ISEN ALL IS DIGITAL! NANTES | yncréa

# Final project
# Sorting algorithms
-
# Team 8

| | | | |
|---|---|---|---|
| **Module:** Advanced algorithm | | **Date :** 29/09/2022 | |
| **Class:** CIR2 | | **Professor**: MONTERO Leandro | |
| Allan CUEFF | Raphaël FOSSE | Mathias PAITIER | Vincent ROCHER |

# Summary

# 1st algorithm: Bubble sort

## How the algorithm works

Bubble sort algorithm is easy to understand. For each i-step, i goes from 0 to n-1 (n is the array's length), his goal is to sort the greatest number by moving it to the right. To do that, at each i step, j comparisons are done between pairs of elements in position j and j+1 (j start at 0 and goes to length-i-1). If the element j is superior to element j+1 they swap position, else nothing happens. In every case, the variable j increase by 1 and the next comparison will be done with the next pair. At the end, the greatest element in at the right of the array and the algorithm will not compare it anymore. The same process is repeated until all the elements are sorted or when there is no swap during an i-step (a Boolean is initialized to false at the beginning of each i-step).

**Let's take as an example with this size six array:**

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Element | 7 | 13 | 2 | 9 | 25 | 3 |

- For the i-step where i = 0, there is j = 6-0-1 = 5 comparison. The Boolean "swapped" is false. Let's see what happens during this 5 j-step:

  - j = 0: comparison between element in position 0 and 1. 7 < 13, there is no swap, so swapped stay at false.

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Element | 7 | 13 | 2 | 9 | 25 | 3 |

  - j = 1: comparison between element in position 1 and 2. 13 > 2 so the values swap, swapped become and will stay true so, the i-step will at least continue to i=1.

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Element | 7 | 13 | 2 | 9 | 25 | 3 |

  - j = 2: comparison between element in position 2 and 3. 13 > 9 so the values swap.

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Elément | 7 | 2 | 13 | 9 | 25 | 3 |

  - j = 3: comparison between element in position 3 and 4. 13 < 25 so the values don't swap and swapped stay at true.

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Elément | 7 | 2 | 9 | 13 | 25 | 3 |

  - j = 4: comparison between element in position 4 and 5. 25 > 3 so the values swap.

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Elément | 7 | 2 | 9 | 13 | 25 | 3 |

  - j = 5: we reach the stopping condition, so there is no more comparison at the i-step for i = 0. The element in position 5 is sorted and will never be compared anymore.

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Elément | 7 | 2 | 9 | 13 | 3 | 25 |

Here is the result after each i-step:

For i = 1, there is j = 6-1-1 = 4 comparison and swapped becomes true at j = 0.

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Element | 2 | 7 | 9 | 3 | 13 | 25 |

- For i = 2, there is j = 6-2-1 = 3 comparison and swapped becomes true at j = 2.

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Element | 2 | 7 | 3 | 9 | 13 | 25 |

- For i = 3, there is j = 6-3-1 = 2 comparison and swapped becomes true at j = 1.

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Element | 2 | 3 | 7 | 9 | 13 | 25 |

- For i = 4, there is j = 6-3-1 = 1 comparison and swapped stay at false so the algorithm stops.

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Element | 2 | 3 | 7 | 9 | 13 | 25 |

The final array is:

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Element | 2 | 3 | 7 | 9 | 13 | 25 |

Here is the pseudo code related to bubble sort and swap, an auxiliary function:

function **swap** (elemA, elemB)

| var = elemA

| elemA = elemB

| elemB = var

end **swap**

```
start Bubble sort (array, size)
|       for i=0 to size-1 do:
|       |       swapped = false
|       |       for j=0 to size-i-1:
|       |       |       if array[j] > array[j+1] then:
|       |       |       |       swap (array[j], array[j+1])
|       |       |       |       swapped = true
|       |       |       end if
|       |       end for
|       |       if swapped is false do
|       |       |       end Bubble sort
|       end for
end Bubble sort
```

## Temporal complexity

Let's calculate the temporal complexity of bubble sort algorithm by using the pseudo code above:

We define n as the array's length. With the Boolean swapped, we can see different cases:

- If the array is already sorted, it is the best case so:
  - The i loop has only one repetition, so the complexity is O (1)
  - The j loop makes n-i-1 repetitions, so the complexity is O (n)
  - Anything else in the program don't depend on any parameter, so its complexity is O (1)

  So, in the best case the complexity is O (1) x O (n) x O (1) = O (n)

- If the array is reversed, it is the worst case. Using the pseudo code bellow:
  - The red part's complexity is O(n). Indeed, the algorithm makes as many loops as there is element in the array.
  - The blue part has also an O (n) complexity because the j loop makes n-i-1 repetition.
  - The green and black parts have an O (1) complexity because the entry parameters don't change the number of operations.
- If the array is shuffled, we have the middle case. It is the same as the worst case but for the first loop it may stop before i = n because of the Boolean swapped. Indeed, it can shorten the algorithm of x iteration if the x first numbers are sorted.

