PSet: Profiling

Table of Contents

Purpose of Assignment

This project seeks to verify empirically the accuracy of those analysis's by measuring performance of each algorithm under specific conditions. Performance measurement or program profiling provides detailed empirical data on algorithm performance at different levels of granularity and measures.

"Profiling" measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Let us use the elapsed times printed by program execution even though it may not be as accurate as special profiling tools. With small input data size, all times will likely be 0.0000 because the clock interval is too large to measure the execution times. In that case, you should try to get sufficiently accurate results with various data sets and/or extra lines of code repetitions. Our focus on this assignment is to compare the time complexity of three sorting algorithms.

Files provided

- profiling.pdf — this file
- profling.cpp — a complete code & a skeleton code for the extra point
- driver.exe, driver — use this to measure the elapse time of sorting for 1 million samples

Step 1. Read profiling.cpp

Read the program **profiling.cpp** and be familiar with how it works.

There are two ways to start the program, profiling.exe. Users may start it by the executable file. Then the program must prompt the user to enter "the number of the maximum sample numbers to sort". If the number entered is less than **STARTING_SAMPLES** (a magic number stored in sort.h), quit the program but with a proper message.

```
// sort.h
const int STARTING_SAMPLES = 1000;
```

Users may give the number of samples in the command line argument. We must check whether or not it is larger than **STARTING_SAMPLES**, exit the program if it is not with a proper message.

Step 2. Build and run profiling.cpp program

- **How to build:** Using your own **`libsort.a`** and **`rand.a`** as you made in the previous lab or use those in ../../lib
  `$ g++ profiling.cpp -I../../include -L../../lib –lsort -lrand –o profiling,`

- **How to run:**
  ```
  $ ./profiling.exe 30000       # PowerShell
  $  profiling 30000            # cmd
  ```

- **How to save the output into a file:**
  ```
  $  ./profiling.exe 50000 > profiling.txt
  $  profiling 50000 > profiing.txt
  ```

- **How to increase the stack size**
  The quicksort with already sorted input samples may not finish completely since it requires **a lot of stack memory**. In this case, you must increase stack since it is only 1megabyte by default. The following command increase the stack size to 16 megabytes.

  By the way, these compiler option does **not** work in the Windows PowerShell nor Atom console, you must change PowerShell to **cmd windows** before run the command.

```
$ cmd
$ g++ -Wl,--stack,16777216 profiling.cpp -I../../include -L../../lib –lsort –lrand –o profiling
$ ./profiling
```

Also **mac users** may look it up more in googling since its syntax may be different a bit.
You may try like this (16M bytes): **`-Wl,-stack_size,0x1000000`**

Mac Users: (Unless you have your own static libraries, you may use `-lsort_mac and –lrand_mac`)

```
$ g++ -Wl,-stack_size -Wl,1000000 profiling.cpp -I../../include -L../../lib –lsort -lrand  -o
profiling

$ ./profiling
```

Step 3. Compare the time complexity

The code, profiling.cpp, are already invoking sorting functions as we need. Now we would like to compare the elapsed time of following cases for three sorting algorithms: **insertionsort, mergesort** and **quicksort** cases, respectively:

1. **Sorted**
2. **Randomly ordered**
3. **Reversed**

Sample Run: (The following output is partially shown to save spaces.)

```
$ ./profiling 10000
The minimum number of entries is set to 1000
Enter the number of max entries to sort: 10000
The maximum sample data size is 10000

        insertionSort(): sorted
          N       repetitions       sort(sec)
       1000            219992        0.000005
       2000            121951        0.000008
       3000             90212        0.000011
       4000             70272        0.000014
       5000             56832        0.000018
       6000             48603        0.000021
       7000             40910        0.000024
       8000             35896        0.000028
       9000             32081        0.000031
      10000             28353        0.000035

        insertionSort(): randomized
          N       repetitions       sort(sec)
       1000              1299        0.000770
       2000               351        0.002855
       3000               152        0.006579
       4000                90        0.011144
       5000                69        0.014565
       6000                44        0.023295
       7000                35        0.028571
       8000                28        0.036643
       9000                21        0.047810
      10000                16        0.063062

        insertionSort(): reversed
          N       repetitions       sort(sec)
       1000               551        0.001815
       2000               202        0.004960
       3000                92        0.010967
       4000                46        0.021761
       5000                33        0.030485
       6000                23        0.043783
       7000                18        0.057889
       8000                13        0.077769
       9000                11        0.095909
      10000                 9        0.118778

         quickSort(): randomized
          N       repetitions       sort(sec)
       1000              1054        0.000949
       2000               586        0.001706
       3000               361        0.002773
       4000               242        0.004145
       5000               210        0.004781
       6000               172        0.005814
       7000               146        0.006897
       8000               120        0.008350
       9000               107        0.009346
      10000               101        0.009941
```

Step 4. Compute the growth rate from the time measured

We safely assume that most algorithms approximately have the order of growth of the running time:

$T(N) \approx a\ N^b$

We want to compute the **constant "a"** and the **growth rate "b"** from data we got from profiling. Once we get a and b, we can compute T(N) where N is 1 billion or 1 million without running the profiling since it may take years or more.

