CS 1XC3 Lab 5 - Benchmarking and Profiling

Feb. 27^{th} - Mar. 3^{rd}, 2023

Contents

Introduction: Benchmarking and Profiling

In introductory programming courses, we focus on the correctness of our code, trying to ensure that programs produce the expected outputs for given inputs. This is only one way in which programs may be evaluated.

In this lab you will be comparing different programs that produce the same outputs using different algorithms. You will use different techniques to see which algorithms are more efficient, in terms of time and memory usage.

Setup

1. Download "lab5.tar.xz" from the course content on Avenue. This archive contains the C source code files we will use in this lab.

2. Create a new directory in your submission repository, "L05".

3. Depending on whether you're doing this lab on your local machine or on the Pascal server, you may be confronted with the problem of transferring files to and from the server. If not, please pay attention to this section anyways, as it has important information.

   The `ssh` (Secure SHell) protocol is for remote login, but doesn't allow us to move files, just commands. To move files, we need `scp` (Secure CoPy).

   ```
   scp username@from_host:file.txt /local/directory/
   # Moving file.txt from remote to local
   $ scp file.txt username@to_host:/remote/directory/
   # Moving file.txt from local to remote
   ```

   In general, this works just like `cp`, but one of the paths includes the user name and server address. Note that the `-r` flag allows you to copy directories and their contents recursively.

   If you're using a GUI interface like PuTTY from a Windows-like system, you may consider using something like WinSCP (`https://winscp.net/eng/index.php`).

   No matter how you accomplish it, copy the compressed archive into your newly created directory in your CS1XC3 repo.

4. Now unpack this archive using the `tar` command. If you are unsure of how to do so, try taking a look at the man page, or stack exchange!

1 Activity: Timing Searching Algorithms [5 points]

First, we will benchmark 3 searching algorithms using `time`. The C source code file "search.c" contains 3 searching algorithms:

- `linearSearch` (mode 0)
- `binarySearch` (mode 1)
- `fibonacciSearch` (mode 2)

1. Compile "search.c". The program is used as follows:

```
$ ./search 0 <number to search for>
# Run search using linear search algorithm
$ ./search 1 <number to search for>
# Run search using binary search algorithm
$ ./search 2 <number to search for>\
# Run search using fibonacci search algorithm
```

   The "search.c" file contains a huge array `arr` of integers containing close to 100,000 entries. The array is sorted in ascending order. All 3 functions in "search.c" return the index at which the first occurrence of the searched value is found. If the search fails and the integer is not found, -1 is returned.

2. Take a look at the code in the provided program. For a description of how these algorithms work, take a look at the following:

   - `https://en.wikipedia.org/wiki/Linear_search`
   - `https://en.wikipedia.org/wiki/Binary_search_algorithm`
   - `https://en.wikipedia.org/wiki/Fibonacci_search_technique`

   You will need some knowledge of how these algorithms work in order to analyse them using the collected data.

3. Run the executable using the time tool, using a search algorithm and search value of your choice.

```
time ./search {0|1|2} <search value>
```

   You'll get a report that looks like the following:

```
Linear Search index: 10510

real  0m0.005s
user  0m0.001s
sys 0m0.003s
```

   - The time delta between when the start and end of the program is reported as `real`.
   - `user` reports the time spent in user-mode code (your code plus libraries)
   - `sys` reports the time spent in system-mode code (kernel mode and sys calls)

4. Run tests as specified by the table below. Keep track of this information in a new comma-separated value file, "timing_results.csv". You only need to record the reported `real` times, we don't need to care about time spent in user vs kernel mode for these tests.

   - All this repetative work should make you think "I wonder if I could write a script to do this..."

Search Value	Linear Search	Binary Search	Fibonacci Search	Notes
1				
4				
22				
37				
22906				
53757				
112591				
361940				
475713				
893766				
996637				
996639				
996652				
-996652				

5. In the Notes column, indicate which search algorithm is fastest for each search value, and why. If there doesn't appear to be a difference between two or more algorithms, explain why you think that is.

6. Commit your CSV file to your git repo in your "L05" directory.

2 Estimating Memory Usage [5 points]

The `time` tool we used in the previous section can be used for more than just timing programs. The GNU version will report a number of different statistics, including "Maximum resident set size", which is the amount of memory allocated to your program by the operating system.

