Devin Gendron
7/8/18
HW2

Problem 1: *(3 points)* Give the asymptotic bounds for T(n) in each of the following recurrences. Make your bounds as tight as possible and justify your answers. *Assume the base cases T(0)=1 and/or T(1) = 1.*

a) $T(n)=2T(n-2)+1$

Not in Master Method form ($T(n) = aT(n/b) + f(n)$) – use:

Muster Method ($T(n) = aT(n-b) +f(n)$)

$a = 2, b = 2, f(n) = 1$

$f(n) = n^0 = 1 \rightarrow d = 0$

Thus, $f(n) = \Theta(n^0)$.

Since $a > 1$, use $\Theta(n^d a^{n/b})$:

$\Theta(n^0 * 2^{n/2}) = \Theta(1 * 2^{n/2}) = \Theta(2^{n/2})$

Therefore, $T(n) = \Theta(2^{n/2})$

b) $T(n)=T(n-1)+n^3$

Not in Master Method form ($T(n) = aT(n/b) + f(n)$) – use:

Muster Method ($T(n) = aT(n-b) +f(n)$)

$a = 1, b = 1, f(n) = n^3$

$f(n) = n^3 \rightarrow d = 3$

Thus, $f(n) = \Theta(n^3)$.

Since $a = 1$, use $\Theta(n^{d+1})$:

$\Theta(n^{3+1}) = \Theta(n^4)$

Therefore, $T(n) = \Theta(n^4)$

c) $T(n)=2T(n/6)+2n^2$

Not in Muster Method form (T(n) = aT(n-b) +f(n)) – use:

Master Method form (T(n) = aT(n/b) + f(n))

a = 2, b = 6, $f(n) = 2n^2$

$\log_b a = \log_6 2 = 0.38685\ldots$

$n^{\log_6 2} = n^{0.38685}$

Case 3: if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$ → $f(n) = \Omega(n^{0.38685 + \varepsilon})$ → verify regularity condition:

$a*f(n/b) <= c*f(n)$

$2(n/6) <= c*(2n^2)$

$c = 2(n/6) * (1/ 2n^2)$

$c = (n / 6n^2)$

$c = (1 / 6n)$

$c = 1/6$ is a solution ($c < 1$)

Thus, $T(n) = \Theta(n^2)$

Problem 2: *(6 points)* The quaternary search algorithm is a modification of the binary search algorithm that splits the input not into two sets of almost-equal sizes, but into four sets of sizes approximately one-fourth.

a) Verbally describe and write pseudo-code for the quaternary search algorithm.

Quaternary Search Algorithm Description:

The Quaternary Search Algorithm, as described in the prompt, is a modification of the binary search algorithm. As we know, the binary search algorithm splits the sorted data set into two pieces and then repeats that process. It does this while continually checking until eventually the value searched for is found, or the location of where it would be located. For the Quaternary Search Algorithm, it splits this data set not into two, but four different set sizes. Just as the binary search algorithm, it will check the middle element of each of these data sets and return the value if it is found or continue its comparisons (< or >) and recursively call the Quaternary Search Algorithm until it is found or its position is found.

PseudoCode:

Citation: http://ijirt.org/master/publishedpaper/IJIRT143908_PAPER.pdf

https://www.geeksforgeeks.org/binary-search/

```
/*

This algorithm is basically an adaptation of the binary search code from geeksforgeeks binary search.

*/

QuaternarySearchAlgo(int data[], int value, low, high)

        int beg = low;

        int part1 = data.size()/4;

        int part2 = data.size()/2;

        int part3 = part1 + part2;

        int end = high;

        if(data[beg] == value)

                return data[beg];

        else if(data[part1] == value)

                return data[part1];

        else if(data[part2] == value)

                return data[part2];

        else if(data[part3] == value)

                return data[part3];

        else if(data[end] == value)

                return data[end];

        else if(data[start] < value < data[part1])
```

```
        return QuaternarySearchAlgo(data, value, beg, part1);
    else if(data[part1] < value < data[part2])
        return QuaternarySearchAlgo(data, value, part1 + 1, part2);
    else if(data[part2] < value < data[part3])
        return QuaternarySearchAlgo(data, value, part2 + 1, part3);
    else if(data[part3] < value < data[end])
        return QuaternarySearchAlgo(data, value, part3 + 1, end);
    else
        return 1;
```

