Technical University of Moldova
Inginerical department S.A.

# Report

**№**: 2

APA

**Subject:** Divide et empera

Author: Terman Emil FAF161
Prof: I. Costiuc

Chisinau 2017

## Objectives:

– to obtain skills in writing efficient code

– to develop analytical thinking

– to be able to correctly find the complexity of sorting algorithms

– to learn the pros and cons of different sorting algorithms

## Tasks:

– to study the divide and conquer method

– to implement algorithms based on this method

# The procsess

In computer science, **divide and conquer** is an algorithm design paradigm based on multi-branched *recursion*. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

This divide and conquer technique is the basis of efficient algorithms for all kinds of problems, such as sorting, merge sort, etc), multiplying large numbers (e.g. the Karatsuba algorithm), finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFTs).

The correctness of a divide and conquer algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations.

# 1 Mergesort

**Algorithm 1**: Mergesort

```
1  function mergeSort(tab: array of int, int firstI, int lastI):
2      int mid
3
4      if (firstI < lastI):
5          mid = (firstI + lastI) / 2
6          mergeSort(tab, firstI, mid)
7          mergeSort(tab, mid + 1, lastI)
8          merge(tab, firstI, mid, lastI)
9
10 function merge(tab: array of int, int firstI, int mid, int lastI):
11     tmp: array[lastI - firstI + 1] of int
12     int i, j, k
13
14     i := k := firstI
15     j := mid + 1
16     while (i <= mid and j <= lastI):
17         if (tab[i] <= tab[j]):
18             tmp[k - firstI] := tab[i]
19             i++
20         else:
21             tmp[k - firstI] := tab[j]
22             j++
23         k++
24
25     while (i <= mid):
26         tmp[k - firstI] := tab[i]
27         k++
28         i++
29
30     while (j <= lastI):
31         tmp[k - firstI] := tab[j]
32         k++
33         j++
34
35     for (i := firstI; i < lastI + 1; i++):
36         tab[i] := tmp[i - firstI]
```

*Mergesort in python* *Mergesort animation*

## 1.1 Time complexity

Line 6, 7: $T\left(\frac{n}{2}\right)$
Line 8: $O(n)$
Let T(n) be the sorting time of an array of length n, then:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = 2T\left(\frac{n}{2}\right) + n$$
$$n = 2^a;\ a = \log_2 n$$
$$T(2^a) = 2T(2^{a-1}) + 2^a$$
$$t_a = 2t_{a-1} + 2^a$$
$$(x-2)\cdot(x-2) = 0$$
$$r_1 = 2;\ r_2 = 2$$
$$t_a = C_1 \cdot 2^a + C_2 \cdot a \cdot 2^a$$
$$t_x = C_1 \cdot n + C_2 \cdot n\log_2 n \Rightarrow$$
$$T(n) = O(n\log_2 n)$$

## 1.2 Advantages

- it's stable: the time complexity doesn't depend on the input
- it's possible to implement an efficient multithreaded program

## 1.3 Disadvantages

- requires a lot of memory
- requires additional stack memory for recursive calls

# 2 Quicksort

**Algorithm 2**: Quicksort

```
function quickSort(tab, firstI, lastI):
    int pi

    if (firstI < lastI):
        pi := partition(tab, firstI, lastI)
        quickSort(tab, firstI, pi)
        quickSort(tab, pi + 1, lastI)

function partition(tab, firstI, lastI):
    int x, i, j

    x := tab[firstI]
    i := firstI - 1
    j := lastI + 1

    while True:
        repeat j-- until tab[j] <= x
        repeat i++ until tab[i] >= x

        if i < j:
            swap(tab[i], tab[j])
        else:
            return j
```

*Quicksort in python* *Quicksort animation*

## 2.1 Time complexity

| Worst case | Best case |
|---|---|
| $T(n) = O(n) + T(0) + T(n-1) = O(n) + T(n-1)$ | $T(n) = O(n) + 2T\left(\frac{n}{2}\right)$ |
| $T(n) = T(n-1) + n$ | $n = 2^a;\ a = \log_2 n$ |
| $(x-1)(x-1)^2 = 0$ | $T(2^a) = 2T(2^{a-1}) + O(2^a)$ |
| $r_1 = 1;\ r_2 = 1;\ r_3 = 1;$ | ... |
| $t_n = C_1 \cdot 1^n + C_2 \cdot n \cdot 1^n + C_3 \cdot n^2 \cdot 1^n$ | $T(n) = O(n \log_2 n)$ |
| $T(n) = O(n^2)$ | |