  So, the temporal complexity for bubble sort in the worst and middle case is O (n) x O (n) x O (1) = $O(n^2)$. As explained before, even if the middle case and the worst case have the same temporal complexity, the middle case will be a little bit faster because it will usually stop before doing every iteration.

  Here is the pseudo code with the complexity detail for the worst case:

```
start Bubble sort (array, size)
|       |       for I from 0 to size-1 do:
|       O (1)           Swapped = false
|       |               |       for j from 0 to size-i-1 do:
|       |               |               |       if array[j] > array[j+1] then:
|     O (n)           O (n)           O (1)           swap (array[j], array[j+1])
|       |               |               |           Swapped = true
|       |               |               |       end if
|       |               |       end for
|       |               |       If swapped = false then:
|       |           O (1)               end Bubble sort
|       |               |       end if
|       |       end for
end Bubble sort


start swap (elemA, elemB)
|       var = elemA
O (1)   elemA = elemB
|       elemB = var
end swap
```

## Comparison of different cases

To have homogeneous results we decided for each array of n elements to run the bubble sort algorithm 5 times, then we do an average of these 5 results. By doing that, we are reducing a possible margin of error. The temporal complexity of the worst case of the algorithm is O ($n^2$), so we decided to set a limit of 10,000 element in the array.

Using the following graph (*figure 1*), we can easily make the comparison between every case. The best case seems instant comparing to the two other case. The middle and worst case have a similar growth. We can see that with 4,000 elements the worst case is about 3 milliseconds longer than the middle case but with 10,000 elements it gets worse, the middle case is about 35 milliseconds faster than the worst case.

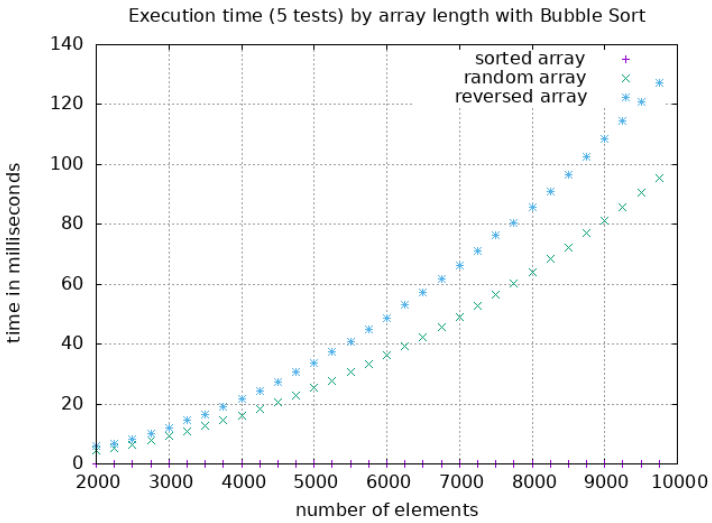



*Figure 1*

## Comparison between theory and practice

On the graph bellow (*figure 2*) we can see a comparison between the middle case represented by the dotted line and a square function that represents the temporal complexity of bubble sort in the middle case (the c = 0.000001 is generated by gnuplot to suit the best to the original graph curve). We can easily conclude that theory and the practice are almost the same. Indeed, on the graph we can see that all middle case's point gravitates around the theorical curve.