b) Give the recurrence for the quaternary search algorithm

$T(n) = T(n/4) + c$

c) Solve the recurrence to determine the asymptotic running time of the algorithm. How does the running time of the quaternary search algorithm compare to that of the binary search algorithm.

Not in Muster Method ($T(n) = aT(n-b) + f(n)$) – use:

Master Method form ($T(n) = aT(n/b) + f(n)$)

$a = 1, b = 4, f(n) = c$

$\log_b a = \log_4 1 = 0$

$n^0 = 1$

$T(n) = 1T(n/4) + 1$

Case 2: if $f(n) = \Theta(n^{\log_b a})$ → then $T(n) = \Theta(n^{\log_b a} \log n)$ → $\Theta(n^0 \log n)$ → $\Theta(\log n)$

Thus, $T(n) = \Theta(\log n)$

When compared, the running time of the Quaternary Search Algorithm and the Binary Search Algorithm are equal.

Problem 3: *(6 points)* Design and analyze a **divide and conquer** algorithm that determines the minimum and maximum value in an unsorted list (array).

a) Verbally describe and write pseudo-code for the min_and_max algorithm.

min_and_max Algorithm Description:

This algorithm is unsorted, so it must work through the data set to determine the min and max. Since this is a divide and conquer algorithm, the algorithm must also be broken up into smaller subsets to determine the correct answer through recursion. The algorithm starts with the base cases to check if its length is 1 or 2, then uses recursion to determine the min and max with some more checking. Since we are looking for the min and max, it would be easier if our array was working with two values. This is where divide comes in to play. We will use recursion until an array of 1 or 2 values is found, and then it will return from there and work up the recursion tree until a proper min and max are found. In my version of min_and_max,

PseudoCode:
Citations:
https://www.youtube.com/watch?v=Cw5cUQp8pgo
https://www.geeksforgeeks.org/maximum-and-minimum-in-an-array/
https://www.youtube.com/watch?v=EHRL2LbS5LU
https://www.youtube.com/watch?v=Dj0TJ_Eko1c

/*
I originally attempted to write this algorithm without doing any research to see how far I could get. I came up with similar logic in regards to solving for one element, then two, then more than 2. I was foolishly returning an array rather than a pointer to an array of a min and max (first position was min, second was max). I was putting my code through my IDE to make sure it was sufficient, but I was having issues. After researching the algorithm, I was able to optimize it with bits and pieces of these sources to get it working. For example, using a struct like in the geeksforgeeks website.
*/

```
struct minmax {
    int min;
    int max;
}

struct minmax min_and_max(int array[], int beg, int end)
    if (beg == end)
            struct minmax mm;
            mm.min = array[beg];
            mm.max = array[end];
    else if (beg == end - 1)
```

```
        struct minmax mm;
        if(array[0] < array[1])
                mm.min = array[beg];
                mm.max = array[end];
        else
                mm.min = array[end];
                mm.max = array[beg];
    else
        int midpt = (beg + end)/2;
        struct minmax lower;
        struct minmax upper;
        struct minmax mm;
        lower = min_and_max(array, beg, midpt);
        upper = min_and_max(array, midpt + 1, end);
        if(lower.max > upper.max)
                mm.max = lower.max
        else
                mm.max = upper.max
         if(lower.min > upper.min)
                mm.min = lower.min
        else
                mm.min = upper.min

    return mm;
```

b) Give the recurrence.
$T(n) = 2T(n/2) + 2$

c) Solve the recurrence to determine the asymptotic running time of the algorithm. How does the theoretical running time of the recursive min_and_max algorithm compare to that of an iterative algorithm for finding the minimum and maximum values of an array.

