COMP 1927: Task 1

Phase 1 – Planning

Peter Kydd (z3367463)

26th August 2013

AIM:

Use the time complexity and other characteristics of various sorting algorithms to determine the identity of four that have been randomly assigned.

INTRODUCTION:

Before describing my specific tests, I will briefly discuss some of the methods and reasoning that I will be employing later on.

Choosing input data:

When selecting set size, it is important to choose sizes that will yield statistically significant results. Using set sizes that give numbers in the low milliseconds range, for example, are not going to allow for accurate determination of time complexity. The program overhead and the margin of error may be enough to distort and give misleading results. It is therefore important to use set sizes that will produce time complexities in the seconds range. The actual set sizes may vary between algorithms, depending on how they perform.

Taking the average of multiple measurements will also improve the accuracy of the overall measurement and increase the chance of correct identification.

Average case time complexity:

The first set of data that I will pass into my algorithms will be randomly generated strings. By taking the average of a large number of tests, this approach will provide the best approximation of the average time complexity of the algorithm.

Using this average case time complexity information, I will separate the sorts that I have been given into smaller groups that may be analysed in more detail.

Boundary case time complexity:

Often, an algorithm's best and worst performance is due to boundary case conditions. For example, the best case time complexity of a bubble sort with an early exit is n comparisons, a linear time complexity. This occurs when the list of items is already in order – a boundary case. However, if that same list is reversed, the sort will run for n^2. This too is a boundary case, and would allow potentially, for its differentiation among a list of others very easily. This particular characteristic will be very helpful for identifying certain algorithms.

Stability:

Another effective method of sort differentiation is the stability of the sort. Stability describes the ability of a sort to maintain the original order of duplicate items in a set. The algorithms that we have been provided are case insensitive – that is, a string of 'AAAA' is considered the same as a string 'aaaa'. This seemingly insignificant piece of information will actually play a large part in determining the stability of the algorithms being tested. For example, the difference between a selection sort and an oblivious bubble sort is not obvious, except that the selection sort will not preserve the original order of the data.

Memory:

Memory is another subtle indicator of sort type. I intend to use memory tools such as Valgrind or linux's pMap to identify whether or not an algorithm has been allocating large amounts of space, or, very specific amounts of space – for example, the selections sort with linked lists can be expected to allocate increasing amounts of space as the set size increases. Similarly, the difference between the bucket and counting sort is largely memory allocation related.

The caveat to using memory as a differentiator is that I am making an assumption that the algorithms have been implemented in C and will use memory in a predictable way. If that is not the case, it should become readily apparent as I record memory usage information throughout my experiment.

METHOD:

Choosing the correct sequence for these tests will be very important for both accuracy and minimization of time required. Initially, I intend to take the average over a large set of tests of the time required to sort random data. This should allow me to separate the sorts into general categories – $O(n^2)$, O(n*log(n)) and the other, less common complexities. Once the sorts are separated into these larger groups, I intend to use specific tests to identify which algorithm they are. These specific tests are described below and are also visually represented in the accompanying flow chart.

O(n*log(n)):

Running this group on linearly ordered data should cause the time complexity of **vanilla quick-sort** to approach n^2. Then, running data that has the first last and middle data values set to a maximum or minimum should cause **the median of three quick-sort** to also approach n^2.

If the time complexity of the sort is still n*log(n) under these conditions, I will test the stability of the sort. If the sort is unstable, it is very likely the **random quick-sort**. Finally, the last two sorts can be distinguished by their memory use – **merge sort** has relatively large amount of memory overhead, whereas the **insertion sort with binary search** does not.

$O(n^2)$:

Running the $O(n^2)$ group with ordered data should be enough to separate it into two separate groups of time complexities – this is a result of the ability of some sorts to take advantage of an already sorted set. In the O(n) sub-group, the next test will be to check the memory use of the sort by using a tool like Valgrind This will show any mallocing or freeing being employed by the sort and should identify **t**he **insertion sort on linked list.** The two remaining O(n) sorts here will be the **bubble early exit** and **the insertion sort**.
Note: these two sorts are almost indistinguishable in their time complexity, and they are both stable. This makes absolutely positive identification possible.

In the $O(n^2)$ sub-group, checking the stability of the sorts should reveal either the **selection sort**, which is unstable and **the oblivious bubble sort**, which is stable.

