ALGORITHMS

Asymptotic Complexity

Java Code Examples

CS 2100 – Data Structures and Algorithms 1

Prof. Nada Basit

What is the Asymptotic Complexity of this code?

```
int returnFirst(int [] arr)  {
     return arr[0];
}
```

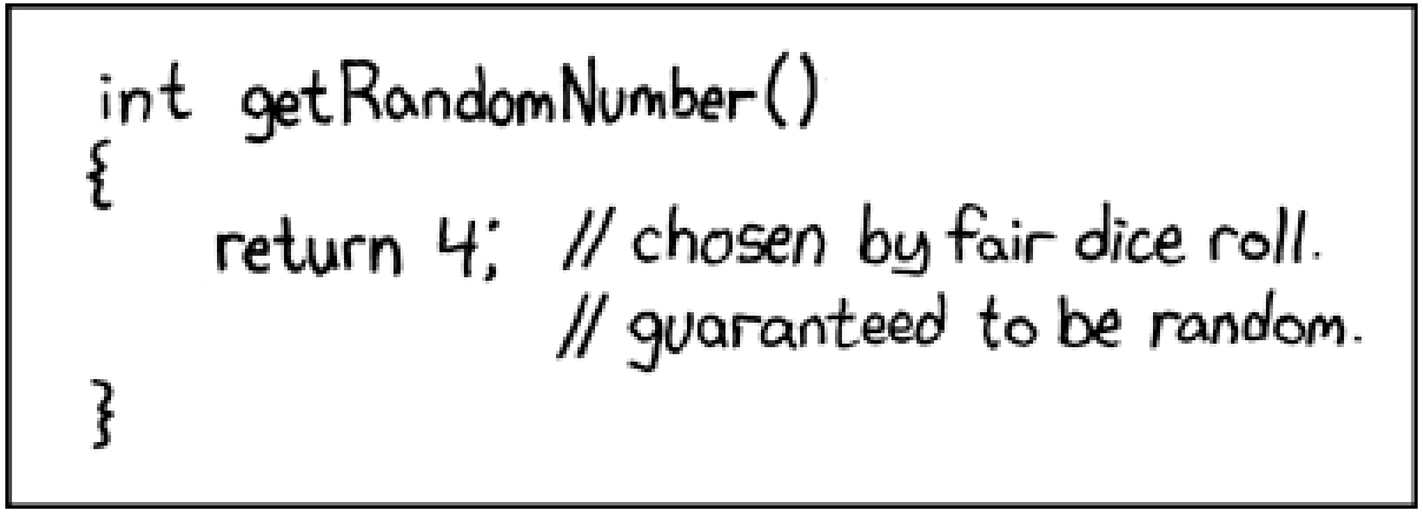
Answer: O(1) – constant time

```
int returnFirst(int [] arr)  {
     return arr[0];
}
```

- This algorithm takes in an int array and **returns the first item in the list**. It will always execute in the same time regardless of the size of the input data set (it doesn't matter if this array has 2 items or 200,000 items; the amount of time it takes to return the first element will always be the same.)

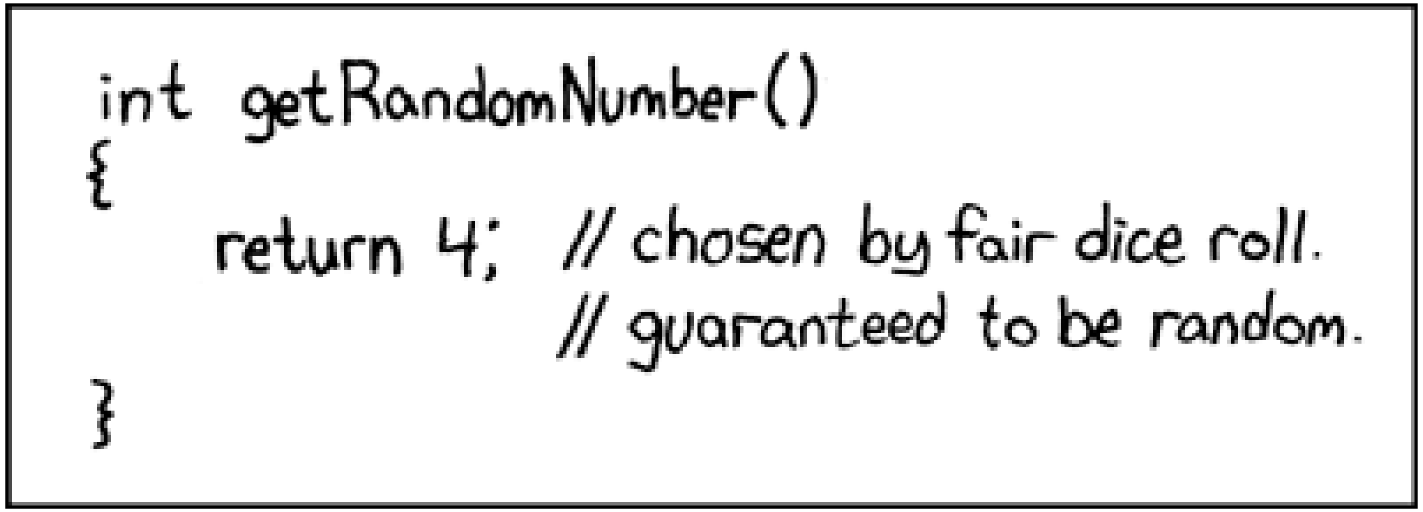
- Therefore, it is an algorithm that runs in **constant time O(1)**
- Note, the "1" represents a constant amount of time, it doesn't imply a single unit of some time.

