COMP 1927: Task 1

Phase 2 – Experimentation and Analysis

Peter Kydd (z3367463)

26th August 2013

CONCLUSION:

After extensive testing, I have concluded that the sorts that have been given consist of the following:

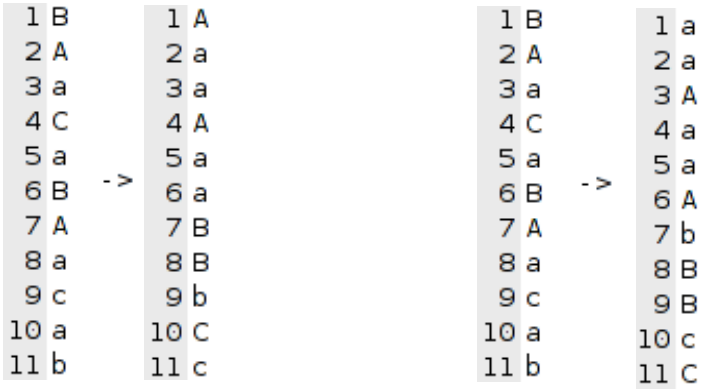
- **Sort 1:** Radix, MSD
- **Sort 2**: Vanilla Selection sort
- **Sort 3**: Bogo Sort
- **Sort 4**: Quicksort, Median of Three

The evidence and rationale that led to thses conclusions will be discussed in more depth below.

GENERAL METHOD:

Faced with such a large number of potential sort identities, I timed each of my assigned sorts with random data over a large range. By initially using data sets increasing by an order of magnitude each time, I was able to generate a range of values that indicated where further testing would be appropriate. A series of tests with smaller increments in size would be conducted in the next testing stage to develop a more detailed profile of the average case time complexity.

The next major differentiating factor was stability. I passed into each sort program a set of data that had a very specific patern of upper and lowercase characters. An example of this idea is shown below.



Stable Unstable

In the stable example (left), the unsorted data has a specific pattern of upper and lower case characters, in this the order is sorted and reproduced on the right, with the specific order maintained and hence, the sort is stable. In the unstable example, (right), the characters are sorted, but the order has not been preserved and the sort program therefore cannot be stable. This technique was applied to all of my sorts and formed the mainstay of my stability testing.

Generating ordered and reverse ordered data for testing was very straightforward. The random string generator that I used previously, allowed very easy control over the number of strings and string length of the random data. By piping that random data of the desired dimensions into the system sort, then piping the result into the sort program that I had been assigned, I could easily produce ordered or reverse ordered data of any particular size. For example:

```
$ ./randomStrings 100 1000 | sort -f | time -p ~cs1927/bin/task1/sort1 -qn
```

Standard ordered data generation method.

Some caveats to be aware of when using this method:

1. Make sure that you are only timing the sort program, not the random generator or the system sort.
2. Use the -f flag with the in built sort program to make sure that it is case insenstive (having tested stability elsewhere). The -r flag allows sorting in reverse order.
3. The -p flag on the time function formats the time data into something easily readable.

With the average time complexity and the stability factors determined for each of my sorts, I could begin to generate specific case tests for each – these specific tests and the reasoning behind them will be discussed in more detail below.

It should be noted that the time complexities, while very important, are not alone indications of one type of sort or another. Because of the other variables involved, it is possible to mistake certain time complexities for one another, particularly if they aren't readily identifiable as in the case of a quadratic or linear sort. In these cases, it is very important to use other characteristics of the sort to determine its identity.

Aditionally, It is worth mentioning that in my phase 1 plan, I had inteded to make use of the memory profiling tool, Valgrind. Very late in phase 1, it was made clear that Valgrind was not available for use for our experiments. The alternative, pmap, was actually quite difficult to configure correctly and produced results that were difficult to interpret. This was an inconvenince, but the timing data and information on the stability of the sorts that I collected turned out to be more than enough information to determine the type of sort that I had.

However, information about memory use that I collected with the Linux System Monitor may give away some information about the implementation of the sorts, and I will speculate on that information in each sort section.

RESULTS:

SORT 1:
Suspected Identity: RADIX, MSD

After thoroughly testing the 100,000 to 1,000,000 range with random data, the time values did not have the characterstics of a linear or quadratic time complexity. Further testing with ordered data produced a very linear result. Reverse order data did not produce a quadratic result, suggesting that the sort was most likely not a quicksort. Overall the time data suggested that the sort may be a radix, but It can be very hard to tell without something to directly compare its performance to.

