Thanhbinh Truong
CS325
April 11, 2018
933233558

Homework #2

Problem 1:

(a)

Recurrence: T(n) = ST(n/2) + O(n)

A = 5, b = 2, c = 1

(1) T(n) = ST(n/2) + O(n)
(2) T(n)= aT(n/b) + O(n^c)

By the master theorem, if the value of c < $\log_b$a then…

$\text{Log}_b a = \log_2 5 > 2 > c$ => T(n) $\Theta(n^{\log_2 5})$

(b) T(n) = 2T(n=1) + O(1)

Difficult to use master theorem since b = 1, log base not defined for any value other than 1

N levels of the tree, at level I, will have 2^i nodes

$\sum_{i=0}^{n} \quad 2^i = -\frac{2n+1}{2-1} = 2^{n+1} -1$

=O(2^n)

(c)

a = 9, each problem = $\frac{n}{3}$ & time “c” = 2, solving in O(n^2)

T(n) = 9T($\frac{n}{3}$) + O(n^2)

$n^c = n^2$ c=2 b=3 a=9

T(n) = O(n^clogn)

$\text{Log}_b a = \log_3 9 = 2$, $\log_b a \equiv 2$

$T(n) \equiv O(n^2 logn)$ – from the 3, Algorithm C is the best choice.

Problem 2:

Initialize r = n – 1 and l = 0

ternarySearch = (l, r ,x) //where x is being searched

```
if( r ≥ l)
    mid1 = l + (r – l)/3;
    mid2 = mid1 + (r+l)/3;

    if(array[mid1] =x ) //x found in mid1
          return mid1;
    if(array[mid2] = x) //x found in mid2
          return mid2;
    if(array[mid1] > x) //x presented in the left 1/3
          return tenarySearch(array, mid1 -1, r,x)
    if*array[mid2] < x) //x is in right 1/3
          return tenarySearch(array, mid2 + 1, r, x)
    else
          return tenarySearch(array, mid1 + 1, mid2 -1, x);

return -1 //if not found in array
```

(b)
ternary algorithm acts like binary search but divides into (3) arrays instead of (2)

$T(n) = T(n/3) + 2$, average time = $\Theta(logn)$
$T(n) = T(n/3) + 2$

(c)
average time = $\Theta(logn)$

The running time of a ternary search compared to the binary search seems faster as the n grows larger. Though theoretically it is faster than binary search, the extra comparisons on the worst case for ternary would make it not practical.

Problem 3:

(a)

```
min_and_max(a[1…..n] of elements)
        if(n == 1)       //array had 1 element
                return (a[1], a[1])
        else if (n==2)
                if(a[1] < a[2])
                        return(a[1], a[2])
                else if
                        return(a[2], a[1])
                else
                        (max_left, min_left = max_and_min(a[1…(n/2)])
                        (max_right, min_right = max_and_min(s(n/2+1…..n])
        if(max_left < max_right)
                max = max_right
        else
                max=max_left
        if(min_left < min_right)
                max=min_left
        else
                min = min_right
        return(min, max)
```

(b)

recursive:

T(n) = # of steps to complete for size = n

- merging linear time
- T(n) = 2 * T(n/2) O(n)

-by master theorem, see recurrence has steady state tree

T(n) = O(n * logn)

M = how many times n is divided by 2 before size of array = 1

$N = 2^m = \log 2^m = \log n$

$M x \log_2 2 = \log_2 n = \log_2 2 = 1$, $m = \log_2 n$

n/2 comparisons for merge at each level

$O(n/2 \log n \Rightarrow O(n \log_2 n)$

(c)

An iterative method would find the minimum and maximum I O(n) times where as the recursive method does the time in $O(n+\log_2 n) = O(n)$

Problem 4:

a.

Step 1: If value at index 0 is greater than value at last index, swap them.

Step 2: Recursively,

a. Stooge sort the initial 2/3rd of the array.
b. Stooge sort the last 2/3rd of the array.
c. Stooge sort the initial 2/3rd again to confirm.

b. Yes, STOOGESORT() function will solve the array

```
#include <iostream>
#include <math.h>

using namespace std;

void STOOGESORT(int array[], int l, int h)
{
        int n = h- l + 1
        if (n==2 && array[l] > array[h])
                swap(array[l], array[h])'
        elseif(n >2)
                int k = floor(n/3); //recursive sort on the last 2/3 elements
                STOOGESORT(array, l+k, h);
                STOOGESORT(array, l, h-k);
}

int main()
{
        int array[] = {4, 9, 1, 2, 3};
        int n = sizeof(array) / sizeof(array[0])
        STOOGESORT(array, 0, n-1);
        For(int i = 0; I < n; i++)
                Printf(array[i]);
                Printf(" ");
        Printf("\n");
        Return 0;
}
```

c.

