Laboratory practice No. 2: Big O Notation

Santiago Isaza Cadavid
Universidad EAFIT
Medellín, Colombia
sisazac@eafit.edu.co

Hamilton Smith Gómez Osorio
Universidad EAFIT
Medellín, Colombia
hsgomezo@eafit.edu.co

September 9, 2018

1) GitHub's codes

When analyzing the Merge sort and Insertion Sort methods, implementing the time counter, we realized that this is effective for problems where the database is very wide since in these is where a progressive trend in the results is evident . In addition, the size of the problem will represent the number of values to be sorted and also has a great influence on the algorithm's complexity.

2) Project Questions Simulation

2.a. Algorithms's chart

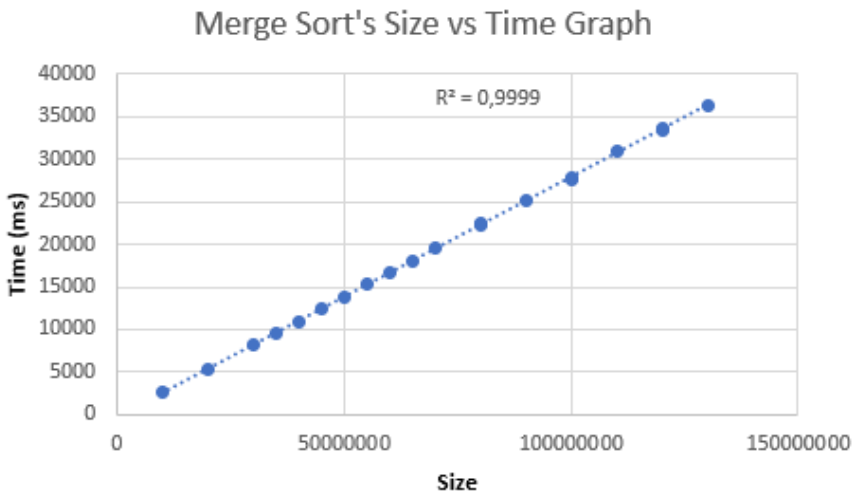
Merge Sort	
Size	Time (ms)
20000000	5355
40000000	10920
60000000	16616
80000000	22398
100000000	27679
120000000	33432
10000000	2650
30000000	8213
50000000	13882
70000000	19504
80000000	22240

90000000	25181
100000000	27831
110000000	31033
120000000	33551
130000000	36289
35000000	9632
45000000	12408
55000000	15274
65000000	18035

Insertion Sort	
Size	Time
100000	3411
110000	4049
120000	4840
130000	5881
140000	6607
150000	7556
160000	8660
170000	9747
180000	10951
190000	12301
200000	13575
210000	14959
220000	16806
230000	18071
240000	19654
250000	21352
260000	23180
270000	25099
280000	26760
290000	28736

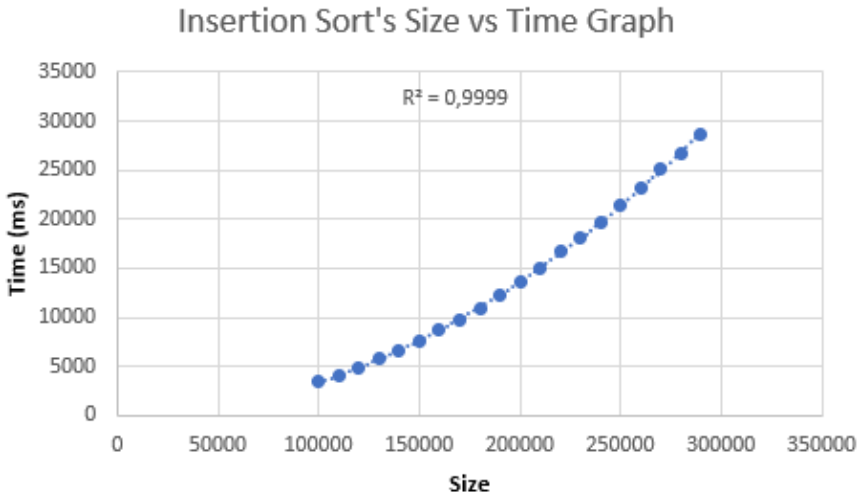
2.b. Algorithms's graphics

Figure 2: Merge Sort



Professor Mauricio Toro Bermudez
Phone: (+57)(4)2619500 Ext. 9473. Office: 19 – 627
E-mail: mtorobe@eafit.edu.co