Bash has it's own version of `time` which it defaults to, so in order to use the GNU version we have to invoke it using it's absolute path in the file system: `/usr/bin/time`. To see the extended statistics, we need to run it in **verbose** mode (`-v`). Many programs have a verbose mode, in which normally hidden status updates and information is provided to the user.

We can run the GNU version of `time` in verbose mode using:

```
/usr/bin/time -v ./search 0|1|2 <search value>
```

Repeat the trials in Part 1 of this lab, and save the results in a new file, "memory_results.csv". This time, record the maximum resident set size instead of real time.

- If you wrote a script in Part 1 to run your trials, you should be able to modify it very quickly to run the current set of trials!

Search Value	Linear Search	Binary Search	Fibonacci Search	Notes
1				
4				
22				
37				
22906				
53757				
112591				
361940				
475713				
893766				
996637				
996639				
996652				
-996652				

In the Notes section, include your observations and why you think the sizes may vary for each of the test cases with the different algorithms. Also, take note of where lower runtime and lower memory cost don't match up.

Don't forget to commit the "memory_results.csv" file to your git repo as well! If you don't commit it, you don't get marked for it!

3 Profiling Factorization Algorithms [5 points]

In this section, you are provided with two algorithms which find the factors of a specific number. The algorithms first calculate all the prime factors of a positive integer greater than 1, and then calculate all groups of factors (that is, all sets of numbers which have a product equal to the target number). The `factor_itr and factor_rec` functions, which solve this problem recursively and iteratively, can be found in "factors.c".

The usage for the "factors.c" program is similar to the search program from the previous sections.

- `factor_rec` (mode 0)
- `factor_itr` (mode 1)

You are tasked with using the `gprof` tool discussed in lecture to analyze the two algorithms provided.

1. Compile the factors.c program using `-pg` flag. This adds code to our C program which records data about the program as it's being run.

```
gcc -pg -o factors factors.c
```

2. We will be running `gprof` for the following numbers:
   - 40, 79, 240, 300, 999, and 4000

   For each of the above, and for both algorithms, run the following:

```
$ ./factors <mode> <number>
$ gprof > factors_<mode>_<number>.txt
```

   For example, your first trial will be

```
$ ./factors 0 4
$ gprof > factors_0_4.txt
```

   The > operator redirects a command's output to a file.

   You will have a collection of 12 files at the end of this process. Please commit them to your repo.

3. Create another file, named "part3.txt", and provide answers to the following questions:
   - Why are all the reported times zero?
   - Why is factors_itr only called once every time the program is run?
   - Is there an obvious mathematical relationship between the input number and the number of recursive calls? What about the number of prime factors each number produces?

4. Commit your "part3.txt" file.

4 Code Coverage Using `gcov` [5 points]

For the last section of this lab you will be using `gcov` to analyse the source code in "factors.c" even further. The `gcov` tool creates code coverage information by annotating your source code. The number of times a line is executed is reported alongside the code itself. Follow the upcoming steps to produce the `gcov` reports in order to help you answer the questions below.

1. Compile "factors.c" using the following command to allow for the use of `gcov` later on:

```
gcc -fprofile-arcs -ftest-coverage -o factors factors.c
```

2. Execute the factors executable (an example for using iterative algorithm to find factors of 4):

```
./factors 1 4
```

3. Run the `gcov` tool:

```
gcov factors.c
```

4. Rename the resulting file, "factors.c.gov" to "factors_1_4.gov"

5. Repeat the above process, so that we have results for the input nubmers 4, 10, 17, and 256, in both recursive and iterative mode, renaming the final file using the naming convention in part 3.

6. Answer the following questions, and save your answers to "part4.txt".

   (a) How are the iterative and recursive algorithms similar and different for the same test cases? Which has more repetition?

   (b) What is the most executed line in the iterative and recursive algorithms? Are they the same?

   (c) Are there any parts of the code which are not exercised?

   (d) If we were to try to optimize either function, where would be a good place to start?

7. Commit all of your "*.gov" files, as well as your "part4.txt", to complete the lab!