## 2.2 Advantages

- it doesn't require additional memory
- quicksort's divide-and-conquer formulation makes it amenable to parallelization using task parallelism

## 2.3 Disadvantages

- it's not as stable as Mergesort, meaning that it may require more iterations. But since it requires less memory, it can still run in less time
- requires additional stack memory for recursive calls

# 3 Insection sort

**Algorithm 3**: Insertion sort

```
function insertionSort(tab: array of int[0..n], int size):
    int i, j

    i := 1
    for i in range(1, size):
        j := i
        while (j > 0):
            if tab[j] < tab[j - 1]:
                Swap tab[j] and tab[j - 1]
            j := j - 1
```

*Insertion sort in python* *Insertion sort animation*

## 3.1 Time complexity

| | |
|---|---|
| Best case | $O(n)$ |
| Worst case | $O(n^2)$ |
| On Average | $O(n^2)$ |

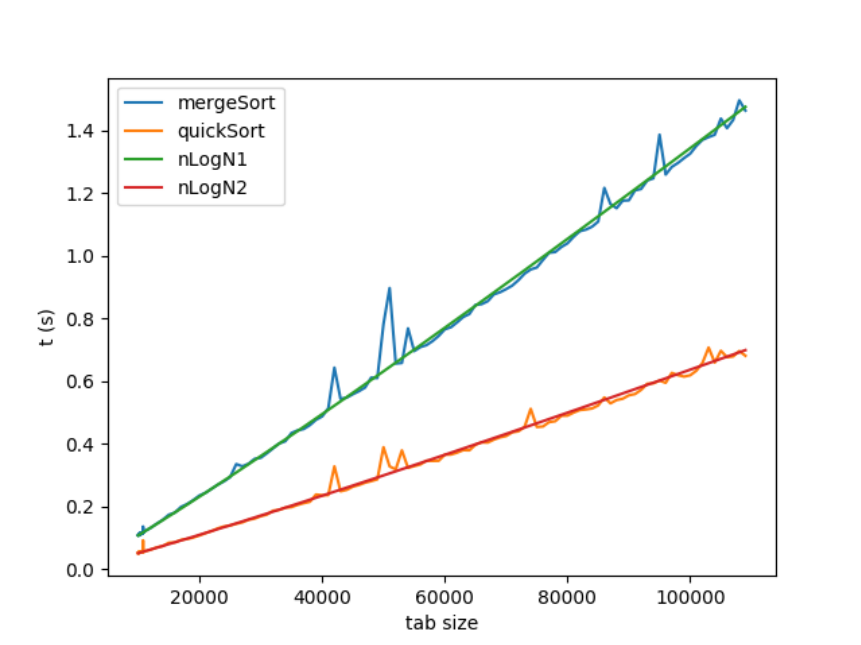
## 3.2 Advantages

- a very short and intuitive algorithm
- doesn't require any additional memory
- it shows a better performance when the table is almost sorted
- it can sort a list as it recives it