Figure 3: Insertion Sort



2.c. *Given the above information, how efficient is merge sort compared with insertion sort for large arrays? Is it appropriate to use insertion sort for a data base with millions of elements?*

Merge Sort's complexity is $O(n \log n)$ while Insertion Sort's is $O(n^2)$, so Insertion Sort takes more time to organize the array, making the first one way more efficient and fast. So Insertion sort is not appropriate for problems with millions of elements because the execution time is bigger than Merge Sort's time, so its behavior will not be ideal.

2.d. *Explain with your own words how does the Codingbat's Array3 exercise maxSpan works. Why?*

This problem ask us to find the max number of elements in an array such that these are within two equal numbers. In order to do this we define two variables, one as a general counter (cont) and the other which stores the largest length found (count2). With two cycles we take care of traversing the arrangement, one ascending from position zero(0) to the length of the arrangement and other cycle in the opposite direction. When two equal numbers are found in the visited positions, cont takes values of the amount of numbers among the detected ones. As the length is being sought, this is repeated until the end of the cycles and,if a longer length is found than the previous one, count2 is replaced to finally return the largest value.

2.e. *Calculate the complexity of the on-line exercise*

i.

```
public int countEvens(int[] nums) {
 int n=0;
  for(int i=0;i<nums.length;i++){ // C_1 *(n + 1)
      if(nums[i]%2==0) n+=1; // C_2 * n
  }
```

```
    return n;  //C_3
  }
```

$$T(n) = C_1 * n + C_2 * (n + 1) + C_3$$
$$T(n) = O(C_1 * n + C_2 * (n + 1) + C_3)$$
$$T(n) = O(n + n + 1)$$
$$T(n) = O(n)$$

The complexity of this algorithm is O(n)

ii.

```
public boolean lucky13(int[] nums) {
   for(int i=0;i<nums.length;i++){ // C_1 * (n + 1)
       if(nums[i]==3 || nums[i]==1) return false; //C_2 * n
   }
   return true; //C_3
}
```

$$T(n) = C_1 * n + C_2 * (n + 1) + C_3$$
$$T(n) = O(C_1 * n + C_2 * (n + 1) + C_3)$$
$$T(n) = O(n + n + 1)$$
$$T(n) = O(n)$$

The complexity of this algorithm is O(n)

iii.

```
public boolean isEverywhere(int[] nums, int val) {
   for(int i=0;i<nums.length-1;i++){ // C_1 * n
       if(nums[i]!=val && nums[i+1]!=val) return false; //C_2 * (n - 1)
   }
   return true;// C_3
}
```

$$T(n) = C_1 * n + C_2 * (n - 1) + C_3$$
$$T(n) = O(C_1 * n + C_2 * (n - 1) + C_3)$$
$$T(n) = O(n + n - 1)$$
$$T(n) = O(n)$$

The complexity of this algorithm is O(n)

iv.

```
 public boolean modThree(int[] nums) {
for(int i=0;i<nums.length-2;i++){ // C_1 * (n - 1)
    if(nums[i]%2==0 && nums[i+1]%2==0 &&
    nums[i+2]%2==0) return true; // C_2 * (n - 2)
    if(nums[i]%2==1 && nums[i+1]%2==1 &&
    nums[i+2]%2==1) return true; // C_3 * (n - 2)
}
    return false; //C_4
}
```

$$T(n) = C_1 * (n-1) + C_2 * (n-2) + C_4$$
$$T(n) = O(C_1 * (n-1) + C_2 * (n-2) + C_4)$$
$$T(n) = O(n + n - 3)$$
$$T(n) = O(n)$$

