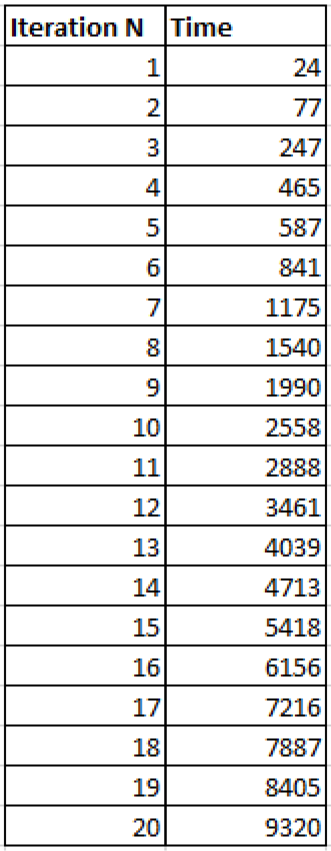
Laboratory practice No. 2: Big O Notation

Juan Sebastián Pérez Salazar
Universidad Eafit
Medellín, Colombia
jsperezs@eafit.edu.co

Yhoan Alejandro Guzmán García
Universidad Eafit
Medellín, Colombia
yaguzmang@eafit.edu.co

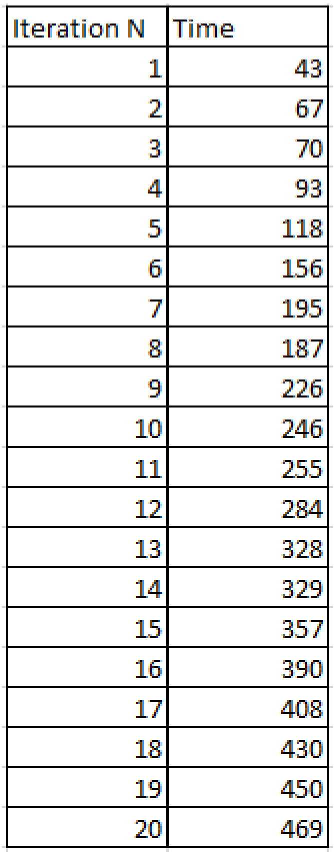
3) Practice for final project defense presentation

1. **Insertion sort table**



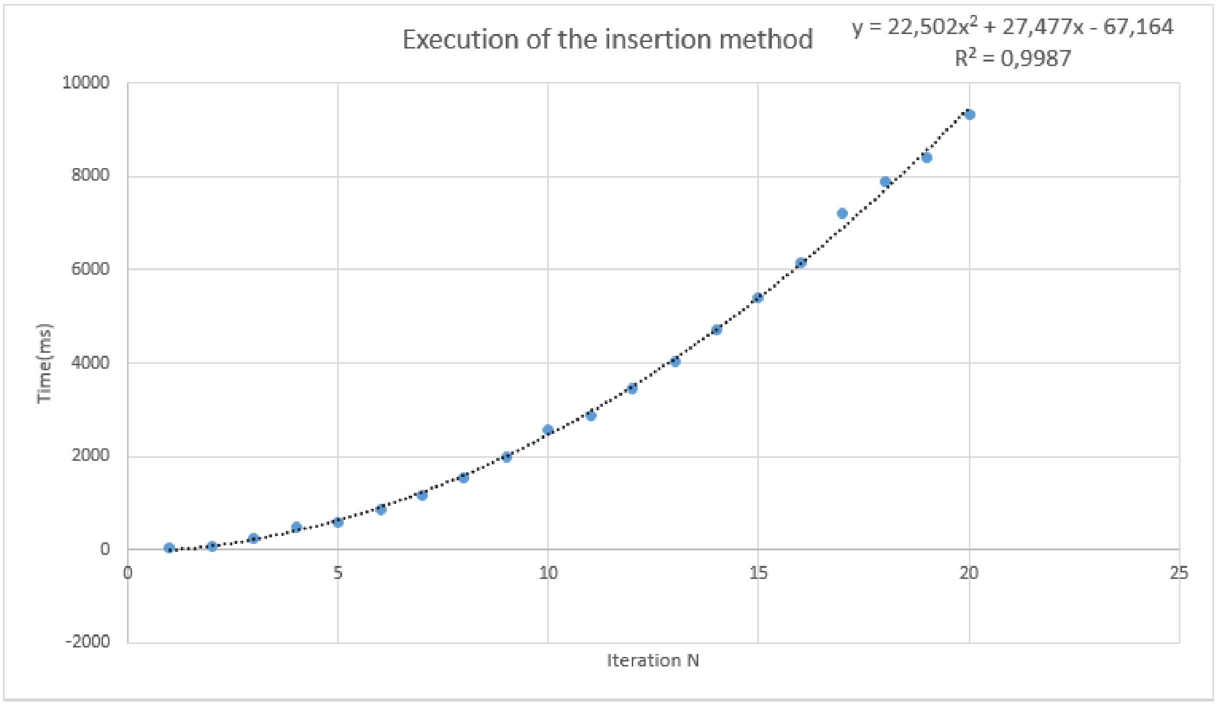
Iteration N	Time
1	24
2	77
3	247
4	465
5	587
6	841
7	1175
8	1540
9	1990
10	2558
11	2888
12	3461
13	4039
14	4713
15	5418
16	6156
17	7216
18	7887
19	8405
20	9320

