Benjamin Fondell
7/9/2017

CS 325 HW 2

1.

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

By the muster method.

a = 1, b = 2, and d = 1

f(n) = **Θ**(n^d)

then, T(n) = **Θ** (n^(d+1)) as a = 1.

therefore, T(n) = **Θ** (n^(1+1)) = **Θ**(n^2)

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

By the muster method.

a = 3, b = 1, d = 0 as n^0 = 1

then, **Θ**(n^d * a^(n/b)) as a>1

then (n^d * a ^(n/b)) = **Θ**(n^0 * 3^(n/1))

thus, t(n) = **Θ**(3^n)

c) $T(n) = 2T(n/8) + 4n^2$

By master theorem.

a=2 , b=8, and f(n) > 0

by case 3: as f(n) = Ω(nlogb^a+3) for some Ɛ > 0

we must solve the regularity af(n/b)<=cf(n) for some c<1

then, 2*4(n/8)^2 <= c * 4n^2 = 2 * (n/8)^2 <= c * n^2 = 2* (n^2/64) <= c * n^2

= n^2/32 <= c*n^2 = 1/32 <= c. Therefore c can be less than 0 so the regularity is true

Thus t(n) = **Θ**(f(n)) = **Θ**(n^2)

2.

A. The quaternary search algorithm is similar to a binary search algorithm except it divides an input of values by four repeatedly instead of dividing the same values by two repeatedly like in binary search. The simplest way to perform this algorithm is with recursion. As this is a search a sorting algorithm no values need to be moved only looked at. The most important part of this algorithm is the division into four sub sets. The pseudocode of this algorithm is as follows.

```
def quaternarySearch( array, value)

        initialize start to first index of array
        initialize first quarter to length of array / 4 -1
        initialize mid point to array / 2 - 1
        initialize third quarter to length of array * 3/4 - 1
        initialize end to the length of the array - 1
        make an array of sections [start, first quarter, mid point, third quarter, end]


        iterate through array of sections from start - end in for loop
                check for value in index marked by section names
                if found return value at index


         if start < value < first quarter
                return quaternarySearch( array [start : first quarter] , value )
        else if first quarter < value < mid point
                return quaternarySearch( array [first quarter : mid point] , value )
        else if mid point < value < third quarter
                return quaternarySearch( array [midpoint : third quarter] , value )
        else if third quarter < value < end
                return quaternarySearch( array [third quarter : end] , value )
        else
                return not found
```

Benjamin Fondell
7/9/2017

B.

Recurrence: $T(n) = T(n/4) + C$

This is because the algorithm divides the values n by 4 so the factor for b is 4. The factor for c is constant as the iteration loop used to search for the value represents the number of comparisons made. Based on this recurrence we may solve using the master method.

C.

By the master method:

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

n^(log^b(a)) = n^(log^4(1)) = n^0

then n^0 = 1 = Θ(1) = Θ(c) so case 2 is used.

Thus T(n) = Θ(n^(log^b(a)) * log^4(n)) = Θ(1 * log^4(n))

thus runtime = T(n) = Θ(log^4(n))

Comparing the worst case runtime for quaternary search and binary search:
worst case runtime binary search: Θ(log^2(n))
worst case runtime quaternary search: Θ(log^4(n))
because the runtime of logarithms with different bases is equivalent when considering asymptotic analysis we can assume that Θ(log^2(n)) = Θ(log^4(n))

3. A.

```
Minimum_Maximum(array[min…max]){
        if length(array) == 1
                return (array[0] , array[0])
        else if length(array) == 2
                if (array [0] < array [1])
                        return (array[0], array[1])
                else
                        return(array [1], array[0])
        else
                (leftMin, leftMax) = Minimum_Maximum( array[0…(length(array) / 2)] )
                (rightMin, rightMax) = Minimum_Maximum( array(length(array) / 2)]..length(array))

                if (leftMax < rightMax
                        max = rightMax
                else
                        max = leftMax

                if(leftMin < rightMin)
                        min = leftMin
                else
                        min = rightMin

                return (min , max)
```

Benjamin Fondell
7/9/2017

B.

The recurrence for the Minimum_Maximum algorithm:

If n == 0 then T(1) = 0 because no comparisons are made.
if n == 2 then T(1) = 1 because only one comparison is made to determine which value is larger.

if n>2 then T(n) = 2T(n/2) + 2

The factor for a = 2 as there are two recursive calls for each run of the algorithm. One to get the left subset of the array and one to get the right subset. The factor for b is 2 because the set of values is divided in half each time the algorithm is called. The f(n) factor is equal to 2 because there are two comparisons made for each call to the function. One to compare each sub array's minimum and one to compare each sub array's maximum. The runtime can be determined with the master theorem.

