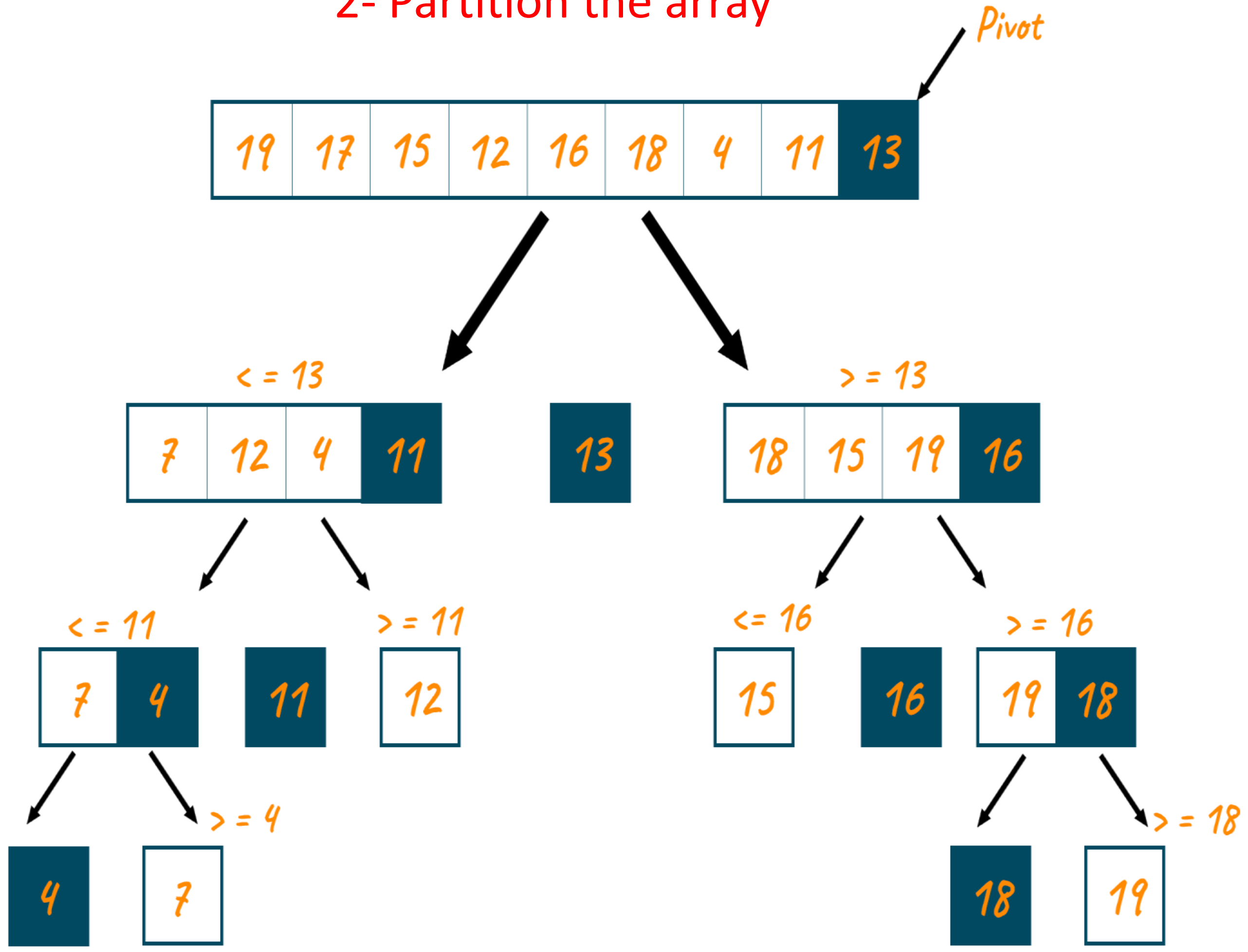
- Like Merge Sort, **QuickSort** is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways:

  - Pick the first element as a pivot.
  - Pick the last element as a pivot
  - Pick a random element as a pivot.
  - Pick median as the pivot.