The complexity of this algorithm is O(n)

v.

```
    public boolean tripleUp(int[] nums) {
    for(int i=0;i<nums.length-2;i++){ // C_1 * (n - 1)
        if(nums[i+1]==nums[i]+1 &&
        nums[i+2]==nums[i]+2) return true; // C_2 * (n - 2)
    }
        return false; //C_3
    }
```

$$T(n) = C_1 * (n-1) + C_2 * (n-2) + C_3$$
$$T(n) = O(C_1 * (n-1) + C_2 * (n-2) + C_3)$$
$$T(n) = O(n + n + -3)$$
$$T(n) = O(n)$$

The complexity of this algorithm is O(n)

vi.

```
public boolean linearIn(int[] outer, int[] inner) {
    int n=0;
    for(int i=0;i<inner.length;i++){   // C_1 * (n + 1)
        for(int j=0;j<outer.length;j++){     //C_2 * (n*m)
            if(inner[i]==outer[j]){       // C_3 * (n*m)
                n++;       //C_4
                break;
            }
```

Professor Mauricio Toro Bermudez
Phone: (+57)(4)2619500 Ext. 9473. Office: 19 – 627
E-mail: mtorobe@eafit.edu.co

```
            }
        }
        return n==inner.length; //C_5
    }
```

$$T(n) = C_1 * (n+1)C_2 + (n*m) + C_3 + (n*m) + C_4 + C_5$$
$$T(n) = O(C_1 * (n+1) + C_2 * (n+1*m) + C_3 * (n*m) + C_4 + C_5)$$
$$T(n) = O(n + 1 + m + n*m + n*m)$$
$$T(n) = O(2n*m)$$
$$T(n) = O(n*m)$$

The complexity of this algorithm is O$(n*m)$

vii.

```
    public int[] seriesUp(int n) {
        int [] arr=new int[n*(n+1)/2]; //C_1
        int num=0; //C_2
        for(int i=1;i<=n;i++){ //C_3 * (n + 1)
            for(int j=1;j<=i;j++){ //C_4 * (n*(n+1))
                arr[num]=j; //C_5
                num++; //C_6
            }
        }
        return arr; // C_7
    }
}
```

$$T(n) = C_1 + C_2 + C_3 * (n+1) + C_4 * (n*(n+1)) + C_5 + C_6 + C_7$$
$$T(n) = O(C_1 + C_2 + C_3 * (n*(n+1)) + C_4 * (n*(n+1)) + C_5 + C_6 + C_7)$$
$$T(n) = O(n^2 + n + n^2 + n)$$
$$T(n) = O(2n^2)$$
$$T(n) = O(n^2)$$

The complexity of this algorithm is O(n^2)

viii.

```
    public int[] fix34(int[] nums) {
         int index=0; // C_1
         for(int i=0;i<nums.length;i++){ //C_2*(n+1)
             if(nums[i]==3){ //C_3*(n+1)
                for(int j=index;j<nums.length;j++){ // C_4*(n*(n+1))
                    if(nums[j]==4){ //C_5*(n*(n+1))
                    index= j; //C_6*(n*(n+1))
                    int aux= nums[j]; //C_7*(n*(n+1))
```

```
                    nums[index]= nums[i+1]; //C_8*(n*(n+1))
                    nums[i+1]= aux; //C_9*(n*(n+1))
                    break;
                    }
                }
            }
        }
        return nums; // C_10
    }
```

$$T(n) = C_1 + C_2 * (n+1) + C_3 * (n+1) + C_4 * (n * (n+1)) + C_5 * (n * (n+1)) + C_6 * (n * (n+1)) + C_7 * (n * (n+1)) + C_8 * (n * (n+1)) + C_9 * (n * (n+1)) + C_10$$