1. N = 1, tot. comparisons = 0
2. N = 2, tot. comparisons = 1
3. N = 3, tot. comparisons = 3*1 + 0 = 3
4. N = 4, tot. comparisons = 4 * 2 + 1 = 9
5. N = 5, tot. comparisons = 5 * 3 + 2 = 17

6. N= N,n * (n-2) + (n-3)

$T(n) = 3T(3n/2) + \Theta(1)$

d.

$T(n) = 3T(3n/2) + \Theta 1$ (substitution method)

$T(n) = 3T(3n/2) + c$

$= 3[3T(3^2n/2^2] + c$

$=3^2T(3^3n/2^3) + 2c + c$

$= 3^2T(3^3n/2^3) + 3c + c$

$= 3^kT(3^kn/2k)+ (3k-1/2)C$

$T(3^kn/2^k) = 1$

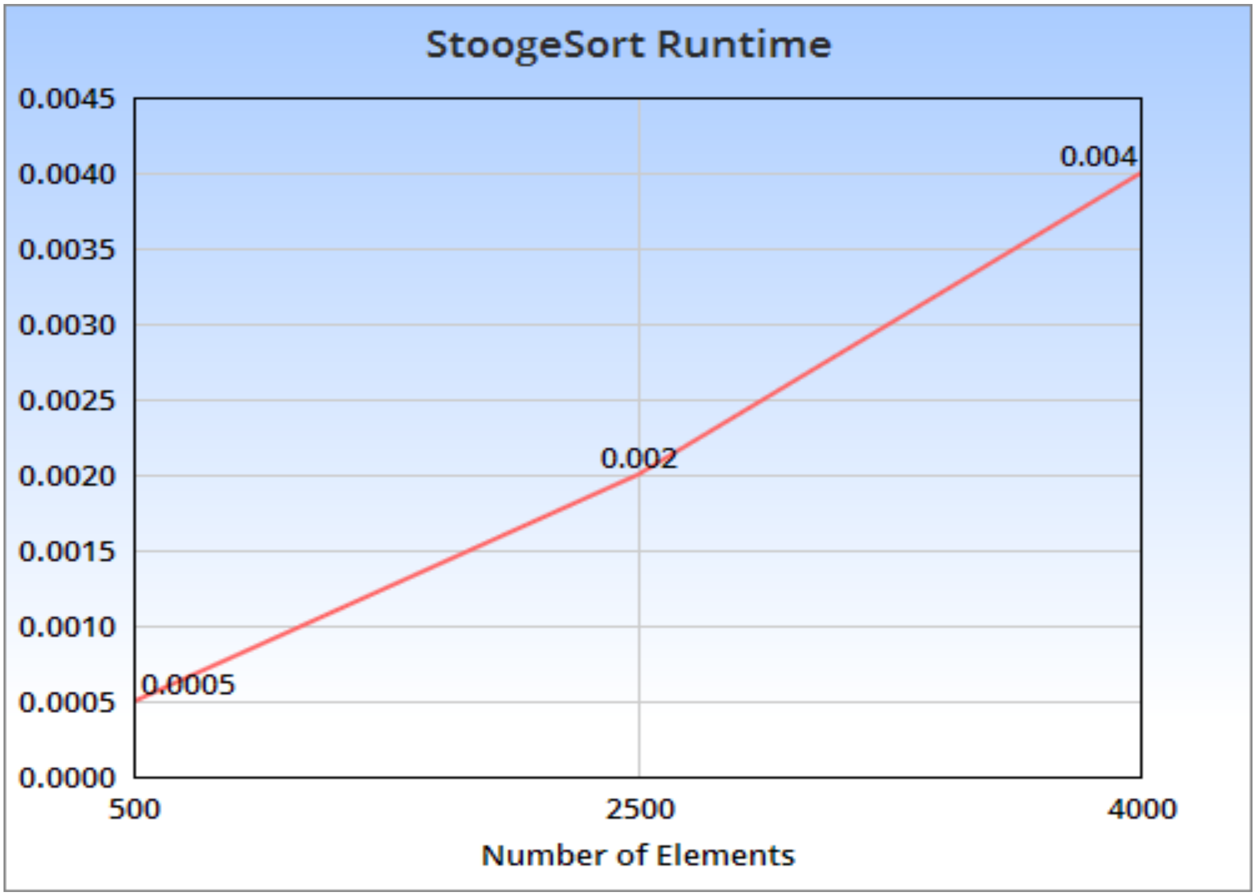
Problem 5:

b)