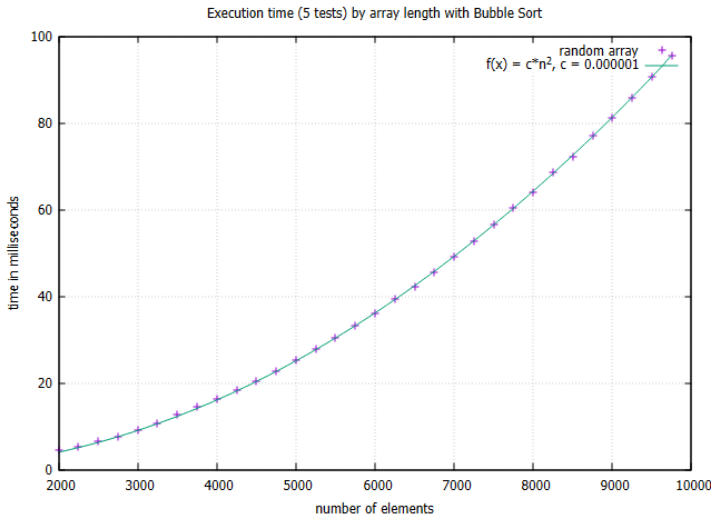



*Figure 2*

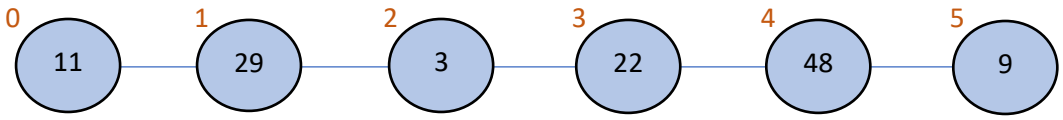
# 2nd algorithm: Heap sort

## How the algorithm works

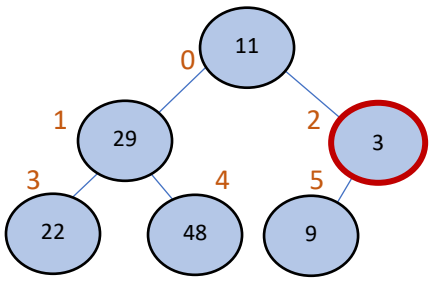
Heap sort uses the principle of heaps to sort an array. A heap can be represented as a binary tree, with the constraint that for a given node, the left and right child's values must be inferior to the node value. This being say, if our array is represented as a heap, we know for sure that the maximum value item is at the root of the heap. The principle of this sort is to take the item at the root of the heap, place it at the end of our result array and then to rebuild the heap to have the next maximum item at root.

This logic implies that we must build the heap before sorting our array, using more time than a Merge Sort for example, but it has the advantage that if we must add a new item to our array, we will be able to place it very fast.

Let's take an example of this sort algorithm execution, this will be our enter array:




We can represent this array as a heap without creating a new structure or using more memory, working on the indexes. We choose to define that the index 0 will be the root of our heap and the maximum index, the last leaf. We can find the left child with the formula (2 * index) + 1. The right child index is (index - 1) * 2. The last node with a child has the index length(array-1)/2.




As said before, the first step will be to build a correct heap, even before trying to sort data. Here is the pseudo code for this step :

*For cursor from (length(array)-1)/2 to 2*

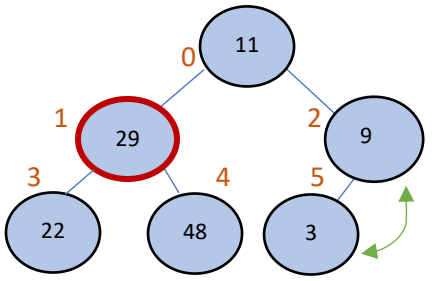
*nodeSieving(array, length(array), cursor);*

That means we are going to start our cursor from position (6-1)/2 = 5/2 = 2

The second step is to explain the principle of "node sieving":

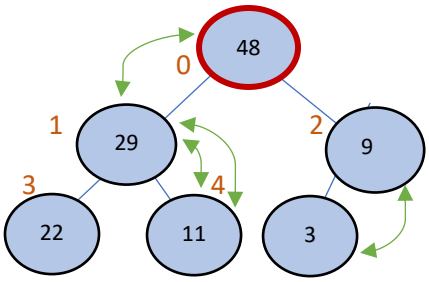
Basically, we are going to want to check if the correct heap conditions are being respected and make it so if it is not already the case.

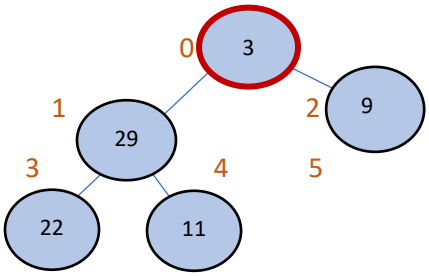
The first step will be to find the max between left and right child, then we check if the max is superior from our current node value. If this is the case, we then swap positions the maximum value child and the node working on the indexes. We then need to call recursively our nodeSieving function on the modified child to check that the heap is still correct after our swap.




Position 5 don't have any child so the node sieving for pos 2 is finished. Next cursor position will be 2-1 = 1

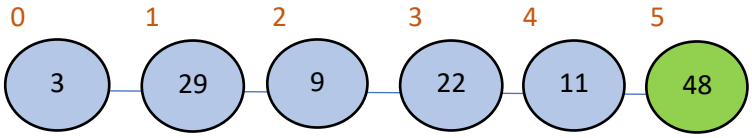
We will repeat the same operations as long as it need for our cursor to arrive at position 0.




Our heap is now correct. We can now start to sort the array. As specified by the definition of a heap, the root is the index with the maximum value in the entire heap. We can grab it and put it to the end of our array by swapping with the last item and start reconstructing the sorted array from the end.

The maximum value being swapped with the last value, we need to reconstruct the heap to find the second maximum value and so on.




