Hsiang Lo

Homework #2

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

Given that the structure of this function is in the form of T(n) = aT(n-b) + f(n), we can "decrease and conquer" it by using and applying the Muster Method.

Provided that a = 2, b = 2 and f(n) = 1 = O(n^0), d = 0, and since a > 1, we can conclude from this information that the equation should take the form of,

T(n) = O(n^d * a^(n/b))

T(n) = O(n^0 * a^(n/2))

T(n) = O(2^(n/2))

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

Given that the structure of this function is in the form of T(n) = aT(n-b) + f(n), we can "decrease and conquer" it by using and applying the Muster Method.

Provided that a = 1, b = 1 and f(n) = n^3 = O(n^3), d = 3, and since a = 1, we can conclude from this information that the equation should take the form of,

T(n) = O(n^(d+1))

T(n) = O(n^ 4)

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

Given that the function is in the form of T(n) = aT(n/b) + f(n), given which, a = 2 which is larger than 1 and b which is 6 is also larger than 1, thereby the Master Theorem is applicable for the given equation.

1. a = 2, b = 6 and f(n) = 2n^2
2. $n^{\log_b a} = n^{\log_6 2}$
3. Comparing f(n) and $n^{\log_b a}$, we see that f(n) is larger than $n^{\log_b a}$, this indicates that the running time is dominated by the cost of the roots.
4. Therefore, the appropriate master case for this equation is in the form of the following if f(n) = Ω $n^{\log_b a}$, then T(n) = Θ(f(n)).

   Since it is Case 3, we must check for **Regularity Condition** which is applicable in the form of af(n/b) <= cf(n) where c < 1.

   af(n/b) <= cf(n)
   2 * 2$(\frac{n}{6})$^2 <= c * 2 n^2
   n^2 / 9 <= 2n^2 * c

   n^2/18 <= n^2 * c

   Thus for c < 1, given c = ¾

   n^(2/18) <= ¾ n^2 , thus every n satisfies this condition, thereby concluding that case 3 satisfies the condition for the regularity condition .

   Finally, T(n) = Θ(f(n)), thus T(n) = Θ(2n^2) or thus **T(n) = Θ(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.

Provided that the quaternary search algorithm is a modification of the binary search algorithm and that splits the input into four sets of sizes approximately into one-fourth, I would image the code will look something like this.

Source for this code - https://stackoverflow.com/questions/39845641/quaternary-search-algorithm

```
QuaternarySearch(A[], value, low, high)

While (high is larger than or equal to 1)

    firstQuarter = low + ((high-low) / 4)

    secondQuarter = low + ((high-low) / 2)

    thirdQuarter = low + (3(high-low) / 4)

    if (search A[firstQuarter] found value)

        return A[firstQuarter]

    else if (search secondQuarter found value)

        return A[secondQuarter]

    else if (search A[thirdQuarter ]found value)

        return A[thirdQuarter]

    else if (search A[firstQuarter]  is larger than value)

        recursively call/return QuaternarySearch(A, value, low, firstQuarter-1)

        //Recursively pass in first quarter

    else if (search secondQuarter  is larger than value)

        recursively call/return QuaternarySearch(A, value, firstQuarter+1,
        secondQuarter)

        //Recursively pass in second quarter

    else if (search A[thirdQuarter]  is larger than value > A[secondQuarter])

        recursively call/return QuaternarySearch(A, value, secondQuarter+1,
        thirdQuarter-1)

    return QuarterSearch(A,value, thirdQuarter + 1, high)
```

//Lastly, check the last part

b) Give the recurrence for the quaternary search algorithm

The recurrence for the quaternary search algorithm is as follows, T(n) = 1T(n/4 + **Θ(1).** This is in compliance to the Master Method. Where 1 is the the number of subproblems, 4 is the size of each subproblem, and the size of work done outside will just be **Θ(1).**

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.

T(n) = 1T(n/4 + **Θ(1)**

Given that the function is in the form of T(n) = aT(n/b) + f(n), given which, a = 1 which is larger than 1 and b which is 4 is also larger than 1, thereby the Master Theorem is applicable for the given equation.

