Laboratory practice No. 2: Big O Notation

Juan S. Cárdenas Rodríguez
Universidad EAFIT
Medellín, Colombia
jscardenar@eafit.edu.co

David Plazas Escudero
Universidad EAFIT
Medellín, Colombia
dplazas@eafit.edu.co

September 9, 2017

1) CODE FOF `ARRAYSUM`, `ARRAYMAX`, `INSERTIONSORT`, `MERGESORT` WITH RANDOM ARRAYS

The `.py` file can be found in the "codigo" folder.

2) ONLINE EXERCISES (`CODINGBAT`)

The source code for all 10 exercises can be found in a `.java` file in "ejercicioenlinea" folder.

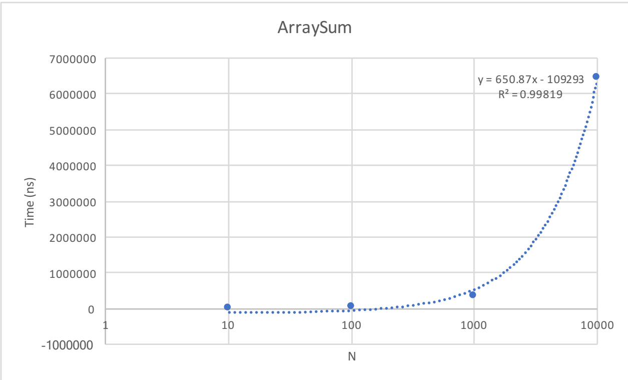
3) SIMULATION OF PROJECT PRESENTATION QUESTIONS

3.a. Time for algorithms

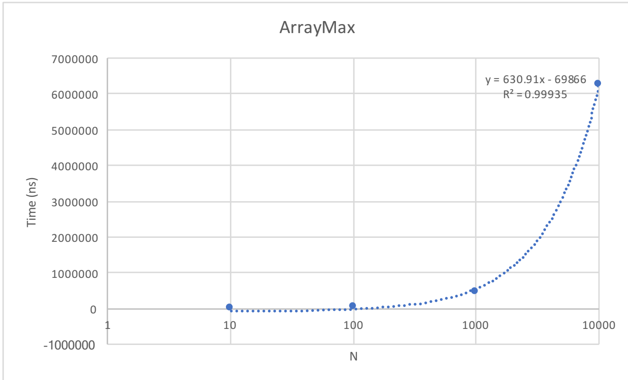
Input/Time(ns)	10	100	1000	100000
ArrayMax	5000	25000	450000	6250000
ArraySum	6000	22000	348000	6410000
MergeSort	31000	291000	3734000	45673000
InsertionSort	10000	445000	47573000	4655923000

Table 1: Execution time for algorithms in nanoseconds.

3.b. *Plots for execution time*

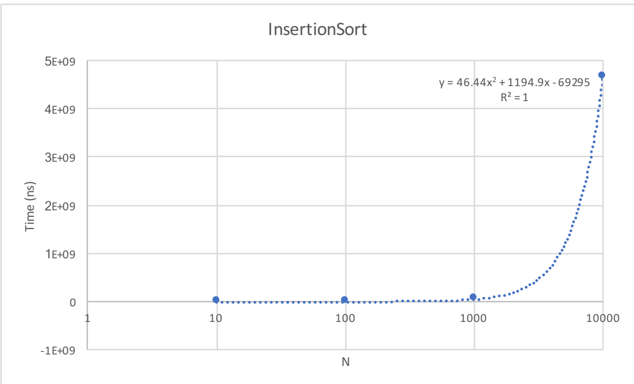


(a) Time vs N for ArraySum

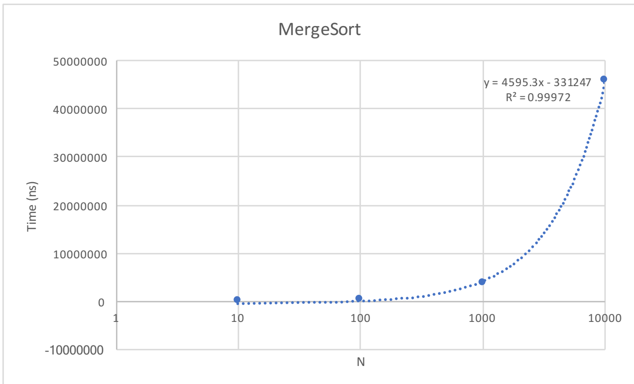


(b) Time vs N for ArrayMax

Figure 1: Array algorithms



(a) Time vs N for InsertionSort



(b) Time vs N for MergeSort

Figure 2: Sort algorithms

3.c. *What can you say between the results obtained and the theoretical Big-O results?*

According to the plots and the value of R, we can say that all of the algorithms execute within the interval of expected results; this is because all the results for R are really close to 1 (even one of them is exactly one), which allow us to affirm that the curve does indeed fit the data.