Not in Muster Method ($T(n) = aT(n-b) + f(n)$) – use:
Master Method form ($T(n) = aT(n/b) + f(n)$)
$a = 2, b = 2, f(n) = 2$
$\log_b a = \log_2 2 = 1$
$n^1 = n$
$n > 2n^0$, so case 1 is used.
Case 1: if $f(n) = O(n^{\log_b a - \varepsilon})$ → then $T(n) = \Theta(n^{\log_b a})$ → $\Theta(n^{\log_2 2})$ → $\Theta(n^1)$
Thus, $T(n) = \Theta(n)$

Iterative Algorithm: https://www.geeksforgeeks.org/maximum-and-minimum-in-an-array/

I used Method 1 for the iterative algorithm. In that example we see that the big O complexity is O(n) with a asymptotic runtime of about $T(n) = \Theta(n) + c$. Thus, we can see that the runtimes of the recursive and iterative min_and_max algorithm are equivalent.

Problem 4: *(5 points)* Consider the following pseudocode for a sorting algorithm.

```
StoogeSort(A[0 ... n - 1])
        if n = 2 and A[0] > A[1]

                swap A[0] and A[1]

        else if n > 2

            m = ceiling(2n/3)

            StoogeSort(A[0 ... m - 1])

            StoogeSort(A[n - m ... n - 1])

            Stoogesort(A[0 ... m - 1])
```

a) Would STOOGESORT still sort correctly if we replaced k = ceiling(2n/3) with k = floor(2n/3)? If yes prove if no give a counterexample. (Hint: what happens when n = 4?)

Let's check. If we input an array of size 4 for StoogeSort, it will pass the first if statement since it is not equal to 2. It will then enter the else if statement since n is greater than 2. Once inside, a "floor" will be calculated which will be m = floor(2*4/3). m will thusly, be equal to 2 since we are working with integers. StoogeSort will then be recursively called with the first two elements from the original array. StoogeSort will then check if the arrays size is equal to 2 (it is) and will swap the values if the first element is greater than the second. Let's say the array data is {3, 1, 5, 2}. So since 3 > 1, they will be swapped and the array will be {1, 3, 5, 2}. StoogeSort will then do this again for the next two elements – so since 5 > 2, they will be swapped and the array will be {1, 3, 2, 5}. Lastly, StoogeSort is called again on the first two elements, but nothing will be done here. As we can see, no, StoogeSort does not sort correctly when k = floor(2n/3). (Although there are cases where it will be lucky and work).

b) State a recurrence for the number of comparisons executed by STOOGESORT.
$T(n) = 3T(2n/3) + c$

c) Solve the recurrence to determine the asymptotic running time.
Not in Muster Method ($T(n) = aT(n-b) + f(n)$) – use:
Master Method form ($T(n) = aT(n/b) + f(n)$)
$a = 3$, $b = 3/2$, $f(n) = c$
$\log_b a = \log_{3/2} 3 = 2.709511\ldots$
$n^{2.71} > f(n)$, so case 1 is used.
Case 1: if $f(n) = O(n^{\log_b a - \varepsilon})$ → then $T(n) = \Theta(n^{\log_b a})$ → $\Theta(n^{\log_{3/2} 3})$ → $\Theta(n^{2.71})$
Thus, $T(n) = \Theta(n^{2.71})$

CS 325 - Homework Assignment 2

Problem 5: *(10 points)*

a) Implement STOOGESORT from Problem 4 to sort an array/vector of integers. Implement the algorithm in C++, your program should compile with g++ stoogesort.cpp. Your program should be able to read inputs from a file called "data.txt" where the first value of each line is the number of integers that need to be sorted, followed by the integers (like in HW 1). The output will be written to a file called "stooge.out".

