1 Lab 9

Date: Apr 2, 2020

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

In the listings which follow, comments are any text extending from a `#` character to end-of-line.

1.1 Aims

The aim of this lab is to allow you to compare the performance of some common sorting algorithms. The lab will give you some exposure to the following topics:

- More shell scripting concepts.
- The use of gnuplot.
- The performance of common sorting algorithms.

1.2 Exercises

1.2.1 Starting up

Follow the *provided directions* for starting up this lab in a new git `lab9` branch and a new `submit/lab9` directory. Start a `script` session to log your interaction into a `lab9.LOG` file.

You will be doing all your work in your `submit/lab9` directory:

```
$ cd ~/i240?/submit/lab9
```

Copy over the exercises directory:

```
$ cp -r ~/cs240/labs/lab9/exercises .
```

The rest of this lab assumes that you are using `bash` as your shell.

1.3 Exercise 1

In this exercise, you will create data files which will be used in future exercises.

Linux distributions come with a seq command. Play with it:

```
$ seq 5
$ seq 2 11
$ seq 2 2 11
```

```
$ seq 11 -2 2
```

The shell also supports looping through consecutive "words" using a **for**-loop. Play with it:

```
$ for x in 22 33 44; do echo $x; done
$ for x in `seq 100 -20 0`; do echo $x; done
```

Recall from the earlier labs that the output of a command within backquotes (or `$(...)`) is expanded into the containing command. So the last command above is as though you had typed:

```
$ for x in 100 80 60 40 20 0; do echo $x; done
```

Another feature of bash is that it defines a shell variable `$RANDOM` which has a different random integer value each time it is accessed. Play with it:

```
$ echo $RANDOM
$ for x in `seq 50`; do echo -n " $RANDOM "; done
```

The range of values generated by `$RANDOM` is limited and the randomness is not supposed to be great, but it is good enough for generating random data.

Make sure you are in the `exercises` directory. Use the above features, to generate the following 3 data files in the `exercises` directory:

`ascending.dat` A file containing 100,000 whitespace separated integers ordered in ascending order.

`descending.dat` A file containing 100,000 whitespace separated integers ordered in descending order.

`random.dat` A file containing 100,000 whitespace separated random integers.

First try out the commands in the shell to generate integers for some small count like 10. Then use output file redirection to capture 100,000 integers in files having the above names. Look at the generated files in an editor to verify.

1.4 Exercise 2: Playing with Different Sorting Algorithms

Change over to the 2-sorts directory. It contains a file sorts.cc which implements 5 sorting algorithms; the implementations are along the lines of what was discussed in class. The algorithms are:

- `insertionSort()`.
- `bubbleSort()`.
- `selectionSort()`.
- `mergeSort()`.
- `quickSort()`.

All algorithms are set up to simply sort an `int`-array.

As explained in class, the first three algorithms above have $O(n^2)$ performance, whereas the last two have $O(n \lg n)$ performance.

IMPORTANT NOTE: To avoid overloading the system please do not use the $O(n^2)$ algorithms to sort more than 10,000 integers.

Compile the program using the `Makefile` which is in the parent directory. Run the program to get a usage message:

```
$ ./main
usage: ./main [-v] ALGORITHM INTS_DATA_FILE|- [N...]
$
```

The arguments are as follows:

`-v` If this option is specified, then the results of each sort is output as a line containing whitespace separated integers.

`ALGORITHM` This required argument must specify one of `insertionSort`, `bubbleSort`, `selectionSort`, `mergeSort` or `quickSort`.

`INTS_DATA_FILE` This required argument must specify a path to a file containing whitespace separated integers to be sorted. If specified as `-`, then integers are read from standard input.

`N...` A possibly empty sequence of positive integers. Specifies the number of integers from `INTS_DATA_FILE` to be sorted. This will be used in subsequent exercises to obtain the run sorts for different values of `N`. If this argument is not specified, then the entire contents of `INTS_DATA_FILE` will be sorted.

Examples:

```
$ seq 100 -8 0 | ./main -v insertionSort -
4 12 20 28 36 44 52 60 68 76 84 92 100
$ echo $RANDOM $RANDOM $RANDOM $RANDOM \
    | ./main -v mergeSort -
9180 15138 15262 22610
$ seq 100 -4 0 | ./main -v selectionSort - 4 7
```

```
88 92 96 100
60 64 68 72 76 80 84
$ seq 100 -4 0 | ./main -v bubbleSort - 4 7
88 92 96 100
60 64 68 72 76 80 84
$ seq 100 -4 0 | ./main  bubbleSort - 4 7
$ alg0="insertionSort bubbleSort selectionSort"
$ alg1="mergeSort quickSort"
$ for a in $alg0 $alg1; do \
    echo "*** $a"; \
    time ./main $a ../random.dat 10000; \
  done
*** insertionSort

real    0m0.208s
user    0m0.203s
sys     0m0.005s
...
$
```

The `time` in the last command above is used to show the time taken for each run of the program. The `real` time is the elapsed "wall-clock" time, `user` and `sys` times are the time spent in user-space, system-space respectively.

