ADVANCED OPERATING SYSTEMS — PROJECT 1
Measuring the performance of sorting algorithms

1 Overview

For this project, our primary goal is to design, implement, and execute a complete experiment. So that your efforts are focused on the experiments themselves, the topic of evaluation will be a set of algorithms that you know and understand well: sorting algorithms.

While asymptotic analysis suggests that a $O(n\ lg\ n)$ algorithm such as *mergesort* is preferrable to a $0(n^2)$ algorithm like *bubble sort*, and that a $O(n)$ non-comparison-based algorithm such as *radix sort* would be better still. However, *better* has a specific meaning for this type of analysis: the time required to complete the computation grows more slowly as the input size increases. This notion of performance comparison between algorithms is powerful and useful, but it fails to consider some important factors.

First, asymptotic analysis is performed on an *algorithm*—an abstract sequence of steps—and not on the actual *implementation* of that algorithm as manifested by a particular program or collection of procedures. Most algorithms can be implemented in many different ways, relying on different data structures, written in different languages, and translated into different underlying assembly and machine code. Those differences can affect performance.

Second, analysis at the algorithmic level assumes a constant amount of time required to perform all operations. For example, analysis of sorting algorithms counts only the comparisons of the elements being sorted. It does not account for time to perform arithmetic used to traverse the elements. More importantly, it does not account for the variable amount of time that different operations require—comparing two numbers does not necessarily require the same time as copying a value from one location to another. Additionally, complex memory hierarchy behavior can make the access of some elements (and other data structures) take highly varying amounts of time.

Finally, it is sometimes not enough to know that, as the input size (n) grows arbitrarily large that one algorithm will require fewer operations (and thus complete its work in less time) than another. For intermediate values of n, it is unclear which algorithm is best.

For this project, you will implement sorting algorithms from all three of the "interesting" asymptotic classes: $O(n^2)$, $O(n\ lg\ n)$, and $O(n)$. You will then measure, in practice, the performance of these sorting algorithms on a variety of inputs. You will then analyze the results and attempt to draw conclusions about which sorting algorithm and implementation to use for different values of n. Finally, and perhaps most importantly, you will find that the measurement and data analysis steps are more messy a task than it first appears.

2 The sorting algorithms

From each of the three asymptotic categories of sorting algorithms, you should choose one. Here are some examples (although you may choose other algorithms not listed here, so long as they belong to one of these categories):

- $O(n^2)$: Bubble sort; insertion sort; selection sort.

- $O(n\ lg\ n)$: Mergesort; heapsort.
- $O(n)$: Radix sort; counting sort.

Coding: For the three sorting algorithms you choose, you must **implement each in C**. Specifically, each such algorithm should be implemented in a separate C program. On the CS department systems, you will find a file that you should copy that implements bubble sort as an example:

```
~sfkaplan/public/cs39/bubble.c
```

In this file, you will find code that can read a sequence of integers from the keyboard (henceforth known as *stdin*), and emit those same integers in sorted order to the terminal (henceforth *stdout*). This code reads the sequence of integers until there are no more, and then it calls the following procedure:

```
void sort (int* array, int size)
```

This procedure sorts all `size` elements of `array`. In this example, I have implemented bubble sort in that procedure to perform the sorting. For each of your chosen sorting algorithms, you should make a copy of this file and change the `sort()` procedure, implementing a different algorithm.

Compiling: When you first write your program and need to debug it, use the following command to compile it[1]:

```
$ gcc -ggdb -o bubble bubble.c
```

This compilation produces an executable image suitable for use with the `gdb` source level debugger, whose use is described below. Once you have debugged your program more thoroughly, you can take advantage of the compiler's optimization capabilities. Specifically, `gcc` is capable of varying levels of optimization—the longer you are willing for compilation to take, the more optimized it tries to make your code. It's `-O`*n* flag allows you to specify an *optimization level* from $n = 0$ to $n = 4$. For example, if you wish to compile your code at optimization level 2, then the following command will do:

```
$ gcc -O2 -o bubble bubble.c
```

Note also that, if you are using `emacs` or `xemacs` as your editor, it can also do useful things to control and process your compilation results. Specifically, instead of issuing the compilation command within a shell, you can do so within (x)emacs using the following command:

[1] Of course, you should replace `bubble` and `bubble.c` with the name of your executable image and source code files, respectively.

```
M-x compile
```

Entering this command will cause (x)emacs to issue the prompt:

```
Compile command:
```

There may be some default command inserted after this prompt (something like: `make -k`); erase that default. You can then enter the same `gcc` commands shown above, and (x)emacs will carry it out. Moreover, the results of the compilation will be shown in a separate buffer within (x)emacs. If there are errors and/or warnings in your source code, (x)emacs will allow you to step through them, one at a time, with the key combination: `C-x ` ` (that's a reverse accent, also known as a *backquote*). With this key combination, (x)emacs will move through each error in your compilation, bringing you to the line of source code to which it is attributed in the other half of the window.

Running and debugging: These programs are designed to read their input—a list of whitespace-separated integers, represented as decimal text—from *stdin*. Similarly, these programs will emit their output—a sorted list of whitespace-separated, decimal text integers—to *stdout*. This type of interface, where a small program reads text from *stdin* and writes text to *stdout*, follows the useful *UNIX design philosophy* that recommends that programs be simple and perform a single task well. We will see the usefulness of that design approach in Section 3.

To run one of these sorting programs in the most basic way, you can enter the command to run it at a shell prompt. You can then enter the list of numbers to sort by hand, completing the list by typing `Control-D`. You will then see (if your program works correctly) the same list of numbers printed by the program, but this time in sorted order. For example:

```
$ ./bubble
8
4
2
19
<Control-D>
2
4
8
19
```

This result is a minor verification that your program works correctly. However, it is an inefficient and inconvenient way to perform more extensive tests and experimental measurements. We would like to be able to read a pre-made list of integers from a file, and to write the sorted list to another file. To do so, we can use *shell redirection*, where we run the program in such a way that *stdin* draws its input from a file instead of the keyboard, and *stdout* emits its output to a file instead

of the terminal. For example, the first command shown here will read a list of numbers from the file `unsorted-numbers.txt`, and write it to the file `sorted-numbers.txt`; the second command will allow you to examine the output file, one screen at a time:

```
$ ./bubble < unsorted-numbers.txt > sorted-numbers.txt
$ less sorted-numbers.txt
```

You may find, with great sadness, that your program does not execute correctly. Perhaps the output is incorrect; perhaps your program never seems to complete; or, you may see an incomplete output ending with the dreaded **Segmentation fault**. How, with a C program, do you perform debugging? One traditional approach whose value is not to be underestimated is the insertion of output statements at carefully chosen points in the program, thus generating debugging output that helps you to understand the inner details of your program's execution.

However, a *source-level debugger* can be a more helpful tool to find your bugs. In particular, `gdb` operates on C programs compiled with `gcc`. In particular, if you compile using the `-ggdb` option, as shown above in Section 2, then `gdb` can do a great deal to help you track program execution and find errors. Moreover, you can run `gdb` within (x)emacs, which can in turn load and highlight the line of code on which the debugger is currently operating.

If you want to debug your sorting program using `gdb` from within (x)emacs, enter this (x)emacs command:

```
M-x gdb
```

Once entered, (x)emacs will prompt you to enter a command to launch `gdb` on some specific program. It will provide a prompt and then a `gdb` command that lacks only the name of an executable image. You should provide that pathname to start `gdb` running within (x)emacs, like so:

```
Run gdb (like this): gdb --annotate=3 bubble
```

I recommend doing a web search for a **`gdb` tutorial**; you should also avail yourself of the standard `gdb` documentation. The following commands will help you to get started with `gdb`:

- `run`: Issue this command to start the program running. It will run either to completion or until an error causes the debugger to stop the program.
- `break`: Set a *breakpoint*—a location in the code at which execution will stop when it is reached. A procedure may be named, or a specific line number in the source code may be used. For example:

  ```
  (gdb) break main
  (gdb) break bubble.c:90
  ```

The first example will cause execution to stop any time `main()` is called—in this case, when the program first begins execution. The second example will stop execution when the it reaches line 90 of the source code file `bubble.c`.

