INTRODUCTION

In this assignment, we created a simulator capable of emulating different ways of organizing a computer's memory space. We used a chained hash table to simulate pages in memory and integers to simulate memory addresses. Our maximum address possible was 33,554,431 (2^{25}-1). Therefore the number of pages is this maximum address divided by the page size.

In terms of storing data, we would take the address of a value, do a simple modulus operator on that address with the page size, and put the address in the resulting hash index. At each index we have a linked list (the "chain"), which means that in a fully occupied memory space, the first element of every linked list corresponds to the first page, the second element corresponds to the second page, and so forth.

Our way of measuring data was through window size and working set sizes. We took 4 different page sizes (64, 128, 256, 512) and for each page size we evaluated how the working set size changed with 8 different window sizes (128, 256, 512, 1024, 2048, 4096, 8192, 16384). The idea is to see how many different pages are accessed per window size. The more distinct pages that are accessed, the more page faults there are.

DATA

See Appendix A

ANALYSIS

Using this simulator and measuring parameters described, we analyzed the performance of two algorithms: quicksort and heapsort. Before running simulations, we already knew the space complexity of heapsort, O(1) and quicksort, O(n log n) (to ensure this space complexity we used iterative quicksort which emulates its own stack, rather than the recursive version of quicksort which can increase to $O(n^2)$). Under these assumptions, we expected to see linear, horizontal lines for heapsort and a curve upwards for quicksort. However, the results were different than what we had expected:

We had made the mistake of confusing big O space complexity with the working set size. Big O is a measurement of different algorithms with increasing input sizes. For these simulations, however, we were not increasing the input sizes. Rather, we were keeping the input size constant and only varying the way in which we measured the page accesses to memory.

Under these realizations, we made the following conclusion: although the space complexity for heapsort is linear (much less than the n log n of quicksort), heapsort must access its own memory, which is spread over multiple pages, very often. Heapsort uses indexes to simulate a

binary tree within an array, and it traverses this tree's "nodes" multiple times. As it jumps between elements in this array, the additional space used does not increase, but the number of accesses to different pages increases significantly. Therefore the number of page swaps observed for heapsort is much greater than for quicksort.

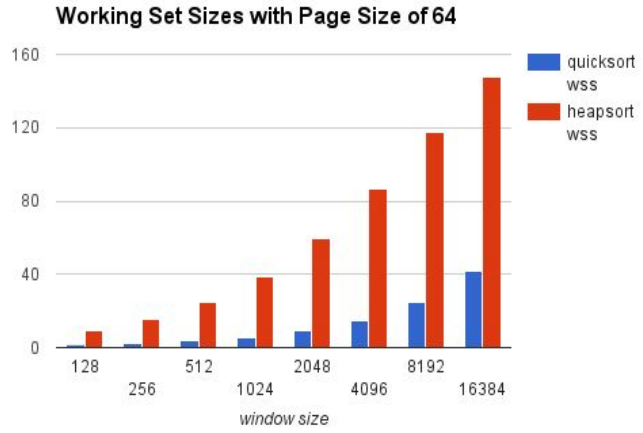
On the other hand, quicksort requires an additional space of at least log n (assuming that the iterative version of quicksort is used. In comparison to heapsort, however, quicksort sorts multiple subarrays at a time. The implication here is that much of the elements in these subarrays are in contiguous memory areas (and therefore in the same page). Partitioning and "creating" these sub-arrays occupy additional memory, but this is the cost of having fewer page swaps.

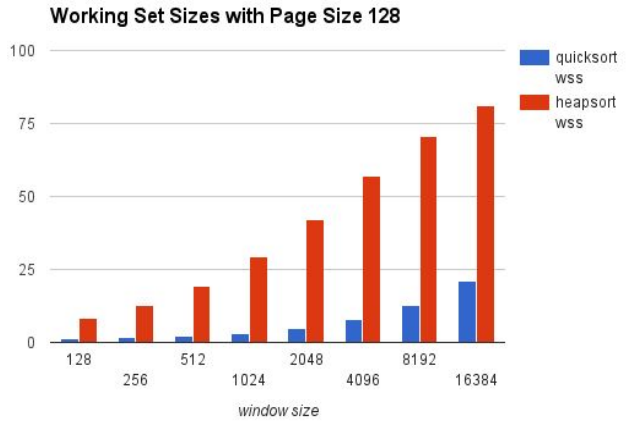
The graphed data also displays the natural correlation between window size and working set. Obviously if more possible page accesses can be captured in a window, the more likely to see distinct page accesses within this window. Additionally, graphs demonstrate the relationship between page size and working set size. Smaller pages means more possible distinct pages and thus a large number of page faults.

