Thanhbinh Truong
CS325
April 7th, 2018
933233558

2. T(n) ={ 2 if n=2
{ 2T(n/2) + n if n = 2^k, for k > 1

T(n) = nlg n

Base: n = 2, T(2) = 2 x lg 2 = 2

Hypothesis: $T^{2^k} = 2^k \lg 2^k$
$= k + 1 = \mathbf{T(2^{k+1}) = 2^{k+1} \lg 2^{2k+1}}$

inductive: $T(2^{k+1} = 2T(2^{k+1}/2) + 2^{k+1}$
$= 2T(2^k) + 2 * 2^k$
$= 2*2^k \lg 2^k + 2 * 2^k$
$= 2 * 2^k (\lg 2^k + 1)$
$= 2^{K+1} (\lg 2k + \lg 2)$
$= \mathbf{2^{k+1} \lg 2^{K+1}}$

3.(a)

$a_1 f_1(n) = O(g(n))$ and $f_2(n) = O(g(n))$ then $f_1(n) = \underline{O}f(_1(n))$

disprove
- $F_1(n) = n$, $f_2(n) = n^2$, $g(n) = n^3$ then $f_1(n) =/ \underline{O}(f_2(n))$

(b) f1(n) = O(g1(n)) and f2(n) = O(g2(n)) then f1(n)+ f2(n)= O(max{g1(n), g2(n)})

By definition there exists a c_1, c_2, n_1, $n_2 > 0$ such that
$f_1(n) \leq c_1 g_1(n))$ for n ⚽_n_1 and $f_2(n) \leq c_2 g_2(n)$ for n ⚽_n_2
Since the functions are asymptotically positive
$f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$
$\leq c_1 \max\{g_1(n), g_2(n)\} + c_2 \max\{g_1(n), g_2(n)\}$
$\leq (c_1 + c_2) \max\{g_1(n), g_2(n)\}$
Let $k = (c_1 + c_2)$ and $n_0 = \max(n_1, n_2)$ then **(-0.5 if n_0 is missing)**
$f_1(n)$ ☁_$f_2(n) \leq k \max\{g_1(n), g_2(n)\}$ for n ⚽_n_0; k, $n_0 > 0$ and by definition
$f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$

5.

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


void merge(int *arr, int l, int m, int r)
{
	int i, j, k;
	int n1 = m - l + 1;
	int n2 = r - m;

	int L[n1], R[n2];
	for (i = 0; i < n1; i++)
	{
		L[i] = arr[l + i];
	}

	for (j = 0; j < n2; j++)
	{
		R[j] = arr[m + 1+ j];
	}

	i = 0;
	j = 0;
	k = l;

	while (i < n1 && j < n2)
	{
		if (L[i] <= R[j])
		{
			arr[k] = L[i];
			i++;
		}
		else
		{
			arr[k] = R[j];
			j++;
		}
	k++;
	}
```

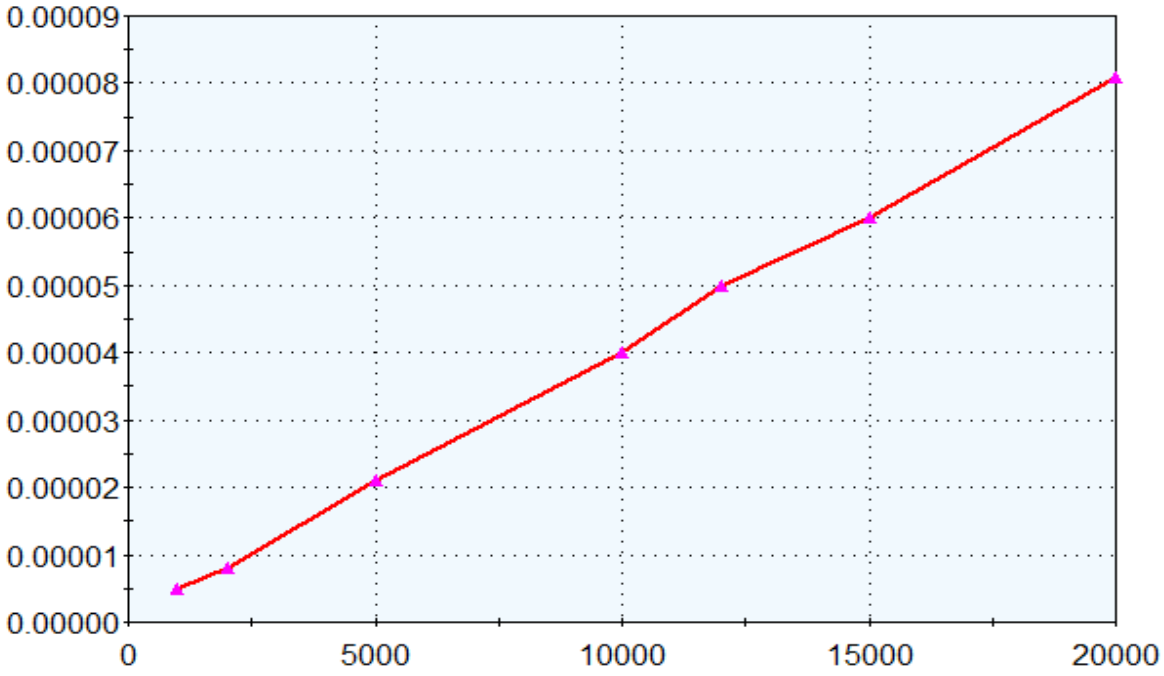
```
	while (i < n1)
	{
		arr[k] = L[i];
		i++;
		k++;
	}