Submit a copy of all your code files and a README file that explains how to compile and run your code in a ZIP file to TEACH. We will only test execution with an input file named data.txt.

b) Now that you have proven that your code runs correctly using the data.txt input file, you can modify the code to collect running time data. Instead of reading arrays from a file to sort, you will now generate arrays of size n containing random integer values and then time how long it takes to sort the arrays. We will not be executing the code that generates the running time data so it does not have to be submitted to TEACH or even execute on flip. Include a "text" copy of the modified code in the written HW submitted in Canvas. You will need at least seven values of t (time) greater than 0. If there is variability in the times between runs of the algorithm you may want to take the average time of several runs for each value of n.

```
/*
 * Author: Devin Gendron
 * Assignment: Homework 2 - Question 5: Stooge Sort
 * Date Due: July 8st - 11:59PM
 */
#include <iostream>
#include <string>
#include <cmath>
#include <cstdlib>
#include <vector>
#include <ctime>
#include <time.h>

/*CITATION:: Discussions in our group chat helped me understand how to fully implement
stooge sort
            as well as the pseudocode.
 */
/*CITATION:: https://www.geeksforgeeks.org/stooge-sort/
 * This citation was used when I hit a wall with stoogeSort.  For some reason, my data
would be
 * returned unsorted.  I commented out the function and just swapped some elements and
when I
 * tested my code, the values were swapped.  Thus, I knew there was an error with
stoogeSort.
 * I eventually realized the error was with my m variable.  I originally attempted to
use the
 * same version as the pseudocode and this was where I hit the snag.  Using the method
used by the
 * link, I could properly ascertain the size of n.  I then used this in my if
statements to be consistent.
 */
void stoogeSort(int data[], int start, int end) {
    if(end - start + 1 == 2 && data[start] > data[end]) {
```