C.

By the master method:

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

n^(log^b(a)) = n^(log^2(2)) = n^1

f(n) = 2 thus f(n) = n^0 so we can use case 1

by case 1:

f(n) = O(n^(log^b(a)-ℰ) such that ℰ = 1

thus, T(n) = **Θ**(n^(log^b(a))

then **Θ**(n^(log^2(2)) = **Θ**(n^1) = **Θ**(n)

therefore the runtime of the minimum maximum algorithm is **Θ**(n)

4.

A.

```
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])
```

Benjamin Fondell
7/9/2017

Case size of array = 0 or 1: The function will do nothing.

Case size of array = 2: The function will simply compare and swap the two values if necessary.

Case size of array > 2: A variable M is initialized to the ceiling of the size of the array * 2 divided by 3. this essentially sets an indexing point 2/3 into the array. The function then makes three recursive calls.

The first recursive call is passed the first 2/3 of the array which will then be sorted by the comparison at the top of the function as it makes subsequent recursive calls.

The second recursive call works similarly to the first but sorts the last 2/3 of the array.

The third recursive call is the same as the first. it sorts the first 2/3 of the array. This third call is necessary in that it combines the sorted subarrays from the first and second call essentially blending the overlapping middle third of the array so the entire array becomes sorted.

The algorithm sort by breaking the array into sub arrays and recursively sorting and then combining the two sub arrays where they meet in the middle. The need to resort the first 2/3 of the array in the third recursive call introduces some inneficiency as it's sorting an array that has already been sorted. There is also inneficeiency in that the two sub arrays share the middle third values of the initial array. This causes significantly more comparisons than if the subarrays were exclusive.

B.

No. Consider if we passed an array of size 4. Then the value for k = (2n/3) = floor(2(4)/3) = floor(8/3) = 2 so k = 2. The first call will pass the first two values of the array [x, x …. to be sorted. This will satisfy the n=2 condition so they will be swapped if necessary. The second recursive call will be passed the last 2 values of the array to be sorted …x x]. These will also be swapped if necessary. The third recursive call will pass the first two values of the array [x x.. to be sorted. Thus the problem with using the floor(2n/3) is that the middle of the array never gets compared or recombined. These values [x x x x] are ignored in the algorithm so the array will likely not be sorted correctly.

C.

The recurrence for STOOGESORT is as follows.

a = 3 because there are three recursive calls
b = 2/3 because the input values are divided into sub arrays of size 2/3

Therefore the recurrence can be written as $T(n) = 3 * T(2n/3) + \Theta(1)$

Benjamin Fondell
7/9/2017

D.

We can solve this recurrence by the master method.

a = 3, b = 2/3, f(n) = c

n^(log^b(a)) = n^(log^(3/2)(3) = (log^10(3)) / (log^10(3/2)) = 2.71

therefore n^log^(3/2)(3) = n^2.71 = Ω(f(n))

Then by case 1:

f(n) = (O(n^(2.71 -ε)) such that ε = 2.71 Therefore, T(n) = **Θ**(n^2.71)

So the runtime for STOOGESORT = **Θ**(n^2.71)