## Temporal complexity

The algorithm explained before can be transcribed in pseudo code as shown:

```
function HeapSort(array)
        | for cursor from (length(array)-1)/2 to 0 do:
        O((n-1)/2)      nodeSieving(array, length(array), cursor)
        | end for
        heapLen = length(array)
        | for i from 0 to length(array)-2 do:
        |       swap(first(array), last(array))
        |       heapLen = heapLen – 1
        O(n)    | for cursor from 0 to (length(array)-1)/2 do:
        |       O((n-1)/2)      nodeSieving(array, heapLen, cursor)
        |       | end for
        | end for
end HeapSort
```

<=> O(((n-1)/2) * log(n) + n * ((n-1)/2) * log(n))

= O(log(n)*n²)

```
start nodeSieving(array, heapLength, cursor)
|       left = 2*cursor+1
|       right = (cursor+1)*2
|       maxValuePos = left
|       | if(left < heapLength) then:
|       |       if(right < heapLength) then:
|       |       |       | if(table[right] > table[left]) then:
|       |       O(1)    |       maxValuePos = right
|       |       |       |       | end if
|       |       |       | end if
|       |       | if(table[maxValuePos] > table[cursor]) then:
|       |       O(1)    swap(array[maxValuePos], array[cursor])
|       |       |       nodeSieving(array, heapLength, maxValuePos)
|       |       | end if
|       | end if
end nodeSieving
```

Recursive call

Most of this algorithm has a basic complexity. Swap has a O(1) complexity and nodeSieving function without recursive call is O(1) too. But we have a recursive call, that check the modified children after a swap. The maximum number of recursive calls is the height of the heap, in the case of a swap from the root that implies modifications to a leaf. Height of the array is log(n), so we have a log(n)*O(1), that correspond to a O(log(n)) complexity. Based on that, we can conclude that this algorithm presents a O(n*logn) complexity.

## Comparison of different cases

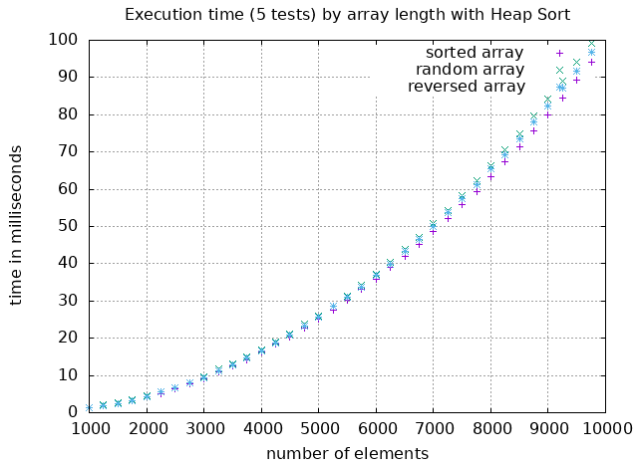



*Figure 3*

Since we must completely rearrange the array to form a correct heap, the elements order in the array doesn't alter the execution in a significative way, it may just reduce the number of basic operations required, which are not considered when calculating the complexity. So, we can't say there is a best and a worst case.

We did tests on sorted, randoms and reverse sorted arrays from 1000 to 10000 elements. From 0 to 1000 (*Figure 3*) and could observe that in fact, the sort time measured for already sorted arrays is slightly less important, but the 3 cases are very similar, and

they have the same tendance so we can confirm that there is no better case.

## Comparison between theory and practice

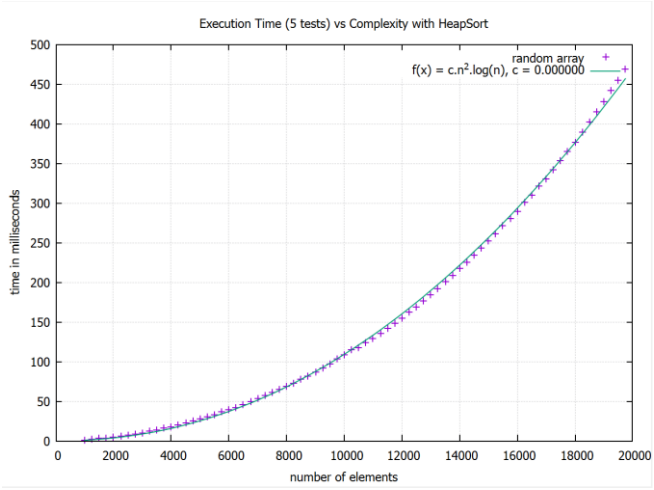



*Figure 4*

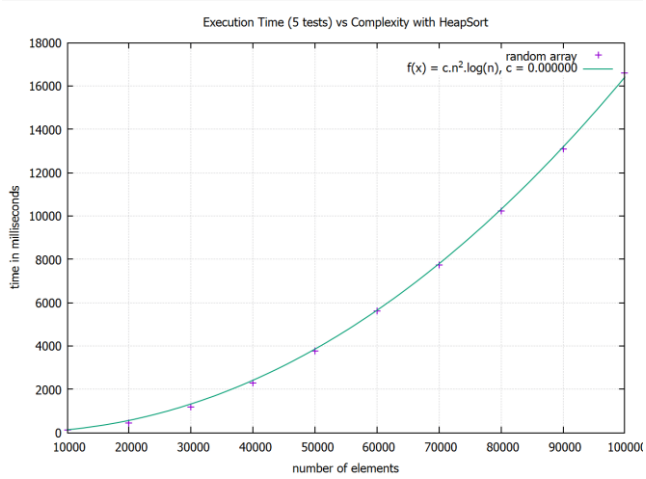



*Figure 5*

We can see with these different tests that the reality we observed testing with random arrays of integers between 0 and 1000 sticks to the complexity we calculed before, for small arrays (*Figure 4*) and larger ones (*Figure 5*).

