**Heap Sort Vs. Insertion Sort**

**1.Motivation**

*What is the importance of this problem in the world.*

Sorting is necessary for other important operations like searching. Without knowing what sorting algorithm should be used on a given program or operations, it has the potential to be extremely inefficient causing the programs and operations that require sorting to run slower / inefficiently, which makes the computers and phones run slower when running these operations and programs. This then creates a gap that without solving this problem could only solved by developing more advanced computers, but even this is a tireless effort since sorting / searching through data sets with millions of data points with an n^2 complexity instead of nlog(n) complexity would require constant technological innovation whereas solving this problem, so that we know to use the nlog(n) sorting algorithm could accomplish the same result.

*What do we want to do?*

We want to know when insertion sort and heap sort should be used instead of the other, in order to accomplish this we are going design and implement both of these sorting algorithms into c++, using assert statements and invariants to ensure correctness, and test them for the worst case, average case and best case arrays of size, gathering data for the time it takes to sort an n size array, then compare this data with the expected time complexity. This will tell us what size of array to use heap sort for and the size of the array to use insertion sort.

*Why?*

Finding the time complexity for insertion sort and heap sort will allow us to figure out when to use the appropriate sorting algorithm. Therefor allowing us to sort any given size of data more efficiently, essentially increasing the speed for anything that requires sorted data, such as searching.

**2.Background**

*What is the historical perspective of algorithms, when and why developed?*

The first ever algorithm was developed around 1700 BC by the Egyptians to multiply two numbers together (source 1). However the first sorting algorithm, Radix sort, was developed in 1901 by Herman Hollerith the founder of IBM, for the purpose of sorting the columns on the punch cards that were used by a machine he made to tally for the U.S. census (source 2). The rational for developing new algorithms is to solve new problems or solve old problems better or at least better in some aspect.

*Which algorithm is easier to understand, why.*

Insertion sort is easier to understand, because the only thing someone needs to understand is the algorithm, which is not complicated, whereas heap sort is still easy to understand, someone would also need to understand heaps and max heaps and need to know how to make max heaps with an algorithm then incorporate this into heap sort.

*How do they work?*

Insertion sort works by only looking at an array at its smallest, a size of 1, then taking the last element in this shorter version of the array and compare its value with the values of elements positioned before it, swapping the positions of the two elements until the value of the element is less than the last element in the shorter array, then repeat this process adding the next element in the array until the size of the array after increasing would be 1 larger than size of the original array.

Heap sort work by taking an array and converting it to a max heap, then swapping the first element in the array with the last, then making the array a max heap again while ignoring the last value in the array, repeat this until every element except 1 is being ignored.

*How do they work what is the complexity.*

Insertion sort is of complexity O(n^2) because for a worst case array, a reversed sorted array, takes 1 comparison for when only looking at the first 2 elements then 2 comparisons for the first 3 up until the last element which takes n-1 comparisons, adding all this comparisons together = n^2/2 – n / 2 which for any constant, c, c \* n^2 will be greater than or equal to = n^2/2 – n / 2

Heap sort is of complexity O(nlogn) because for a worst case array, each time an array is adjusted back to a max heap it takes lg(k), with k = to the number of elements not being ignored, adding all of these up gives log(n!) which for any constant, c, c \* n lg(n) is greater than or equal to log(n!)

**2. Procedures**

**Over all Structure of the Program**

*Classes with two sorting methods*

class HeapSrtCls

{

public:

void HeapSort(int arr[], int n);

void MaxHeapify(int arr[], int pos, int n);

void MakeMaxHeap(int arr[], int n);

};

class InsertionSrtCls

{

public:

void InsertionSort(int arr[], int n);

};

*Driver that uses these methods to sort the data*

int main()

{

const int MAX\_NUM\_ELEMENTS = 10000;

HeapSrtCls Heap;

InsertionSrtCls Insert;

int \* arrayH;

int \* arrayI;

int temp;

srand(time(NULL));

for(int i = 0; i <= MAX\_NUM\_ELEMENTS; i++)

{

delete [] arrayH;

delete [] arrayI;

arrayH = new int [i];

arrayI = new int[i];

for(int j = 0; j < i; j++)