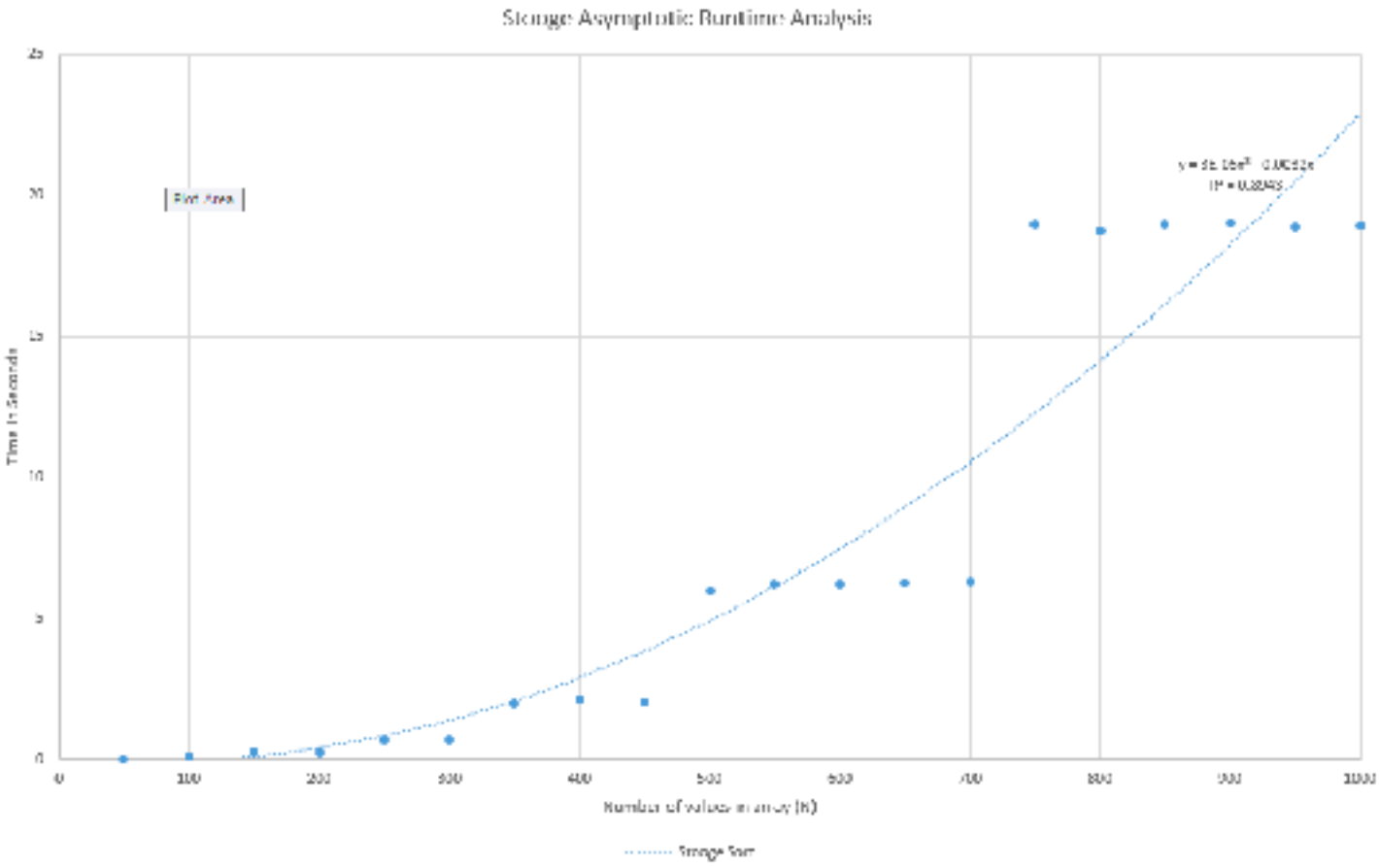
5.

B.