Quick Sort Algorithm

1- Select the pivot
2- Partition the array



- The key process in **quickSort** is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.

Stop recursion at one Element

Partition Algorithm

Partitioning the array

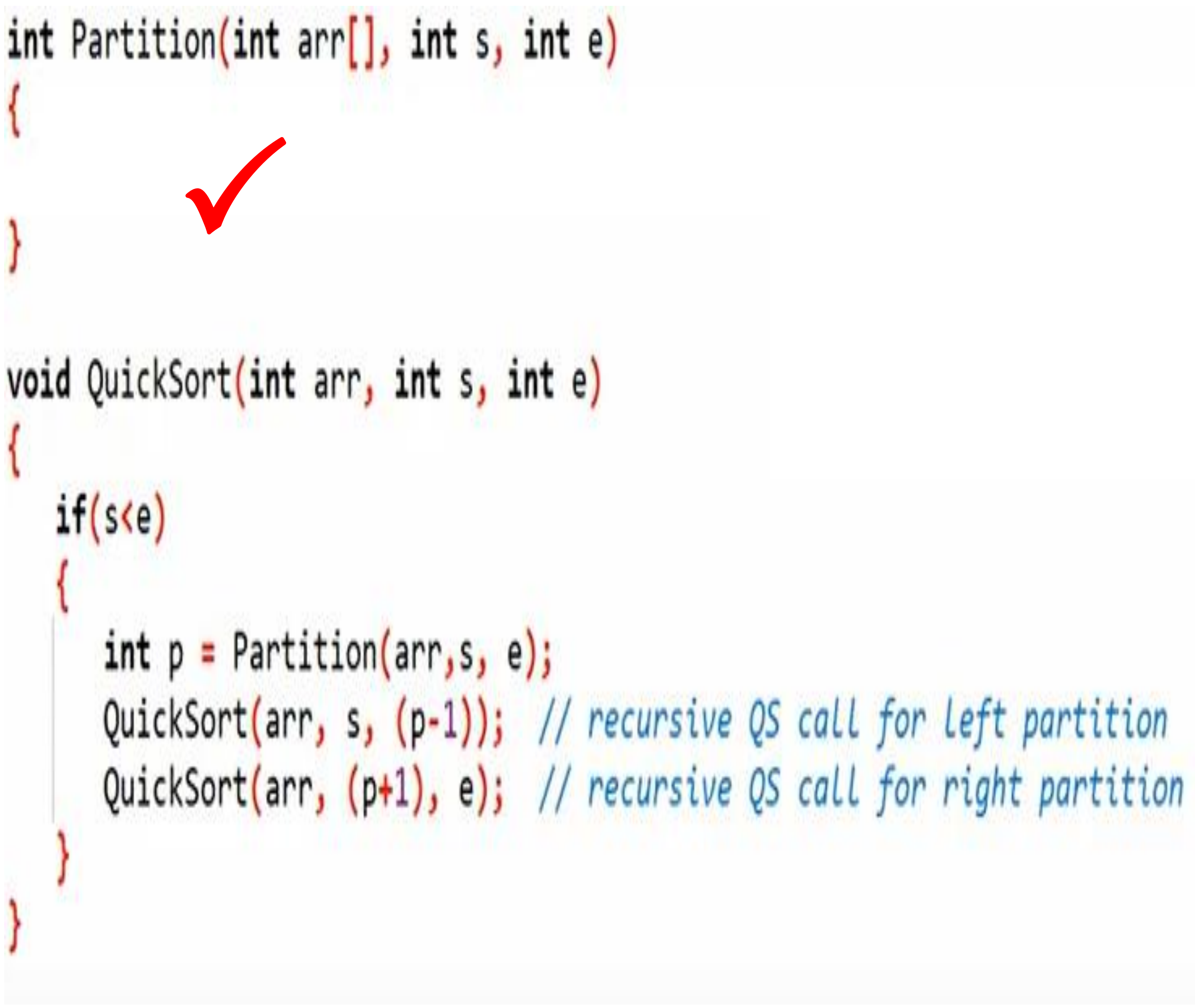
The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p \,..\, r]$ in place.

```
PARTITION($A, p, r$)
1  $x = A[r]$
2  $i = p - 1$
3  for $j = p$ to $r - 1$
4        if $A[j] \leq x$
5            $i = i + 1$
6            exchange $A[i]$ with $A[j]$
7  exchange $A[i + 1]$ with $A[r]$
8  return $i + 1$
```

