**Amarilda Celhaka**

**CMP SCI 3130**

**Project #2**

**10/23/2018**

**Analysis of the Sorting Algorithms**

**Selection Sort**

|  |  |  |
| --- | --- | --- |
| Type of array | Size | Measured Time (ms) |
| Random Array | 100 | 0.018336 |
|  | 1000 | 1.53706 |
|  | 10000 | 248.71 |
| Semi-Sorted Array | 100 | 0.019394 |
|  | 1000 | 1.58184 |
|  | 10000 | 151.73 |
| Sorted | 100 | 0.01622 |
|  | 1000 | 1.19325 |
|  | 10000 | 151.73 |

Time complexity for Selection Sort for every case is is θ(n^2)

Selection sort is an algorithm that will traverse through the entire array of elements and will check for the smallest element. It will find the smallest element and position it on the beginning of the array if we are sorting in increasing order. So, for any time its being iterated through the array the number of the sorted elements increases and the unsorted elements gets smaller. For any element sorted, want check again if its in right place.

Selection sort for any case it will require about the same time because the number of checks will be the same, it will go over each element and the number of comparison will decrease by one for any time an iteration happens.

Since these comparisons will be performed every time, the running time will not have much of a difference when compared to a sorted array, random array or semi-sorted. A random array will require a little more time because the number of swaps will be higher compared to a semi-sorted or sorted array. Our data also show that the running time of selection sort for a semi sorted array is almost the same as the running time of a second array, approximately 151 ms for a 10,000 element array. The running time is very small when the array is of size 100 or 1000. We notice that the running time for these arrays goes from 0.002~2ms. This time is very small compared for the running time of a bigger array. Something that also need to be pointed out is that the difference between a 10,000 sorted and semi-sorted with a random array differs with about 100ms which is a considerable amount of time.

Graph also displays how the data correlates to one another. Cost #of times

int j;

for (int i = 0; i < size - 1; i++){ c1 n-1+1=n

int min = i; //setting min as the position of I c2 n-1

for (j = i + 1; j < size; j++){ c3 ∑(n-i+1)

if (array[j] < array[min]){ c4 ∑(n-i)=N

min = j; c5 n < N

exchange(&array[min], &array[i]);

T(n) = c1\*n + c2\*(n-1) + c3\*∑(n-i+1)+c4\*∑(n-i)+c5\*n

= n(c1+c2) -c2 + c3(n+1)(n-1) + c3(n+2)/2 \* n-1 + c4\*n +c4((n+2)/2 \* n-1) + c5\*n

= An2 + Bn + c

T(n) is O(n^2) for the worst case,it will require a little bit more time than average case

which is T(n) is θ(n^2) and for the best case will be a lower bound T(n) is Ω(n^2)

All the above proof is clearly displayed on the graph, where the worst case graph is growing faster than the average case(semi-sorted) and semi-sorted is requiring a little more time than the best case, a sorted array.

**Insertion Sort**

|  |  |  |
| --- | --- | --- |
| Type of array | Size | Measured Time (ms) |
| Random Array | 100 | 0.009873 |
|  | 1000 | 0.988737 |
|  | 10000 | 108.655 |
| Semi-Sorted | 100 | 0.003173 |
|  | 1000 | 0.134347 |
|  | 10000 | 17.6351 |
| Sorted | 100 | 0.001058 |
|  | 1000 | 0.005289 |
|  | 10000 | 0.042667 |
|  |  |  |

Time complexity of Insertion Sort

Tbest(n) is θ(n).

Tavg (n) is θ(n^2).

Tworst (n) is θ(n^2).

Insertion Sort will still go through the array and will compare each element with another. So it will walk through the first two elements and compare which one is bigger and swap elements if it is necessary. It will do so until the entire array is sorted. The number of comparisons different from selection sort will be different each time. We compare each element every time with the key element which will be every element of the array from 1-n.