Merge sort table

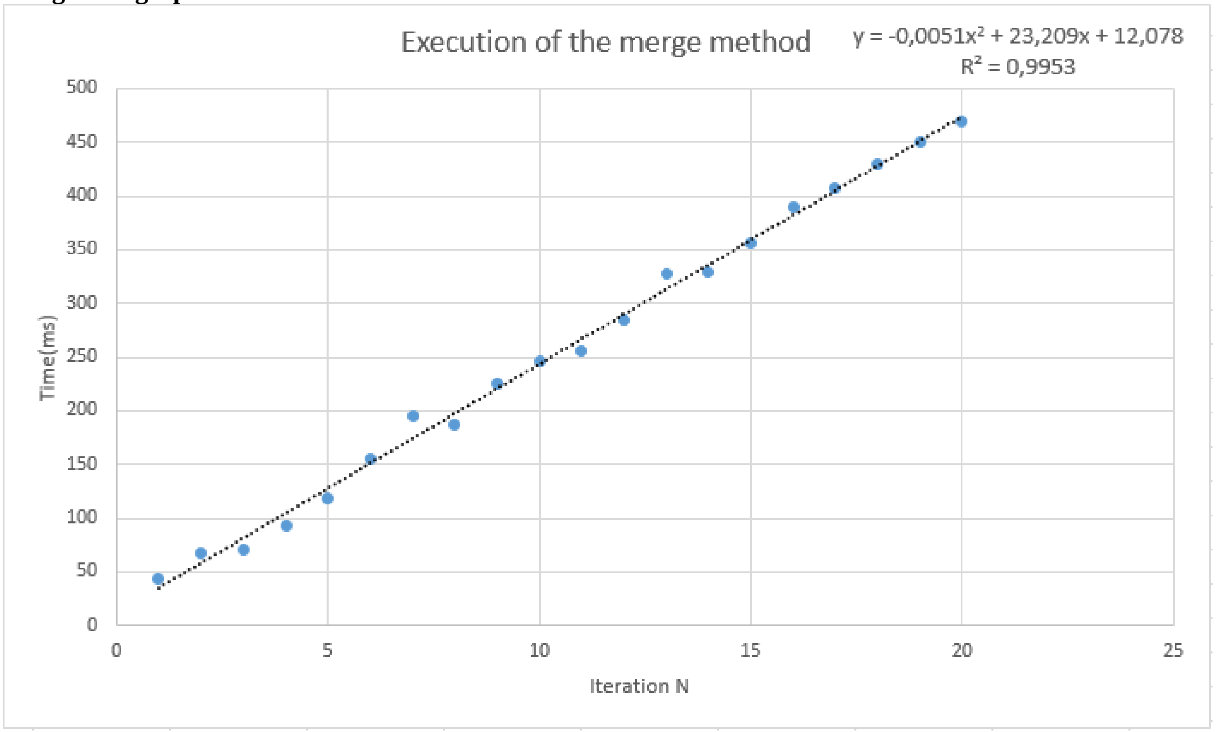


Iteration N	Time
1	43
2	67
3	70
4	93
5	118
6	156
7	195
8	187
9	226
10	246
11	255
12	284
13	328
14	329
15	357
16	390
17	408
18	430
19	450
20	469

2. **Insertion sort graphic**



Merge sort graphic



3. Taking into account the results of the time table and graphs, we can affirm that the merge sort method is much more effective when dealing with large arrays in comparison with the insertion sort method. For large sizes merge sort tends to have a complexity of O(nlog(n)) while insertion sort tends to have a complexity of $O(n^2)$. Therefore, we do not recommend using the insertion sort method when it comes to databases with millions of data, due to the fact that it grows in time indefinitely, while with the merge sort method this time would have a tendency more efficient.

