**Dzu Pham**

**Professor Gerry Howser**

**Computer Science 253 Algorithm**

**October 16, 2013**

Putting Insertion Sort to the Test

**Motivation**

There are many different ways to solve a problem, today we look at how to sort a list of numbers to get it into ascending order. Three sorts will be put to the test to see its performance with a process of an array of numbers in C++. Insertion sort, Merge sort, and Heap sort are the three sorts that will be used. The main focus will be seeing is how each sort performs with different sizes of array (an array is also known as list outside of computer science). The sorts will be tested against 3 types of arrays, (1) randomly sorted array, (2) backwards sorted array, and (3) fully sorted array in the range of 10,000 elements to 90,000. Any other size array will not be accounted. Each test will measure the amount of time it takes to fully sort the array.

Insertion Sort is expected to perform better with arrays on smaller size and arrays that are nearly too fully sort. Merge sort and heap sort will be expected to have similar outputs as the run time for each is identical. Both algorithms are not the same however so one algorithm will have to run faster, and that will be heap sort.

**Background**

Insertion sort is just one of the many other sorting algorithms developed in computer science. It is not the most efficient but a very simple algorithm to understand. The best way to describe how insertion sort works is by comparing it as a deck of playing cards.

*“We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each card already in the hand, from right to left. At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.”[[1]](#footnote-1)*

To show mathematically how long it takes for the insertion to sort any length of the array is with the equation C(n2). C is a constant that depends on the processing power of the machine, which in this lab can be ignored. The reason that the constant can be taken out is because as n approaches infinity, then the constant will be negligible. The benefit of have n2 is that it accounts for the worst case scenario, which means that insertion sort will not run longer than n2. The smaller the output from this question will result in a faster sorted time, so as the array gets larger, the time will exponentially increase to compensate for the time it takes the algorithm to completely sort the array.

Merge sort and Heap sort runs at O(n log n) which is a slower growing run time then insertion sort. The way these sort works is by breaking up the array into parts and solving each part separately. This is a method is called a binary tree, breaking up the array into two pieces and branching off downwards. Both sorts are similar in run time but differ in what makes up how they sort the array of numbers. The merge sort algorithm works like the “divide-and-conquer” method;

*“****Divide****: Divide the n-element sequence to be sorted into subsequence of n/2 elements each.*

***Conquer****: Sort the two subsequence recursively using merge sort.*

***Combine****: Merge the two sorted subsequence to produce the sorted answer.”*

Heap sort is the final algorithm that will be used, this sort combines the better aspects of both insertion sort and merge sort. This algorithm focuses more on dealing with branches, which will be left and right. These two branches will be connected to the parent; the parent can also be a subtree to another left or right branch as well. This type of algorithm is more complex than the other two sorts, which is why this algorithm is broken into parts;

*“The* ***Max-Heapify*** *procedure, which runs in O(lg n) time, is the key to maintain the max-heap property.*

*The* ***Build-Max-Heap*** *procedure, which runs in linear time, produces a max-heap from an unordered input array.*

*The* ***HeapSor****t procedure, which runs in O(n lg n) time, sorts an array in place.”*

These three parts are the key feature in keeping the run time of this algorithm at O(lg n) time. The run time of heap sort is now showed to be very similar to merge sort because both sorts use the binary tree method.

**Procedures**

1. Designed a pseudocode for testing the three algorithm
2. Create pre and post conditions before testing the algorithm
3. Express the invariants and support it
4. Implement the pseudocode in a C++ program
5. Validate correctness of the program with pre and post conditions created
6. Testing phase
7. Measure the run time with different sizes of array
8. Record multiple test of the algorithm
9. Create table and sort results, spice it up
10. Finish with conclusion, address the problems encountered

**Pseudocode (includes Pre/Post conditions)**

Inside int main //*main body of the program*

Pre: The array is created to a certain size, filled with numbers (sorted or not)

Post: A[1] < A[2] … A[n] && A[i] < A[j-1]

Invariant: A[i … j-1] && A[i] < A[j-1]

Initialization: Prior to first iteration, i is set to 1 and j is set to 2.