From our tests and measurements, we notice that the running time of a sorted array will be very fast for all array sizes. The time difference of the time between one array to another will be less than milliseconds. From out data seems that the correlation between the time and the array size will be linear. This is also shown on our graph where the line for the running time of insertion sort is linear, so time complexity for the best case, a sorted array will be T(n) is Ω(n)

A big difference we see from the data collected for a sorted array and a random array. Because the number of comparisons and swaps is higher since the array is created with random numbers, the time it will require to sort the array is higher. This is also supported from our measurement. My insertion sort algorithm sorted a random 100 element array for about 0.0098 ms, and sorted a 10,000 array for 108 ms. So, the time will be almost 107 ms higher. Looking at the graph it also reflects that the time complexity of Insertion sort for the worst case will be O(n^2). The graph for the random array has a parabolic shape, the same it is for the average case, a semi-sorted array. The line of a semi-sorted array in our graph shows that it is an upper and lower bound. Time complexity for an average case will be θ(n^2).

**Bubble Sort Without Swaps**

|  |  |  |
| --- | --- | --- |
| Type of array | Size | Measured Time (ms) |
| Random Array | 100 | 0.1928881 |
|  | 1000 | 7.8429 |
|  | 10000 | 792.844 |
| Semi-Sorted Array | 100 | 0.025388 |
|  | 1000 | 1.94715 |
|  | 10000 | 179.371 |
| Sorted | 100 | 0.013399 |
|  | 1000 | 1.2391 |
|  | 10000 | 141.118 |
|  |  |  |

Time complexity of Bubble Sort without swaps is the same for 3 cases

T(n) is θ(n^2).

Bubble Sort is a very simple sorting algorithm. With two for loops, this algorithm, for the first iteration of the first for loop, will go through the entire array and check for each adjacent element if the a[j] > a[j+1]. If yes than the elements will swap with one another. It will do so until the number of comparisons will be 1. This is like selection sort. For an array of the same size the number of comparisons for selection and bubble sort will be the same.

What I noticed on bubble Sort is that for a random array (the worst case) the running time for a 10,000 will be the maximum of the measured time for all sorting algorithms. It will be approximately 0.8 seconds. The time it takes to measure the time for a 100, to 1000, and to 10,000 will grow with the same rate as polynomials. This is also reflected on the graph where the running time for every case will be θ(n^2). For a random array, semi-sorted and sorted running time will drastically grow when the size grows as a factor of 10.

For a worst case the graph will be growing faster than best and average case, however all of them grow with the same rate as polynomials.

for (int i = 0; i < size - 1; i++){ n-1+1

for (int j = 0; j < size - i - 1; j++){ ∑n-i-1+1

if (array[j] > array[j + 1]){ ∑n-i-1

//exchange(&array[j], &array[j + 1]);

T(n) = c1\*n+ c2\*∑(n-i)+c3\*∑(n-i-1)

= C1\*n + c2(n+1)(n-1) + c2(n+2)/2 \* n-1 + c3\*n +c3((n+2)/2 \* n-1)

= An2 + Bn + c

Theoretically we proofed that the Time Complexity for Bubble Sort will be T(n) is θ(n^2). Our measured data as well as graph supports the theoretical results.

**Bubble Sort With Swaps**

|  |  |  |
| --- | --- | --- |
| Type of array | Size | Measured Time (ms) |
| Random Array | 100 | 0.070523 |
|  | 1000 | 6.38589 |
|  | 10000 | 714.273 |
| Semi-Sorted | 100 | 0.016925 |
|  | 1000 | 1.95314 |
|  | 10000 | 215.276 |
| Sorted | 100 | 0.000705 |
|  | 1000 | 0.002821 |
|  | 10000 | 0.023978 |

Bubble sort with Swaps

Tbest(n) is θ(n).

Tavg (n) is θ(n^2).

Tworst (n) is θ(n^2).