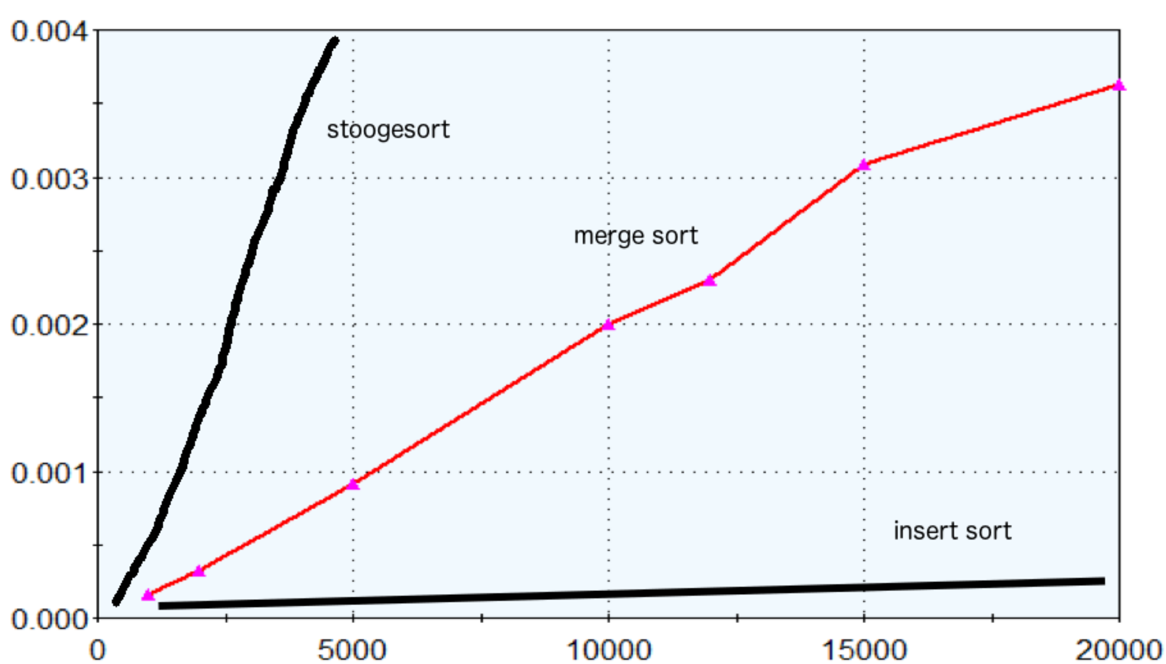
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

void printArr(int arr[], int arrSize)
{
	int i;
	for(i=0; i < arrSize; i++)
	{
		print("%d", arr[i]);
	}
	printf("\n");
}

void sort(int arr[], int i, j)
{
	if arr[i] > arr[j]
	{
		int holder = arr[i];
		arr[i] = arr[j];
		arr[j] = holder;
	}

	if((j-i) < 1)
	{
		int t =(int)ceil((j-i+1/3);
		sort(arr, i, (j-t)):
		sort(arr, (i+t), j);
		sort(arr, i, (j-t)):
```

```
	}
	return;
}

int main()
{
	int n;
	int i;
	int time;
	int arr[n];
	int arrSize;

	srand(time(NULL));

	printf("Random numbers between 1 - 250 will be inserted into an array\n");

	arrSize = rand % 100;

	for(i = 0; i < arrSize; i++)
	{
		int num = rand() % 200;
		array[i] = num;
		printf("%d\n", num);
	}

	clock_t t;
	t=clock();

	sort(array, 0, (arrSize-1));

	t=clock() - t;

	time = ((double)t)/CLOCKS_PER_SEC;

	printf("Sorted Array:\n");
	printArr(arr, arrSize)
	printf("\n");
	printf("Time taken: %f", time)
}
```

c)


StoogeSort Runtime
0.0045
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
0.004
0.002
0.0005
500
2500
4000
Number of Elements

Merge Sort


0.004
0.003
0.002
0.001
0.000
stoogesort
merge sort
insert sort
0
5000
10000
15000
20000

d)
A quadratic would be the best fit with the stoogesort data set.
With n on the x axis and time on the y axis,

$T(n) = n^{2.71}$