1. a = 1, b = 4 and f(n) = **Θ(1)**
2. $n^{\log_b a} = n^{\log_4 1} = n^{log1/log4}$ = n^0 = 1
3. Comparing f(n) and $n^{\log_b a}$, we see that f(n) is equal to $n^{\log_b a}$, this indicates that the running time is evenly distributed,
4. Therefore, the appropriate master case for this equation is in the form of the following if f(n) = **Θ** $n^{\log_b a}$, then T(n) = Θ($n^{\log_b a}$ log (n)).

T(n) = Θ($n^{\log_b a}$ log (n))
T(n) = Θ(1 ∗ log (n)).
T(n) = Θ(log (n))

The running time for both the quaternary search algorithm and the binary search algorithm turns out to have the same running time, which is log(n).

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

- For problem 3, I will be referencing the divide and conquer algorithm and analyze provided by this website - http://somnathkayal.blogspot.com/2012/08/finding-maximum-and-minimum-using.html

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

Iteratively

```
StraightMaxMin(a,n,max,min)
      Set max and min to a[1];
      for I equal to 2 up to n do the following
           if(a[i] > max) then set the max toa[i];
           if(a[i] > min) then set the min to a[i];
```

Recursively

```
MaxMin(i, j, max, min)
{
     If I is equal to j then max and min to a[i];
     else if I is equal to j-1
            if (a[i] < a[j]) then set max to a[j] and min to a[i];
            else set max to a[i] and set min to a[j]
     else
           set mid  to ( i + j )/2;
           Recursively call MaxMin( i, mid, max, min );
           Recursviely call again MaxMin( mid+1, j, max1, min1 );
           if (max < max1) then set max equal to max1;
           if (min > min1) then set min equal to min1;
     }
}
```

b) Give the recurrence.

The recurrence for the provided algorithm for finding max and min is equal to 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.

Given that the function is in the form of T(n) = aT(n/b) + f(n), given which, a = 2 which is larger than 1 and b which is 2 is also larger than 1, thereby the Master Theorem is applicable for the given equation.

1. a = 2, b = 6 and f(n) = 2
2. $n^{\log_b a} = n^{\log_2 2}$ = n^1 = n

3. Comparing f(n) and $n^{\log_b a}$, we see that f(n) is smaller than $n^{\log_b a}$ on all cases where n >= 2 this indicates that the running time is dominated by the cost of the roots.
4. Therefore, the appropriate master case for this equation is in the form of the following if f(n) = O $n^{\log_b a}$, then T(n) = O(f(n)).

Finally, T(n) = O(f(n)), thus T(n) = O($n^{\log_2 2}$) or thus T(n) = O(n^1) => **T(n) = O(n)**

The average case for the straight forward iterative method will on average check 2(n-1) variables as opposed to 3n/2, so this concludes to a 25% increase in efficiency using the divide and conquer method

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?)

STOOGESORT will not work correctly because when n = 4, m = ceiling(2(4)/3), which would mean m = ceiling(8/3), , = ceiling(2). Passing this information into the program, we can see that in the first recursive function, StoogeSort(A[0,1]), and this is error-some since n will never be able to reach 2 thereafter.

. b) State a recurrence for the number of comparisons executed by STOOGESORT.

. T(n) = 3T(2n/3) + Θ(1)

. c) Solve the recurrence to determine the asymptotic running time.

- In this case, a = 3, which indicates that the number of subproblems is 3.
- With b = 3/2, indicating that each subproblem size is 3/2.
- And Θ(1) is the constant time operation

Given that the function is in the form of T(n) = aT(n/b) + f(n), given which, a = 1 which is larger than 1 and b which is 4 is also larger than 1, thereby the Master Theorem is applicable for the given equation.

5. a = 3, b = 3/2 and f(n) = **Θ(1)**
6. $n^{\log_b a} = n^{\log_{3/2} 3}$ = log 3 / log (3/2) = log(3)/ (log(3) – log(2)) = 2.7
7. Comparing f(n) and $n^{\log_b a}$, we see that f(n) is less than $n^{\log_b a}$, this indicates that the running time is heavy on the leaves. Therefore, the appropriate master case for this equation is in the form of the following if f(n) = **O** $n^{\log_b a}$, then T(n) = Θ($n^{\log_b a}$)).