Maintenance: At start and end of each loop, A[j-1] < A[j]

Termination: j = n-size, by Invariant A[i] < A[j-1]

create array [n-size]

for i to n-size

//*this is where the different types of array is built*\*

array [i] = x //*fully sorted, using 1 to n-size*

array [i] = backwards

array [i] = random

call insertion sort

Insertion Sort

for j = 2 to length

key = A[j]

i = j – 1

**while** i > 0 and A[i] > key

A[i+1] = A[i]

i = i -1

A[i+1] = key

Merge Sort

if(p<r)

Pre: The array is created to a certain size, filled with numbers (sorted or not)

Post: A[p…k - 1] = A[p…r] && A[k] < A[k+1]

Invariant: L[i-1] < A[k] and R[j-1] < A[k]

Initialization: before first iteration, i = j = 1. Both L[i] and R[j] is made form A[].

Maintenance: The smallest element form L[i] and R[j] is copied into A[k]. k and i/j is incremented.

Termination: When k = r+1, the array A[k] is sorted in ascending order.

q = floor (p+r)/2

mergeSort(A,p,q)

mergeSort(A,q+1, r)

merge(A,p,q,r)

merge (A,p,q,r)

n1 = q - p + 1

n2 = r - q

Create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]

for i = 1 to n1

do L[i] = A[p + i - 1]

for j = 1 to n2

do R[j] = A[q + j ]

L[n1 + 1] = infinty

R[n2 + 1] = infinty

i = 1

j = 1

for k = p to r

if L[i ] <= R[ j]

then A[k] = L[i]

i = i + 1

else A[k] = R[j]

j = j + 1

Heap Sort

BuildHeap(A)

Pre: The array is created to a certain size, filled with numbers (sorted or not)

Post: A[1] < A[2] … A[n] && A[i] < A[j-1]

Invariant: A[i] < A[j-1] && A[i] > A[i\*2]

Initialization: Prior to first iteration, i = floor(n/2).

Maintenance: if the child is higher than the parent then Max-heap is called. i is decrementing after each for loop, moving down the array

Termination: i = 0, which is the end of the array and everything is now sorted.

for i <- length(A) downto 2 {

exchange A[1] <-> A[i]

heapsize <- heapsize -1

Heapify(A, 1)

BuildHeap(A) {

heapsize <- length(A)

for i <- floor( length/2 ) downto 1

Heapify(A, i)

Heapify(A, i)

l <- left(i)

r <- right(i)

if (l<=heapsize) and (A[l]>A[i])

largest <- l

else

largest <- i

if (r<=heapsize) and (A[r]>A[largest])

largest <- r

if (largest != i) {

exchange A[i] <-> A[largest]

Heapify(A, largest)

//*Types of array*\*\*

Backwards sorted array

Returns (n-size – 1) each time it is called

Randomly generated array

Returns a random number of (n-size)

\*To minimize the complexity of creating the array, each test phase 2 out of the 3 array is commented (//) out so that only one of the array is actually being used. In the code posted at the end you will see 2 of the arrays commented out.

\*\*This is not a part of the main body because they are build outside of the program; they are called when they are needed to be used, so the program does not process this code unless it is called.

**Problems** **Encountered**

The algorithm’s code did not match the way the array was set up, the array was zero based while the algorithm was created to accommodate a one based array. Adjustments were made to increment how the array was created and modify the driver programs to not read the zero point of the array and start with the one slot. Another major problem encountered was the complier used, with a piece of code that was created for the C++ 2011 version, new methods had to be used for the current complier (putty) to compile the new code.

**Testing** **Plan**

All of the data points on the measured output are the mean of multiple test runs of each algorithm. Displaying the mean will show the general curve of how the function will work.