# 3rd algorithm: Stupid sort

## How the algorithm works

The stupid sort algorithm is simple, the goal is as it says in his name, to be the slowest as possible, so the stupid as possible. The principle of it, is to shuffle the array while it is not sorted. Thus, while the array is not sort, we shuffle each element of with a random element. As random function in C is actually only pseudo-random, we used the time value to initiate it.

**Here is the pseudo code related to stupid sort, and the auxiliary function swap and verify:**

```
start stupidSort(array)
verify = false
    | while not verify(array):
    |         | for i from 0 to length(array)-1 do:
    |         |         swap(array[i], array[rand() % (len)])
    |         | end for
    | end while
end stupidSort
start verify(array, size):
    | unsigned int valid = 1
    | for I from 0 to size-1 do:
    |         | if array[i] = array[I+1] then:
    |         |         Valid = 0
    |         | end if
    | end for
end verify
```

**Example :**

Iteration = 1:

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Element | 1 | 35 | 2 | 12 | 27 | 40 |

The array is not swap so the algo is going swap it.

Iteration = 2:

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Element | 27 | 40 | 12 | 2 | 1 | 35 |

Each element has been swap with another one but the array is still not sort. So, the algo is gone swap it.

Iteration = 3:

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Element | 1 | 35 | 27 | 12 | 2 | 40 |

Each element has been swap with another one but the array is still not sort. So, the algo is gone swap it again.

Iteration = 4:

| Position | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Element | 1 | 2 | 12 | 27 | 35 | 40 |

The array is sorted, so the algorithm end.

## Temporal complexity

**Best case:** The first action of the algorithm is to check if the array is sort. So, if the array is already sort, the algorithm is not gone modify it and end. So, the expected complexity is **O(n).**

**Middle case:** If all the elements are different there are n! possible permutations. Therefore, the probability that the array is already sort is $\frac{1}{n!}$, this means that we will have to do n! permutations to generate a sort array. Moreover, each swap has a O(n) complexity, and the verification a O(n) complexity. So, the expected complexity is O(n*n*n!) = **$O(n^2*n!)$**

**Worst case:** The algorithm mixes the array while it is not sorted, one would think that it will never end. However, in reality this doesn't happen and we can prove it with the infinite monkey theorem (which is why stupid sort is also call monkey sort). According to this theorem, if a monkey is typing on a keyboard during on unlimited amount of time, it will almost surely type any text possible. In the case of our algorithm, for an unlimited amount of time it will it will generate a sorted array.

**Example of the complexity in pseudo code :**

```
start stupidSort(array, size):
|               |  verify = false
|               |  while verify(array, size) is false do:
|               |               |  for I from 0 to size-1 do:
O(n²*n!)   O(n!*n)      O(n)            O(1)     swap(array[i], array[rand() % (len)])
|               |               |  end for
|               |  end while
end stupidSort


start verify (array, size):
      |  valid = true
      |  for I from 0 to size-1 do :
      |        |  if array[i] = array[I+1] then:
   O(n)    O(1)         valid = false
      |        |  end if
      |  end for
end verify
```

## Comparison of different cases

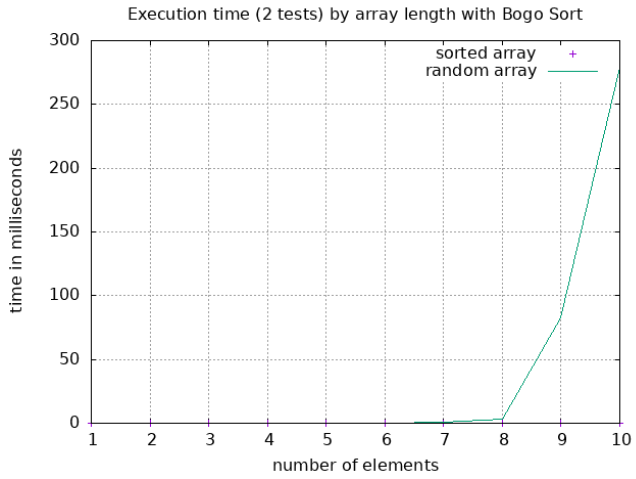



*Figure 6*

Here (*figure 6*) we wanted to generate thanks to 2 arrays. One is the sort and represent the best case (purple), in the other a random one and represent the middle and the worst case(green). After some try, we see that the stupid sort was slow, so we decide to do the test with just a few elements (here 10 elements).

**Sort array (purple cross):**

The graph does not show any big variation and an execution time close to 1ms, this is because the complexity for a random array is O(n). Thus, in this case, the number of elements is really low so as the execution time.