   T(n) = Θ($n^{\log_b a}$)
   T(n) = Θ(n(log 3 / log (3/2))).
   There for the final function is T(n) = Θ(n^2. 7)

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. 5B Is attached at the bottom of the file

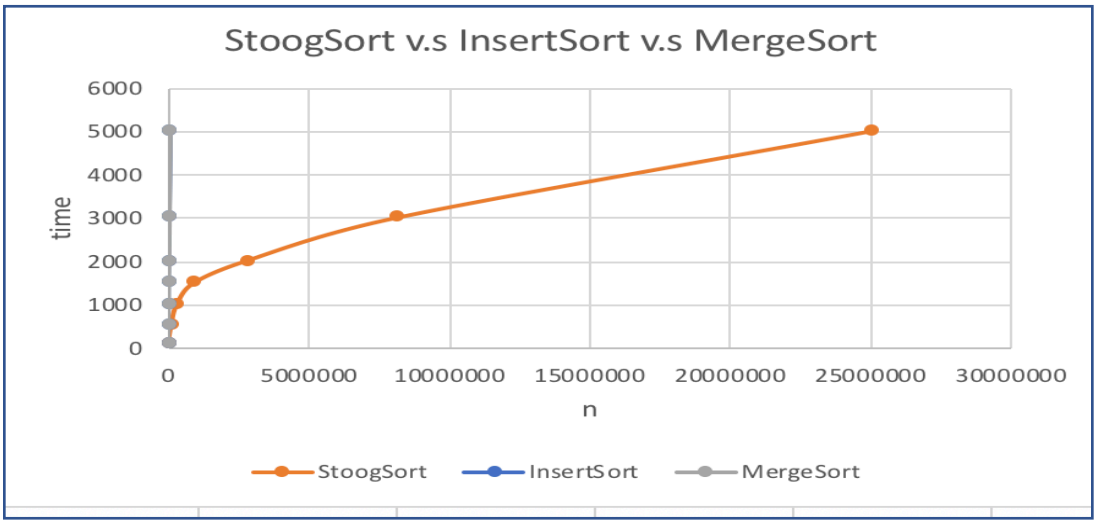
. StoogeSort

```
Sort – 100  Count – 1201
Sort – 500  Count – 103776
Sort – 1000 Count – 310829
Sort – 1500 Count – 927956
Sort – 2000 Count – 2763743
Sort – 3000 Count – 8133502
Sort – 5000 Count – 25035677
```

. 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.

-The table/chart on the right has the data for the three sorts, StoogSort, InsertSort and MergeSort. And the graph below details the difference in performance . As clearly visible. the running time for StoogSort is significantly slower for the same amount of sort when compare to that of Insert sort or merge Sort.

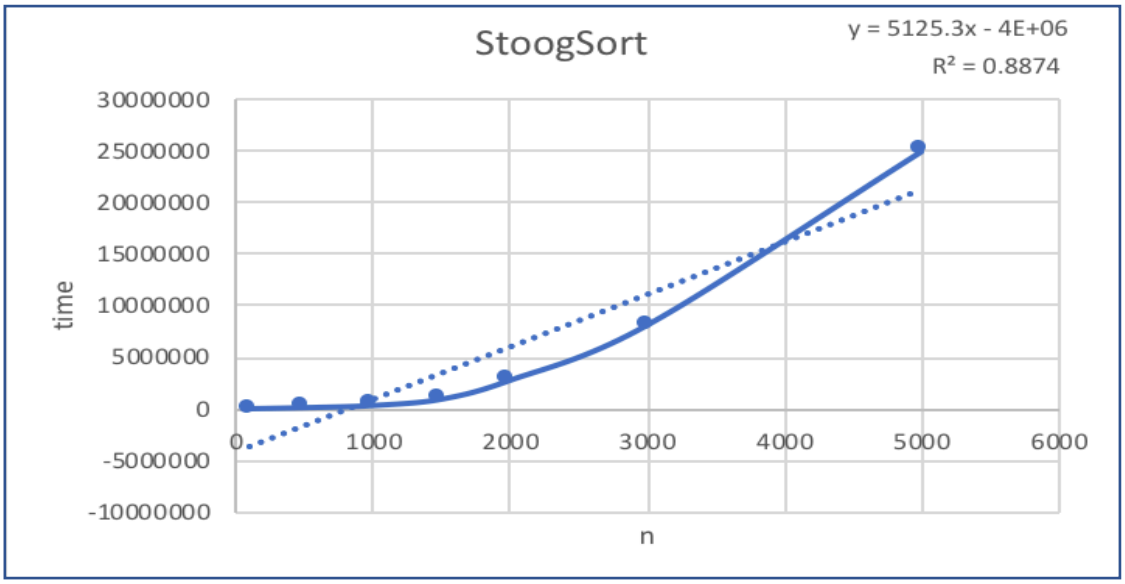
Time	StoogSort	InsertionSort	MergeSort
100	1201	18	14
500	103776	380	77
1000	310829	1118	151
1500	927956	2151	234
2000	2763743	3907	360
3000	8133502	9829	494
5000	25035677	23494	857



