Measuring Algorithm Efficiency Using Big O Notation

- Algorithm design is to develop a mathematical process for solving a problem. Algorithm analysis is to predict the performance of an algorithm.
- The Big O notation obtains a function for measuring algorithm time complexity based on the input size.
- The Big O notation estimates the execution time of an algorithm in relation to the input size. If the time is not related to the input size, the algorithm is said to take constant time with the notation O(1). For example, a method that retrieves an element at a given index in an array takes constant time because the time does not grow as the size of the array increases.

Consider the time complexity for the following loop:

```
for (int i = 1; i <= n; i++) {
    k = k + 5;
}
```

It is a constant time, c, for executing

```
k = k + 5;
```

Since the loop is executed n times, the time complexity for the loop is

T(n) = (a constant c) * n = O(n).

```
class bigO{
    public static void main(String args[ ]){

        getTime(100000000);
        getTime(150000000);
        getTime(200000000);

    }
    static void getTime(long n){
        long stime=System.currentTimeMillis();
        long k=0;
        for(long i=0;i<=n;i++)
            k=k+5;
        long etime=System.currentTimeMillis();
        System.out.println("Execution time for n="+n+" is "+(etime-stime)+"ms");
    }
}
```

```
C:\javap>java bigO
Execution time for n=100000000 is 31ms
Execution time for n=150000000 is 39ms
Execution time for n=200000000 is 47ms
```

What is the time complexity for the following loop?

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        k = k + i + j;
    }
}
```

It is a constant time, c, for executing

```
k = k + i + j;
```

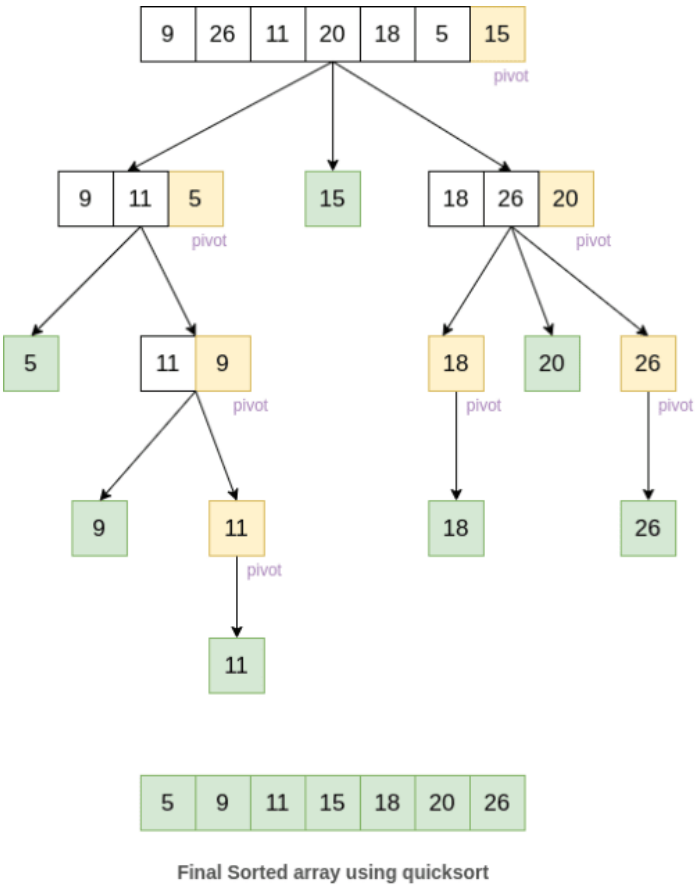
The outer loop executes n times. For each iteration in the outer loop, the inner loop is executed n times. Thus, the time complexity for the loop is

$T(n) = (a\ constant\ c) * n * n = O(n^2)$

An algorithm with the $O(n^2)$ time complexity is called a quadratic algorithm and it exhibits a quadratic growth rate. The quadratic algorithm grows quickly as the problem size increases. If you double the input size, the time for the algorithm is quadrupled. Algorithms with a nested loop are often quadratic.

Quick Sort