https://www.youtube.com/watch?v=MZaf_9IZCrc

Pseudo Code

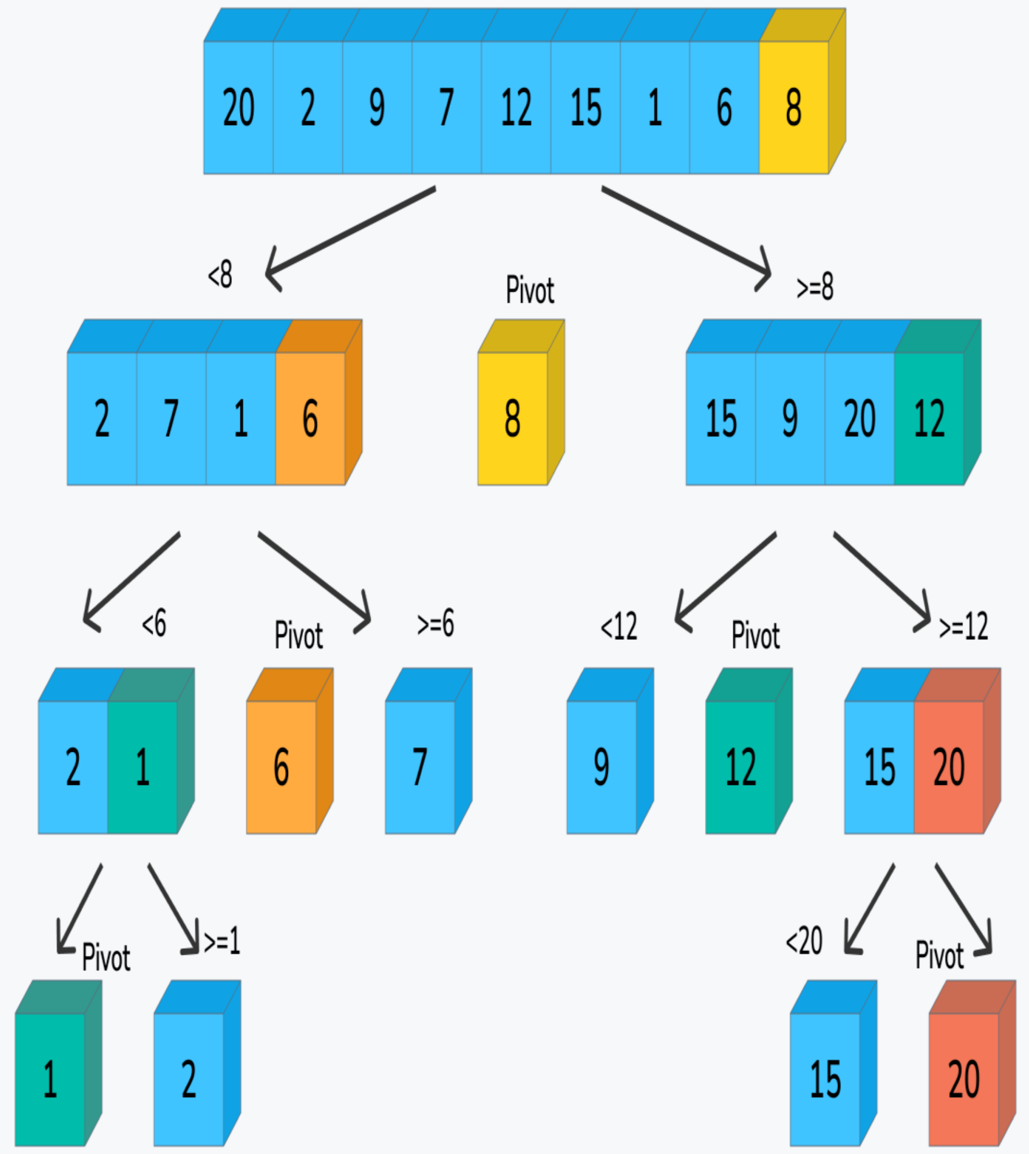
```
QuickSort(A, p, q)
if p < q
   then r <- Partition(A, p, q)
         QuickSort( A, p, r-1)
         QuickSort( A, r+1, q)
```



```
int Partition(int arr[], int s, int e)
{
        ✓

}

void QuickSort(int arr, int s, int e)
{
   if(s<e)
   {
      int p = Partition(arr,s, e);
      QuickSort(arr, s, (p-1));  // recursive QS call for left partition
      QuickSort(arr, (p+1), e);  // recursive QS call for right partition
   }
}
```