$$T(n) = O(C_1 + C_2 * (n+1) + C_3 * (n+1) + C_4 * (n * (n+1)) + C_5 * (n * (n+1)) + C_6 * (n * (n+1)) + C_7 * (n * (n+1)) + C_8 * (n * (n+1)) + C_9 * (n * (n+1)) + C_10)$$

$$T(n) = O(n+1+n+1+n^2+n+n^2+n+n^2+n+n^2+n+n^2+n+n^2+n)$$

$$T(n) = O(6n^2+3n)$$

$$T(n) = O(n^2)$$

The complexity of this algorithm is O(n^2)

ix.

```
    public int maxSpan(int[] nums) {
        int cont=0;  // C_1
        int cont2=0; // C_2
        for(int i=0;i<nums.length;i++){ // C_3*(n+1)
            for(int j=nums.length-1;j>=0;j--){ // C_4*(n*(n+1))
                if(nums[i]==nums[j]) cont= (j-i)+1;// C_5*(n*(n+1))
                if(cont>cont2){ // C_6*(n*(n+1))
                    cont2=cont; //C_7*(n*(n+1))
                    cont=0; //C_8*(n*(n+1))
                }
            }
        }
        return cont2; //C_9
    }
```

$$T(n) = C_1 + C_2 + C_3 * (n+1) + C_4 * (n * (n+1)) + C_5 * (n * (n+1)) + C_6 * (n * (n+1)) + C_7 * (n * (n+1)) + C_8 * (n * (n+1) + C_9)$$

$$T(n) = O(C_1 + C_2 + C_3 * (n+1) + C_4 * (n * (n+1)) + C_5 * (n * (n+1)) + C_6 * (n * (n+1)) + C_7 * (n * (n+1)) + C_8 * (n * (n+1) + C_9))$$

$$T(n) = O(n + 1 + n^2 + n + n^2 + n + n^2 + n + n^2 + n + n^2 + n)$$

$$T(n) = O(5n^2 + n)$$

$$T(n) = O(n^2)$$

The complexity of this algorithm is O(n^2)

x. Title: SquareUp CodingBat solution Author: Elentok, O. Date: 14 jul 2012 Code version: 1.0 Availability: https://github.com Taken from https://bit.ly/2MUhkrA

```
    public int[] squareUp(int n)
    {
        int[] arr = new int[n*n]; // C_1
            int p; //C_2
            for(int i = 1; i <= n; i++){ // C_3*(n+1)
                    p = n * i - 1; // C_4*(n+1)
                    for(int j = 1; j <= i; j++, p--){ //C_5*(n*(n+1))
                            arr[p] = j;// C_6*(n*(n+1))
                }
        }
      return arr; //C_7
    }
}
```

$$T(n) = C_1 + C_2 + C_3 * (n+1) + C_4 * (n+1) + C_5 * (n * (n+1)) + C_6 * (n * (n+1)) + C_7$$

$$T(n) = O(C_1 + C_2 + C_3 * (n+1) + C_4 * (n+1) + C_5 * (n * (n+1)) + C_6 * (n * (n+1)) + C_7)$$

$$T(n) = O(n + 1 + n + 1 + n^2 + n + n^2 + n)$$

$$T(n) = O(2n^2 + 4n)$$

$$T(n) = O(n^2)$$

The complexity of this algorithm is O(n^2)

2.f. Explain what the variables (n, m...) means in the previous exercises

2.6.1 countEvens

The variable n means the size of the array.

2.6.2 lucky13

The variable n means the size of the array.

2.6.3 isEverywhere

The variable n means the size of the array.

2.6.4 modThree

The variable n means the size of the array.

2.6.5 tripleUp

The variable n means the size of the array.

2.6.6 linearIn

The variable n means the size of the outer array and the variable m means the size of the inner array.

2.6.7 seriesUp

The variable n means the size of the array.