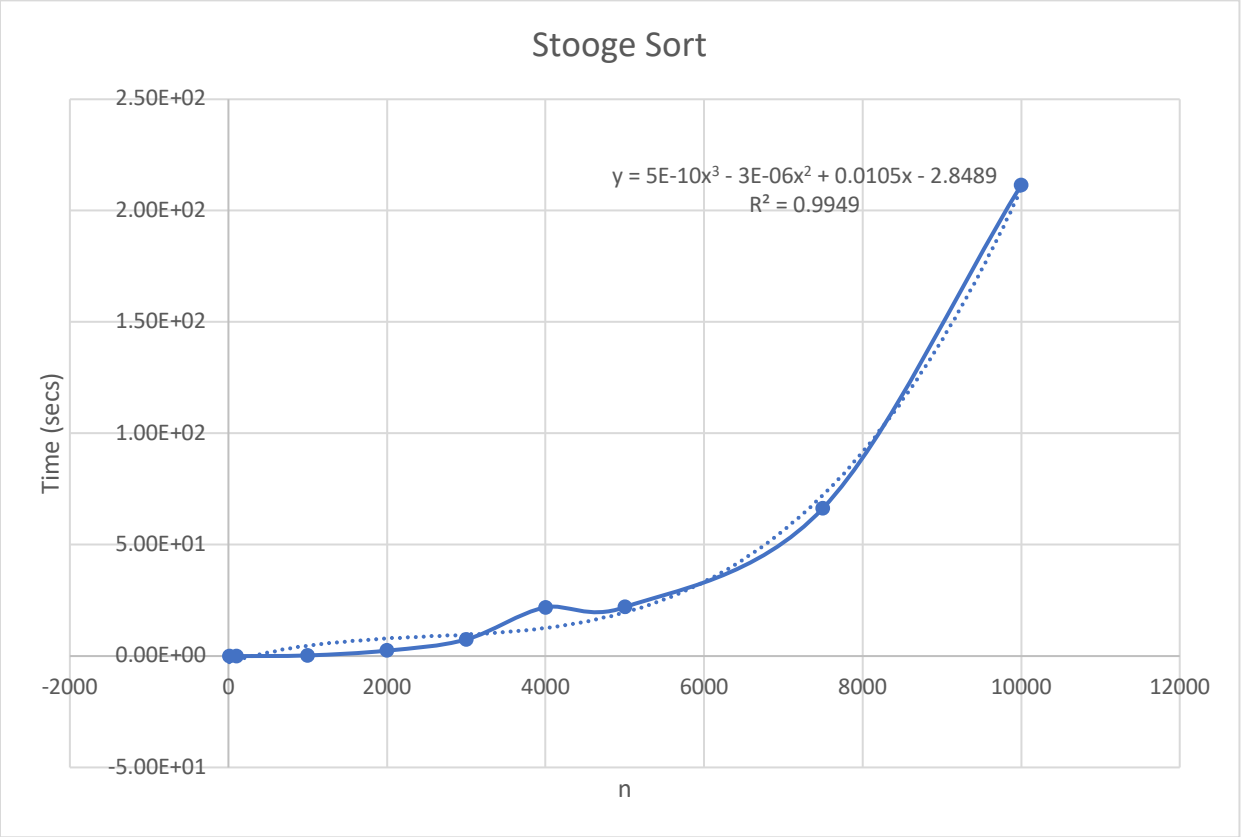
```
        int temp;
        temp = data[start];
        data[start] = data[end];
        data[end] = temp;
    }
    else if (end - start + 1 > 2) {
        int m;
        //https://stackoverflow.com/questions/2745074/fast-ceiling-of-an-integer-
division-in-c-c
        //m = (2 * end + 1 + 2) / 3;
        m = (end - start + 1) / 3;

        //stoogeSort(data, start, m - 1 + start);
        //stoogeSort(data, end - m + 1, end);
        //stoogeSort(data, start, m - 1 + start);
        stoogeSort(data, start, end - m);
        stoogeSort(data, start + m, end);
        stoogeSort(data, start, end - m);

    }
}

int main() {
    //set for randomized numbers
    srand(time(NULL));

    //variables
    std::vector <int> vec;
    int num_to_sort;
    int choice;
    int random;
    bool menu = false;
    double runtime = 0.0;

    //while loop menu
    while(menu == false) {
        //Print Options
        std::cout << "Please Choose your Preferred Size of n: " << std::endl;
        std::cout << "1. 10" << std::endl;
        std::cout << "2. 100" << std::endl;
        std::cout << "3. 1000" << std::endl;
        std::cout << "4. 2000" << std::endl;
        std::cout << "5. 3000" << std::endl;
        std::cout << "6. 4000" << std::endl;
        std::cout << "7. 5000" << std::endl;
        std::cout << "8. 7500" << std::endl;
        std::cout << "9. 10000" << std::endl;

        //read in user input
        std::cin >> choice;

        //fill vector based off user input
        switch(choice) {
            case 1 : //generates 5,000 n
                for(int i = 0; i < 10; i++) {
                    random = (rand() % 10000 + 1);
                    vec.push_back(random);
                    //std::cout << "data = " << data[i] << std::endl;
                }
                num_to_sort = 10;
                menu = true;
```