Tracing Merge Sort   Algorithm

Basic time analysis

- The main goal is to learn how to analyze algorithms for efficiency.

- The analysis of algorithms attempts to optimize some critical resource. The critical resource usually chosen is running time.

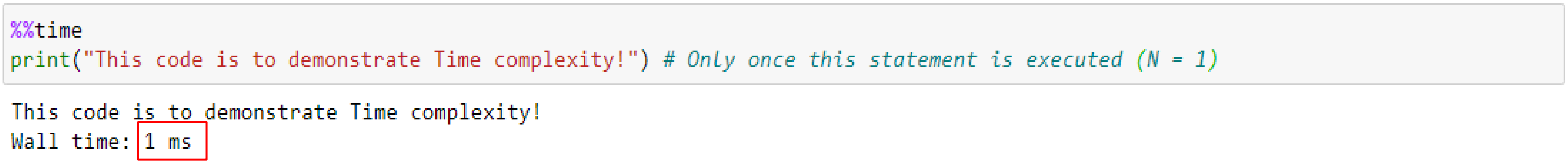
- Time is effected by what?

    1. Hardware.
    2. Compiler.
    3. Input of the algorithm (input size).

Basic time analysis (continued)

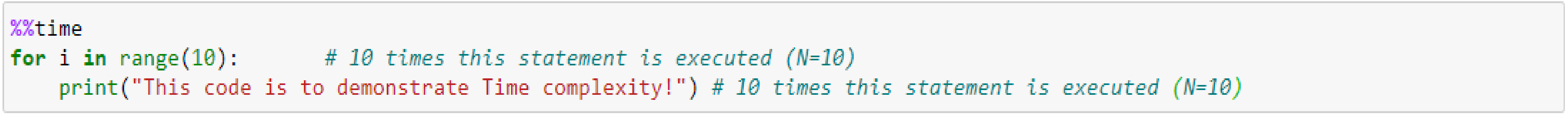
- The best notion for input size depends on the problem being studied. For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the number of items in the input: for example, the array size n for sorting.
- The most significant factor effecting on running time is n. Thus, running time represents as a function in *n*, T(n).
- The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed
- An algorithm with more operations will have a lengthier running time. The number of operations of an algorithm in proportion to the size of an input also affects the algorithm’s running time.

