Lab N°02 : Sorting Algorithms

Exercise II.1 (Initialization)

Q1) Write a function $\boldsymbol{manual\_init(int\ Tab[], int\ n)}$ that manually initializes the elements of the $\boldsymbol{Tab}$ array of $\boldsymbol{n}$ integers with integer values (User enters the array values).

Q2) Write a function $\boldsymbol{auto\_init(int\ Tab[], int\ n)}$ that generates $\boldsymbol{Tab}$ array elements of $\boldsymbol{n}$ random integers in the range of $\mathbf{1}$ to $\mathbf{100.000}$.

N.B.: ***rand()*** can be found in **#include <cstdlib>**.

Q3) Write a function $\boldsymbol{display(int\ Tab[], int\ n)}$ that displays the elements of the $\boldsymbol{Tab}$ array of $\boldsymbol{n}$ integers.

Q4) Write a function $\boldsymbol{is\_sort(int\ Tab[], int\ n)}$ that returns **True** or **False** depending on whether the $\boldsymbol{Tab}$ array of $\boldsymbol{n}$ integers is sorted in ***ascending*** order or not.

Exercise II.2 (Bubble Sort)

Q1) Write a function $\boldsymbol{bubble\_sort(int\ Tab[], int\ n)}$ sorting the array $\boldsymbol{Tab}$ using the conventional Bubble sort algorithm presented in the lecture. Then use the $\mathbf{is\_sort(\dots)}$ (Exercise II.1) function to check that the implementation is correct, by randomly generating arrays of numbers.

Q2) One of the disadvantages of the Bubble sort algorithm is that if the array is sorted before the end of the assumed number of loops, the algorithm continues to execute the remaining loops. This extra work reduces the performance of the Bubble sort algorithm.

Write a function $\boldsymbol{enhanced\_bubble\_sort(int\ Tab[], int\ n)}$ to enhance the Bubble Sort algorithm that can prevent the algorithm from doing this extra work.

Q3) Give the time complexity for both the conventional and enhanced versions of Bubble Sort.

Q4) Investigate the impact of diverse input datasets on the performance of Bubble Sort. How does Bubble Sort perform with datasets that are already sorted, reverse sorted, and containing many duplicate elements?

Exercise II.3 (Selection Sort)

Q1) Write a function $\boldsymbol{selection\_sort(int\ Tab[], int\ n)}$ sorting the array $\boldsymbol{Tab}$ using the fundamental Selection sort algorithm presented in the lecture. Then use the $\mathbf{is\_sort(\dots)}$ (Exercise II.1) function to check that the implementation is correct, by randomly generating arrays of numbers.

Q2) One of the disadvantages of the Selection sort algorithm is that the smallest element obtained in the inner loop must always be swapped with the first element.

Write a function $\boldsymbol{improved\_selection\_sort(int\ Tab[], int\ n)}$ to improve the Selection Sort algorithm.

Q3) Conduct a time complexity analysis for both the basic and advanced versions of Selection Sort.

Q4) Discuss why Selection Sort might not be the best choice for large datasets. What specific properties of Selection Sort lead to this?

Exercise II.4 (Insertion Sort)

Q1) Write a function $\boldsymbol{insertion\_sort(int\ Tab[], int\ n)}$ sorting the array $\boldsymbol{Tab}$ using the traditional Insertion sort algorithm presented in the lecture. Then use the $\mathbf{is\_sort(...)}$ (Exercise II.1) function to check that the implementation is correct, by randomly generating arrays of numbers.

Q2) Conduct a time complexity analysis for the basic Insertion Sort.

Q3) Explore the scenarios where Insertion Sort might outperform more complex sorting algorithms. What are the characteristics of such datasets?

Exercise II.5 (Quicksort)

Quick sort is a recursive sort whose idea is as follows:

- We start by choosing any element of the array (called the pivot), e.g., the last one (the first or a randomly chosen element).
- We go through the array once, separating elements below the pivot from those above it. This provides an array divided into three parts: a first sub-array of elements less than or equal to the pivot, a second sub-array containing only the pivot, and a third sub-array containing all elements strictly greater than the pivot.
- We repeat the same operation on the first and third of these sub-tables, until they are of length $\mathbf{1}$.

Q1) Write a function $\boldsymbol{quick\_sort(int\ Tab[], int\ left, int\ right)}$ sorting the array $\boldsymbol{Tab}$ using the basic Quicksort algorithm presented in the lecture (The pivot is chosen as the last element). Then use the $\mathbf{is\_sort(...)}$ (Exercise II.1) function to check that the implementation is correct, by randomly generating arrays of numbers.

Q2) Rewrite the function $\boldsymbol{quick\_sort(int\ Tab[], int\ left, int\ right)}$ from the previous question, choosing the pivot as the first element.

Q3) Rewrite the function $\boldsymbol{quick\_sort(int\ Tab[], int\ left, int\ right)}$ from the previous question, choosing the pivot as the random element.

Q4) Conduct a time complexity analysis for the basic Quicksort.

Q5) Discuss the impact of pivot selection on the performance of Quicksort. How does choosing a random pivot improve the algorithm?

Exercise II.6 (Merge Sort)

The Merge Sort consists of recursively performing the following operation:

- Cut the data to be sorted into two roughly equal parts
- Sort the data in each part
- Merge the two parts

The recursion stops at some point, as we end up with lists of 1 element, at which point sorting is trivial.

Q1) Write a function $\boldsymbol{Merge\_sort(int\ Tab[], int\ left, int\ right)}$ sorting the array $\boldsymbol{Tab}$ using the basic Merge sort algorithm presented in the lecture. Then use the $\mathbf{is\_sort(...)}$ (Exercise II.1) function to check that the implementation is correct, by randomly generating arrays of numbers.

Q2) Conduct a time complexity analysis for the basic Merge sort.

Q3) Explain why Merge Sort is considered a stable sorting algorithm. What feature of Merge Sort ensures its stability?

Exercise II.7 (Merge Sort)

We'll try to measure the performance of the following five (5) sorting algorithms on the same set of values: Bubble Sort, Selection Sort, Insertion Sort, Quicksort, and Merge Sort. Write a program that:

Q1) Initializes five (5) identical arrays of $N = 10$ values using the $rand()$ function;

Q2) Sort each array using one of the algorithms implemented above;

Q3) Measures the execution time of each of these algorithms using the $clock()$ function;

This function returns the processor time used since the start of the current program.

So, to measure the execution time of one of the algorithms, call this function just before and just after it, then proceed by differences;

```
int main()
	{
		clock_t T1, T2; double time;

		T1 = clock();		// take the clock before the block
		{ Instruction block }
		T2 = clock();		// takes clock after function call

		// calculates execution time in seconds
		time = (double)(T2 - T1) / (double)CLOCKS_PER_SEC;
		cout << "execution time is: " << time<< "second(s) "<<endl;
	}
```

Q4) Fill in the table by running the five programs with different values of N.

	Execution times (Runtimes)				
N	**Bubble Sort**	**Selection Sort**	**Insertion Sort**	**Quicksort**	**Merge Sort**
10					
100					
1000					
10000					
100000					
1000000					
10000000					

Q5) Compare the five (5) algorithms (to do this, represent the graphs of the five (5) algorithms in the same figure). Which of the five (5) algorithms is better (or more efficient)? Justify your answer.