The next test I applied was designed to detect radix sorts. I used the two following sets of data, each 200,000 strings long, 20 characters wide, to test the sort:

```
 6 faaaaaaaaaaaaaaa
 7 gaaaaaaaaaaaaaaa
 8 haaaaaaaaaaaaaaa
 9 iaaaaaaaaaaaaaaa
10 jaaaaaaaaaaaaaaa
11 baaaaaaaaaaaaaaa
12 aaaaaaaaaaaaaaaa
```

MSD data

```
16 aaaaaaaaaaaaaaat
17 aaaaaaaaaaaaaaag
18 aaaaaaaaaaaaaaah
19 aaaaaaaaaaaaaaai
20 aaaaaaaaaaaaaaaj
21 aaaaaaaaaaaaaaab
22 aaaaaaaaaaaaaaaa
```

LSD data

On the left data was created with a variable most significant digit and on the right, was created with a variable least significant digit. Data on the left would allow MSD radix sorts to be quick and efficient, while slowing the LSD radix sort. The data on the right would cause the reverse of this situation to occur.

Applying the sort to the data on the left yielded an average time of 1.52 seconds, while the data on the right took over 600 seconds to be sorted, with the same number of strings. Based on the time complexity, stability and time characteristics when applied to radix biased data, this sort strongly demonstrated the qualities of the MSD radix.

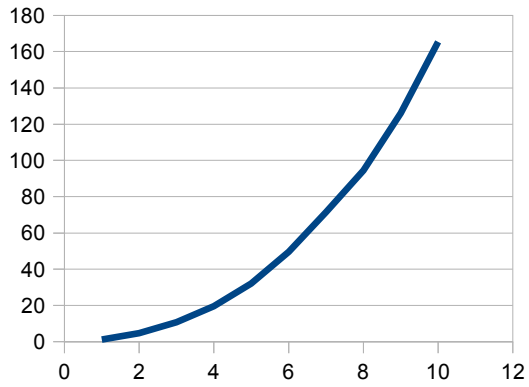
Possible Implementation Information:

Checking the memory use of this application with Linux's System Monitor gives some insight into the possible implementation of the sort. While sorting, the memory use of the program was stable (once the data set had been created). This suggests that the program is not using recursive calls to sort the data, and is most likely using an array or some other, non-dynamic form of storage. This is in line with expectations for a Radix-type sort.

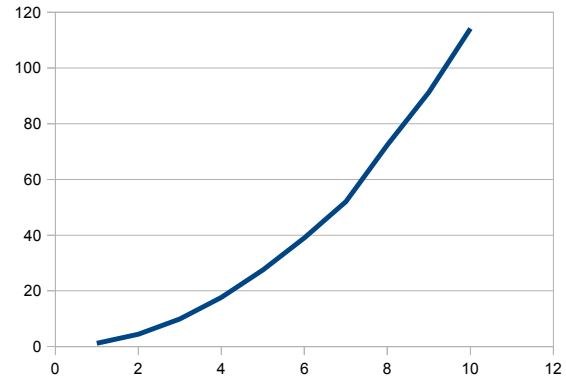
Sort 2, and Sort 3, for the same size data set, both used 114.5MB of memory, consistently. However, this sort used 129.9MB, just over 15MB of extra space. I hypothesise that this may have something to do with the stability of the sort – perhaps an extra array was used to maintain stability, as is the case with some radix sort implementations.

SORT2:
Suspected Identity: VANILLA SELECTION SORT

Upon testing with random data, this sort had very clear quadratic bahviour. When tested with orderd and reverse ordered data, it's time complexity remained almost perfectly quadratic, where an increase in data size of 10 times resulted in a time increase of 100 times. The random, and ordered data graphs are shown below.



Random Data



Ordered Data

*The x axis represents size of data set x10^5
**The y axis represents the average time in seconds taken to complete the sort

It was also unstable when tested with the method outlined above. The only unstable sort with consistently quadratic time complexity is the vanilla selection sort.

Possible Implementation Information:
As with Sort 1 and Sort 3, memory use here was consitent and did not increase once it had reached a stabe state. This suggests the use of an array for data storage. Less memory was used by this sort compared to Sort 1 – this sort is also unstable. There may be a correlation between those two facts – for example, there may only be one array that the data is being shuffled around in – hence the instability of the sort.