Basic time analysis (continued)

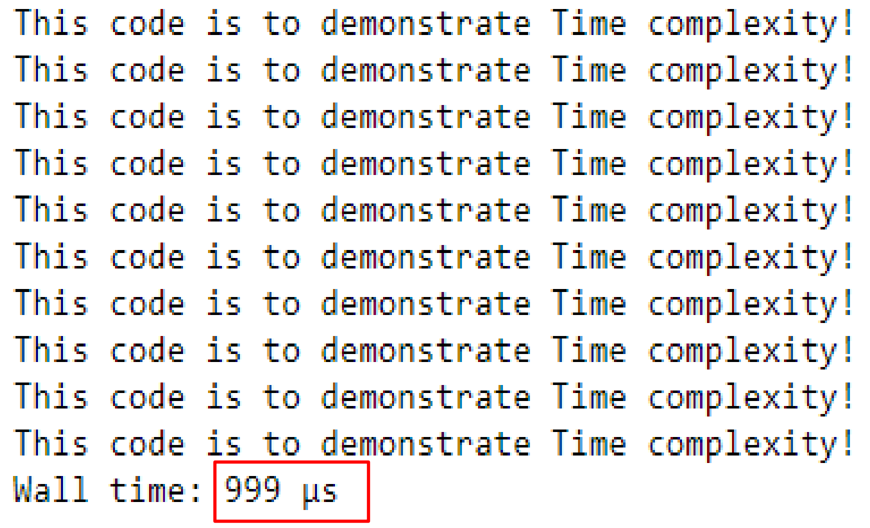


```
%%time
print("This code is to demonstrate Time complexity!") # Only once this statement is executed (N = 1)
```

```
This code is to demonstrate Time complexity!
Wall time: 1 ms
```



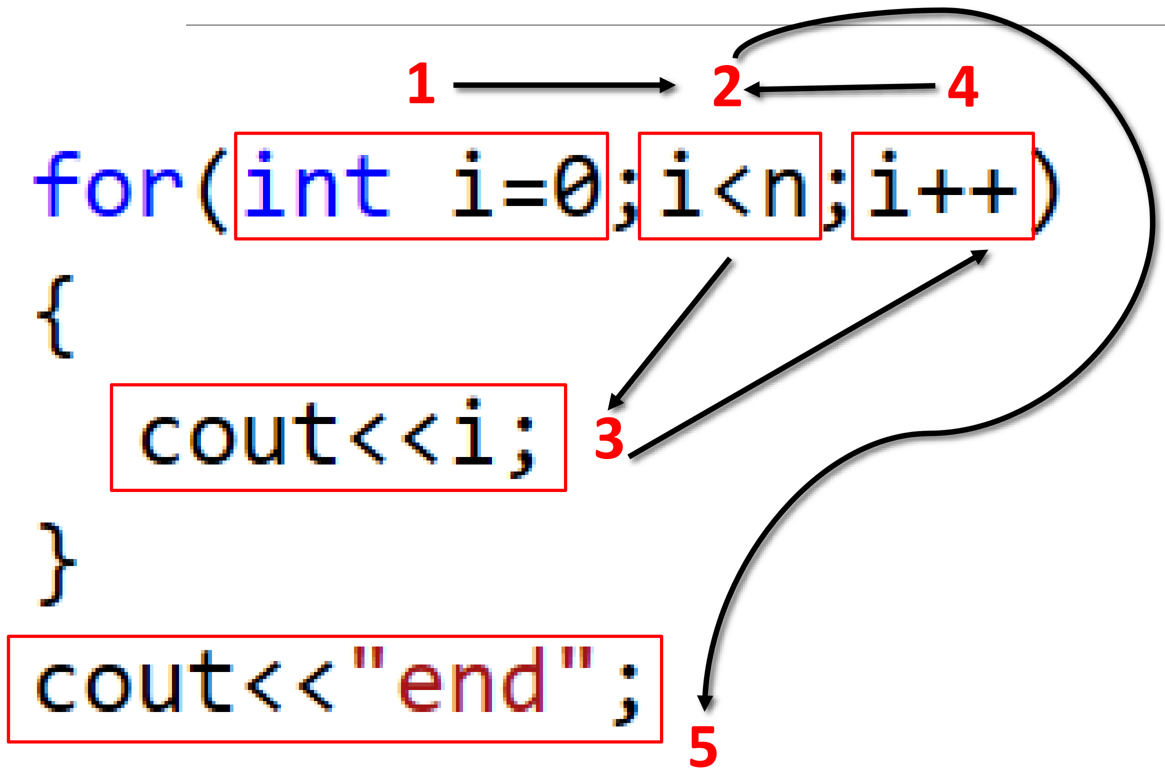
```
%%time
for i in range(10):        # 10 times this statement is executed (N=10)
    print("This code is to demonstrate Time complexity!") # 10 times this statement is executed (N=10)
```



```
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
Wall time: 999 µs
```