{

temp = rand() % 10000 + 1;

arrayH[j] = temp;

arrayI[j] = temp;

}

Heap.HeapSort(arrayH, i);

Insert.InsertionSort(arrayI, i);

}

return 0;

}

*Graphical display for visualizing the layout of the classes/methods*

**Pseudo code with Correct Program Headers for usability**

void HeapSort(int arr[], int n)

{

MakeMaxHeap(arr(1..n), n)

for i = n down to 2

swap arr(1) and arr(i)

MaxHeapify(arr(1..n), 1, i-1)

}

void MaxHeapify(int arr[], int pos, int n)

{

index = 2\* pos

if index > n

return

if index < n

if arr(index + 1) > arr(index)

index += 1

if arr(index) > arr(pos)

swap arr(index) and arr(pos)

MaxHeapify(arr(1..n), index, n)

}

void MakeMaxHeap(int arr[], int n)

{

for i = floor (n / 2) down to 1

MaxHeapify(arr(1..n), i, n)

}

void InsertionSort(int arr[], int n)

{

for i = 2 up to n

j = i

while arr(j) < arr(j-1) and j – 1 > 0

swap arr(j) and arr(j-1)

j = j - 1

}

**Pre/Post Conditions, Invariants in Pseudo code major loop invariants**

PreCondition:

arr(1..n) is an array of numbers

void HeapSort(int arr[], int n)

{

MakeMaxHeap(arr(1..n), n)

for i = n down to 2

Invariant: arr(1..i) is max heap, arr(i+1..n) is sorted

swap arr(1) and arr(i)

MaxHeapify(arr(1..n), 1, i-1)

Invariant: arr(1..i-1) is max heap, arr(i..n) is sorted

}

PostCondition:

arr(1..n) is sorted

PreCondition:

arr(pos+1…n) has the max heap property

void MaxHeapify(int arr[], int pos, int n)

{

Invariant: arr(pos + 1…n) has the max heap property

index = 2\* pos

if index > n

return

if index < n

if arr(index + 1) > arr(index)

index += 1

if arr(index) > arr(pos)

swap arr(index) and arr(pos)

MaxHeapify(arr(1..n), index, n)

Invariant: arr(pos…n) has the max heap property

}

PostCondition:

arr(pos…n) has the max heap property

PreCondition:

arr(1..n) is an array of numbers

void MakeMaxHeap(int arr[], int n)

{

for i = floor (n / 2) down to 1

Invariant: arr(i+1…n) has max heap property

MaxHeapify(arr(1..n), i, n)

Invariant: arr(i…n) has the max heap property

}

PostCondition:

arr(1...n) has the max heap property

PreCondition:

arr(1) is sorted

void InsertionSort(int arr[], int n)

{

for i = 2 up to n

Invariant: arr(1…i-1) is sorted

j = i

while arr(j) < arr(j-1) and j – 1 > 0

swap arr(j) and arr(j-1)

j = j – 1

Invariant: arr(1…i) is sorted

}

PostCondition:

arr(1…n) is sorted

**Invariants as they will be implemented in the program**

There are 3 types of invariants in the pseudocode

1. That some part of an array is sorted

bool isSorted(arr(1..n), start, stop)

this function will itterate from start to stop checking to make sure that the next element in the array is >= the previous, if this is ever false the function returns false

assert(isSorted(arr(1..n),start,stop))

2. That some part of an array has the max heap property

bool isMaxHeap(arr(1..n), start, stop)

this function will itterate from start to stop checking that if an index has child nodes that those child nodes are <= the node being indexed, if this is ever false the function returns false

assert(isMaxHeap(arr(1..n), start, stop))

3. That an array is made of numbers

This does not need an assert statement since a c++ program will not compile if the functions that are expecting an array of integers get anything that is not promotable to an array of integers

**3.Testing Plan**

*Describe what kind of data will be tested?*

The number of comparisons made to sort an array is what is being tested, this number will be different for each of the sorting algorithms and for best, worst, and average case.

*is it real data or manufactured synthetic test*

It is a manufactured synthetic test since the data is not real and has no meaning since it is randomly generated, but the data will be a good representation of how real data will work.