SORT3:
Suspected Identity: BOGO SORT

This was the first sort that I identified and the sort that required the least testing. When using random data to sort my sorts into their average time complexity groups, this sort would not finish, even with small sets of data (100 to 1000). It was also found to be unstable. Reducing the data set size and complexity, I was able to achieve sorting on eight strings of five characters in less than a second. This indicated that a data set range of five to fifteen strings may be a good range for testing. The results follow:

SORT3	Test1	Test2	Test3	Average
5 char x 6 strings	0	0	0	-
5 char x 7 strings	0.09	0.02	0.02	-
5 char x 8 strings	0.13	0.56	1.28	-
5 char x 9 strings	11.63	0.34	9.85	-
5 char x 10 strings	16.45	106.96	6.7	-

As shown above, the results became wildly unpredictable with anything over 8 strings so I stopped testing at 10 strings. This very poor performance with a small set of data is indicative of the bogo sort. Additionally, the large range, unpredictable times and the unstable nature of the sort also point to it being the bogo sort.

Possible Implementation Information:

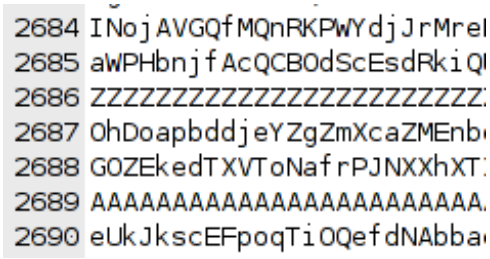
The memory profile of this sort is identical to Sort 2 for the same set of data, and the same implementation implications apply here, so I will not repeat them.

However, one curious thing that I discovered by accident about this sort is that it does not change its randomisation routine. That is, it randomises data in the same way, every time. I discovered this by passing in very small sets of data very quickly with a random data generator that uses the system time as a seed. I found that when I passed data in within the same second (that is, the random generator is creating the same data set), the bogo sort took exactly the same amount of random time to sort it.

This was confirmed by passing multiple longer sets in, multiple times. Even thought he implementation is 'randomly' shuffling the data, it took the same random amount of time, each time for each data set. This suggests that the shuffle is not random at all, rather, that it simulates randomness, but uses the same randomising routine each time.

SORT4:
Suspected Identity: QUICKSORT, MEDIAN OF THREE

Upon passing random data through this sort, the results were not linear, nor were they quadratic. Further, random and ordered data had almost identical time complexities. This, in addition to the unstable nature of the sort, ledt me to believe that it may have been a quicksort. To test this idea, I used a file made up of 15,000 lines of random strings, and inserted 500 lines of strings that were comprised entirely of A's and Z's. Ideally, this would cause at least some of the pivots selected to be extreme values, and thus cause poor performance. An example of this data can be seen below.



Depite the fact that only 1 in 60 lines were made up entirely of A's or Z's, the performance reduction was extreme. When 1,000,000 lines of random data with a string length of 100 characters were piped through, this sort took ~350 seconds to process it. When the above 'extreme value' lines were added, the sort took well over 1300 seconds for the same amount of data.

This strongly indicated that the sort was of the quicksort variety. Based on these performance characteristics, we could come to the conclusion that the sort was either median of three, or randomised quicksort.

An important consideration is that in the implementation given to us, the randiomised quicksort does not randomise pivot selection, but rather the entire list. This is important, as it allows for easy diffrentiation of the random from the median of three quicksort.

By passing a short list of data, made up of a particular pattern of upper and lower case letters (similar to the one used for my stability testing), I was able to detect any randomisation of the list itself. I am aware that the quicksorts are not stable, but the list was only slightly out of order – most of the paterns that I passed in were still present in the output. This semi-stability was consistent, and suggests that the quicksort type was most likely not of the random type. Hence, by deduction, the quicksort must have been of the median of three type.

Possible Implementation Information:
Using the system monitor, I detected the same amount of memory use as in Sort 2 and Sort 3, however, the memory values slowly increased as the program ran. This is expected for a recursively called sort like quicksort. As the program runs and recursively calls itself, more and more memory is allocated on the stack. As a result, the memory consumption increased.
This not only strengthens my case that this sort is a quicksort, but also provides a window into the implementation of the sort.