What is the Asymptotic Complexity of this code?



xkcd.com

Answer: O(1) – constant time



xkcd.com

- Here is another example of a **constant time algorithm**. The algorithm will once again always execute in the same amount of time regardless of the size of the problem / size of the input data. Therefore, since the size of the problem is not factored.

Code also considered O(1) – constant time

```
int getRandom(int [] arr)  {
     int randomInt = (int)(arr.length * Math.random());
     return randomInt;
}
```

- This is yet another example of an algorithm that operates in **constant time, or O(1).**
- Note that merely including the array as an input to this method does **NOT** factor into the complexity.

What is the Asymptotic Complexity of this code?

```
void method1(int [] arr)  {
     int n = arr.length;
     for(int i = n - 1 ; i >= 0; i = i - 3)  {
          System.out.println(arr[i]);
     }
}
```

Answer: O(n)

```
void method1(int [] arr)  {
      int n = arr.length;
      for(int i = n - 1 ; i >= 0; i = i - 3)  {
            System.out.println(arr[i]);
      }
}
```

- Consider this example of printing out every third element in an int array (starting from the end). The problem size, n, is the length of the array, so this algorithm will be O(n).
- **In actuality, this will be a 1/3*n complexity**, but because we are only concerned with the highest-order term (n) (constant 1/3 ignored), we say this will run in **O(n).**
- The algorithm performs the exact same behavior (printing out an element) n times. The total time spent on this operation is a*n, where a is the time it takes to perform the operation once.

Answer: O(n) – continued…

```
void method1(int [] arr)  {
     int n = arr.length;                           // ‘c’
     for(int i = n - 1 ; i >= 0; i = i - 3)  { // ‘b’
          System.out.println(arr[i]);              // ‘a’
     }
}
```

- Now, this is not the only thing that is done in the algorithm. The value of i is incremented and is compared to n each time through the loop. This adds an additional time of b*n to the run time, for some constant b.
- Furthermore, i and n both have to be initialized; this adds some constant amount c to the running time. The exact running time would then be (a+b)*n+c, where the constants a, b, and c depend on factors such as how the code is compiled and what computer it is run on. Using the fact that c is less than or equal to c*n for any positive integer n, we can say that the run time is less than or equal to (a+b+c)*n. That is, the run time is less than or equal to a constant times n.
- By definition, this means that the run time for this algorithm is O(n).

What is the Asymptotic Complexity of this code?

```
public static someCoolSort( int[] A, int n ) {
    for (int i = 0; i < n; i++) {
    // Do n passes through the array...
        for (int j = 0; j < n-1; j++) {
        // Do n-1 passes through the array...
            if ( A[j] > A[j+1] ) {
            // A[j] and A[j+1] are out of order, so swap them
                int temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

Answer: $O(n^2)$

```
public static someCoolSort( int[] A, int n ) {
    for (int i = 0; i < n; i++) {
    // Do n passes through the array...
        for (int j = 0; j < n-1; j++) {
        // Do n-1 passes through the array...
            if ( A[j] > A[j+1] ) {
            // A[j] and A[j+1] are out of order, so swap them
                int temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

- Here, the parameter **n** represents the problem size. The **outer** for loop in the method is executed **n times**. Each time the outer for loop is executed, the **inner** for loop is executed **n-1 times**, so the if statement is executed n*(n-1) times. This is **n^2-n**. So, the run time of this algorithm is **$O(n^2)$.**

Answer: $O(n^2)$ – continued…

- Here, the parameter **n** represents the problem size. The **outer** for loop in the method is executed **n times**. Each time the outer for loop is executed, the **inner** for loop is executed **n-1 times**, so the if statement is executed n*(n-1) times.
- This is **n^2-n**, but since lower order terms are not significant in an asymptotic analysis, it's good enough to say that the if statement is executed about n^2 times. Furthermore, if we look at other operations -- the assignment statements, incrementing i and j, etc. -- none of them are executed more than n^2 times**, so the run time of this algorithm is $O(n^2)$.**

