The following materials have been collected from the numerous sources such as Stanford CS106 and Harvard CS50 including my own and my students over the years of teaching and experiences of programming. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Please send any comments or criticisms to idebtor@gmail.com. Your assistances and comments will be appreciated.

Project Set 5: 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

- Pset05Profile.pdf – this file
- **profiling_s.exe, profiling_q.exe** – produced when you run **selectionSort.exe** and **quicksort.exe**, respectively, in Step 2.
- profiling.xlsx – intermediate file you produce in Step 2 and Step 3 to write a report.
- sortx.exe – Final executable

Step 1. Implementing getStep()

Read and run the program **profiling1.cpp** provided in pset. Currently the step increases in linear scale such as 100, 200, … , 1000, 1100, 1200. In order to measure the performance, this step 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. Assume that you have a magic number STARTING_SAMPLES = 100.

const int STARTING_SAMPLES = 100;

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;
……
……
```

Hint: getStep() returns **1** for **[0..9]**, **10** for **[10..99]**, **100** for **[100..999]** and so on.

Step 2. Build and run executables

Now we would like to compare the elapsed time of two sort algorithms, **selectionSort** and **quicksort** which have **O(n^2) and O(n log n)** , respectively. Get the elapsed times of the selectionSort and quicksort shown below. Use the **profiling1.cpp** program for both selectionSort() and quicksort(). To use two sorting algorithms, simply replace the function call when necessary and recompile the program so that you can build two executables.

Sample runs of profiling_s.exe for selectionSort()

```
lg14z970\user c:/users/user/DropBox/nowic/src> profiling_s 20000
Getting the number of max entries from the command line...
The maximum sample data size is 20000.
         n         repetitions        sort(sec)
       100               50700         0.000020
       200               13975         0.000072
       300                6385         0.000157
       400                3195         0.000313
       500                2677         0.000374
       600                1915         0.000522
       700                1418         0.000705
       800                1083         0.000923
       900                 880         0.001136
      1000                 708         0.001412
      2000                 153         0.006542
      3000                  68         0.014838
      4000                  46         0.022435
      5000                  28         0.036929
      6000                  19         0.055211
      7000                  15         0.069467
      8000                  12         0.085750
      9000                  10         0.107700
     10000                   8         0.131500
     20000                   2         0.526000
```

Sample runs of profiling_q.exe for quicksort()

```
lg14z970\user c:/users/user/DropBox/nowic/src> profiling_q 20000
Getting the number of max entries from the command line...
The maximum sample data size is 20000.
         n        repetitions       sort(sec)
       100             111270        0.000009
       200              58042        0.000017
       300              34960        0.000029
       400              24053        0.000042
       500              17040        0.000059
       600              13997        0.000071
       700              11182        0.000090
 . . .
      3000               1939        0.000516
      4000               1446        0.000692
      5000               1177        0.000850
      6000                928        0.001078
      7000                800        0.001250
      8000                720        0.001389
      9000                622        0.001608
     10000                553        0.001808
     20000                261        0.003843
```

- 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 Problem 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.

Hint

- **How to compile:**
  ```
  g++ profiling1.cpp selection.cpp quicksort.cpp -o profiling_s  #using selectionSort()
  g++ profiling1.cpp selection.cpp quicksort.cpp -o profiling_q  #using quicksort()
  ```
- **How to run:** (selectioSort case example)
  ```
  ./profiling_s 30000
  ```
- **How to save the output into a file:** (selectionSort case example)
  ```
  ./profiling_s 30000 > profiling_s.txt
  ```

Step 3 – Compute O(n^2) and O(n log n)

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

1. Using profiling_s.txt for selectionSort() and profiling_q.txt for quicksort(), respectively, **compute** the order of growth of the running time as a function of n using the output you got from Step 2 and fill the following table for their comparison.

   We can assume that the running time obeys a power law $T(n) \approx a\, n^{b}$. Based on the elapsed time between n = 10,000 and n = 20,000 shown in the table below, **compute** the actual constant **b**.

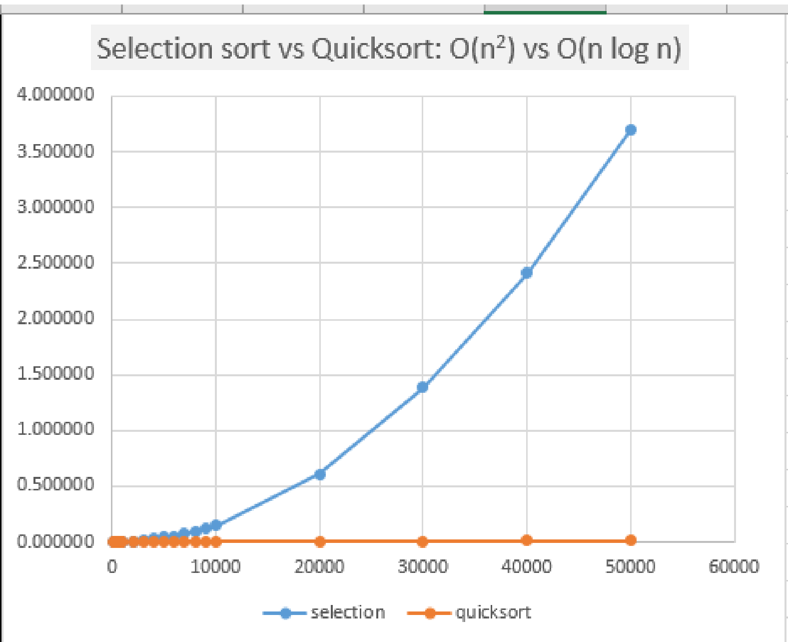
   You must show how you get your answer. You may use a calculator and compute up to two digits after the decimal separator. It should be close to 2.0 for the selection sort and 1.2 ~ 1.4 for the quicksort. Use the time units such as second, days, and years when you fill the white blanks in the table.

	$T(n) \approx a\,n^b$, (a=1.0, b =) Selection sort O(N^2)			$T(n) \approx a\,n^b$, (a=1.0, b =) Quick sort O(**N log N)**		
N	10,000	Million	Billion	10,000	Million	Billion
My computer						
	x - known	100^b * x				

2. 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 Selection sort and Quicksort for plotting and report.

n	selection	quicksort
100	0.000025	0.000009
200	0.000067	0.000019
300	0.000144	0.000031
400	0.000250	0.000059
500	0.000371	0.000072
600	0.000529	0.000087
700	0.000778	0.000106
800	0.001013	0.000128
900	0.001202	0.000150
1000	0.001520	0.000152
2000	0.005929	0.000378
3000	0.013453	0.000569
4000	0.025643	0.000742
5000	0.041280	0.000907
6000	0.051950	0.001290
7000	0.075643	0.001458
8000	0.092818	0.001531
9000	0.115889	0.001810
10000	0.150571	0.002134
20000	0.605000	0.003929
30000	1.383000	0.006711
40000	2.414000	0.009202
50000	3.690000	0.010600



Step 4 – Be ready for all "sorts" of profiling

In Step 4, we want to make most job of **main()** in profiling1.cpp into **sortProfiling()** function. Implement this function in a new file called profiling2.cpp using profiing1.cpp.

1. Copy profiling1.c into profiling2.cpp
2. Implement sortProfiling() function in profiling2.cpp.

- Most contents of the main() goes to sortProfiling() function and its proto-type would be as shown below:

```
void sortProfiling(void (*sortFunc)(int *, int), int *list, int n,
                   int starting_samples = STARTING_SAMPLES) {

   double duration;

   cout << "          n\t repetitions\t   sort(sec)\n";
   for (int i = starting_samples; i <= n; i += getStep(i)) {
      …
      …
```

- Since most contents of the main move to sortProfiling() function, the main() contains only the following:
  - accept or set the number of samples N
  - allocate list by N
  - invoke sortProfiling() as shown below:
  - sortProfiling(selectionSort, list, N, starting_samples);
    or
    sortProfiling(quickSort, list, N, starting_samples);
  - free list

3. Using sortProfiling(), make sure that profiling2.cpp works exactly like profiling1.cpp.

Step 5 – Passing a function pointer as a parameter

In Step 5, let the user have many choices of sort algorithm to run the profiling of sorting along with other options that you implemented in PSet2. You add an option **p** in sortDriver3.cpp which is copied from sortDriver2.cpp and call sortProfiling() defined in profiling2.cpp.

1. Copy sortDriver2.cpp in Pset2 into sortDriver3.cpp.
2. Add "profiling" menu item "p" as shown below:

```
C:\WINDOWS\system32\cmd.exe
Enter a number of samples to sort: 10000

       SORTING OPTIONS:   [Selection Sort, size=10000, randomized=Y]
       a - algorithm to run
       n - n sample size and initialize
       r - randomize(shuffle) input samples
       p - profiling
       s - sort()
       d - display input or output
       Command(q to quit):
```

3. Add the function proto-type at the top of sortDriver3.cpp and define the magic number `STARTING_SAMPLES`.

```
const int STARTING_SAMPLES = 100;
void sortProfiling(void (*sortFunc)(int *, int), int *list, int n,
int starting_samples = STARTING_SAMPLES);
```

Implement the option **p** and invokes sortProfiling() with a sort algorithm chosen. 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.

4. Compile with all files necessary. You must comment out the main() part in **profiling2.cpp** by setting #if 0 just above main() since we are using main() in sortDriver3.cpp.

   If you successfully made **libsort.a** before, use the following command to build sort.exe.

   ```
   g++ sortDriver3.cpp, profiling2.cpp -I../../include -L../../lib -lnowic -lsort -o sort
   ```

   If you have not made **libsort.a** before, use the following command to build sort.exe.

   ```
   g++ sortDriver3.cpp, profiling2.cpp insertion.cpp, selection.cpp, quicksort.cpp,
   bubble.cpp -I../../include -L../../lib -lnowic -o sort
   ```

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

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

- Step 1, 4, 5: profiling1.cpp, profiling2.cpp, sortDriver3.cpp
- Step 2, 3: 2~3 pages long report – report.doc, profiling_s.txt, profiling_q.txt
- **Don't submit** selection.cpp, quicksort.cpp etc.

Due and Grade points

- Due: 11:55 pm, April 7, 2019
- 3 points