Note that the times for quicksort and mergesort are much smaller than those for the other sorts.

1.5 Exercise 3: Sorting Algorithm Operations

Important measures for different sorting algorithms are the number of comparison and swap operations.

Change over to the 3-stats directory. It contains essentially a copy of the code from the previous exercise except for one change: in the previous exercise, the comparison operations were hardcoded as `<`, `>`, etc. in the code and swaps were hardcoded to a `swap()` function. In this exercise, they have been replaced by virtual functions of an `Ops class` in ops.hh with implementation in ops.cc.

```
class Ops {
public:

  /** return < 0, == 0, > 0 if a < b, a == b, a > b */
  virtual int compare(int a, int b);

  /** swap a[i] and a[j] */
```

```
  virtual void swap(int a[], int i, int j);
};
```

Extend this class to count the number of `compare()` and `swap()` operations. Specifically:

1. Define a new class which inherits from the above class.
2. This class should define members which maintain counters for the number of calls to `compare()` and `swap()`.
3. Override the `compare()` and `swap()` functions. Have them merely wrap the corresponding functions in the base class by calling the corresponding base class function (using syntax like `this->Ops::compare()`). Additionally, they should also update the appropriate counter.

Add code to `main.cc` so that when run, the program will output three **tab-separated columns** containing the following headings:

`n` The number of integers being sorted.

`compares` The total number of calls to `compare()` for sorting the `n` integers.

`swaps` The total number of calls to `compare()` for sorting the `n` integers.

An example output:

```
$ ./main quickSort ../random.dat `seq 10000 10000 100000`
n	compares	swaps
10000	209967	38265
20000	447434	81683
30000	713555	126515
40000	967632	172602
50000	1314331	217209
60000	1551919	267604
70000	1889324	315032
80000	2170144	365113
90000	2436807	418455
100000	2714756	469392
$ for f in `seq 100000`; do echo $f; done > ../ascending.dat
$ ./main insertionSort ../ascending.dat `seq 2000 2000 10000`
n	compares	swaps
2000	1999	0
4000	3999	0
6000	5999	0
8000	7999	0
10000	9999	0
$ ./main insertionSort ../descending.dat `seq 2000 2000 10000`
n	compares	swaps
```

```
2000     1999000 1999000
4000     7998000 7998000
6000     17997000        17997000
8000     31996000        31996000
10000    49995000        49995000
$
```

Note that `insertionSort()` performs no swaps when the data is already sorted.

Collect data:

```
# alg0 and alg1 as defined in previous exercise
$ for a in $alg0; do \
    ./main $a ../random.dat `seq 1000 1000 10000` > $a.dat; \
  done
$ for a in $alg1; do \
    ./main $a ../random.dat `seq 10000 10000 100000` > $a.dat; \
  done
```

You can look at the generated `*.dat` files using a text editor, but you can also view the data as a graphical plot:

1. Use gnuplot to convert each data file into a `png` image:
   ```
   $ for a in $alg0 $alg1; do ../plot.gp $a.dat > $a.png; done
   ```

2. If you are on a graphical terminal, you can view the `.png` images:
   ```
   $ for f in $alg0 $alg1; do display $f.png; done
   ```

   If you are not on a graphical terminal, download the `*.png` files onto your local workstation and view them using a browser or any image viewer.

1.6 Exercise 4: Sorting Algorithm Times

Change over to the 4-time directory. It contains a `now` module specified by now.hh which contains a `now()` function which returns the number of milliseconds since Jan 1, 1970. You can use that function to time each call to a `sort()` function.

Copy over the files from Exercise 2. Change the code around the call to `sort()` within `go()` in `main.cc` to:

```
long t0 = now();
sort(a, n);
```

```
long t1 = now();
//t1 - t0 contains time for sort() in millis
```

As you did in the previous exercise, add code to `main.cc` so that when run, the program will output two **tab-separated columns** containing the following headings:

`n` The number of integers being sorted.

`time` The amount of time in millis taken for the call to `sort()`.

Example output:

```
$ ./main insertionSort ../random.dat `seq 1000 1000 10000`
n	time
1000	11
2000	26
3000	25
4000	33
5000	51
6000	73
7000	100
8000	131
9000	164
10000	204
$ ./main quickSort ../random.dat `seq 10000 10000 100000`
n	time
10000	6
20000	10
30000	7
40000	6
50000	7
60000	6
70000	8
80000	9
90000	10
100000	11
$
```

Once you have your changes working, collect the timing data for all the algorithms as in the previous exercise. Use the plot.gp gnuplot program to obtain `.png` plots. View the plots using `display` if you have a graphical terminal; otherwise download and view on your workstation.

1.6.1 Winding Up

Follow the *provided directions* for winding up this lab. Terminate your `script` session producing the log file `lab9.LOG` in your `lab9` directory. Add all your files to git and commit. Then merge your `lab9` branch into the `master` branch and commit your changes.