**Insertion Sort**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array Size | Expected Output | | | Measured Output | | |
| **Random**  **Sorted** | **Backward Sorted** | **Fully Sorted** | **Random**  **Sorted** | **Backwards Sorted** | **Fully Sorted** |
| 10000 | 0 | 0 | 0 | 0.040747 | 0.16011 | 0.000101 |
| 20000 | 1 | 1 | 0 | 0.158557 | 0.631645 | 0.000213 |
| 30000 | 1 | 1 | 0 | 0.35982 | 1.42653 | 0.000303 |
| 40000 | 2 | 2 | 0 | 0.627624 | 2.52807 | 0.000407 |
| 50000 | 2 | 2 | 1 | 0.988156 | 3.96217 | 0.000521 |
| 60000 | 3 | 4 | 1 | 1.46604 | 5.71146 | 0.000617 |
| 70000 | 5 | 5 | 1 | 1.94556 | 7.76066 | 0.000713 |
| 80000 | 7 | 6 | 1 | 2.54856 | 10.1296 | 0.000825 |
| 90000 | 9 | 8 | 1 | 3.2112 | 12.8423 | 0.000912 |

**Merge Sort**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array Size | Expected Output | | | Measured Output | | |
| **Random**  **Sorted** | **Backward Sorted** | **Fully Sorted** | **Random**  **Sorted** | **Backwards Sorted** | **Fully Sorted** |
| 10000 | 0 | 0 | 0 | 0.002334 | 0.003861 | 0.004081 |
| 20000 | 1 | 1 | 0 | 0.004882 | 0.0084 | 0.007607 |
| 30000 | 1 | 1 | 1 | 0.007756 | 0.013053 | 0.013395 |
| 40000 | 2 | 2 | 1 | 0.011046 | 0.015707 | 0.013405 |
| 50000 | 3 | 3 | 1 | 0.014165 | 0.021797 | 0.013402 |
| 60000 | 3 | 4 | 1 | 0.016224 | 0.015016 | 0.022735 |
| 70000 | 4 | 4 | 1 | 0.019272 | 0.017273 | 0.022558 |
| 80000 | 4 | 5 | 2 | 0.022265 | 0.026401 | 0.028047 |
| 90000 | 5 | 5 | 2 | 0.025246 | 0.028704 | 0.029904 |

**Heap Sort**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array Size | Expected Output | | | Measured Output | | |
| **Random**  **Sorted** | **Backward Sorted** | **Fully Sorted** | **Random**  **Sorted** | **Backwards Sorted** | **Fully Sorted** |
| 10000 | 0 | 0 | 0 | 0.002404 | 0.00392 | 0.002497 |
| 20000 | 1 | 1 | 0 | 0.005148 | 0.00886 | 0.005373 |
| 30000 | 1 | 1 | 1 | 0.008278 | 0.013437 | 0.008387 |
| 40000 | 2 | 2 | 1 | 0.011053 | 0.014399 | 0.01119 |
| 50000 | 3 | 3 | 1 | 0.014282 | 0.019139 | 0.01484 |
| 60000 | 3 | 4 | 1 | 0.017442 | 0.021787 | 0.018092 |
| 70000 | 4 | 4 | 1 | 0.020691 | 0.020961 | 0.021252 |
| 80000 | 4 | 5 | 2 | 0.024772 | 0.025315 | 0.024911 |
| 90000 | 5 | 5 | 2 | 0.027434 | 0.027057 | 0.028099 |

**Performance** **Results**

Figure 1 - 3 Insertion Sort

Figure

Figure

Figure

In figure 3 shows that it would take very little time for sorting the array. It is very hard to see the curve as the numbers increase.

Figure 4-6 Merge Sort

Figure

Figure

Figure

Figure 7-9 Heap Sort

Figure

Figure

Figure

**Conclusions**

After testing the program, Heap sort proves to be the fastest overall. However heap sort was not that much faster than Merge sort. By displaying the graph of the data point, it shows that merge sort and heap sort has similar growth functions, reinforcing the face that the run time for both algorithms is similar. The original theory that insertion sort will be the fastest on fully sorted arrays and the smaller sizes arrays were correct and that merge sort and heap sort will overlap in a few run times’ points.

**Appendix A – Source Code**

1. Quoted from “Introduction to Algorithms 3rd Edition” [↑](#footnote-ref-1)