Comparing Algorithms

S1.02

1 Introduction

The goal of this project is to understand and compare different sorting algorithms. You will work individually and you should produce a **pdf report** explaining your findings, that you must hand in together with all the **source code** (each will be **graded separately**). Everything must be coded and executed under linux.

In the sorting problem, we are given a vector `V` of `n` comparable elements and the goal is to produce another vector (possibly in the same memory location) such that `V[i]` $\leq$ `V[i+1]` for $0 \leq \texttt{i} \leq n-2$.

2 Methodology

You will compare (i) several different sorting algorithms and (ii) three different types of vectors (iii) of several different sizes. The vectors contain elements that of type `int` between 0 and $1024n$, where n is the size of the vector. The three types of vector are the following.

1. All elements are independent uniformly distributed random `int` values between 0 and $1024n$.
2. The first $n/2$ elements satisfy $\texttt{v[i]} = 1024 \cdot \texttt{i} + r$, where r is a random value between 0 and 1023, chosen independently for each element. The remaining $n/2$ elements are random numbers as before.
3. The first $n/2$ elements satisfy $\texttt{v[i]} = 1024 \cdot (\texttt{n} - \texttt{i}) + r$, where r is a random value between 0 and 1023, chosen independently for each element. The remaining $n/2$ elements are random numbers as before.

The number of elements n goes from 1000 to 100000 elements. You will measure the running time $t(n)$ of the sorting algorithm for an input of size n and produce two graphs with n on the x-axis and $t(n)$ on the y-axis (using *sagemath*). The graphs should be as follows:

- Each sorting **algorithm** will have a **different color**, and a **legend** will tell which is which.
- The first type of input should be drawn as **solid line**, the second as a **dashed line**, and the third as a **dotted line**.
- **Two graphs** should be produced, by changing the scale of the y-axis. In the first one, we should be able to compare the slow $O(n^2)$ running times, and in the other one we should be able to compare the fast $O(n \log n)$ running times. Clearly distinguish both cases in your report.

You must use the `g++` compiler with the `-Ofast` compiler option on all tests. A single bash script called `build.sh` should:

1. Compile all your code.
2. Run all tests.
3. Create all plots.

You may abort the execution if the running time of one algorithm goes over 10 minutes. (The linux command `timeout` may be useful for that.) If an algorithm crashes in some situation, that should be included in the report. Notice that your C++ code can produce a data file for *sagemath* (possibly by redirecting the standard output to a file). You can write your sagemath plotting code in a file called sagemath script `myplot.sage`, that will create all plots. Then, you can run your code with `sage myplot.sage`.

An example script is presented below. In this case, the program takes two command line arguments: the size of the vector to sort and the content of the vector ((1) all random, (2) half sorted, or (3) half reversed).

```
#!/usr/bin/bash

g++ -Wall -Ofast -c main.cpp

for alg in stdsort qsort selection insertion quicksortdet quicksortrnd
do
  echo Algorithm: $alg
  g++ -Wall -Ofast -c $alg.cpp
  g++ -Wall -Ofast -o $alg main.o $alg.o
  for t in 1 2 3
  do
    rm $alg$t.data
    for i in ‘seq 1 100‘
    do
      n=${i}000
      echo $alg $n $t
      timeout 10m ./$alg $n $t >> $alg$t.data
      ret=$?
      if [[ $ret -gt 120 ]]
      then
        echo TIMEOUT OR CORE DUMP ON $alg $t
        break
      fi
    done
  done
done

echo Creating plots
sage plot.sage
```

The pdf report should explain each sorting algorithm **with your own words**, show the plots, and compare the algorithms. I expect around 4 pages of text (not including the plots). Feel free to write the report in English or in French (language mistakes will not be graded). Both pdf and code should be sent as a `.tar.gz` file. No other format is accepted (renaming a different file as `.tar.gz` is not acceptable).

3 Sorting Algorithms

You should compare **at least six** out of several different algorithms. You should choose at least one out of:

1. The one available as `std::sort`.
2. The one available as `std::stable_sort`.
3. The one available as `qsort` in `cstdlib`.

Choose at least one out of:

4. A selection sort you will code yourself.
5. A bubble sort you will code yourself.

Choose at least one out of:

6. An insertion sort you will code yourself.
7. The algorithm from the following paper that you will code yourself. `https://arxiv.org/abs/2110.01111`

Compare both:

8. The quicksort algorithm presented below.
9. The modified quicksort algorithm as described below.

The C++ source code below presents a very simple implementation of the quicksort algorithm. On line 8, we choose the pivot to be the first element of the input vector. An important variation that you should implement is to modify that line in order to use a random element of the input vector as the pivot.

```
 1 #include <vector>
 2 #include <iostream>
 3
 4 void quickSort(std::vector<int> &v) {
 5     if(v.size() >= 1) {
 6       int pivot = v[0];
 7       std::vector<int> v1, v2, v3;
 8       for(int x : v) {
 9           if(x < pivot)
10               v1.push_back(x);
11           else if (x > pivot)
12               v3.push_back(x);
13           else // x == pivot
14               v2.push_back(x);
15       }
16
17       quickSort(v1);
18       quickSort(v3);
19
```

```cpp
      v = v1;
      v.insert(v.end(), v2.begin(), v2.end());
      v.insert(v.end(), v3.begin(), v3.end());
    }
}

int main() {
    std::vector<int> v = {5,3,1,2,7,6,7,5,1,3,4,0,9,8};

    quickSort(v);

    for(int x : v)
        std::cout << "␣" << x;

    std::cout << std::endl;
    return 0;
}
```

Sometimes the code may produce a stack overflow because of the recursion depth. This can be avoided with the compiler option `-fsplit-stack` that allows the stack to grow as needed. Identify in your report in which cases this option has been necessary.