Bubble Sort with Swaps algorithms is the same algorithms as bubble sort but it has another feature added to make it more efficient. Algorithm compares each case the number of swaps. This is done because if the number of swaps will be zero than the array will be sorted, so the code can finish executing instead of checking until it reaches the last element of the array. We noticed that for a worst case scenario the running time of the bubble sort with and without swaps does not differ at all, a difference of tenth of millisecond’s, which for a 714ms value will not make a big difference. However, for a sorted array the time will very small. And it makes sense because once it iterates once through the entire array and the number of swaps will be 0, than that will be an indication that the array is sorted and the executing will end, taking fewer milliseconds. This algorithm is very efficient for a sorted array. Our graph clearly demonstrates our measured time and running complexity which for the best case, a sorted array will be O(n)

**Quick Sort**

|  |  |  |
| --- | --- | --- |
| Type of array | Size | Measured Time (ms) |
| Random Array | 100 | 0.019394 |
|  | 1000 | 0.278567 |
|  | 10000 | 3.14877 |
| Semi-Sorted Array | 100 | 0.15233 |
|  | 1000 | 3.6993 |
|  | 10000 | 7.67823 |
| Sorted | 100 | 0.137873 |
|  | 1000 | 11.1346 |
|  | 10000 | 89.25932 |
|  |  |  |

Worst case Ο(n^2)

Best case θ(nlogn)

Average case O(nlogn)

Quick sort algorithm is a conquer and divide sorting algorithm which will sort the elements by diving the array around the pivot element.

Uniquely from the previous sorting algorithms we analyzed worst case for quick sort will be the when the array is sorted, or elements will be the same. Because if the array is sorted the pivot element will be the rightmost or leftmost element, portioning the array with a subarray of size 0 and another one of size n-1. This will require more time for our algorithm to sort all elements. So the running time for the worst case will be Ο(n^2). Graph displays graphically our data, which for a sorted array the running time will be very high compared to the average case or best case.

For the worst and average case, running time will be about the same. This could be supported by our data on the table as well as the graph. For the average case the running time line will be an upper bound, and time complexity will be O(nlogn)

Instead for a best case will be upper and lower bound of nlogn which is θ(nlogn).

quickSort(int array[], int p, int r){

if (p < r) //if pivot < righmost element, pivot is in the right position

{

q = partion(array, p, r); //call partion function which will put pivot in the right

quickSort(array, p, q - 1); //call quickSort for the two new subarrays

quickSort(array, q + 1, r);

}

Quick sort is a recursive algorithm, and the running time will depend by the array input. For the worst case since the elements will split into very unequal sizes the recurrence will look like:

T(n) = T(n-1) + θ(n) . Solving it with the backwards substitution we will get that the recurrence will be θ(n^2).

For the best case instead, we will have the same recurrence as the merge sort. Because in an average case the array will be split into two equal subproblems.

T(n) = 2T(n/2) + θ(n). Using again master theorem as in merge sort recurrence, we get that the Time will be θ(nlogn).

Theoretical analysis is also supported by experimental data, and graphically displayed on the above graph.

**Merge Sort**

|  |  |  |
| --- | --- | --- |
| Type of array | Size | Measured Time (ms) |
| Random Array | 100 | 0.097322 |
|  | 1000 | 0.885068 |
|  | 10000 | 8.0245 |
| Semi-Sorted | 100 | 0.116363 |
|  | 1000 | 0.786335 |
|  | 10000 | 7.55023 |
| Sorted | 100 | 0.114247 |
|  | 1000 | 0.771525 |
|  | 10000 | 8.48396 |
|  |  |  |

**Merge Sort Time Complexity, for all cases is** **θ(nlogn)**

Merge sort, in the same way as Quick Sort is a divide and conquer algorithm, which will divide the array in two parts, with the same size, and merge these two halves. The merge function then will sort this two subarrays.

Merge sort is a very efficient algorithm, for each case running time is almost the same. As we can see on the table the running time for a 10,000 array will be approximately 8 ms. This running time was never seen in the previous algorithms, making merge sort a very efficient algorithm for each case. For a 1000 array the measured time is still the same with fewer millisecond’s of difference, the running time will be about 0.8 milliseconds.

mergeSort(array, p, r)

if (p < r)

{

q = (p + r) / 2; //q will be the mid element

mergeSort(array, p, q);

mergeSort(array, q + 1, r);

merge(array, p, q, r);

}

Because it is a recursive algorithms, calls itself twice in the same function, it will have two subproblems with size n/2 each since the array is split in halves. The time complexity will be

T(n) = 2T(n/2) + θ(n). Solving it using recurrence Tree or master method

We find that : 2 = 2^1, so the Time Complexity will be T(n) is θ(nlogn) for each case.

Theoretical analysis is also supported by our experimental analysis and graphically