LIÈGE université
Sciences Appliquées

INFO2050: Advanced Programming
Assignement 1 - Report
October 15^{th}, 2020

Alain Eros Prestige	Musoni	s202208
Ayawo Désiré	Dandji	s197206

Introduction

In this assignment we were asked to Implement Sorting algorithms such as quickSort, HeapSort, placeSort and RecSort. After the implementation we are supposed to analyse them in terms of complexity and then answer the questions that were asked. We have proceeded in an organised way where we simply analyse algorithms after testing them by using the average of 10 experiments each as asked.

1 Algorithms seen in Class: Experimental analysis

In this experiment we used our computer with specifications of 2.7GHz Dual-Core Intel Core i5, 8GB of RAM DDR3 for testing purposes. This may have some impact on the results we observed during the experiment.

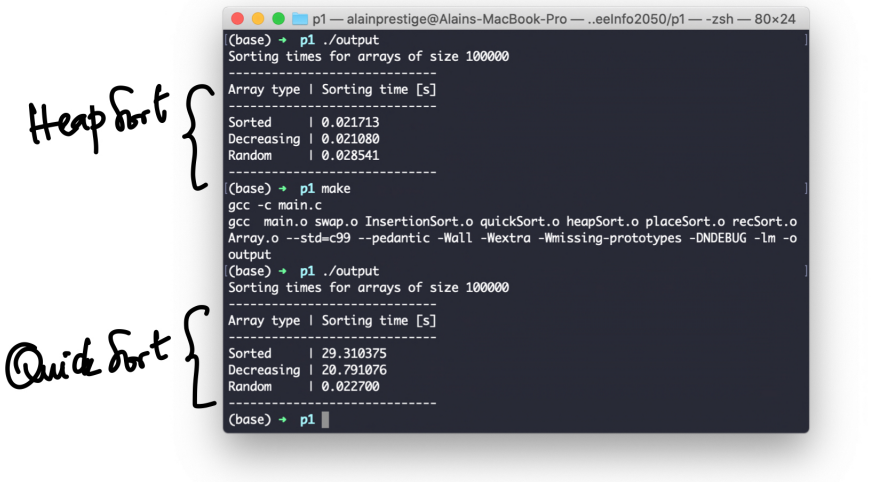
1.a Average Execution time table of the three algorithms

Type de tableau	aleatoire			croissant		
Taille	10^3	10^4	10^5	10^3	10^4	10^5
InsertionSort	0.000074	0.006734	0.678231	0.000005	0.000041	0.000404
QuickSort	0.001687	0.026052	0.0225423	0.002949	0.285984	29.878381
HeapSort	0.000166	0.001957	0.0278607	0.000154	0.001923	0.022098

1.b Discussion on the results for each table type

Theoretically we know that the best algorithm performance is seen by increasing the size of the array to sort. In our case we see that the time to sort always increases if the size of the array increases which correspond to what know theoretically.

Theoretically heapSort is faster than quickSort since its complexity in the worst case scenario is expressed by the order of $\theta(n \log n)$ while the quickSort experience difficulties in worst case scenario when the array is almost or already sorted as we can observe in the table above. Althought this is the case in theory, quickSort remains faster and widely used in practical due to its few comparisons and swaps which leads to fast performance, that explains the use of less memory cache.



For the insertionSort, we know that theoretically it's complexity is linear in the best case scenario, the best case scenario being when the array is already sorted but it is quadratic in the worst case scenario which is also confirmed in our execution table. In conclusion we notice that the quickSort algorithm has the best performance in practice when the array is not already sorted but if n is large and we don't know the state of the array in advance, we should obviously use heapSort due to its guarantee when n is large.

2 PlaceSort

2.a PlaceSort pseudo-code

```
placeSort(array, index, k, length)
    initialIndex=index
    if initialIndex == length-1
        return
    else
        do
            k = findNewPlace(array, initialIndex, length)
            if initialIndex != k and array[initialIndex] != array[k]
                swap(array[initialIndex],arra[k])
            else if initialIndex != k && array[initialIndex]==array[k] && k==(initialIndex+1)
                        swapPlace(&array[initialIndex], &array[k+1])
            else
                initialIndex = initialIndex + 1
                break
        while(initialIndex != k)
        placeSort(array, initialIndex, k, length)
```

Where index is the index of a given element, k is the ideal index of that element in the sorted ar-

ray and the length is the size of the array to sort.
The findNewPlace function returns an Ideal place in the sorted: array.

```
findNewPlace(array, index, length)
    size_t count=0
    for j=0 to length
    if array[j] < array[index]
        count++;
return count;
```

2.b Experimental analysis of placeSort

Average execution table:

Type de tableau	aleatoire			croissant		
Taille	10^3	10^4	10^5	10^3	10^4	10^5
InsertionSort	0.000074	0.006734	0.678231	0.000005	0.000041	0.000404
QuickSort	0.001687	0.026052	0.0225423	0.002949	0.285984	29.878381
HeapSort	0.000166	0.001957	0.0278607	0.000154	0.001923	0.022098
PlaceSort	0.002359	0.494734	49.029116	0.002532	0.243581	23.86636

PlaceSort performs well for arrays of small size but less so when the array size becomes big. Along side other algorithms placeSort is not better in performance than others in general and we can see that in the table above. Without generalization, we observe that when the array is already sorted it performs better than the quickSort algorithm even if they are a bit similar in the implementation.