3.d. *What happens to `insertionSort` for large N?*

As we see in the graphics and the values, insertion sort has a asymptotical complexity of n^2. In this manner, we can see that if we use insertion sort for big numbers it will take a enourmous amount of time, so it will not be efficient in any shape or form.

3.e. What happens to `arraySum` for large N? Why does `insertionSort` increase faster?

As we know, the *Big-O* notation tells us how does the function behave for large values of N. `arraySum` only makes one simple recursive call, thus `arraySum` is $O(n)$. This tells us that the execution time for `arraySum` increases linearly, proportional to the value of N. On the other hand, `insertionSort` is $O(n^2)$, which makes `arraySum` n times more efficent for the worst-case scenario.

3.f. How efficient is `mergeSort` with respect to `insertionSort` for large arrays?

The asymptotic complexity of `insertionSort` is $O(n^2)$, whereas for `mergeSort` is $O(n\log(n))$; it can be proved that $n\log(n) < n^2, \forall n \geq 0$, therefore, `mergeSort` is more efficent than `insertionSort`.

3.g. How does *maxSpan* work?

```
def maxSpan(array):
  arr = []
  max = 0
  for i in range(len(array)):
    for j in range(i,len(array)):
      if array[i] == array[j]:
        c = j-i+1
    arr.append(c)
  for i in range(len(arr)):
    if arr[i] > max:
    max = arr[i]
  return max
```

It works fairly easy. First for every data in the array of integers it moves through the same array to the same index through the end of the array searching for the number again. If it finds it again it sets the variable "c" to the numbers it has between them; it does this until the array ends. Then, it searches the array to find the biggest span and returns that number.

3.h. Array II

i.

```
public int[] zeroFront(int[] nums) {                // c0
    boolean [] used = new boolean [nums.length]; // c1
    int cont = 0;                                // c2
    for (int i = 0; i < nums.length; i++) {      // c3*n
      if(nums[i] == 0) {                         // c4*n
        if (i != cont) {                         // c5*n
          nums[i] = nums[cont];                  // c6+n
          nums[cont] = 0;                        // c7*n
```

```
      }
      cont++;                                  // c8*n
    }
  }
  return nums;                                 // c9
}
```

Therefore, `zeroFront` is $O(c_0 + c_1 + c_2 + c_9 + (c_3 + c_4 + c_5 + c_6 + c_7 + c_8)n)$, where n is the length of `nums`. Applying the sum and product properties, `zeroFont` is $O(n)$.

ii.

```
public int[] notAlone(int[] nums, int val) {   // c0
    for(int i = 1; i < nums.length-1; i++) {   // c1*n
      if(nums[i] == val && nums[i-1] != val
        && nums[i+1] != val) {                 // c2*n
        if (nums[i-1] > nums[i+1])             // c3*n
          nums[i] = nums[i-1];                 // c4*n
        else                                   // c5*n
          nums[i] = nums[i+1];                 // c6*n
      }
    }
    return nums;                               // c7
  }
```

Therefore, `notAlone` is $O(c_0 + c_7 + (c_1 + c_2 + c_3 + c_4 + c_5 + c_6)n)$, where n is the length of `nums`. Applying the sum and product properties, `notAlone` is $O(n)$.

iii.

```
public boolean tripleUp(int[] nums) {            // c0
  for (int i = 0; i < nums.length - 2; i++) {    // c1*(n-2)
    if(nums[i] + 1 == nums[i+1] && nums[i]
     + 2 == nums[i+2]) return true;              // c2*(n-2)
  }
  return false;                                  // c3
}
```

`tripleUp` is $O(c_0 + c_3 + (c_1 + c_2)(n - 2))$, where n is the size of `nums`. When we apply the product and sum properties, `tripleUp` is $O(n)$.

iv.

```
public int[] tenRun(int[] nums) {              // c0
  int tempMult = 0;                            // c1
  boolean used = false;                        // c2
  for(int i = 0; i < nums.length; i++) {       // c3*n
    if (nums[i] % 10 == 0) {                   // c4*n
      used = true;                             // c5*n
      tempMult = nums[i];                      // c6*n
    }
```

```
    if(used)                                        // c7*n
      nums[i] = tempMult;                           // c8*n
  }
  return nums;                                      // c9
}
```

tenRun is $O(c_0 + c_1 + c_2 + c_9 + (c_3 + c_4 + c_5 + c_6 + c_7 + c_8)n)$, where n is the length of nums. When we apply the product and sum properties of the *big-O* notation, yields that tenRun is $O(n)$.

v.

```
public int[] shiftLeft(int[] nums) {               // c0
  int [] mod = new int[nums.length];               // c1
  if (nums.length==1) return nums;                 // c2
  for (int i=1; i<nums.length; i++) {              // c3*n
    mod[nums.length-1]=nums[0];                    // c4*n
    mod[i-1]=nums[i];                              // c5*n
  }
  return mod;                                      // c6
}
```

shiftleft is $O(c_0 + c_1 + c_2 + c_6 + (c_3 + c_4 + c_5)n)$, where n is the size of nums; which implies that shiftLeft is $O(n)$.

