PSet: Profiling – Performance Analysis

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.

"Program 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 two sorting algorithms.

Files provided

- profiling.pdf – this file
- profling.cpp – a skeleton code Step 1 and 2
- **profilingx.exe** – for pc, a solution for Step 1and 2
- **profiling – for mac**
- **sortDriver3.cpp – a skeleton code for Step 4**
- sortx.exe – for pc, a solution for Step 4
- sortx - for mac

Step 1. Handling user's input and complete getStep()

Read and run the program **profiling.cpp** provided in this pset.

Task 1: User's input

Run **profilingx.exe** and be familiar with how it works.

There are two ways to start the program, profilingx.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 = 500;
```

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. Run sortx.exe to see the error message.

Once you get the max sample size N, you must allocate the memory accordingly and deallocate after use.

NOTICE: To make profiling.cpp simple, **don't use** nowic library functions such as GetInt(), and GetChar() and so on.

Task 2: getStep()

Currently the step increases in linear scale such as 100, 200, … , 1000, 1100, 1200. In order to measure the performance, the step size should be incremented by 100 between 100 and 1000; the step size will be 1,000 between 1000 and 10,000; From 10,000 to 100,000, the step size will be 10,000 and so on. Rewrite getStep() function accordingly.

Implement **getStep()** function such that it returns **1** for **[0..9]**, **10** for **[10..99]**, **100** for **[100..999]**, **1000 for [1000..9999]**, and so on. The variable step defined in the program would be many different values, depending on the number of samples. The sample sizes could reach up to **billions.**

You should **not** code something like below:

```
// this is the way of coding.
if (n == 100) step = 100;
if (n == 1000) step = 1000;
if (n == 10000) step = 10000;
……
……
…
```

Step 2. Build and run executables

The skeleton code, profiling.cpp, are already invoking sorting functions as we need. Now we would like to compare the elapsed time of following cases:

1. **insertionSort()**
   A. Best case – O(n), Input data is already sorted
   B. Average case - O(n^2), Input data is randomly ordered
   C. Worst case - O(n^2), Input data is reversely ordered
2. **quickSort()**
   A. Average case - O(n log n), Input data is randomly ordered

Sample run:

```
$ ./profiling 10000
The minimum number of entries is set to 500
Enter the number of max entries to sort: 10000
The maximum sample data size is 10000

        insertionSort(): already sorted - best case.
        Data will NOT be randomized before use.
         n       repetitions       sort(sec)
       500            351798        0.000003
       600            318514        0.000003
       700            282230        0.000004
       800            250027        0.000004
       900            236931        0.000004
      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 - average case.
        Randomized Data will be used during sorting.
         n       repetitions       sort(sec)
       500              5543        0.000180
       600              3908        0.000256
       700              2674        0.000374
       800              2097        0.000477
       900              1604        0.000623
      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(): sorted reversed - worst case.
      Data will NOT be randomized before use.
        n        repetitions       sort(sec)
      500               2934        0.000341
      600               2241        0.000446
      700               1500        0.000667
      800                930        0.001075
      900               1010        0.000990
     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 - average case.
      Randomized Data will be used during sorting.
        n        repetitions       sort(sec)
      500               2631        0.000380
      600               1522        0.000657
      700               1533        0.000652
      800               1396        0.000716
      900                605        0.001653
     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
```

- You are going to use these files to draw a graph to show the **growth rate** of the algorithm as the sample size n increases and compare them in Step 2.
- Make sure that you have the appropriate function calls before you redirect the output. You may need to recompile after you switch the sort function.

Tips and Hints

Read the skeleton code provided and follow the instruction properly.

The quicksort really runs worst if the input data is already sorted. To test the worst-case quicksort, you must pass a sorted data. For other cases, you just pass the randomized data.

- **How to compile:** Using your own **libsort.a** as you made it in **lab6** before.