2.c Complexity analysis in time of placeSort

Looking at the table above, we observe a big change when n increases but also the array types has a huge influence on the performance of the placeSort. We observe that when the array is sorted or almost sorted, place sort will be much faster than when the array is unsorted. This is due to the implementation of place sort where we have 2 loops and a recursive call and in each loop we make a comparison n times which results to approximately quadratic in the worst case scenario.

3 RecSort

3.a Experimental analysis of RecSort

Average execution table:

Type de tableau	aleatoire			croissant		
Taille	10^3	10^4	10^5	10^3	10^4	10^5
InsertionSort	0.000074	0.006734	0.678231	0.000005	0.000041	0.000404
QuickSort	0.001687	0.026052	0.0225423	0.002949	0.285984	29.878381
HeapSort	0.000166	0.001957	0.0278607	0.000154	0.001923	0.022098
PlaceSort	0.002359	0.494734	49.029116	0.002532	0.243581	23.86636
RecSort	0.003195	0.311923	31.231808	0.003891	0.393769	38.742483

RecSort also does not have the best performance compared to others but by observing in the table above we notice that it performs better than the place sort when the array is not already sorted(random).

3.b Complexity analysis in time of RecSort

RecSort complexity is not better than the others in terms of performance. It is all about finding the best partition index using the give place algorithm and then use the divide and conquer methodology as in mergeSort, this result to the the algorithm no being an in-place algorithm because it uses additional memory. Looking at the table above, it is better to use RecSort algorithm when n(size of the array) is smaller because the performance increases quadratically when n is big.

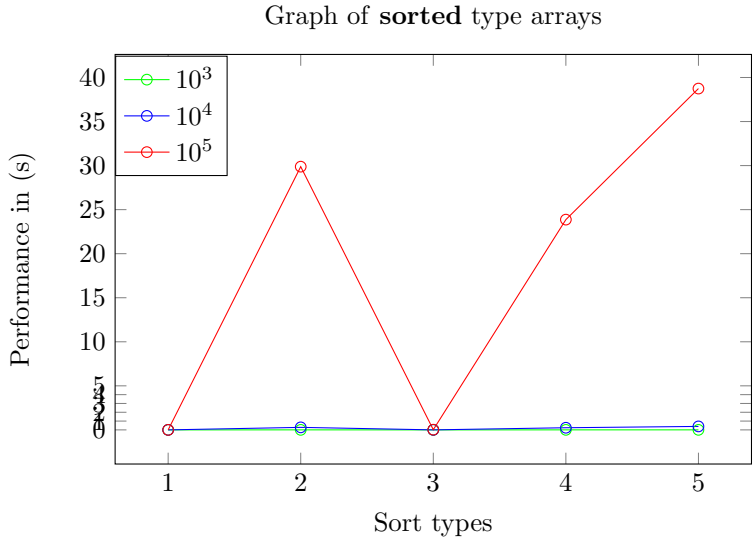
3.c Conclusion on the benefits of PlaceSort and RecSort

The benefits of using placeSort is that it will use less swaps and comparisons compared to other sorts we see that it performs well when the array is almost or already sorted which was not the case for quickSort. RecSort's benefits is that it uses a place algorithm which enables it to find a good spot for partitioning, this enables it to divide the array such that all elements on the left are smaller than the pivot and all the elements on the right are bigger. This will make it perform well when the array is not a sorted or almost sorted array in advance.

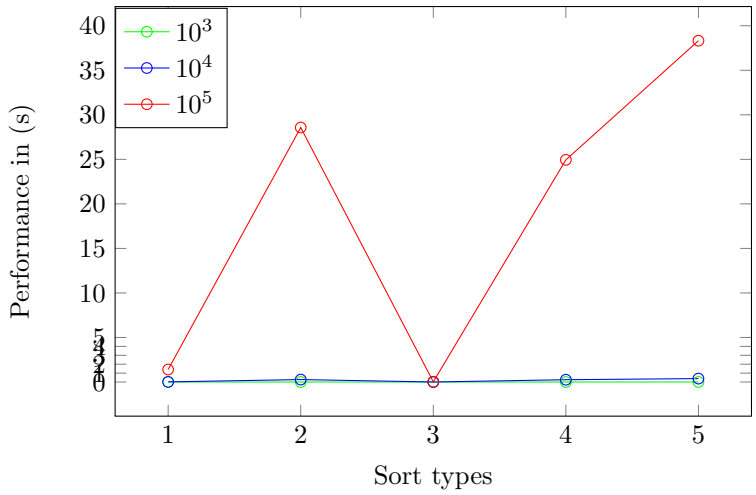
4 Conclusion

This graph shows the overall measurements we observed for each algorithm. Where :

- 1:InsertionSort
- 2:QuickSort
- 3:HeapSort
- 4:PlaceSort
- 5:RecSort



Graph of **Decreasing** type arrays



Graph of **Random** type arrays