To predict running times, multiply the last observed running time by 2^b and double N, continuing as long as desired. Let us compute b using the double ratio.

Since $T(N) \approx a\ N^b$, $T(2N) = a\ (2N)^b$, then

$$\frac{T(2N)}{T(N)} = \frac{a(2N)^b}{aN^b} = \frac{2^b(N)^b}{N^b} = 2^b$$

Log both sides

$$log\frac{T(2N)}{T(N)} = \log 2^b$$

$$b = log\frac{T(2N)}{T(N)}$$

In this example, let us choose N = 4000 or 2N = 8000, an average case of the insertion sort shown above. Recall that log we use here is **log base 2.**

$$b = log\frac{T(2N)}{T(N)} = log\frac{t2(8000)}{t1(4000)} = log\frac{0.036643}{0.011144} = 1.717$$

For b in the table below, you must show how you get your answer. You may use a calculator and compute up to two digits after the decimal separator. In my case, I have got 1.717 for the average case of insertionsort. It will be close to 2.0 for the worst case of insertionsort, 1.2 ~ 1.5 for the average case quicksort.

Now, we use this b = 1.717 to solve for a when $N = 4000,\ T(N) = 0.011144$ in the following:

$$T(N) = a\ N^{1.717}$$

$$0.011144 = a\ (4000)^{1.717}$$

$$a = \frac{0.011144}{(4000)^{1.717}}$$
$$a = 7.27\text{x}10^{-9}$$

Therefore, we have the growth rate **b = 1.717, the constant** $a = 7.27\text{x}10^{-9}$ for the insertion sort average case.

Step 5. The time complexity for a million samples

In this step, the question we want to answer **is "How long will my program take, as function of the input size?"** To help answer this question, we plot data with the problem size N on the x-axis and the running time T(N) on the y-axis. Let's suppose we have the running time, as a function of the input size,

$$\mathrm{T(N)} = a\,N^{b},$$

where **a** is a constant and **b** is a growth rate. Even though we already computed a = 7.27 x 10^{-9} and b = 1.717 in the previous step, let's compute the constant **a** again when N = 8000 as shown below,

$$\mathrm{T(8000)} = 0.036643 = \mathrm{a}\ 8000^{\,1.717}$$

$$\mathrm{a} = \frac{0.036643}{8000^{\,1.717}} = 7.269\,x\,10^{-9}$$

$$\mathrm{T(N)} = 7.27\,\mathrm{x}10^{-9}\ \mathrm{N}^{\,1.717}$$

In conclusion, we confirmed that two constant a's the same. Now you can estimate the elapsed time for one million samples or billion samples as well. With this a and b, we can compute an estimated time for T(N) for N = 1,000,000 or 1 billion samples without running the program, right?

Compute the estimated time for one million samples based your computation of **the grow rate b and the constant a** in your machine. In your report, show your exact steps how you compute the estimated time. Use the constant "a" that you computed with more samples. **Use a proper time unit. For example, don't say 1,234.57 sec., but 20 min 35 sec.**

Also fill the blanks with the elapsed times both estimated and measured in your computer while running them with 1 million samples. **Use driver.exe for this purpose.**

	$\mathrm{T(N)} \approx a\,N^{b}$, **a =** **b =** insertionsort – Best		$\mathrm{T(N)} \approx a\,N^{b}$, **a =** **b =** insertionsort – Average		$\mathrm{T(N)} \approx a\,N^{b}$, **a =** **b =** insertionsort – Worst	
N	10,000	Time for Million	10,000	Time for Million	10,000	Time for Million
Time		Estimated:		Estimated:		Estimated:
N	20,000		20,000		20,000	
Time		Measured:		Measured:		Measured:

	$\mathrm{T(N)} \approx a\,N^{b}$, **a =** **b =** Average quicksort **O(N log N): randomized**	
N	10,000	Time for Million
Time		Estimated:
N	20,000	
Time		Measured:

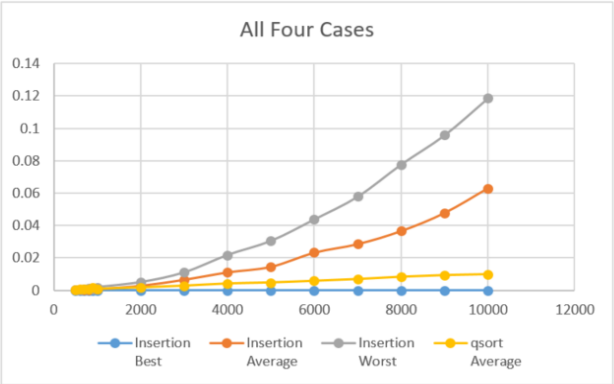
	$T(N) \approx a\, N^b$, a = b = Average mergesort **O(N log N): randomized**	
N	10,000	Time for Million
Time		Estimated:
N	20,000	
Time		Measured:

Step 6. Plot the time complexity to compare

Plot the data sets that you got from the previous step to compare them graphically as shown below. You may use Excel Chart(분산형) to plot them. An output example combined data from `insertionsort`, and `quicksort` for plotting and report.

In your report, you must include average `mergesort` case in the following chart.

n	Insertion Best	Insertion Average	Insertion Worst	qsort Average
500	0.000003	0.000180	0.000341	0.000380
600	0.000003	0.000256	0.000446	0.000657
700	0.000004	0.000374	0.000667	0.000652
800	0.000004	0.000477	0.001075	0.000716
900	0.000004	0.000623	0.000990	0.001653
1000	0.000005	0.000770	0.001815	0.000949
2000	0.000008	0.002855	0.004960	0.001706
3000	0.000011	0.006579	0.010967	0.002773
4000	0.000014	0.011144	0.021761	0.004145
5000	0.000018	0.014565	0.030485	0.004781
6000	0.000021	0.023295	0.043783	0.005814
7000	0.000024	0.028571	0.057889	0.006897
8000	0.000028	0.036643	0.077769	0.008350
9000	0.000031	0.047810	0.095909	0.009346
10000	0.000035	0.063062	0.118778	0.009941



Step 7. Applying DRY in profiling.cpp

Copy **`profiling.cpp`** into **`profile.cpp`**. Submit **`profile.cpp`** once your implement this step. The final output of **`profile.cpp`** should be the same as that of **`profiling.cpp.`**

There is a set of code that repeated three times in profile.cpp. It violates the rule of our in-house coding principle, **DRY** – Do not repeat yourself. As you see, only differences in three sections of code are sorting function and printing statement. We do not want to repeat the same code many times when we want to test more sorting functions in future. I n conclusion, we want to put those sorting functions and its description in a string array. For example, we may put the sorting description as shown below:

```
vector<string> sort_st = {"insertionsort", "mergesort", "quicksort"};
```

How about the sorting functions?

You already know about the function pointer and `insertionsort` may be defined using a function pointer `sort_fp` as shown below:

```
void (*sort_fp)(int*, int, bool (*comp)(int, int)) = insertionsort;
```

Now you just need to know how to declare an array of function pointers. It is shown below:

```
void (*sort_fp[])(int*, int, bool (*comp)(int, int));
```

By now, you know how to initialize this array with three sorting algorithms as usual.

By now, you know how to code the rest `profiling.cpp` without violating **DRY** and **NMN** like 3.

Submitting your solution

- **On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.**
  Signed: _____ Section: _____ Student Number: _____
- Make sure your code **compiles** and **runs** right before you submit it. Don't make "a tiny last-minute change" and assume your code still compiles. You will not receive sympathy for code that "almost" works.
- If you only manage to work out the Project problem partially before the deadline, you still need to turn it in. However, don't turn it in if it does not compile and run.
- Place your source files in the folder you and I are sharing.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again as long as it is **before the deadline**. You will have to resubmit any related files together, even if you only change one. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit

Submit the following files on Piazza PSet folder.

- **ProfilingReport.docx** or one in a readable format: your repost must include the followings:
  1. Screen capture of profiling.exe output.
  2. Complete the performance analysis tables with your data – estimated and measured. Show your computation steps for a, b and estimated time for all **five cases** (worst, average and best cases for `insertionsort`, and two average cases for `mergesort` and `quicksort`).
  3. Include the excel chart and graph for comparing those **five cases.**
  4. **Describe your observation** of the time complexity of the five sorting algorithm cases.
- **profile.cpp**
  1. Both DRY and NMN applied

Due and Grade

- Due: 11:55 pm