```
$ g++ profiling.cpp printList.cpp -I../../include -L../../lib -lsort -o profiling,
```

- **How to run:**
```
$ ./profiling 100000        # PowerShell
$  profiling 100000         # cmd
```

- **How to save the output into a file:**
```
$  ./profiling 100000 > profiling.txt
$  profiling 100000 > profiing.txt
```

- **How to increase the stack size**
The worst-case quicksort 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, you must change PowerShell to cmd windows before run the command.

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

Step 3 – Compute the time complexity of best/average/worst cases

We would like to do the performance analysis with our programs and data. 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.

Let's suppose that you can get the growth rate b in the following step. Then we want to estimate the elapsed time of one samples. We can use one of our data points to solve for a – for example,

$$\mathrm{T(8000)} = 0.036643 = \mathrm{a}\ 8000^{\,1.98}$$

$$\mathrm{a} = \frac{0.036643}{8000^{\,1.98}} = 6.85\ x\ 10^{-10}$$

$$\mathrm{T(N)} = 6.85\ x\ 10^{-10}\ \mathrm{x}\ \mathrm{N}^{\,1.98}$$

With this equation, you can estimate the elapsed time for one million samples or billion samples as well.

__Compute__ the **growth rate** __b__ of the running time as a function of n using the output (profiling.txt) you got from Step 2 and fill the following table for their comparison. Assume that the running time obeys a power law $\mathrm{T(n)} \approx a\, n^{b}$. Based on the elapsed time between n = 4,000 and n = 8,000 shown in the table below, **compute** the actual constant **b**.

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. It should be close to 2.0 for the insertionSort, 1.2 ~ 1.5 for the average case quickSort.

Estimate the elapsed time for one million samples based your computation of b in your machine. Fill the blank in the table below. Show your exact steps how you compute the estimated time using the **growth rate b** and write it in your report. Use a proper time unit. For example, don't say 1,234.57 sec., but 20 min 35 sec.

Fill blanks in the table below with the elapsed time actually measured in your computer while running them with 1 million samples.

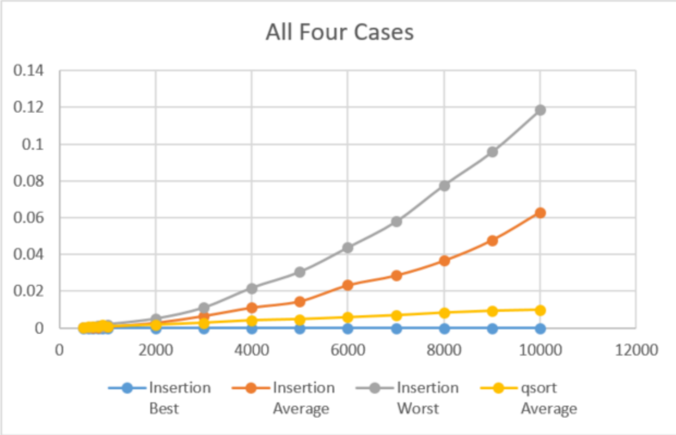
Hints and Tips: Refer to **DiscreteMath.pdf** provided for this computation.

	$\mathrm{T(n)} \approx a\, n^{b}$, ($a$ =1, **b=**) insertionSort – Best		$\mathrm{T(n)} \approx a\, n^{b}$, ($a$ =1, **b =**) insertionSort – Average		$\mathrm{T(n)} \approx a\, n^{b}$, ($a$ =1, **b =**) insertionSort – Worst	
N	4,000	Time for Million	4,000	Time for Million	4,000	Time for Million
Time		Estimated:		Estimated:		Estimated:
N	8,000		8,000		8,000	
Time		Measured:		Measured:		Measured:

	$\mathrm{T(n)} \approx a\, n^{b}$, ($a$ =1, **b =**) Average qsort O**(N log N)**	
N	4,000	Time for Million
Time		Estimated:
N	8,000	
Time		Measured:

Plot the data sets that you got from Step 2 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.

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 4 – Be ready for all "sorts" of profiling

In this step, let the user have many choices of sort algorithm to run the profiling of sorti with other options that you implemented in PSet2. For this purpose, add two options **p** amd **o** in **sortDriver3.cpp** and call profiling() defined in **profiling.cpp.**

1. Add "p" option which invokes profiling() in profiling.cpp with a sort algorithm chosen.

   Add the function proto-type at the top of sortDriver3.cpp as needed.

   When you invoke it, you have to pass the function pointer as an argument.

   If the number of samples are less than `STARTING_SAMPLES` , print the error message such that the user changes the number of samples much larger than `STARTING_SAMPLES` .

2. Add an option **"v"** which sets **the list as sorted list** in ascending or descending order. It toggles. If the current list is in ascending order, set it in descending order and vice versa. This option enables us to test sorting algorithms with a reverse ordered form of data sets.

3. Before compiling, you must comment out the main() part in **profiling.cpp** by setting #if 0 just above main() since we are using main() in sortDriver3.cpp.

   Use your own **libsort.a** as you made it lab6 before, use the following command to build sort.exe.

   ```
   g++ sortDriver3.cpp, profiling.cpp printList.cpp -I../../include -L../../lib -lnowic -lsort -o sort
   ```

4. You may check your implementation with **sortx.exe** provided.

Step 5 – QuickSort and Selection Sort

Now, we would like to use this new menu-driven profiling program developed in Step 4 and apply for selectionSort().

Task 1: Run profiling for insertionSort and quickSort using this sort.exe and make sure that it produces the same results by profiling.exe.

Task 2: Do profiling for the following three cases of selectionSort().

- Case 1. Input data is already sorted
- Case 2. Input data is randomly ordered
- Case 3. Input data is reversely ordered

	$\mathrm{T(n)} \approx a\, n^{b}$, ($a$ =1, b=) selectionSort – Case 1		$\mathrm{T(n)} \approx a\, n^{b}$, ($a$ =1, b =) selectionSort – Case 2		$\mathrm{T(n)} \approx a\, n^{b}$, ($a$ =1, b =) selectionSort – Case 3	
N	4,000	Time for Million	4,000	Time for Million	4,000	Time for Million
Time		Estimated:		Estimated:		Estimated:
N	8,000		8,000		8,000	
Time		Measured:		Measured:		Measured:

Task 3. Compare the results with insertionSort() and write about your findings.

Task 4. We have run quickSort() with only randomized data set (so-called average case). Run it with different kind of data sets such as sorted and reversed. Observe the results and write what you found in the report.

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

- profiling.cpp, sortDriver3.cpp
- report.docx: at least 4 pages long report includes the followings:

- Screen capture of profiling.exe output
- Complete The performance analysis table
- The excel chart and graph for comparing best/average/worst cases
- Comparison and analysis of algorithms: For example, Insertion vs quick sort, Timing, Stack problem, best/average/worst cases analysis

Due and Grade points

- Due: 11:55 pm, Oct 4, 2019
- 5 points