CMPSC 463: Name That Sort!

In this assignment we will put the *Science* back in *Computer Science*! You are given a black box which contains implementations of 11 different sorting algorithms, in a random order. Your mission is to use everything that you have learned about sorting algorithms to identify the order of the sorting algorithm. You will need to use everything you have learned about sorting algorithms to complete this assignment. including your in-depth understanding of the advantages and limitations of asymptotic running time analysis, worst-case and average-case running times of sorts, sort stability, differences auxiliary space usage, and pathological inputs.

You will not be able to examine the code for the sorts, and, therefore, you will need to base your answers on externally observable factors, including the running time for the sorts for different inputs; the output of the sorts; errors, like stack overflows, that can be induced in some sorts under special inputs; and memory usage.

To organize your exploration of the sorts, you will frame your analyses as a series of experiments. You will start by stating a hypothesis about what you expect to see from some or all of the sorting algorithms (in the form `if … then … because …`). For example, you might write:

> *Hypothesis:* **If** given randomly ordered input of size 30, **then** most of the sorts will finish very quickly except Bogo Sort, **because** the expected running time of Bogo Sort is O(n!), while all of the other sorts run in polynomial time.

Then you will create an input file that will be provided to one or more of the sorts. If the input file is small enough, you could create it by hand. Otherwise, you will probably want to write a program to generate the input. In your deliverable, you should provide a short description of the input, for example:

> *Input File:* Input with 30 values to be sorted, in a random order.

You will then run the experiment on one or more of the sorts, and report the results in a table like the one below.

> *Results (running time in ms):*

Sort Number	1	2	3	4	5	6	7	8	9	10	11
Time (s)	1	1	> 120	0	1	0	1	0	0	1	1

Finally, you will report your conclusions based on the results. For example, you might write:

> *Conclusions:* Since sort #3 had to be killed before it was able to complete, and all of the others took 1 ms or less, I conclude that sort #3 is Bogo Sort.

The Black Box

The sorts are implemented in an exe file, `nameThatSort` (for Linux). You should run these programs on a SUN lab computer.

To run the program, you will use the following command:

```
nameThatSort [ID] [Sort#] [Output] < [input]
```

where

- `[ID]` is the last four digits of your PSU id. This is the 9 digit number starting with "9" that appears on the PSU id card. Everyone will have a unique mapping based on their ID.
- `[Sort#]` is a number between 1 and 11, inclusive, used to select one of the sorts that are implemented in the program.
- `[Output]` is either 'Y' if you want to see the sorted output or 'N' if you do not.
- `[input]` is an input file that you create which you will use to try to distinguish between the different sort algorithms. The format of the file is given below.

The program will output the input in sorted order (if you set `Output` to `Y`), the time it took to do the sorting (i.e. the program's running time minus the time for input and output), and the amount of auxiliary memory used by the sort.

The following sort algorithms from the lecture notes have been implemented as sorts 1 through 11:

a. Bogo Sort*
b. Bubble Sort*
c. Counting Sort
   To determine the size of the universal set, this algorithm first scans the input to find the maximum value. Then, we set $|U| = max_{0 \le i < n}(A[i])+1$
d. Heap Sort
e. Insertion Sort
f. Merge Sort
g. Quick Sort
   This is the first version of Quick Sort that we studied, i.e. a single pivot chosen from end of the array segment
h. LSD Radix Sort
   The number of iterations, k, is based on the maximum value in the array, i.e. for a zero-based array A with n elements: $k = \begin{cases} 0, & if\ \forall i \in \{0,1,2,\dots,n-1\}, A[i] = 0 \\ max_{0 \le i < n, A[i] \neq 0}(1 + \log_{32} A[i]), & otherwise \end{cases}$
i. Randomized Quick Sort
   Using a single pivot.
j. Selection Sort
k. Stooge Sort

*: See implementation notes at end of this assignment

Input File Format

The first line of the input file is a number *n,* an integer between 1 and 100,000,000, inclusive. Following this line are *n* lines each providing an item consisting of an integer key and a string value. These items are to be sorted based on the integer key. The integer keys to be sorted must be between 1 and 100,000,000, inclusive. The string value can be helpful to determine how the sorting program is handling duplicate keys.