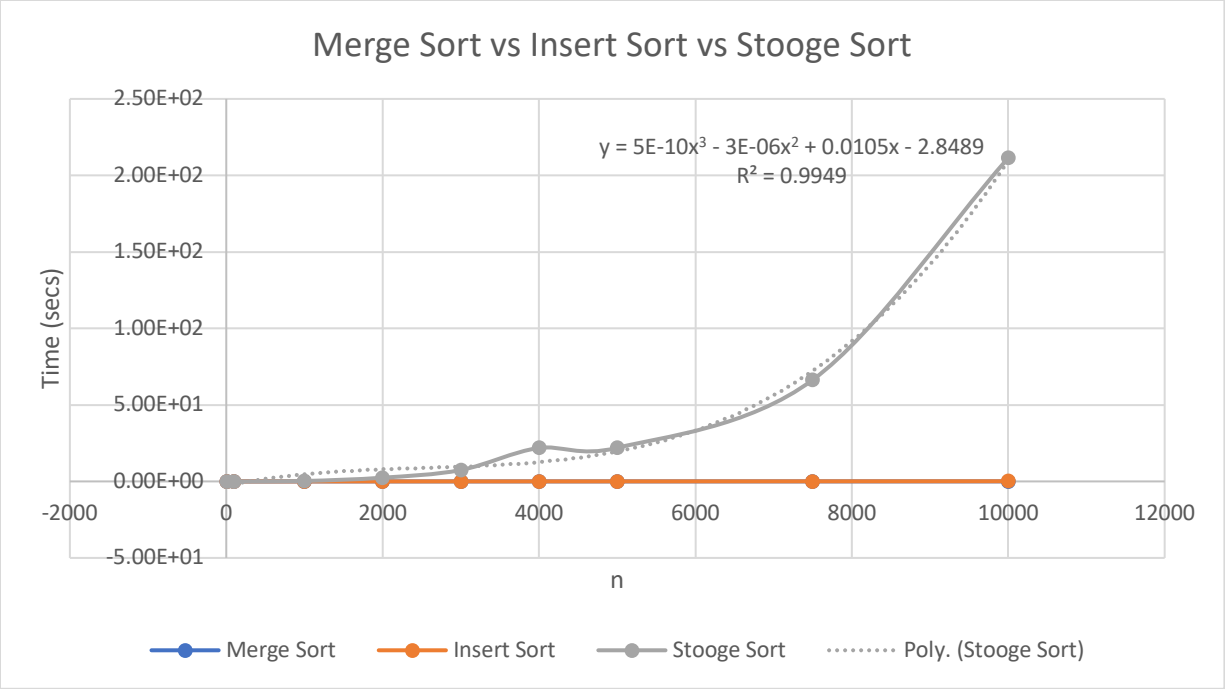
```
            break;
        case 2 : //generates 10,000 n
            for(int i = 0; i < 100; i++) {
                random = (rand() % 10000 + 1);
                vec.push_back(random);
            }
            num_to_sort = 100;
            menu = true;
            break;
        case 3 : //generates 15,000 n
            for(int i = 0; i < 1000; i++) {
                random = (rand() % 10000 + 1);
                vec.push_back(random);
            }
            num_to_sort = 1000;
            menu = true;
            break;
        case 4 : //generates 20,000 n
            for(int i = 0; i < 2000; i++) {
                random = (rand() % 10000 + 1);
                vec.push_back(random);
            }
            num_to_sort = 2000;
            menu = true;
            break;
        case 5 : //generates 30,000 n
            for(int i = 0; i < 3000; i++) {
                random = (rand() % 10000 + 1);
                vec.push_back(random);
            }
            num_to_sort = 3000;
            menu = true;
            break;
        case 6 : //generates 40,000 n
            for(int i = 0; i < 4000; i++) {
                random = (rand() % 10000 + 1);
                vec.push_back(random);
            }
            num_to_sort = 4000;
            menu = true;
            break;
        case 7 : //generates 50,000 n
            for(int i = 0; i < 5000; i++) {
                random = (rand() % 10000 + 1);
                vec.push_back(random);
            }
            num_to_sort = 5000;
            menu = true;
            break;
        case 8 : //generates 60,000 n
            for(int i = 0; i < 7500; i++) {
                random = (rand() % 10000 + 1);
                vec.push_back(random);
            }
            num_to_sort = 7500;
            menu = true;
            break;
        case 9 : //generates 70,000 n
            for(int i = 0; i < 10000; i++) {
                random = (rand() % 10000 + 1);
                vec.push_back(random);
```

```
                }
                num_to_sort = 10000;
                menu = true;
                break;
            default : //if error
                std::cout << "Error: please choose 1 - 9 only." << std::endl;
                menu = false;
                break;
        }
    }

    std::cout << "Stooge Sort Commencing" << std::endl;
    std::cout << "" << std::endl;

    //set array
    int data[num_to_sort];

    //fill array from vector
    for(int i = 0; i < num_to_sort; i++)
    {
        data[i] = vec[i];
    }

    //CITATION: https://stackoverflow.com/questions/5248915/execution-time-of-c-
program
    //start clock
    clock_t begin = clock();

    //stooge sort function
    stoogeSort(data, 0, num_to_sort - 1);

    //end clock
    clock_t end = clock();

    //compute time
    runtime = (double)(end - begin) / CLOCKS_PER_SEC;

    //print data
    std::cout << "n = " << num_to_sort << std::endl;
    std:: cout << "Stooge Sort RunTime: " << runtime << std::endl;
    std::cout << "" << std::endl;
    std::cout << "Stooge Sort Read and Write Program Ending" << std::endl;

    return 0;
}
```