	while (j < n2)
	{
		arr[k] = R[j];
		j++;
		k++;
	}
}


void mergeSort(int *arr, int l, int r)
{
	if (l < r)
	{
		int m = l+(r-l)/2;
		mergeSort(arr, l, m);
		mergeSort(arr, m+1, r);
		merge(arr, l, m, r);
	}
}

void printArray(int A[], int size)
{
	int i;

	for (i=0; i < size; i++)
	{
		printf("%d ", A[i]);
	}

	printf("\n");
}



void insertionSort(int *arr, int n)
{
	int i, j,key;
```

```
	for(i = 1; i < n; i++)
	{
		key = arr[i];
		j = i-1;

		while (j >= 0 && arr[j] > key)
		{
			arr[j+1] = arr[j];
			j = j-1;
		}
		arr[j+1] = key;
	}

}

void generateRandomArray(int *arr, int size)
{
	for(int i=0; i<size; i++)
	{
		arr[i] = rand() % 10000;
	}
}


int main()
{

	int rows = 7;

	int **arr = (int **) malloc(rows * sizeof(int *));

	//array of the sizes of the different arrays
	int array_size[7] = {1000, 2000, 5000, 10000, 12000, 15000, 20000};

	//array of size 1000, 2000, 5000, 10000, 120000, 15000, 20000
	arr[0] = (int *) malloc(1000 * sizeof(int));
	arr[1] = (int *) malloc(2000 * sizeof(int));
	arr[2] = (int *) malloc(5000 * sizeof(int));
	arr[3] = (int *) malloc(10000 * sizeof(int));
	arr[4] = (int *) malloc(12000 * sizeof(int));
	arr[5] = (int *) malloc(15000 * sizeof(int));
	arr[6] = (int *) malloc(20000 * sizeof(int));
```

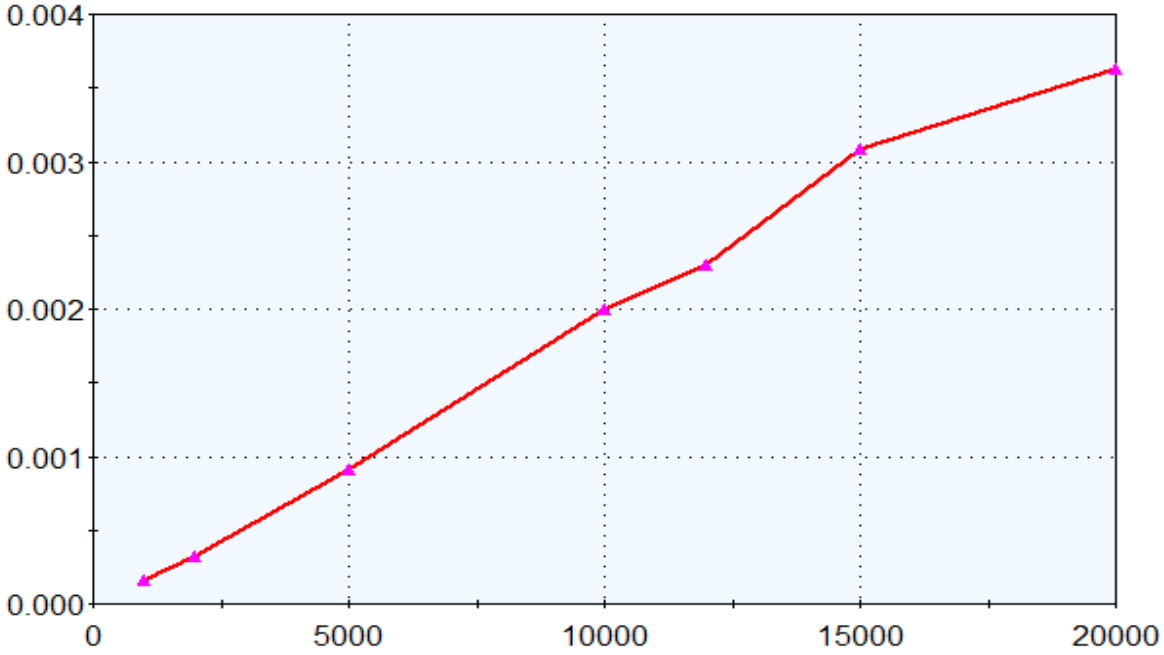
```
	double timeMergeSort[7];			//to hold running time
	double timeInsertionSort[7];			//to hold running time