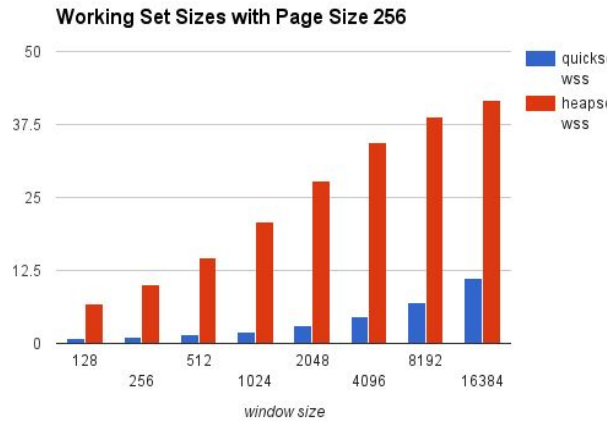
CONCLUSION

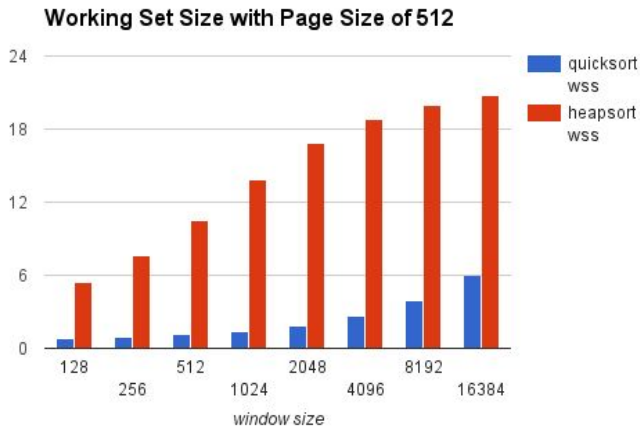
Heapsort takes no additional space to sort a set of numbers. Iterative quicksort takes log n additional space to sort the same set of numbers. However, heapsort creates a binary tree inside of its array using indexes, while quicksort creates subarrays and sorts them accordingly. Our results show that heapsort takes many more page swaps to execute than quicksort. In terms of practical applications, there is a tradeoff between preserving space (which favours heapsort) and saving time (clock cycles) by having as few page swaps as possible (which favours quicksort). From our experiments, we know that the quicksort algorithm consistently requires less page swaps than the heapsort algorithm.

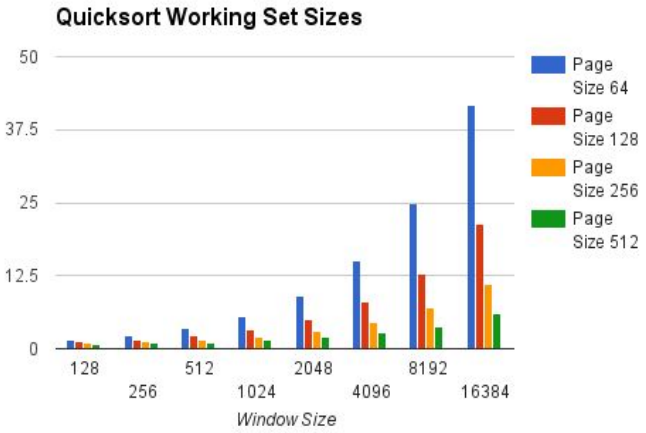
Appendix A

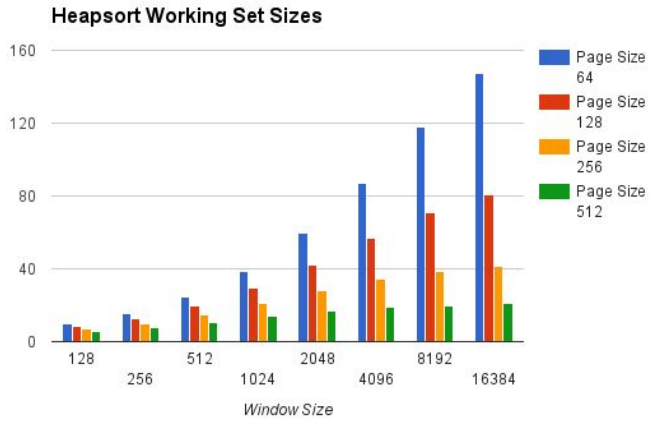

Working Set Sizes with Page Size of 64
160
120
80
40
0
128
256
512
1024
2048
4096
8192
16384
window size
quicksort wss
heapsort wss


Working Set Sizes with Page Size 128
100
75
50
25
0
128
256
512
1024
2048
4096
8192
16384
window size
quicksort wss
heapsort wss


Working Set Sizes with Page Size 256
50
37.5
25
12.5
0
128
256
512
1024
2048
4096
8192
16384
window size
quicks wss
heaps wss


Working Set Size with Page Size of 512
24
18
12
6
0
128
256
512
1024
2048
4096
8192
16384
window size
quicksort wss
heapsort wss


Quicksort Working Set Sizes
50
37.5
25
12.5
0
128
256
512
1024
2048
4096
8192
16384
Window Size
Page Size 64
Page Size 128
Page Size 256
Page Size 512


Heapsort Working Set Sizes
160
120
80
40
0
128
256
512
1024
2048
4096
8192
16384
Window Size
Page Size 64
Page Size 128
Page Size 256
Page Size 512