- `next`: Execute the next line of source code. Any calls to procedures on that line will be performed without interruption (unless a breakpoint or error within those proceudres causes the debugger to take control).

- `step`: Execute the next line of source code, but jump **into** any procedure calls on that line. That is, unlike `next`, a procedure will be called and, before any of its lines of code are executed, control will be given back to the debugger, allowing you to move through the code in the procedure itself.

- `print` <*variable name*>: Print the value of the given variable. Notice that the variable must in scope for the current line of source code for the variable to be accessible with this command. Also note that this command is flexible in that it can print elements of arrays (that is, if `x` is a pointer to an array of integers, you can request that `x[3]` be printed).

- `print/x` <*variable name*>: Like the `print` command above, but the value shown is printed in hexidecimal. Often valuable when examining pointer values.

- `backtrace`: Examine the call stack. The current sequence of procedure calls that brought the program's execution to its current moment are shown, including the values or arguments passed. Each activation frame is numbered, with the current frame at the top of the stack as frame 0, the procedure that called the current one as frame 1, and so on down to the initial call to `main()` at the bottom of the stack.

- `frame` <*frame number*>: Move into the scope of the given frame number, where the `backtrace` command lists the available frame numbers. This command allows you to move into the scope of the caller of the current procedure and examine its variable values; you can then move to the scope of the caller of that procedure, and so on.

When using `gdb` within (x)emacs, the editor will split the window and show you the source code line on which the debugger is operating. When frame numbers are changed, the editor likewise changes the source code associated with that frame. Being able to see the code that is current for the debugger is helpful in determining which variables and frames to examine.

Verifying correctness: Your program may seem to run correctly on small inputs, and it may not crash on larger inputs, but in the absence of some type of proof of correctness, how can you be more certain that your implementation is correct? Timing results on a program that fails to correctly perform the sorting task are meaningless. Thus, we need some way to automatically verify the output of these sorting programs.

The aforementioned UNIX design philosophy is helpful here. UNIX(-like) systems come with a standard utility, `sort`. (Ironic, no?) When called with the `-n -C` options, it will determine whether its input is sorted. Specifically, every program, when completed, sets a *result status* to indicate whether it completed its work correctly and without error. Typically, a status of 0 indicates, success, while any other status value implies failure of one type or another. In the case of the

`sort -n -C` command, if we examine the `status` environment variable immediately after its execution, a 0 will indicate that the input was sorted, while a status of 1 indicates that it was not sorted. For example:

```
$ ./bubble < unsorted.txt > sorted.txt
$ sort -n -C < unsorted.txt
$ echo $status
1
$ sort -n -C < sorted.txt
$ echo $status
0
```

When much larger inputs are used in the actual experiments described in Section 3, this approach to verifying correctness is essential. No person can practically check a 1 million element list of integers to ensure that they are in sorted order.

3 Experimental design

Now that you have written and debugged your three sorting programs, how do you test their performance? The goal is to determine how these programs behave on inputs of varying lengths and inputs. Moreover, we want our empirical evaluation to be carefully designed to account for variation in running times. You will perform two sets of experiments, one using a simple, low-resolution timer, and another using a more complex, high-resolution timing mechanism.

3.1 Inputs and their generation

In order to meaningfully test your sorting programs, you must provide inputs that may yield differentiable behavior. Thus, you need programs that can synthetically generate lists of integers to be sorted. For example, you should create a program that generates a list of k randomly selected integers whose values range from 0 to m, and emits those integers as text. If we wanted $k = 5$ and $m = 100$, then we would invoke the program like this:

```
$ ./generate_random 5 100
70
6
44
92
23
```

Once this program is created, you can either redirect its output into a file that can later be used as an unsorted input file to a sorting program. Alternatively, the list of numbers need not be stored in a file at all. Instead, it can be *piped* from the number generating program straight into a sorting program, like this:

```
$ ./generate_random 5 100 | ./bubble
6
23
44
70
92
```

With this construct, the absolute value sign (`|`) is the *pipe*, which is really a FIFO queue that connects the *stdout* of the first process to the *stdin* of the second process. We can also extend the use of the piping concept to not only feed the random values into the sorting program, but also to feed the output of the sorting program into the verifier:

```
$ ./generate_random 5 100 | ./bubble | sort -n -C
$ echo $status
0
```

This type of use of the verifier ensures that you are always measuring the results of a correct program.