**Random array (green curve):**

In this case, the difference is clear, from 7 to 10 the curve grows incrementally fast. Thus, this show that for just one more element, the time is multiplied by 30 between 8 elements and 9 elements and by 3 between 9 elements and 10 elements. We can see that, the more elements, the more time, but not proportionally, due to the random.

## Comparison between theory and practice

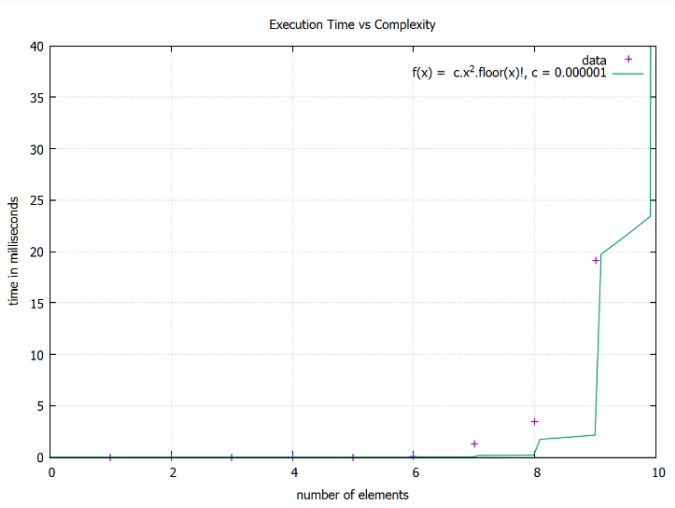



*Figure 7*

This graph (*figure 7*) shows 2 curves, the one with the purple cross is the execution time of stupid sort from a random array, and the green line represent the theoretical model create with the complexity of the algorithm. As we look at it, we see that the theoretical model is still longer than the real one, except for the last one with 10 elements. Thus, it shows what probability do and due to the random, for each point, it will around the expected time, but some like for 10 elements we will be much longer or shorter because of the probability. But, for many tries, we will see the result towards to the theorical model.

# 4th algorithm: Radix sort

## How the algorithm works

Radix sort is the hardest algorithm to understand, because to make this algorithm work, we need another sorting system. There are many sorting algorithms to use with radix sort, and we decide to use counting sort.

Radix function by focusing on the unit, then the ten, then the hundreds… For each power of ten radix use another algorithm to sort this number. Radix isolate the unit then sort the numbers then take the ten sort them…

start radix_sort(array, size):

| max = array[0]

| for from 0 to size-1 do:

| | if array[i] > max then:

| | | max = array[i]

| | end if

| end for

| for power=1 until $\frac{max}{place}$ > 0 do:

| | counting_sort(array, size, power)

| | power=power*10

| end for

end radix_sort

start counting_sort(array, size, power):

| output[size+1]

| count[size+1]

| for i=0 to 9 do:

| | count[i]=0

| end for

| for i=0 to size do:

| | count[$\frac{array[i]}{power}$%10]= count[$\frac{array[i]}{power}$%10]+1

| end for

| for i=0 to 10 do:

| | count[i]=count[i]+count[i-1]

| end for

| for i=size-1 to 0 do:

| | output[count[$\frac{array[i]}{power}$%10]-1] = array[i]

| | count[$\frac{array[i]}{power}$%10]= count[$\frac{array[i]}{power}$%10]-1

| end for

| for i=0 to size do:

| | array[i]=output[i]

| end for

end counting sort

**So now let's have an example:**

array:

| position | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| element | 185 | 921 | 347 | 55 |

- max (before the loop) = 187 = > max (after the loop)= 921
- power = 1 so $\frac{921}{1}$ = 921 > 0 so we execute for the first-time counting sort
- we create output with a size = 5 and count with a size = 10

So, in the next loop we fill count with 0 and we get:

| position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| element | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

We add 1 to the element in count wo have the position who correspond the unit of array and we finish this loop with count:

| position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| element | 0 | 1 | 0 | 0 | 0 | 2 | 0 | 1 | 0 | 0 |

Now, we add to an element, the element of the previous element. And we get after the loop:

| position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| element | 0 | 1 | 1 | 1 | 1 | 3 | 3 | 4 | 4 | 4 |

Then we fill output with the value of array in position i and put them in position count[$\frac{array[i]}{power}$%10]-1 (as an example we take i=3 so array[3] = 55 and $\frac{array[3]}{power}$%10 = 5 and count[5]-1 = 2 so 55 will be in position 2 in output). At the end of the loop, we get:

count:

| position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| element | 0 | 0 | 1 | 1 | 1 | 1 | 3 | 3 | 4 | 4 |

Output:

| position | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| element | 921 | 185 | 55 | 347 | 0 |

- and the last loop replace the element of array by the element (sorted) of output and we finally have a sorted array:

| position | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| element | 921 | 185 | 55 | 347 |

- Once counting finished the algorithm return in radix and repeat counting until $\frac{max}{power} > 0$.