```
class qsort{
        public static void main(String args[ ]){
                int A[]={9,26,11,20,18,5,15};
                int b=0,e=A.length-1;
                //qsrec(A,b,e);
                //for(int i=0;i<A.length;i++)
                part(A,b,e);
          for(int i=0;i<A.length;i++)
                  System.out.println(A[i]);
        }
        static void qsrec(int A[ ],int b,int e){
                if(b<e){
                int p=part(A,b,e);
                qsrec(A,b,p-1);
                qsrec(A,p+1,e);
                }
        }
        static int part(int A[ ],int b,int e){
                int p=A[e];
                int i=b-1;
                for(int j=b;j<=e-1;j++){
                        if(A[j]<=p){
                                i++;
                                int t=A[i];
                                A[i]=A[j];
                                A[j]=t;
                        }
                }
                int t=A[i+1];
                A[i+1]=A[e];
                A[e]=t;
                return i+1;
        }
}
```



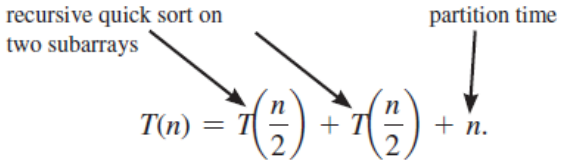
Final Sorted array using quicksort

To partition an array of *n* elements, it takes *n* comparisons and *n* moves in the worst case. Thus, the time required for partition is *O*(*n*).

In the worst case, the pivot divides the array each time into one big subarray with the other array empty. The size of the big subarray is one less than the one before divided. The algorithm

requires (*n* - 1) + (*n* - 2) + g + 2 + 1 = $O(n^2)$ time.

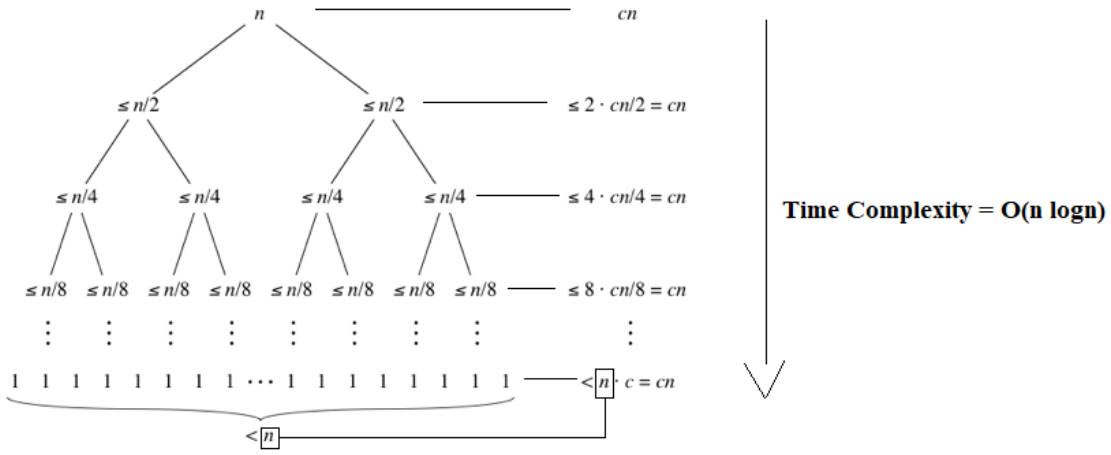
In the best case, the pivot divides the array each time into two parts of about the same size. Let *T*(*n*) denote the time required for sorting an array of *n* elements using quick sort.



T(*n*) = *O*(*n* log *n*)

On the average, the pivot will not divide the array into two parts of the same size or one empty part each time. Statistically, the sizes of the two parts are very close. Therefore, the average time is *O*(*n* log*n*).

Quick Sort Best Case Scenario



Quick Sort- Worst Case Scenario