. 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?

The equation for StoogSort is y = 5125.3x- 4E + 06 and R^2 is 0.8874. And the equation derived was T(n) = Θ(n(log 3 / log (3/2))).

Given that R^2 is decently close to 1, we are relatively confident in the data presented. The shape of the curve is more of a staircase shape when looking at individual data? But when compiled, it is more of a exponential plot. The experimental running time when compared to the theoretical running time of the algorithm, we see that there is similarly between the two



```
//Source COde Referenced from wikipedia and is also provided by the teacher
//from the homework assignment. Also from
// rosettacode.org/wiki/Sorting_algorithms/Stooge_sort#C.2B.2B as well

#include <iostream>
#include <fstream>

using namespace std;

/*****************************************************************************
* Description: THe purpose of this class is to sort the given data in a "stooge" algorithm. This
algorithm
 * is referenced from an online source that is written above

*****************************************************************************/

void sort( int* arr, int start, int end )
{
      if( arr[start] > arr[end - 1] )
      {
         swap( arr[start], arr[end - 1] );
      }
      int n = end - start;
      if( n > 2 )
      {
         n /= 3; sort( arr, start, end - n );
         sort( arr, start + n, end );
         sort( arr, start, end - n );
      }

}

/*****************************************************************************
************************
 * Description: This is the main menu that receive input, process it through the file and output the
result

*****************************************************************************
**********************/

 int main( int argc, char* argv[] )
{
   int count; //Number of input

   count = 100;
   int arr1[count];
   for(int i = 0; i < count; i++)
   {
      arr1[i] = rand() % 1000;
   }

   clock_t t;
   t = clock();

   sort(arr1, 0, count);
   t = clock() - t;
```

```
cout << "Sort - Count = 100: " <<  t << endl;
//-----

count = 500;
int arr2[count];
for(int i = 0; i < count; i++){
   arr2[i] = rand() % 1000;
}
t = clock();
sort(arr2, 0, count);
t = clock() - t;
cout << "Sort - Count = 500: " <<  t << endl;

count = 1000;
int arr3[count];
for(int i = 0; i < count; i++){
   arr3[i] = rand() % 1000;
}
t = clock();
sort(arr3, 0, count);
t = clock() - t;
cout << "Sort - Count = 1000: " <<  t << endl;

count = 1500;
int arr4[count];
for(int i = 0; i < count; i++){
   arr4[i] = rand() % 1000;
}
t = clock();
sort(arr4, 0, count);
t = clock() - t;
cout << "Sort - Count = 1500: " <<  t << endl;

count = 2000;
int arr5[count];
for(int i = 0; i < count; i++){
   arr5[i] = rand() % 1000;
}
t = clock();
 sort(arr5, 0, count);
t = clock() - t;
cout << "Sort - Count = 2000: " <<  t << endl;

count = 3000;
int arr6[count];
for(int i = 0; i < count; i++){
   arr6[i] = rand() % 1000;
}
t = clock();
 sort(arr6, 0, count);
t = clock() - t;
cout << "Sort - Count = 3000: " <<  t << endl;


count = 5000;
int arr7[count];
```

```
    for(int i = 0; i < count; i++){
        arr7[i] = rand() % 1000;
    }
    t = clock();
    sort(arr7, 0, count);
    t = clock() - t;
    cout << "Sort - Count - 5000: " <<  t << endl;
    return 0;

}
```