O(n+r):

There are only two sorts in this category – the counting sort and the bucket sort. Both have fairly short sorting times and potentially high memory use. However, the memory allocation for bucket sort (in this implementation) is fixed, whereas the counting sort implementation will vary depending on the amount of

data passed into it. I intend to use Valgrind, to identify **bucket sort** and **counting sort,** by observing the amount of memory that is allocated with respect to data set size for each.

O(n*(d/k)):
(where d = base of string (ie, 26), and k = number of keys)

The LSD and MSD radix sorts are distinguishable for a few reasons. Primarily, in their standard implementations, the **MSD** variant is unstable (without a separate array to store values in), whereas the **LSD** variant is stable. Additionally, as the key size increases, their performance will start to diverge, particularly with large ranges where MSD becomes more efficient. This is because MSD Radix sorts will sort the set into large groups based on the first digit. Subsequently, it only has to make comparisons within that group, rather than with the entire set, unlike LSD Radix. So, in addition to stability testing, I will be varying the length of the strings that are being passed into the sort, in order to for the LSD radix sort performance to diminish.

O(n*n!):

The **Bogo sort** is one of a kind. It will have a random time complexity for each set of data and is unbounded (it could potentially run forever). As a result, it would be best to limit the number of values passed into it. It should return wildly different values for the same entered data. For example, a 10 value set may be sorted instantly, or in over an hour (or longer). It will be fairly easy to identify by this erratic timing behavior

$O(n^{3/2})$:

The shells sorts and the quadratic selection sort will be the only sorts with performance approaching $O(n^{3/2})$.Passing ordered data through the shell sorts will separate them from the **quadratic selection sort**, where the shell sorts yield approximately O(n) time complexity and the quadratic selection sort will remain $O(n^{3/2})$.

The shell sorts are slightly trickier to differentiate. We can expect the shell(3k+1) to outperform the shell(4k) as the set size increases, however, I am unsure at this stage to what extent this will be the case. I will additionally use data that has every fourth string with an identical value. This may help reduce the performance of **the shell 4k** sort enough to differentiate it from the **shell (3k+1)** sort.

EXPECTATIONS:

As always with empirically derived data, the results will not be as clear as they have been represented here. Misidentification will be a possibility. Using an average time value over a number of sort passes should mitigate this issue to some degree, however, the possibility remains that the timings will not be exactly their theoretical value. To further complicate things, there are several algorithms that even under ideal circumstances will have very similar performance. In these cases, I will provide the evidence for each algorithm and then describe why I believe it is one or the other, or, why it could be either.

Appendix:

Appendix A – Sample Input Data

- Ordered strings:
  “aaaaa”, “bbbbb”. “ccccc”, “ddddd”…
- Ordered strings with capitals (to check for stability):
  “AAAaa”, “aaaaa”. “CCccc”, “cccccc”
- Reverse ordered data:
  “zzzzz”, “yyyyy”, “xxxxx”, “wwwww”
- Random ordered data:
  “alSKF”., “hIOjg”, “fjeIG”, “IASDj”

*Note: string lengths will be longer in the actual tests than the examples above unless otherwise stated.

Appendix B – General Sort Characteristics

Algorithm	Best Case	Average Case	Worst Case	Stability
Oblivious Bubble	n^2	n^2	n^2	Stable
Bubble sort with early exit	n	n^2	n^2	Stable
Vanilla insertion sort	n	n^2	n^2	Stable
Insertion sort on list	n	n^2	n^2	Stable
Insertion sort with binary search	n*log(n)	n^2	n^2	Stable
Vanilla selection sort	n^2	n^2	n^2	Unstable
Quadratic selection sort	$\mathbf{n^{3/2}}$	$\mathbf{n^{3/2}}$	$\mathbf{n^{3/2}}$	Unstable
Merge sort	n*log(n)	n*log(n)	n*log(n)	Stable
Vanilla Quick sort	n*log(n)	n*log(n)	n^2	Unstable
Quick sort Median of three	n*log(n)	n*log(n)	n^2	Unstable
Randomized Quick-sort	n*log(n)	n*log(n)	n^2	Unstable
Shellsort (3K+1)	n	$n^{3/2}$	$n^{3/2}$	Unstable
Shellsort (4K)	n	$n^{3/2}$	n^2	Unstable
Bogo Sort	n	n*n!	unbounded	Unstable
Radix MSD	-	n*(d/k)	n*(d/k)	Unstable
Radix LSD	-	n*(d/k)	n*(d/k)	Stable
Bucket sort	n+r	n+r	n+r	Stable
Counting sort	n+r	n+r	n+r	Stable

Appendix C – Test Flow Chart