## Temporal complexity

Radix sort doesn't have a best or worst case, each time it goes through all the number. Compared to the other algorithms, radix have two variables in his complexity, the first one is, like the others, the array's size n, and the second the power of ten of the maximum value d. So, we finally have the complexity:

start radix_sort(array, size): -> O(d.3n + n)

| int max = array[0]

| | for i=0 to size-1 do:

| | if array[i] > max then:

| O(n) max = array[i]

| | end if

| | end for

| | for power=1 until $\frac{max}{place} > 0$ do:

| O(d) do counting_sort(array, size, power) -> O(3n)

| | power=power*10

| | end for

end radix_sort

start counting_sort(array, size, power): -> O(3n+20)

| int output[size+1]

| | for i=0 to 10 do:

| O(10) count[i]=0

| | end for

| | for i=0 to size do:

| O(n) count[$\frac{array[i]}{power}$%10]= count[$\frac{array[i]}{power}$%10]+1

| | end for

| | for i=0 to 10 do:

| O(10) count[i]=count[i]+count[i-1]

| | end for

| | for i=size-1 to 0 do:

| O(n) output[count[$\frac{array[i]}{power}$%10]-1] = array[i]

| | count[$\frac{array[i]}{power}$%10]= count[$\frac{array[i]}{power}$%10]-1

| | end for

| | for i=0 to size do:

| O(n) array[i]=output[i]

| | end for

end counting sort

## Comparison of different cases

Each point is an average between 5 time from 5 different array with the same size. With this system we reduce the probability of mistake even though with radix only the power of ten or the size of the array have an influence on the time needed not the value. We decide to go to the power of ten 9 for the maximum otherwise we would have some issue with the number maximum an int can get and cause some problem.

We can see on the graph (*figure 8*) that the power of ten influence the growth of the time needed. If the power of ten is high, we will need more time. The blue curve takes more than 3 times more time than the purple one confirming than the power of ten influence the time needed, and we can maybe theorise about a link between the power of ten and the time.

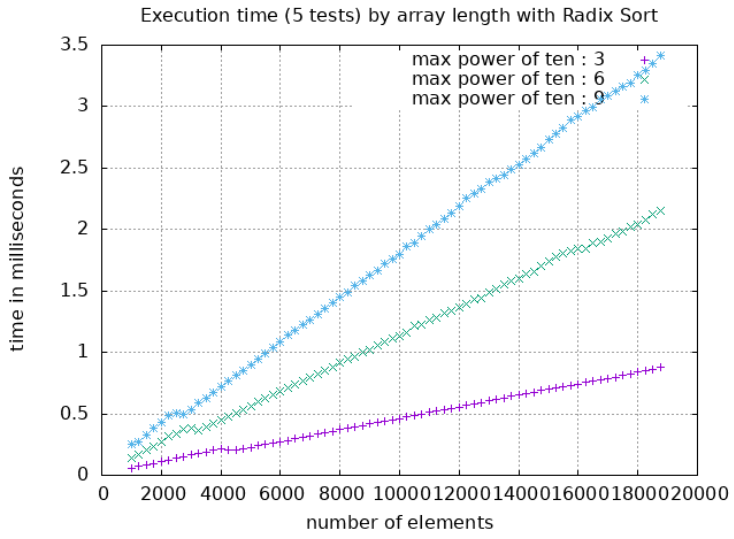



*Figure 8*

## Comparison between theory and practice

The graph below (*figure 9*) shows 2 curves, the first one composed by the purple dot is the execution time of radix with a max power of ten at 6, and the green line represent the theoretical model create with the complexity of the algorithm. We can see that the purple dot is always near the green line, and we can conclude that the real attempt is true with a minimum gap.