c) Plot the running time data you collected on an individual graph with n on the x-axis and time on the y-axis. You may use Excel, Matlab, R or any other software. Also plot the data from Stooge algorithm together on a combined graph with your results for merge and insertion sort from HW1.

n	Merge Sort	Insert Sort	Stooge Sort
10	1.80E-05	3.00E-06	6.00E-06
100	0.000175	2.00E-05	0.001819
1000	0.00192	0.001668	0.286595
2000	0.003668	0.005817	2.43782
3000	0.005521	0.0126	7.45989
4000	0.006727	0.021588	21.9188
5000	0.00909	0.031365	21.962
7500	0.012928	0.061442	66.4078
10000	0.018193	0.105959	211.445





d) What type of curve best fits the StoogeSort data set? Give the equation of the curve that best "fits" the data and draw that curve on the graphs of created in part c). How does your experimental running time compare to the theoretical running time of the algorithm?

According to the data, our graph's trendline is x^3. So, the curve that best fits this data set is similar to a cubic parabola where n >= 0. I would consider this an exponential curve that just increases at a faster rate (runs slower) than say Insert Sort which is x^2.

The recurrence would be:
$T(n) = 3T(2n/3) + \Theta(1)$

Asymptotic Runtime Complexity:
-Use Master method: $T(n) = aT(n/b) + f(n)$
$a = 3, b = 3/2, f(n) = 1;$

$\log_b a = \log_{3/2} 3 = 2.709511\ldots$

$n^{\log_b a} = n^{\log_{3/2} 3} = n^{2.71}$

Compare root to leaf: $n > f(n)$ → use Case 1

Case 1:

if $f(n) = O(n^{\log_b a - \varepsilon})$ → then $T(n) = \Theta(n^{\log_b a})$ → $\Theta(n^{\log_{3/2} 3})$ → $\Theta(n^{2.71})$

Thus, $T(n) = \Theta(n^{2.71})$

As we can see, the theoretically runtime ($n^{2.71}$) is a bit faster than what we observed in our experimental runtime (n^3). However, a n^3 vs $n^{2.71}$ difference in runtime isn't too bad considering the small data size.

All Citations:

https://www.youtube.com/watch?v=Cw5cUQp8pgo

https://www.geeksforgeeks.org/maximum-and-minimum-in-an-array/

https://www.youtube.com/watch?v=EHRL2LbS5LU

https://www.youtube.com/watch?v=Dj0TJ_Eko1c

https://www.geeksforgeeks.org/maximum-and-minimum-in-an-array/

http://ijirt.org/master/publishedpaper/IJIRT143908_PAPER.pdf

https://www.geeksforgeeks.org/binary-search/