Appendix A:
Data:

Sort 1: Radix MSD

RANDOM DATA:

Data Size:	Test1	Test2	Test3	Average
10000				0.01
100000	1.03	1.04	1.03	1.03
200000	2.57	2.59	2.6	2.59
300000	4.69	4.7	4.7	4.7
400000	7.34	7.38	7.34	7.36
500000	10.59	10.55	10.54	10.57
600000	14.34	14.34	14.4	14.36
700000	18.61	18.6	18.6	18.6
800000	23.44	23.46	13.43	23.44
900000	28.74	28.73	28.8	28.78
1000000	34.64	34.69	34.63	34.65

ORDERED DATA:

Data Size	Test1	Test2	Test3	Average
1000	0.02	0.02	0.02	0.02
10000	0.08	0.08	0.07	0.08
50000	0.38	0.36	0.36	0.37
75000	0.57	0.56	0.54	0.55
100000	0.73	0.74	0.76	0.76
1000000	7.36	7.4	7.38	7.38

REVERSE DATA:

Data Size	Test1	Test2	Test3	Average
1000	0.01	0.01	0.01	0.01
10000	0.08	0.09	0.08	0.08
100000	1.28	1.28	1.28	1.28
1000000	60.98	60.85	60.88	60.9

RADIX SPECIFIC TESTING:

RADIX TESTING				
LSD*	TEST1	TEST2	TEST3	AVERAGE
	600++	600++	-	~600++
MSD	TEST1	TEST2	TEST3	AVERAGE
	1.52	1.5	1.49	1.5

*User terminated for the sake of time.

Sort 2: Vanilla Selection Sort

RANDOM DATA:

Data Size	Test1	Test2	Test3	Average
10000	1.23	1.24	1.23	1.23
20000	4.8	4.81	4.81	4.8
30000	10.75	10.75	10.78	10.76
40000	19.35	19.56	19.6	19.52
50000	32.04	31.84	32.49	32.17
60000	48.73	49.32	49.75	49.6
70000	68.82	74.59	71.06	71.49
80000	92.84	95.88	94.84	94.52
90000	127.23	126.91	126.8	126.1
100000	166.19	162.75	167.23	165.39

ORDERED DATA:

Data Size	Test1	Test2	Test3	Average
10000	1.15	0.02	0.02	0.03
20000	4.45	1.16	1.15	1.15
30000	9.93	9.8	9.95	9.9
40000	17.68	17.75	17.8	17.75
50000	27.51	27.54	27.48	27.6
60000	39.2	39.4	39.3	39.3
70000	52	62.89	63.07	63
80000	72.3	72.15	72.2	72.25
90000	91.24			
100000	114.2	113.12	114.91	114.2

REVERSE DATA:

Data Size	Test1	Test2	Test3	Average
1000	0.04	0.04	0.04	0.04
10000	3.05	3.06	3.04	3.05
100000	372.99	369.7	370.85	371
1000000				

Sort 3: Bogo Sort

RANDOM DATA:

BOGO					
SORT3	Test1	Test2	Test3	Average	UNSTABLE
5 x 6	0	0	0	-	
5 x 7	0.09	0.02	0.02	-	
5 x 8	0.13	0.56	1.28	-	
5 x 9	11.63	0.34	9.85	-	
5 x 10	16.45	106.96	6.7	-	

ORDERED DATA:
N/A

REVERSE DATA:
N/A

Sort 4: Quicksort, Median of Three

RANDOM DATA:

Data Size	Test1	Test2	Test3	Average
100000	10.58	10.61	10.59	10.59
200000	30.28	30.49	30.33	30.35
300000	56.43	56.13	56.25	56.3
400000	87.9	87.77	88	87.8
500000	124.49	124.4	125.23	124.9
600000	165.87	166	165.82	165.9
700000	211.38	210.93	210.3	210.9
800000	260.74	260.76	260.38	260.5
900000	313.39	313.6	312.96	313.4
1000000	368.58	367.9	368.28	368.1

ORDERED DATA:

	Test1	Test2	Test3	Average
100000	8.93	8.98	9	8.95
200000	25.09	25.13	25.08	25.1
300000	46.17	46.18	46.18	46.17
400000	71.63	70.22	72.83	71.56
500000	100.86	101.86	100.68	101.13
600000	132.89	132.72	133.22	132.6
700000	168.95	168.8	168.56	168.77
800000	208.48	207.98	207.88	208.08
900000	249.37	249.68	249.32	249.46
1000000	292.16	290.74	291.45	294.45

REVERSE DATA:

Data Size	Test1	Test2	Test3	Average
1000	0.03	0.04	0.04	0.04
10000	0.48	0.48	0.47	0.48
100000	13.17	13.2	13.19	13.19
1000000	479.21	480.13	481.87	480.13

QUICKSORT SPECIFIC TEST DATA:

Data Size	TEST1	TEST2	TEST3	AVERAGE
1000	0.18	0.18	0.18	0.18
10000	4.72	4.74	4.71	4.73
100000	146.83	147.36	146.88	
1000000	1100+++			