By now, you could have concluded that when an algorithm uses statements that get executed only once, will always require the same amount of time, and when the statement is in loop condition, the time required increases depending on the number of times the loop is set to run. And, when an algorithm has a combination of both single executed statements and LOOP statements or with nested LOOP statements, the time increases proportionately, based on the number of times each statement gets executed.

Determine the relationship between the input size and running time (Simple Example)



```
for(int i=0;i<n;i++)
{
    cout<<i;
}
cout<<"end";
```

#	Times
1	1
2	n+1
3	n
4	n
5	1

Assume that each statement **(CPU)** takes time **c,** where **c** is a constant.
F(n)=T(n)= c*(1+(n+1)+n+n+1) = c*(3n+3)

EX: If c=1 milliseconds
T(5)= 18 milliseconds
T(1000)=3003 milliseconds

Example2

The most Tricky statement

If n is even, How many times statement 1, 2, 3, 4 are executed?

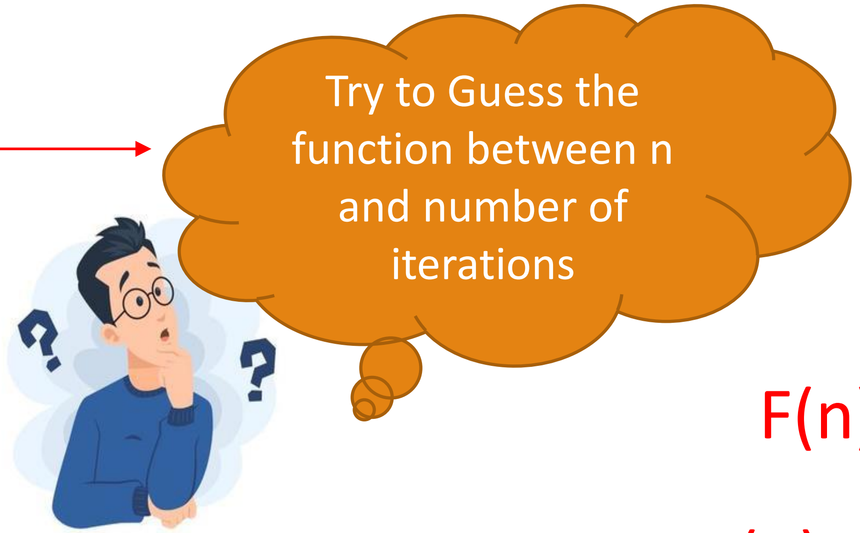
```
         1           2        4
for(int  i=0;i<n;i+=2)
{
   cout<<i; 3
}
```

#	Times
1	1
2	
3	
4	

Test: n=6

i=0	i<6?	T
i=2	i<6?	T
i=4	i<6?	T
i=6	i<6?	F

4 iterations


Try to Guess the function between n and number of iterations

F(n)=n-2

OR

F(n)= (n/2)+1

Example2

The most Tricky statement

How many times statement 1, 2, 3, 4 are executed?

```
for(int  i=0;i<n;i+=2)
{
   cout<<i;
}
```

1: `int i=0` 2: `i<n` 3: `cout<<i;` 4: `i+=2`

#	Times
1	1
2	
3	
4	

Test: n=8

i=0	i<8?	T
i=2	i<8?	T
i=4	i<8?	T
i=6	i<8?	T
i=8	i<8?	F

5 iterations

We can conclude that

F(n)=n-2 X

F(n)= (n/2)+1 ✓

Example2

The most Tricky statement

How many times statement 1, 2, 3, 4 are executed?