*Boundary cases n=0 n> maximum size?*

With how my driver will be set up a pointer will be set = new array[i] and i iterates from 0 to the maximum size in a loop, so n can not be greater than the maximum size and n = 0 will be tested even though it behaves in the same was as a n=1 array and should not create any complications

*best case, worst case, average case data – random*

average case will take in random numbers generated with rand, once all of the random arrays are sorted counting all the comparisons for average case data, then before moving to the next size array the arrays with the sorted data will be sorted again counting the number of comparisons for best case data, then these arrays will be reversed and sorted counting the comparisons for the worst case data

*time charts in the form of graphs comparing the algorithms side by side by varying sizes of data*

*Determine when the efficeincy in behavior is visible for two algorithms for arrays of size larger than 1000*

The behavior should be apparent at any size greater than 1000 since in the graph above of the theoretical compelxity, Insertion sort with n^2 starts to grow much faster than heap sort with nlg(n) at an array size of 50, but nlg(n) is always <= n^2

*Find c, n corresponding to the data and algorithms*

nlg(n) <= n^2 for all values so c \* n^2 for any c >= 1 n >= 0

but the other way is more interesting

n^2 <= c \* nlg(n) one combination of c and n that works is c = 3 n > 9

*What size of data will be tested? what data will be tested.*

The size of data that will be tested is from 0 – 1000, the data being tested is the number of comparisons for best case, worst case, and average case for both sorting algorithms.

**Part02**

**Correctness of Program**

**Implementation**

For these two parts see the 4 submitted files

Part2WithAssert.cpp – The entire expected program with comments and assert statements

Part2NoAssert.cpp – The entire expected program with comments except the assert statements are commented out for accurate time calculations

assertData.txt – The output of Part2WithAssert.cpp used to create the graphs

noAssertData.txt - The output of Part2NoAssert.cpp used to create the graphs

**List of Problems**

When asking to make a max heap position 0 would not change because 2 \* 0 = 0, the child nodes for each node changed when the root node started at 0, so instead of the child nodes being at 2 \* i, and 2 \* i +1, they were actually at 2i + 1, and 2i + 2

Fixing other issues arising from switching from the first element in the array = 1 to the first element = 0

Adding how long the sorts took, I hade a dialema on how to approach this since getting the time for how long a function takes is simple but getting the time for how long certain parts of the function took only(without assert statements), would require a huge restructure of my entire program, so I decided to take the time for the sorts function with the assert statements on a separate file “Part2WithAssert.cpp”, and take the time without assert functions “Part2NoAssert.cpp”

**Performance**

Insertion Theoretical Average and worst is under Insertion Worst

Insertion Best is under Insertion Theoretical Best

Heap Average, Best, and Worst are all on top of each other and are noticibly above Insertion Best if zoomed in enough

**Conclussion**

After looking at the data for Heap sort and Insertion sorts Average case, Best Case, and Worst Case, for Average case Heap sort if faster for n >= 35 and Worst case Heap sort is faster for n >= 12, but for best case Insertion sort is faster for n >= 0

So Heap Sort is the faster sorting algorithm to use if the expected or average array size is over 35, but under that array size of 35 insertion sort is the faster sorting algorithm.

When comparing the data with the theoretical complexity of Heap sort and Insertion sort, we find that best, worst, and average cases of Heap sort are all slower than the theoretical complexity but are of the same order n(lgn), for insertion sort, worst case follows its theoretical complexity n^2 almost exactly, average case is faster than its theoretical complexity n^2 but still in order with it, best case is slower than the theoretical complexity of n but only by about 3 times, so there are of the same order.

**References**

“Write a Program to Reverse an Array or String.” *GeeksforGeeks*, 8 Sept. 2020, www.geeksforgeeks.org/write-a-program-to-reverse-an-array-or-string/.

**Sources:**

*Source 1*

[Timeline of algorithms - Wikipedia](https://en.wikipedia.org/wiki/Timeline_of_algorithms)

*Source 2*

[Byte Sized Episode 3: The First Ever Sorting Algorithm - DEV Community](https://dev.to/bytesized/byte-sized-episode-3-the-first-ever-sorting-algorithm-55kh)

*Soruce 3: reverse function*

[Write a program to reverse an array or string - GeeksforGeeks](https://www.geeksforgeeks.org/write-a-program-to-reverse-an-array-or-string/)