Here is an example of a file with 5 numbers to be sorted:

```
5
100 a
2 b
20 c
1 d
1000000 e
```

Rubric

You will be evaluated based on the following criteria:

1) Soundness of your experiments (28 points).
    a. Are the hypotheses well-reasoned?
    b. Are the input files appropriate for the hypotheses?
    c. Are the results listed?
    d. Are the conclusions justified based on the results obtained?
2) Correctness of your conclusions (72 points). Each correct identification is worth 6 points. Partial credit of 1.5 or 3 points per sort will be given based on the following:
    a. Correctly identify sort stability (1.5 points)
    b. Correctly distinguish between $\Omega(n^2)$ and $o(n^2)$ algorithms. (1.5 points)

Deliverables

Note handwritten entries will not be accepted. You must submit an electronic copy of your submission as a Word or PDF file with the following:

1) Your Name
2) The last 4 digits of your PSU id
3) For each input file (or set of related input files) that you created, provide:
    a. A hypothesis about what you expect to see from some or all of the sorting algorithms
    b. A description of the contents of the input file
    c. A table with the results of your testing
    d. Conclusions that you drew from these results
4) The mapping of Sort# to sort algorithm supported by your experiments.

Implementation Notes

```
bogoSort(A[1..n])
    while (!isSorted(A))
        randomly permute A
    end while
end bogoSort
```

```
bubbleSort(A[1..n])
    swapped = true;
    end = n
    while (swapped)
        swapped = false;

        for i = 2 to end
            if (A[i-1].key > A[i].key) then
                swap(A[i-1],A[i])
                swapped = true;
            end if
        end for
        end = end - 1
    end while
end bubbleSort
```

FAQ's

Q: Once I have determined the identify of a sort, do I need to include it in further tests?

No, once you have determined the identity of a sort, you do not need to include it in subsequent tests.

Q: I know that our asymptotic notation is limited to describing the scalability of algorithms. How can I use this to try to identify sorts?

First, it is important to recognize that our asymptotic running time analysis is limited to describing the scalability of algorithms. It assumes LARGE input sizes. For practical purposes, it would be difficult to draw conclusions from inputs of size 10,000 or smaller.

Moreover, it may be difficult to distinguish between algorithms even if their running time is very different based on a single input file. Since scalability characterizes the running time as input size increases, you will want to use multiple input files in a single test, with varying number of inputs. ***With the exception of identifying BogoSort, any test involving running time must use two different inputs that are compared.*** This will allow you to observe how running time changes with changes in input file size or input file properties.

Q: Will I be able to use just observations of running time on random input to distinguish between the sorts?

No. The differences in timing for sorts that have the same asymptotic running time on generic input will be too subtle in most cases. You will need to create additional tests, including ones that do not rely solely on running time.

Q: How do I interrupt the sort program if it is taking too long to complete?

Some of the sorts may take centuries (or longer) to complete for some input sizes. You may want to interrupt your program. To do so, you can type <ctrl> c (the control key and 'c' at the same time). In the Unix lab, if this does not stop your sort, you can kill it by typing the following:

```
pkill java
```

Note that this will kill all java processes that you are running.

Q: When I am running my program, how can I ensure that the timing is as accurate as possible?

First of all, you want to make sure that garbage collector is not producing output when you are timing the programs. In addition, you may want to run your program 3 times, and choose the smallest of the running time to reduce the impact of disk caching and other running programs on the timing.

In addition, if you are running your program in the Unix lab, you will want to make sure that there are not significant other processes running at the same time. At a command prompt, type the command `top`. This command will list the other processes that are running, as well as the status of resources. Type 'q' to quit the top program.

Below is the first few lines of sample output from this command:

```
top - 09:01:40 up 5 days, 22:01,  3 users,  load average: 13.20, 13.13, 12.16
Tasks: 240 total,   4 running, 236 sleeping,   0 stopped,   0 zombie
%Cpu(s): 80.9 us, 16.0 sy,  0.0 ni,  3.0 id,  0.1 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 32828312 total,   415492 free,  4176892 used, 28235928 buff/cache
KiB Swap:    92156 total,        0 free,    92156 used. 27802124 avail Mem
```

Note that in the output above more than 80% of the CPU is being used. A lot of the physical memory is also in use. This would not be a good machine to do your tests on. You want a machine whose output from this program starts more like below:

```
top - 09:18:08 up 2 days, 35 min,  1 user,  load average: 0.02, 0.01, 0.00
Tasks: 216 total,   1 running, 215 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.3 us,  0.1 sy,  0.0 ni, 99.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 32828360 total, 27498052 free,   925408 used,  4404900 buff/cache
KiB Swap:    92156 total,    92156 free,        0 used. 31406732 avail Mem
```