4. The algorithm of the MaxSpan method of Array3 is as follows:

```
public int maxSpan(int[] nums) {
    int max = 0;
    for(int i = 0; i <= (nums.length/2); i++){
      for(int j = nums.length-1; j >= 0; j--){
        if(nums[i] == nums[j]){
          if((j – i + 1) > max) max = j – i + 1;
          break;
        }
      }
    }
    return max;
  }
```

This method consists of finding which is the highest rank between the appearance to the left of a number and the appearance to the right of it, the first thing that is done is to define the variable max, which holds the highest rank found. Then proceed to perform a nested cycle, the first is responsible for traversing the array with the positions to which their appearance will be sought more to the right. The second cycle is responsible for traversing the array in an inverse manner and this evaluates by means of a condition if an occurrence is found to the right. At the moment this is found the algorithm calculates (j - i) (the algorithm adds 1 because the positions of the array start from 0) to find the range between the two positions and if this value is greater than the one that is find in max, replace the old value of max with the new one and break the internal cycle to continue with the next iteration of i. After the completion of the cycles, the algorithm returns the value found in max.

5. Array2EvenOdd

- **Code and tags**

```
public int[] evenOdd(int[] nums) {
        int evens = 0;
        int odds = nums.length-1;
        int[] evensOdds = new int[nums.length];
        for (int i = 0; i < nums.length; i++) {  // O(n)
            if (nums[i] % 2 == 0) { //O(n)
                evensOdds[evens] = nums[i];
                evens++;
            }
            else {
                evensOdds[odds] = nums[i];
                odds--;
            }
        }
        return evensOdds; // O(1)
```

```
    }
```

- **Notation Big O**
  O(n + n + 1)
  O(n); sum rule and product rule

Array2Sum28

- **Code and tags**

```
public boolean sum28(int[] nums) {
    int cont =0;
    for(int i =0; i<nums.length; i++){ // O(n)
      if(nums[i]==2) cont += nums[i]; // O(n)
    }
    return cont==8; //O(1)
}
```

- **Notation Big O**
  O(n + n + 1)
  O(n); sum rule and product rule

Array2TripleUp

- **Code and tags**

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

- **Notation Big O**
  O(n + n + 1)
  O(n); sum rule and product rule

Array2FizzBuzz

- **Code and tags**

```
public String[] fizzBuzz(int start, int end) {
    int length = end - start;
    String[] fbArray = new String[length];
    for(int i = 0 ; i < length; i++){ // O(n)
        if(start % 3 == 0){ // O(n)
            if(start % 5 == 0){
                fbArray[i] = "FizzBuzz";
            }else{
                fbArray[i] = "Fizz";
            }
        }else if (start % 5 == 0){
            fbArray[i] = "Buzz";
        }else{
            fbArray[i] = start+"";
        }
        start++;
    }
```

```
        return fbArray; // O(1)
    }
```
- **Notation Big O**
  O(n + n + 1)
  O(n); sum rule and product rule

Array2ZeroMax
- **Code and tags**

```
    public int[] zeroMax(int[] nums) {
        int max = 0;
        for (int i = nums.length - 1; i >= 0 ;i--){ // O(n)
            if(nums[i] > max && nums[i] % 2 !=0){ // O(n)
                max = nums[i];
            }
            if(nums[i] == 0){ // O(n)
                nums[i] = max;
            }
        }
        return nums; // O(1)
    }
```
- **Notation Big O**
  O(n + n + n + 1)
  O(n); sum rule and product rule

Array3CountClumps
- **Code and tags**

```
        public int countClumps(int[] nums) {
        int clumps = 0;
        boolean lastClumpMatch = false;
        for (int i = 0; (i + 1)< nums.length; i++) { // O(n)
            if (nums[i] == nums[i+1] && !lastClumpMatch) { // O(n)
                lastClumpMatch = true;
                clumps++;
            }
            else if (nums[i] != nums[i+1]) { // O(n)
                lastClumpMatch = false;
            }
        }
        return clumps; // O(1)
    }
```
- **Notation Big O**
  O(n + n + n + 1)
  O(n); sum rule and product rule