3.i. Array III

i.

```
public int[] seriesUp(int n) {                     // c0
  int no = n*(n+1)/2;                              // c2
  int [] nums = new int [no];                      // c3
  int a = 0;                                       // c4
  for (int i = 1; i <= n; i++) {                   // c5*n
    for (int j = 1; j <= i; j++) {                 // c6*n*n
      nums[a] = j;                                 // c7*n*n
      a++;                                         // c8*n*n
    }
  }
  return nums;                                     // c9
}
```

seriesUp is $O(c_0+c_1+c_2+c_3+c_4+c_9+c_5n+(c_6+c_7+c_8)n^2)$, then seriesUp is $O(n^2)$.

ii.

```
public int countClumps(int[] nums) {               // c0
  int c = 0;                                       // c1
  for (int i = 0; i < nums.length-1; i++) {        // c2*n
    if (nums[i] == nums[i+1]) {                    // c3*n
      for (int j = i; j < nums.length; j++) {      // c4*n*n
```

```
          if (nums[j] != nums[i]) {                    // c5*n*n
            i = j;                                     // c6*n*n
            c++;                                       // c7*n*n
          }
          if (c == 0 && j == nums.length-1) {          // c8*n*n
            c++;                                       // c9*n*n
          }
        }
      }
    }
    return c;                                          // c10
  }
```

countClumps is $O(c_0 + c_1 + c_10 + (c2 + c_3)n + (c_4 + c_5 + c_6 + c_7 + c_8 + c_9)n^2)$, where n is the length of nums; then countClumps is $O(n^2)$.

iii.

```
  public boolean linearIn(int[] outer,
    int[] inner) {                                     // c1
    int j = 0;                                         // c2
    int c = 0;                                         // c3
    if (inner.length == 0) return true;                // c4
    for (int i = 0; i < outer.length; i++) {           // c5*n
      if (inner[j] == outer[i]) {                      // c6*n
        j++;                                           // c7*n
        if (j==inner.length) {                         // c8*n
          return true;                                 // c9*n
        }
      }
    }
    return false;                                      // c10
  }
```

linearIn is $O(c_1 + c_2 + c_3 + c_4 + c_10 + (c_5 + c_6 + c_7 + c_8 + c_9)n)$, where n is the size of outer; this implies that linearIn is $O(n)$.

iv.

```
  public int[] fix45(int[] nums) {                     // c1
    boolean [] arr = new boolean[nums.length];         // c2
    for (int i = 0; i < nums.length-1; i++) {          // c3*n
      if (nums[i] == 4 && nums[i+1] == 5) {            // c4*n
        arr[i+1] = true;                               // c5*n
      } else  if (nums[i] == 4 && nums[i+1] != 5) { // c6*n
        for (int j = 0; j < nums.length; j++) {        // c7*n*n
          if (nums[j] == 5 && arr[j] == false) {       // c8*n*n
```

```
                nums[j] = nums[i+1];                    // c9*n*n
                nums[i+1] = 5;                          // c10*n*n
                arr[i+1] = true;                        // c11*n*n
                break;                                  // c12*n*n
              }
            }
          }
        }
        return nums;                                    // c13
      }
```

fix45 is $O(c_1 + c_2 + c_13 + (c_3 + c_4 + c_5 + c_6)n + (c_7 + c_8 + c_9 + c_10 + c_11 + c_12)n^2)$, where n represents the length of nums; this implies that fix45 is $O(n^2)$.

v.

```
    public boolean canBalance(int[] nums) {             // c0
      int sumRight;                                     // c1
      int sumLeft;                                      // c2
      for (int i = 1; i < nums.length; i++) {           // c3*n
        sumLeft = 0;                                    // c4*n
        sumRight = 0;                                   // c5*n
        for (int j = 0; j < i; j++) {                   // c6*n*n
          sumLeft += nums[j];                           // c7*n*n
        }
        for (int j = i; j < nums.length; j++) {         // c8*n*n
          sumRight += nums[j];                          // c9*n*n
        }
        if (sumRight == sumLeft) {                      // c10*n
          return true;                                  // c11*n
        }
      }
      return false;                                     // c12
    }
```

canBalance is $O(c_0 + c_1 + c_2 + (c_3 + c_4 + c_5 + c_10 + c_11)n + (c_6 + c_7 + c_8 + c_9)n^2)$, where n is the size of nums; therefore canBalance is $O(n^2)$.

4) EXAM SIMULATION

i. c) $O(n + m)$

ii. d) $O(n * m)$

iii. b) $O(ancho)$

iv. b) $O(n^3)$

v. d) $O(n^2)$

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