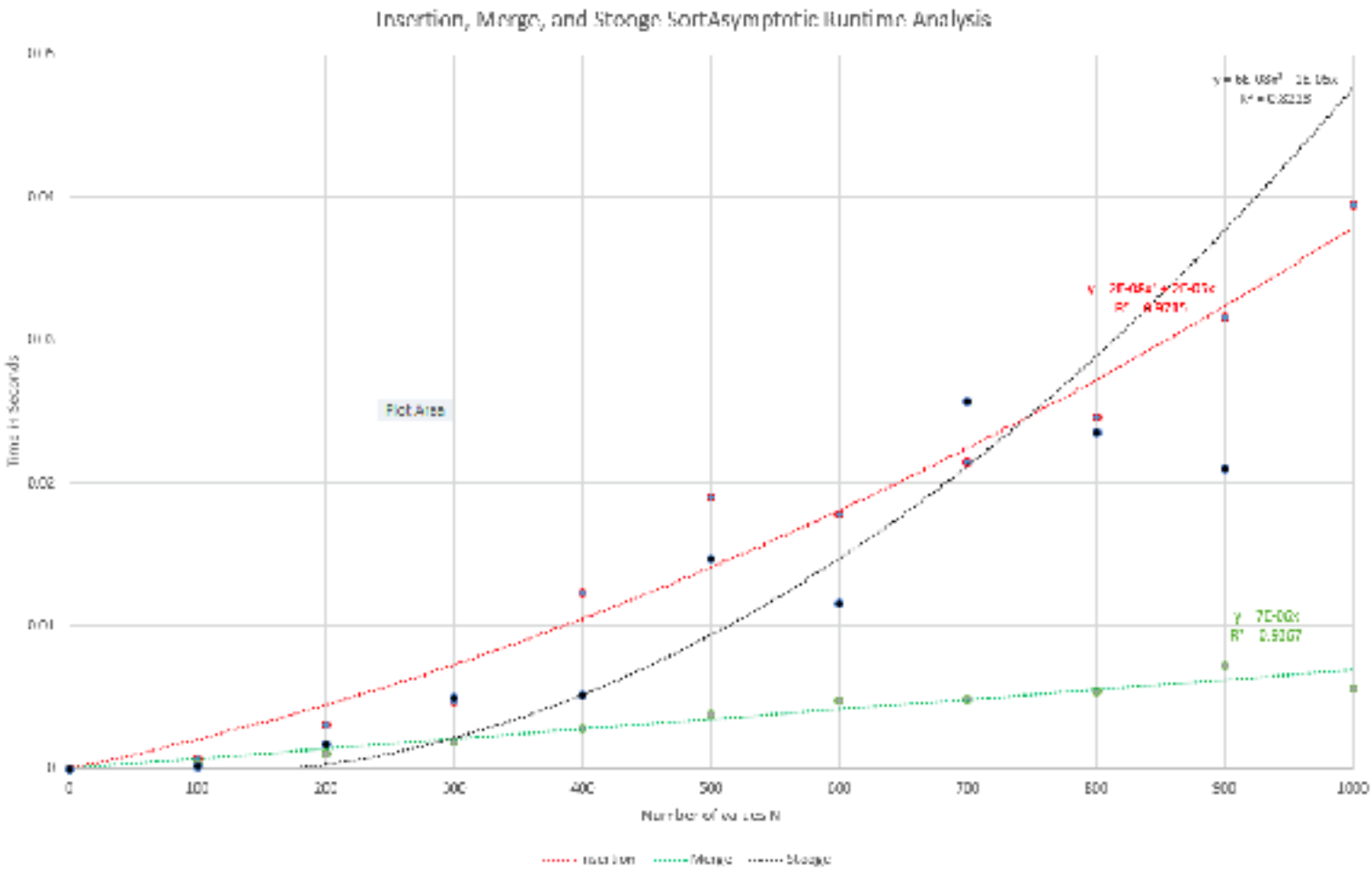
```
import random
import time

def load_file(fname):
    with open(fname) as inf:
        data = [[int(i) for i in line.split()] for line in inf]
    return data

# stoogesort algorithm sourced and modified from ideone.com

def stoogeSort(L):
    stoogeSortRec(L, 0, len(L))

def stoogeSortRec(L, i, j):
    if L[j-1] < L[i]:
        tmp = L[i]
        L[i] = L[j-1]
        L[j-1] = tmp
    if j-i >= 3:
        t = (j-i) // 3
        stoogeSortRec(L, i, j-t)
        stoogeSortRec(L, i+t, j)
        stoogeSortRec(L, i, j-t)


def sample_wr(population, k):
    n = len(population)
    _random, _int = random.random, int
    result = [None] * k
    for i in xrange(k):
        j = _int(_random() * n)
        result[i] = population[j]
    return result

def main():

    sizes = [10,20,30,40,50,60,70,80,90,100]
    for index in range(0,len(sizes)):
        print sizes[index]
        numbers = sample_wr(xrange(0,10000),sizes[index])
        start_time = time.clock()
        stoogeSort(numbers)
        print("---%s seconds---" % (time.clock() - start_time))

main()
```

Benjamin Fondell
7/9/2017

C.



Benjamin Fondell
7/9/2017



D.The line that best fits Stooge sort's runtime analysis for values in the range of 0 and 1000 is a polynomial line with order of 2. It is an exponential line with characterized by the equation: y = 3E-05x2 - 0.0032x

The line of the graph has the following value for its relation to the plotted points.
$R^2 = 0.8943$

E. Given the theoretical runtime of Θ(n^2.71) I found my graphs are roughly reflective of this runtime. I would expect as the values increased and if the collection of data was more thorough the results on the graph would be more accurate. For the graph reflecting merge, insertion, and stooge sort, I believe this result is not an accurate depiction of the runtime comparison between stooge sort and the other algorithms.

Benjamin Fondell
7/9/2017

Stooge sort is an extremely inefficient algorithm. With the exception of sorting the smallest values the algorithm will be slower than almost every other algorithm.