UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
School of Computer Science & Applied Mathematics

Introduction to Algorithms & Programming
Brute-force sorting

1 Introduction

In this lab, we'll be implementing a few of the sorting procedures we saw in the lecture notes. The main aim here is to measure the running time (wall-clock time) of each sort method empirically as the size of the input increases. We know that in theory, selection sort and bubblesort have $O(n^2)$ time complexity, but what does this mean in practice? Let's find out!

Please make sure to read exactly what needs to be uploaded to Moodle to be marked!

2 A strategy for measuring running times

To measure the time a function takes to execute, we need a way of recording the time just before the function was called and the time just after the function has ended. For input of size N, is this sufficient? No! It's not enough, because these measurements are noisy! Your computer is busy doing a million other things at the same time, so if a function takes 500 nanoseconds to complete, it most likely will take a different amount of time if you ran it again! Therefore, we need to average our results over multiple runs to reduce the noise as much as possible. Pseudocode to do this might look like this:

```
total_time = 0
n_runs = 5
for i in range(0, n_runs):
    start_time = get_time()
    f() # run function
    end_time = get_time()
    time_elapsed = end_time - start_time
    total_time += time_elapsed
average_time = total_time / n_runs
return average_time
```

The above executes a function 5 times and records the average time it took to run! We could increase the number of runs beyond this to reduce the variance caused by noise in our measurements, but for this lab, 5 should be sufficient.

Side note: compiled languages like C and C++ are good candidates for this kind of measurement, since most of the optimisation of the code happens at compile time. In contrast, languages like Java employ just-in-time optimisation, which means that code

actually gets faster the longer the program runs! This would be a massive problem if we're trying to make precise measurements, and so special care should be taken when using such languages.

The above will produce a single averaged measurement of how long the function took to execute. However, we want to observe the effect as we vary the size of the input. We therefore need to use an outer loop that generates input of various sizes, and then use the pseudocode above to record the time taken for each of these inputs. Overall pseudocode would look like this:

```
times = list()
n_runs = 5
for N in range(start_size, end_size, step_size):
    total_time = 0
    for i in range(0, n_runs):
        start_time = get_time()
        f(N) # run function with input size N
        end_time = get_time()
        time_elapsed = end_time - start_time
        total_time += time_elapsed
    average_time = total_time / n_runs
    times.append(average_time)
return times
```

This will return a list of times that vary with a change in input sizes (N). We can then plot these results to understand what's going on!

3 Random numbers in C++

Since C++11, random numbers are generated using objects in the `<random>` header file. There are two objects of interest: the *generator*, which actually generates the numbers, and the *distribution*, which determines the distribution that numbers should be drawn from (e.g. Gaussian, uniform, Poisson, etc). The code below creates a generator and uniform distribution of integers from 1 to 10, then uses it to generate a random number according to that distribution:

```
    random_device rd;
    std::mt19937 generator(rd());
    std::uniform_int_distribution<> dist(1, 10);
    int number = dist(generator); // a random number
```

4 Timing in C++

Since C++11, timing in C++ can be done using the STL library and the `<chrono>` header file. The code below shows how to get the current time, and how to compute the difference between two times in nanoseconds:

```cpp
auto start = chrono::high_resolution_clock::now();
// some code to time here
auto finish = chrono::high_resolution_clock::now();
long long timeTaken = chrono::duration_cast<chrono::nanoseconds>(finish - start).count();
```

5 Measuring sort algorithms

The aim of this lab is to measure the running times of sorting algorithms. In particular, we will be looking at selection sort, bubblesort, and bubblesort with the escape clause. Using the above information, your tasks for this lab are the following.

5.1 Generate input

Write a function that accepts an integer N, the size of the input, and returns a vector of size N, where each element is a random integer uniformly drawn from the range 1 to 1000000. Be sure to verify that the numbers are all completely random!

5.2 Sorting algorithms

Implement selection sort, bubblesort and bubblesort algorithm with escape clause as in the notes. It would be a good idea to verify that it works by testing it with some input and verifying that it does in fact sort the numbers. Each of these sorts should be written as their own separate function that accept a vector of integers and sort them.

5.3 Time the algorithms

Use the pseudocode provided to time how long each algorithm takes as a function of the input size. Timing should be measured in nanoseconds. The input size should start at $N = 1000$ and go up to (and including) $N = 20000$ in steps of 1000. Note that all algorithms should be given the same data. For example, for an vector of size 1000, all algorithms should be given as input that exact same vector (not different vectors of size 1000.) Average these results over 5 runs (the vectors between runs should differ, so the vector of size 1000 on run 1 should be different from the same sized vector on run 2).

5.4 Save the data

Save the results in a text file as a comma separated list. The text file should therefore contain 3 lines (one for each algorithm) and each line should contain 20 values, separated by commas. You may save the results manually, or use C++ to write them to file (the latter is obviously preferred).

5.5 Plot the data

Download the Python file on Moodle and modify it by providing the relevant information. Run this Python file to produce a plot of the running times of the three algorithms which will be displayed and saved to file (you will need to have matplotlib installed). Verify that the running times are in fact quadratic (with potentially different constants/lower-order terms). If they are not, you'll need to fix the mistake somewhere in your code!

5.6 Upload to Moodle

Once you are happy and satisfied with everything, upload the following to Moodle. **Each of these files will be marked, so missing files will mean missing marks! Please upload everything!**

1. A single C++ source file containing all the code you used to generate the timing data. Name this file `sorts.cpp`.
2. The text file where the timing data was saved. Name this file `data.txt`.
3. The Python file used to plot the graphs. Name this file `plot.py`.
4. The PDF file with the actual plots. Name this file `sorting.pdf`.
5. A text file with short answers to the following questions:
   (a) Which sort had the best performance empirically?
   (b) Why might this be the case? Provide a reason.

   Name this file `analysis.txt`.