What is the Asymptotic Complexity of this code?

```
public double restaurantRanking (List < Restaurant > rList ){
    for ( Restaurant r : rList ) {
        int rank = 0;
        int cost = r.getCostAsInt();
        for ( Restaurant s : rList ) {
            if (!s.equals (r)) {
                int sCost = s.getCostAsInt();
                if ( sCost < cost ) // r costs more than s
                    rank ++;
            }
        }
        r.setRanking (rank);
    }
}
```

Answer: $O(n^2)$

```
public double restaurantRanking (List < Restaurant > rList ){
    for ( Restaurant r : rList ) {
        int rank = 0;
        int cost = r.getCostAsInt();
        for ( Restaurant s : rList ) {
            if (!s.equals (r)) {
                int sCost = s.getCostAsInt();
                if ( sCost < cost ) // r costs more than s
                    rank ++;
            }
        }
        r.setRanking (rank);
    }
}
```

What is the Asymptotic Complexity of this code?

```
public int doSomething(int n) {
    int sum = 0;
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            for (int l = 0; l < n; l++) {
                sum += j * k / (l + 1);
            }
        }
    }
    return sum;
}
```

Answer: O(n3)

```
public int doSomething(int n) {
    int sum = 0;
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            for (int l = 0; l < n; l++) {
                sum += j * k / (l + 1);
            }
        }
    }
    return sum;
}
```

Three nested for-loops (each of them are on an **order of n**): **$O(n^3)$**

What is the Asymptotic Complexity of this code?

```
void anotherMethod(int [] arr) {
    for(int p = 0; p < arr.length; p++) {
        for(int r = 0; r < arr.length; r++) {
            System.out.println(Math.log(arr[p]));
        }
    }
    for(int s = 0; s < arr.length; s++) {
         System.out.println(arr[s]);
    }
}
```

Answer: $n^2 + n = O(n^2)$

```
void anotherMethod(int [] arr) {
    for(int p = 0; p < arr.length; p++) {
        for(int r = 0; r < arr.length; r++) {  // 2 nested for-loops
            System.out.println(Math.log(arr[p]));
        }
    }
    for(int s = 0; s < arr.length; s++) {  // sequential after the first 2
        System.out.println(arr[s]);
    }
}
```

- Two nested for-loops followed by (not nested) another for-loop (each of them are on an **order of n**). Sequential for-loops don't count the same as nested ones! Therefore, it would be $n^2 + n$ = **$O(n^2)$**

What is the Asymptotic Complexity of this code?

```
void doesSomethingElse(int n) {
    for (int i = 1; i < n; i = i * 2) {
        System.out.println("I'm looking at: " + i);
    }
}
```

Answer: O(log(n))

```
void doesSomethingElse(int n) {
    for (int i = 1; i < n; i = i * 2) {
        System.out.println("I'm looking at: " + i);
    }
}
```

- The running time grows in proportion to the **logarithm** of the input (in this case, log to the base 2). Therefore, if ‘**n**’ is 8, the output will be the following:

  I’m looking at: 1
  I’m looking at: 2
  I’m looking at: 4

  This algorithm ran **log(8) = 3 times.**

Answer: O(log(n)) – continued…

```
void doesSomethingElse(int n) {
    for (int i = 1; i < n; i = i * 2) {
        System.out.println("I'm looking at: " + i);
    }
}
```

- Other algorithms that we will see (later) that also have a run-time of **log(n)** are **searching algorithms** that are either tree based (binary tree in particular) or any other algorithm that **cuts the search space in half at every iteration**. The relationship of eliminating a half then another half is a **logarithmic** one making those algorithms **O(log(n))**.

What is the Asymptotic Complexity of this code?

```
void funMethod(ArrayList<Integer> arr) {
    for (int i = 1; i < arr.size(); i++) {
        if (!(arr.contains(i)))
            System.out.println(arr.get(i));
    }
}
```

Answer: $O(n^2)$

```
void funMethod(ArrayList<Integer> arr) {
    for (int i = 1; i < arr.size(); i++) {
        if (!(arr.contains(i))) // Takes O(n) time for this!
            System.out.println(arr.get(i));
    }
}
```

- The running time of this algorithm is **$O(n^2)$** because **contains() takes linear time to run, i.e. O(n)**. Not only that, contains() is acting on the **same data** (the "arr" ArrayList.)
  So, in the **worst-case scenario** (going through the entire list), the running time will be **quadratic** -- **$O(n^2)$**