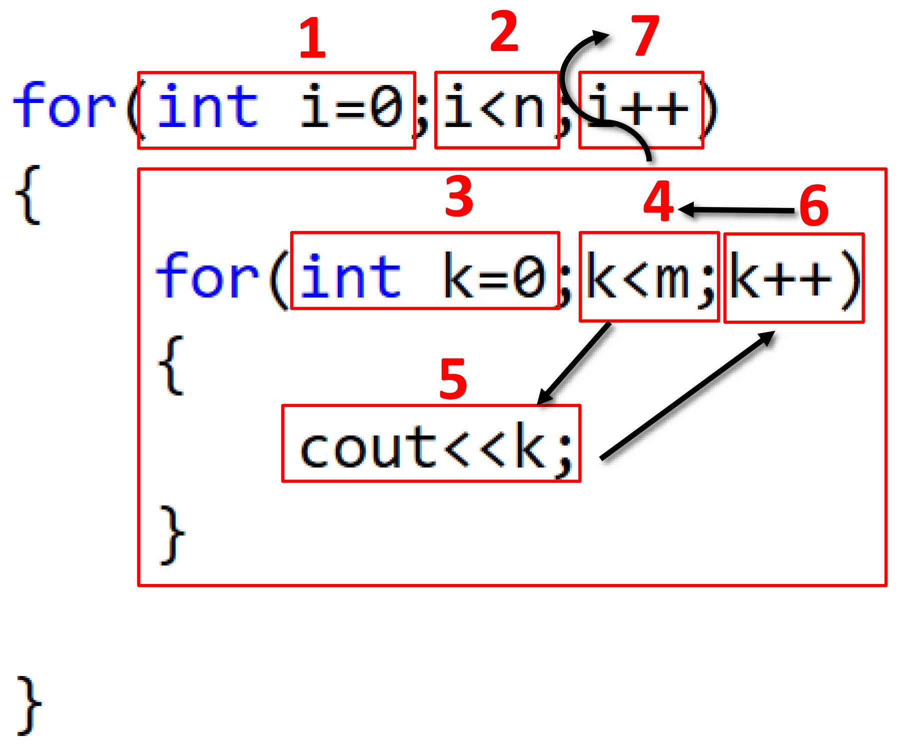
```
for(int i=0;i<n;i+=2)
{
   cout<<i;
}
```

1: `int i=0`
2: `i<n`
3: `cout<<i;`
4: `i+=2`

#	Times
1	1
2	(n/2)+1
3	n/2
4	n/2

Assume that each statement **(CPU)** takes time **c,** where **c** is a constant.
F(n)=T(n)= c*(1+((n/2)+1)+(n/2)+(n/2)) = c*(3(n/2)+2).

Example3 (nested loop)



```
for(int i=0;i<n;i++)
{
      for(int k=0;k<m;k++)
      {
            cout<<k;
      }

}
```

#	Times
1	1
2	n+1
3	1
4	m+1
5	m
6	m
7	n

n

#	Times
1	1
2	n+1
3	n
4	(m+1)*n
5	m*n
6	m*n
7	n

Assume that each statement **(CPU)** takes time **c,** where **c** is a constant.
F(n)=T(n)= c*(1+(n+1)+n+ ((m+1)*n))+(m*n)+(m*n)+n)
F(n)=T(n)=2+4n+3mn

Example3

```
int n=2,m=3;
       1      2   10
for(int i=0;i<n;i++)
{
            3     4  9
    for(int j=0;j<m;j++)
    {
                5     6   8
        for(int k=0;k<m;k++)
        {
                7
            cout<<k;
        }
    }
}
```

2 How many times statement 10 is executed?

6 How many times statement 9 is executed?

18 How many times statement 7 is executed?

If c=1 then T(n)=?

$T(n)= 2+4n+4mn+3nm^2$