2.6.8 fix34

The variable n means the size of the array.

2.6.9 maxSpan

The variable n means the size of the array.

2.6.10 squareUp

The variable n means the size of the array.

3) Midterm Simulation

3.a. Exercise 1

c) O(n+m)

3.b. Exercise 2

d) O($m * n$)

3.c. Exercise 3

b) O(ancho)

3.d. Exercise 4

b) O(n^3)

3.e. Exercise 5

d) O(n^2)

3.f. Exercise 6

a) T(n)= T(n-1)+T(n-2)+C

3.g. Exercise 7

3.7.1 Worst case-scenario number of steps

T(n)=T(n-1)+C

3.7.2 Asymptotic Complexity

O(n)

3.h. Exercise 8

The mystery(n) function executes $n * \sqrt{n}$ steps

3.i. Exercise 9

d) Executes more than $n^2 + n * m$

3.j. Exercise 10

a) Executes less than $n * \log n$ steps

3.k. Exercise 11

c) Executes T(n) = T(n-1)+T(n-2)+C steps

3.l. Exercise 12

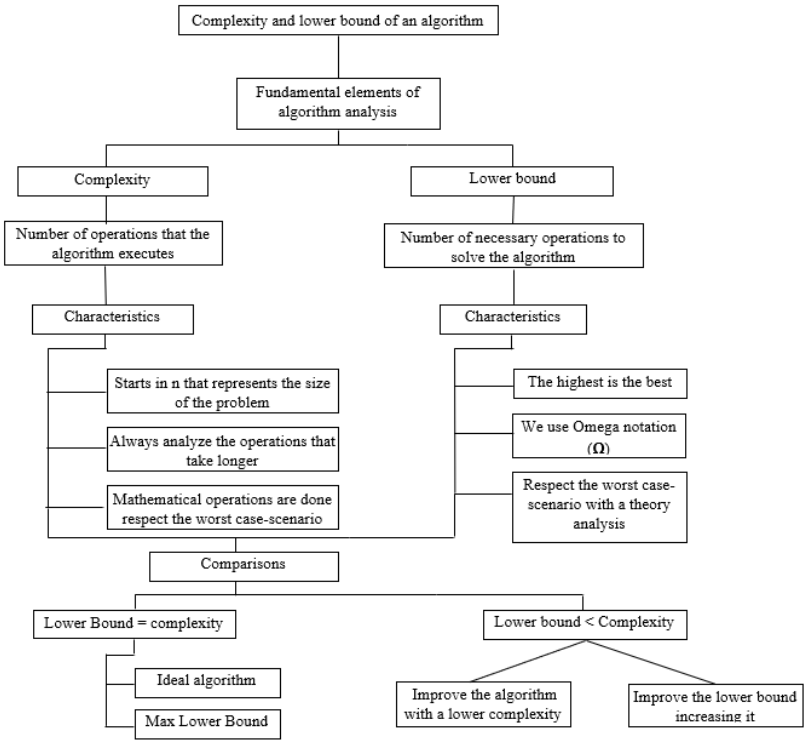
b) O($m\sqrt{n}$)

3.m. Exercise 13

a) $O(n^3)$

4) Recommended reading

The complexity of an algorithm is the number of operations that it must perform in order to get a solution. To determine it, several elements need to be considered. Mainly, a n size that represents the total amount of operations of the problem and the change that it suffers when developing. It the end, the sequence of operations that takes more time to solve within the algorithm will be the worst case and owe mathematical procedures related to that case there is a general expression of execution. To know the efficiency of a problem, it is important to analyze what is known as a lower bound that refers to a function that limits the problem and represents the temporal complexity necessary to solve it. For this, the lower bound is searched, and, if the temporal complexity of the algorithm is equal to this, the optimal solution has been found.



Professor Mauricio Toro Bermudez
Phone: (+57)(4)2619500 Ext. 9473. Office: 19 – 627
E-mail: mtorobe@eafit.edu.co