4 Technical Remarks

4.1 Measuring Time

To measure time in microseconds you can use:

```cpp
#include <iostream>
#include <chrono>

int main() {
    std::chrono::steady_clock::time_point t_start =
        std::chrono::steady_clock::now();

    size_t sum = 0;
    for(int i = 1; i <= 10000; i++)
        sum += i;

    std::chrono::steady_clock::time_point t_end =
        std::chrono::steady_clock::now();
    size_t duration =
        std::chrono::duration_cast<std::chrono::microseconds>
        (t_end - t_start).count();

    std::cout << "The␣sum␣of␣1␣to␣10000␣is␣" << sum << std::endl;
    std::cout << "And␣it␣took␣" << duration <<
        "␣microseconds␣to␣calculate␣it" << std::endl;
    return 0;
}
```

4.2 Generating Random Numbers

To generate random numbers, you can use the old C random number generator `rand()` from `cstdlib`. The `rand()` function returns an `int` between `0` and `RAND_MAX`. A value between `0` and `n-1` may be obtained using the remainder `rand() % n`.

Another alternative is to use the much more complex (and accurate) C++ random number generator, as in the example below.

```cpp
#include <cstdint>
#include <iostream>
#include <random>

int main() {
  static std::random_device rd;  // To obtain a seed
  static std::mt19937 gen(rd()); // Standard generator
  int min = 0, max = 10;

  std::uniform_int_distribution<int> distrib(min, max);
  std::cout<<"Min:␣"<<min<<"␣Max:␣"<<max<<std::endl;
  for (size_t k=0; k<8; k++) {
    int n = distrib(gen); // Random int from 0 to 10
    std::cout << n << '␣';
  }
  std::cout<<std::endl;

  std::uniform_int_distribution<size_t> distrib2(min,max);
  for (size_t k=0; k<8; k++) {
    size_t n = distrib2(gen); // Random size_t from 0 to 10
    std::cout << n << '␣';
  }
  std::cout<<std::endl;

  return 0;
}
```

4.3 Organizing Your Code

Remember that copying and pasting your own code is bad, and will make it harder for you to correct the same problem over multiple parts of the code. Instead, you can create an `hpp` file with a sorting function with a unique signature that you define. In your code, you can convert the exact format whenever necessary, in order to call other sorting functions. Then you can link your code with the proper function. For example you can code your main benchmark code in a file name `main.cpp`. This code can include a file `sort.hpp` that you create with one function with a signature of your choice. Then, you can have selection sort in a file named `sort4.cpp`, and insertion sort in a file named `sort6.cpp`. To link with selection sort, you use:

```
g++ -Wall -Ofast -c main.cpp
g++ -Wall -Ofast -c sort4.cpp
g++ -Wall -Ofast -o main main.o sort4.o
```

Alternatively, to link with insertion sort, you use:

```
g++ -Wall -Ofast -c main.cpp
g++ -Wall -Ofast -c sort6.cpp
g++ -Wall -Ofast -o main main.o sort6.o
```

4.4 Plotting

To build graphic representation of data with *sagemath*, you can use `line(data)` or `point(data)`; `data` must be a list of coordinates such as
`data = [[0,0], [1,3], [2,6]]` or `data = [(0,0), (1, 3), (2,6)]`.

To display the graph, use the command `show()`. To save the graph, use the command `save()`, which exports to pdf and svg file formats, among others.

```
data = [[0,0], [1,3], [2,6]]
p = line(data)
show(p)  # or p.show()
save(p, 'test.pdf') # or p.save('test.pdf')
```

```
data = [[0,0], [.5, sqrt(3)/2], [1,0]]
q = point(data)
q.show()
```

You can also use list of x and y coordinates with the command `zip`.

```
x = [0, 0.5, 1]
y = [0, sqrt(3)/2, 0]
g = line(zip(x,y))+point(zip(x,y),rgbcolor= 'red')
g.show()
```

Note that addition allows us to superimpose several graphic objects.

Last example :

```
data = [[6*cos(pi*i/100)+5*cos((6/2)*pi*i/100),
          6*sin(pi*i/100)-5*sin((6/2)*pi*i/100)]
         for i in range(200)]
p = line(data, rgbcolor=(1/8,1/4,1/2))
q = point(data, rgbcolor=(1/8,1/4,1/2))
t = text("hypotrochoid", (5,4), rgbcolor=(1,0,0))
show(p + q + t)
```

4.5 Other Sagemath Features

You can use most python code inside sagemath. For example to read a file with x and y coordinates separated by space and one line for each (x, y) pair, you can use:

```
def readFile(filename):
  v = []
  with open(filename) as f:
    for line in f:
      v.append([float(x) for x in line.split()])

  return v
```

All these topics about *sagemath* will be explored for four hours of S1.02 - R1.07 very soon.

4.6 Submission Format and Date

You must produce a pdf document with your findings and submit it together with your source code through Ametice in a `.tar.gz` file. The date is available on Ametice. Projects sent later will be penalized.

4.7 Academic Dishonesty

You are not allowed to copy any code from other students. You are not allowed to have anyone else doing this work for or with you. You are allowed to ask for explanations and use any references you want, but the code and report must be written by you alone without copying, except for the pieces of code given herein. Every reference used must be indicated in your report. You are not allowed to use any AI to generate text or code for this assignment.

Violation will be punished severely and French law establishes sanctions that may include exclusion from every public education institution for a period of time or permanently.