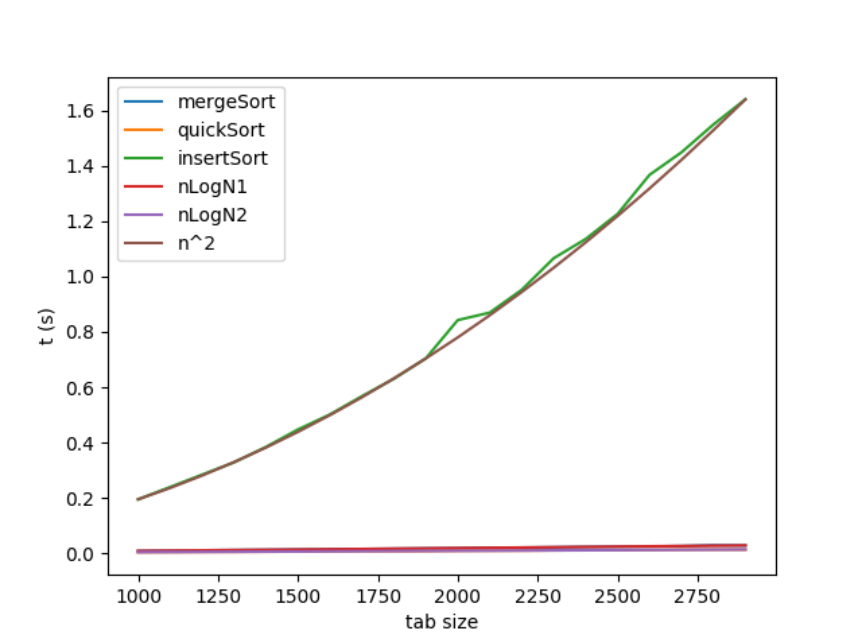
## 3.3 Disadvantages

- it's much less efficient on large tables than Quick or Merge sort

# 4 Exceution time diagram




**Img 1**: Merge and Quick sort time diagrams




**Img 2**: Merge, Quick and Insert sort time diagrams

In this graph, the nLogN plots are calculated using:

$$f(x) = x \log_2 x * K$$

Where K is the minimum value divided by f($x_0$). The same goes for $f(x) = x^2$.

# 5 Conclusion

In this laboratory work, 3 different algorithms were studied: Mergesort, Quicksort and Insertion sort. From the time diagrams, it can be noticed that Quicksort has the least coputation time and Mergesort required almost twice the amount of time, even though, it has a stable $log_2 n$ complexity. The reason why Mergesort is slower, is that it requires additional time to manage the additional memory it needs. So, the fastest and most practical method would be Quicksort.

But, insertion sort isn't too bad too. It can be seen on the **Img 2** time diagram, that it doesn't stand a chance agains the divide and conquer algorithms in execution time, but it doesn't negate its most important advantage: more elements can be added while sorting, which can prove much more efficient in practice.

The reason why Divide and conquer algorithms are fast, is that it divides a very complex problem in many, very simple problems. This concept is a basic rule in general: Divide the complex problems in many small problems, until those little problems are simple enough to be resolved.

# 6 Anexes

**Pyhon Code 1**: Mergesort

```
def mergeSort(tab, firstI, lastI):
    if firstI < lastI:
        mid = int((firstI + lastI) / 2)

        mergeSort(tab, firstI, mid)
        mergeSort(tab, mid + 1, lastI)

        _merge(tab, firstI, mid, lastI)

def _merge(tab, firstI, mid, lastI):
    tmp = [0] * (lastI - firstI + 1)
    i = k = firstI
    j = mid + 1
    while i <= mid and j <= lastI:
        if tab[i] <= tab[j]:
            tmp[k - firstI] = tab[i]
            i += 1
        else:
            tmp[k - firstI] = tab[j]
            j += 1
        k += 1

    while i <= mid:
        tmp[k - firstI] = tab[i]
        k, i = k + 1, i + 1

    while j <= lastI:
        tmp[k - firstI] = tab[j]
        k, j = k + 1, j + 1

    #Copy tmp[] in tab[]
    for i in range(firstI, lastI + 1):
        tab[i] = tmp[i - firstI]
```

**Pyhon Code 2**: Quicksort

```
def quickSort(tab, firstI, lastI):
    iterations = 0
    if firstI < lastI:
        iterations += 1
        pi = _partition(tab, firstI, lastI)
        iterations += quickSort(tab, firstI, pi)
        iterations += quickSort(tab, pi + 1, lastI)
    return iterations

def _partition(tab, firstI, lastI):
    x = tab[firstI]
    i = firstI - 1
    j = lastI + 1

    while True:
        while True:
            j -= 1
            if tab[j] <= x: break
        while True:
            i += 1
            if tab[i] >= x: break

        if i < j:
            tab[i], tab[j] = tab[j], tab[i]
        else:
            return j
```

**Pyhon Code 3**: InsertionSort

```
def insertionSort(tab, i = 0, j = 0):
    for i in range(1, len(tab)):
        j = i
        while j > 0:
            if tab[j] < tab[j - 1]:
                tab[j], tab[j - 1] = tab[j - 1], tab[j]
            j -= 1
```