Note that you should test a number of different patterns of inputs to the sorting program:

- **Random:** The above, if you write the `generate_random` program, will allow you to test the sorting programs on lists of random values.
- **Pre-sorted:** The performance of some algorithms and implementations may vary dramatically, particularly from their asymptotic worst-cases, if the input is already sorted. You can use the `sort` utility to pre-sort the randomly generated input before it reaches your sorting program:

  ```
  $ ./generate_random 5 100 | sort -n | ./bubble | sort -n -C
  ```

- **Pre-reverse-sorted:** Again, performance may varying dramatically if the input to a sorting program is in reverse-sorted order. Again, the standard `sort` utility can do this work for us:

  ```
  $ ./generate_random 5 100 | sort -n -r | ./bubble | sort -n -C
  ```

- **Patterned:** You may imagine other patterns for the inputs to the sorting programs that may yield interesting results, at least on one particular sorting program. Write utilities that can create these patterns for you.

In addition to generating this variety of input patterns, you need to test a range of input lengths by varying k. The value of m matters only if you use counting sort as one of your algorithms, in which case you should vary that value as well.

3.2 Timing runs

There are many ways to time how long a program or a portion of code takes to run. Each has its limitations and problems, and so we do not expect to obtain identical results from each. We will use two different timing mechanisms, and later we will need to reconcile their results.

3.2.1 The jiffy timer

There is a simple utility that allows one to measure the time required to execute a process. This utility simply querries the operating system kernel for information about how the process's time was spent. The kernel, for its part, maintains data about how much time each process spends running, broken down into a number of categories:

- **Real time:** The amount of time that passes between the beginning of the process's execution and its termination, irrespective of the processing time spent on other processes, the time spend waiting on disk accesses, etc.
- **Virtual user time:** The time that the CPU spends running your process's user-level (that is, non-kernel) code. Time spent in the kernel, time spent running other processes, and time spent waiting on input from various devices (disk, network, keyboard, etc.) are not included.
- **Virtual system time:** The time that the CPU spends running kernel code on behalf of your process. That is, if your process performs a system call (for example, to `open()` or `read()` from a file), then the time spent running the kernel code for those requests is attributed to this virtual system time.

A number of other statistics are also maintained by this utility, but for now, we will ignore them. This timer has a resolution of $\frac{1}{100}^{th}$ of a second, also known as a *jiffy*. This resolution is chosen because it is the frequency of a typical clock interrupt used to drive pre-emptive multitasking. It is, of course, a coarse resolution for measuring computation, since tens of millions of instruction cycles may pass within one jiffy.

To use this timer, do the following:

```
$ /usr/bin/time ./bubble < unsorted.txt > sorted.txt
1.50user 0.23system 0:03.46elapsed 50%CPU (0avgtext+0avgdata 0maxresident
0inputs+0outputs (0major+286minor)pagefaults 0swaps
```

Here the *user* portion is the virtual user time, the *system* portion is the virtual system time, the *elapsed* is the real time, and the *50%* is the portion of the elapsed time spent on this process (which should be the same as the sum of the *user* and *system* time).

Your challenge will be to figure out how to use this timer to obtain meaningful results about your sorting programs. You may need to average over multiple runs, and you may need to use larger inputs, both in order to work within the coarseness of the timing granularity.

3.2.2 The zen timer

A second timer requires adding code to your programs. Make copies of the following two files on the CS department systems:

```
~sfkaplan/public/cs39/timer.h
~sfkaplan/public/cs39/test-timer.c
```

This timer uses the *cycle counter* to obtain its timing information. That is, it indicates the number of processor clock cycles that have passed since the CPU was initiated. To see how it is used, examine the sample code in `test-timer.c`.

[SFHK: MORE COMING...]

4 Executing the experiments and collecting the data

[SFHK: COMING SOON.]

5 Analyzing the data

[SFHK: COMING SOON.]

6 Presentation and drawing conclusions

[SFHK: COMING SOON.]

This assignment is due on ???