Array3Fix34
- **Code and tags**

```
    public int[] fix34(int[] nums) {
        int mem = nums.length-1;
        for(int i=0; i<nums.length; i++){ // O(n)
          if(nums[i]==3){ // O(n)
          for(int j=mem; j>=0; j--){ // O(n^2)
            if(nums[j]==4){ // O(n^2)
              int temp = nums[i+1];
              nums[i+1] = nums[j];
              nums[j] = temp;
              mem = j;
              break;
            }
          }
        }
      }
      return nums; // O(1)
    }
```

- **Notation Big O**

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

  $O(n^2)$; sum rule and product rule

Array3LinearIn

- **Code and tags**

```
    public boolean linearIn(int[] outer, int[] inner) {
      int start =0;
      for(int i=0; i<inner.length;i++){ // O(n)
        int num = inner[i];
        for(int j=start; j<outer.length;j++){ //O(n*m)
          if(outer[j]==num){ // O(n*m)
            start=j;
            break;
          }
          if(j==(outer.length-1) && outer[j]!=num) return false; // O(n*m)
        }
      }
      return true; // O(1)
    }
```

- **Notation Big O**

  $O(n + n*m + n*m + n*m + 1)$

  $O(n + n*m + n*m + n*m)$; sum rule

  $O(n*m)$; sum rule and product rule

Array3SeriesUp

- **Code and tags**

```
    public int[] seriesUp(int n) {
      int[] array = new int[n*(n + 1)/2];
      int count = 0;
      for (int i = 1; i <= n; i++){ // O(n)
        for (int j = 1; j <= i; j++){ // O(n^2)
          array[count] = j;
          count++;
        }
      }
      return array; // O(1)
```

```
}
```

- **Notation Big O**

  O(n + n^2 + 1)
  O(n + n^2); sum rule
  O(n^2); sum rule

Array3SquareUp

- **Code and tags**

```
public int[] squareUp(int n) {
    int[] arr = new int[n*n];
    int cont = 0;
    for(int i=1; i<=n; i++){ // O(n)
      for(int j=n; j>=1; j--){ // O(n^2)
        if((i-j)<0){ // O(n^2)
          arr[cont]=0;
        }else arr[cont]=j;
        cont++;
      }
    }
    return arr; // O(1)
}
```

- **Notation Big O**

  O(n + n^2 + n^2 + 1)
  O(n + n^2 + n^2); sum rule
  O(n^2); sum rule and product rule

Array3MaxSpan (Extra)

- **Code and tags**

```
public int maxSpan(int[] nums) {
    int max=0;
    for(int i=0; i<=(nums.length/2); i++){ // O(n)
      for(int j=nums.length-1; j>=0; j--){ // O(n^2)
        if(nums[i]==nums[j]){ // O(n^2)
          if((j-i+1)>max) max = j-i+1; // O(n^2)
          break;
        }
      }
    }
    return max; // O(1)
}
```

- **Notation Big O**

  O(n + n^2 + n^2 + n^2 + 1)
  O(n + n^2 + n^2 + n^2); sum rule
  O(n^2); sum rule and product rule

6. In the previous exercises, you can see that 'n' is the size of the arrangement in most of the occasions and therefore the size to where the cycle is performed. Each time an iteration of the cycle is performed, the auxiliary variable either 'i' or 'j' decreases or increases depending on where you want to arrive, until you reach the fulfillment of the detention condition that can be up to a certain value or a value that changes through a variable. On the other hand, 'm' is a variable that can be taken as a cut-off point for a cycle or as the size of another arrangement, depending on the case, this variable can be the start or end of a cycle. These two variables can be used separately or together (in a nested cycle) depending on what is searched in the algorithm.