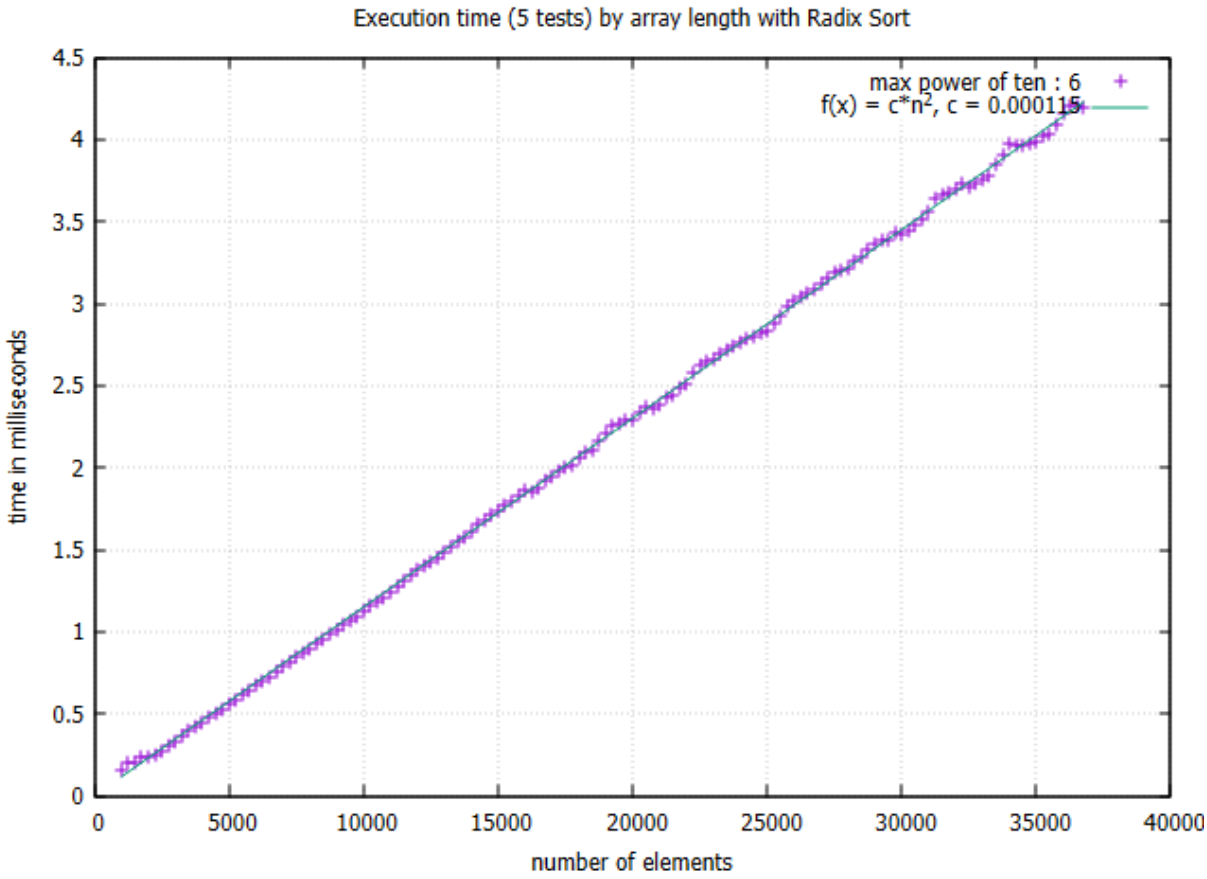



*Figure 99*

# Conclusion:

These two graphs (*figure 10 & 11*) show a comparation between three of our sorting algorithms, we exclude stupid sort because of the infinite time he can possibly take. The one on the left show a comparation between the worst case, and the one on the right show a comparison between the middle case.

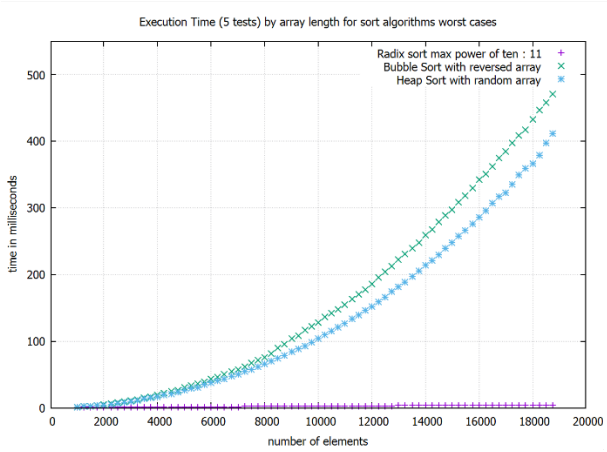



*Figure 10 10*

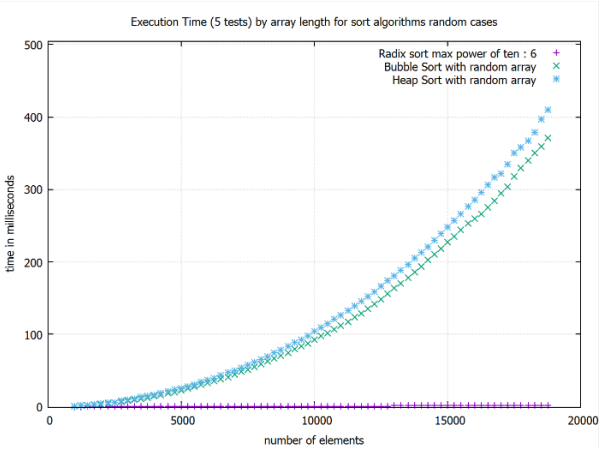



*Figure 1111*

On the first one (*figure 10*) we can see that in worst case the slowest of all the algorithm is Bubble sort (who is the only one who have a worst case) but isn't far from Heap sort. However, these two-sorting system are still slow compared to Radix sort who in comparation with the two others don't even seem to progress even whit the power of ten et 11.

The second graph (*figure 11*) is quite like the first one at the difference here that the slowest is now Heap sort. In a middle case, Bubble sort is faster than Heap sort the rest of the time. But there are still slow compared to Radix sort who still don't seem to move on the graph.