4) Practice for midterms

1. C) O(n + m)
2. D) O(m $_x$ n)
3. B) O(width)
4. B) $O(n^3)$
5. D) $O(n^2)$
6. A) T(n) = T(n - 1) + C
7. 7.1 T(n) = T(n - 1) + C
   7.2 O(n)
8. B) The mistery function(n) executes O(n*sqrt(n))
9. D) It executes more than n $_x$ m steps
10. C) It executes less than n*log(n) steps
11. C) It executes T(n) = T(n - 1) + T(n - 2) + C steps
12. B) O(m*sqrt(n))
13. A) $O(n^3)$

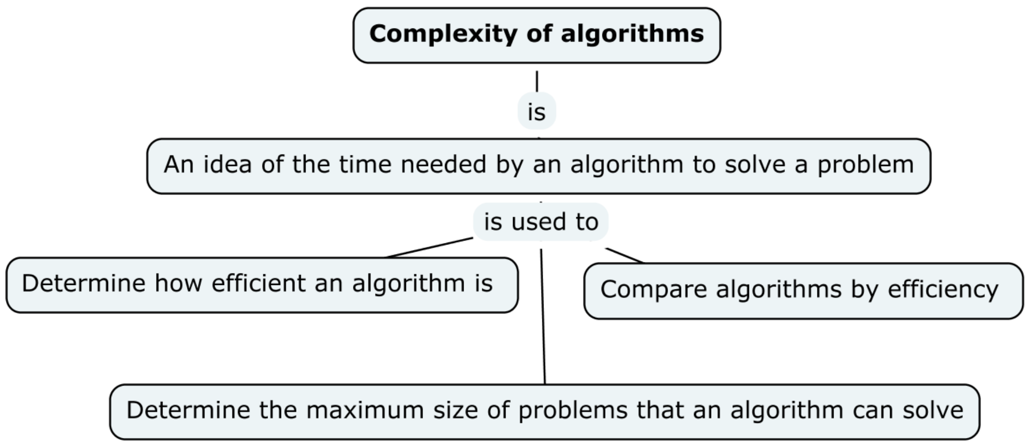
5) Recommended reading (optional)

a) complexity of algorithms and asymptotic lower bound

b) Temporal complexity is the idea of the time consumed by an algorithm to solve a problem, it is used to determine how efficient the algorithm is and how good it is compared to others, under certain parameters. Calculating the complexity of an algorithm also allows knowing the size of problems that can be solved, this is used mostly in cases where the complexity of an algorithm is exponential and solving a problem with a fairly large size would take it even years to resolve.
On the other hand, the lower bound of a problem, which is defined as the least time complexity required for any algorithm which can be used to solve it, helps us to establish a minimum of efficiency (or a maximum of complexity, as you want to see) that an algorithm has to have in order to be useful in solving a problem.

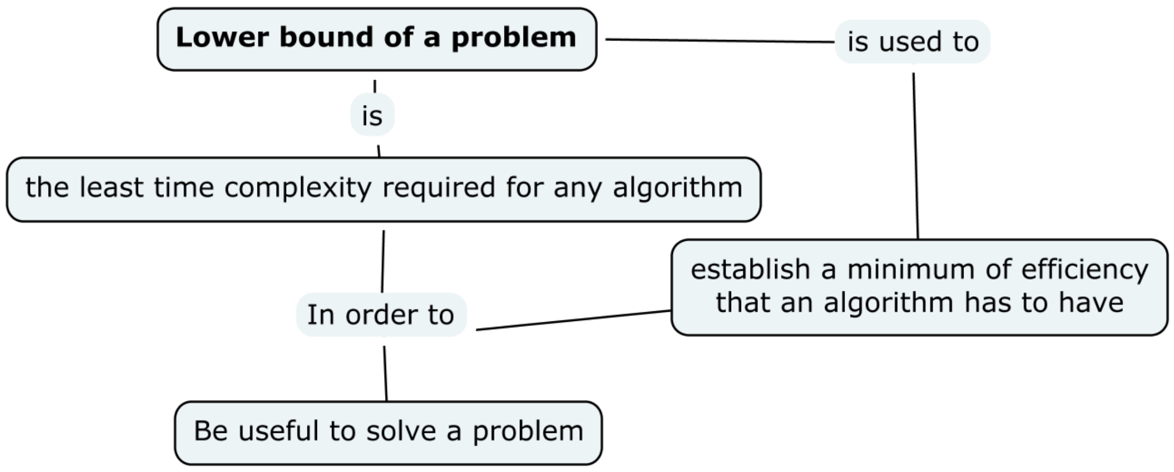
c) Concept maps:

1)



Complexity of algorithms
is
An idea of the time needed by an algorithm to solve a problem
is used to
Determine how efficient an algorithm is
Compare algorithms by efficiency
Determine the maximum size of problems that an algorithm can solve

2)



Lower bound of a problem
is used to
is
the least time complexity required for any algorithm
establish a minimum of efficiency that an algorithm has to have
In order to
Be useful to solve a problem