	for(int i=0; i<7; i++)
	{
		generateRandomArray(arr[i], array_size[i]);			//fill array with
random numbers
	}

	for(int i=0; i<7; i++)				//get mergesort running time
	{
	clock_t t;
	t = clock();

	mergeSort(arr[i], 0, array_size[i]-1);

	t = clock() - t;				//end time - start time

	timeMergeSort[i] = ((double)t)/CLOCKS_PER_SEC;

	}


	for(int i=0; i<7; i++)				//get insertionsort running time
	{
		clock_t t;
		t = clock();

		insertionSort(arr[i], array_size[i]);

		t = clock() - t;			//end time - start time

		timeInsertionSort[i] = ((double)t)/CLOCKS_PER_SEC;

	}

	printf("\nTimes of Merge Sort : \n");

	for(int i=0;i<7; i++)
	{
		printf("%f ", timeMergeSort[i]);
	}
```

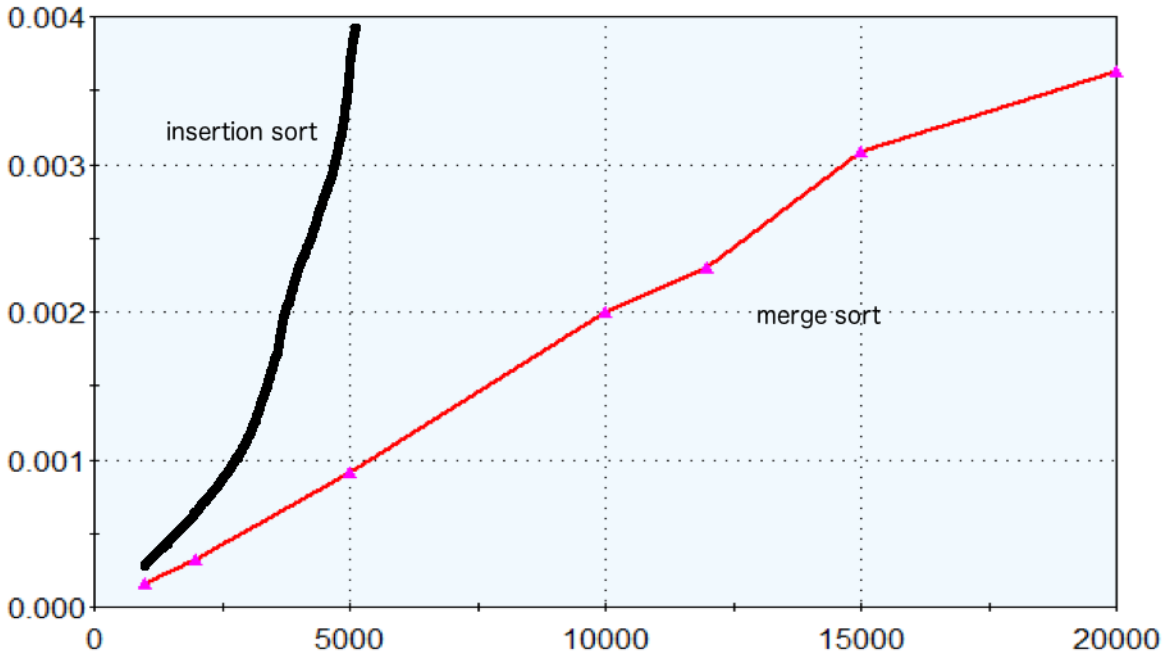
```
	printf("\nTimes of Insertion Sort : \n");

	for(int i=0;i<7; i++)
	{
		printf("%f ", timeInsertionSort[i]);
	}

double mergeSortTotalTime, insertionSortTotalTime;

	for(int i=0;i<7; i++)
	{
		mergeSortTotalTime += timeMergeSort[i];
	}

	for(int i=0;i<7; i++)
	{
		insertionSortTotalTime += timeInsertionSort[i];
	}

	printf("\nAvg time(Merge Sort) = \%f seconds", mergeSortTotalTime/7);

	printf("\nAvg time(Insertion Sort) = %f seconds", insertionSortTotalTime/7);

return 0;

}
```

Insertion Sort



Merge Sort





c) The merge sort graph seems to represent the data the best.

d) For the merge sort, the best curve that fit it is the polynomial curve. For the insertion sort algorithm, it seems as though the linear curve best.

e) The experimental runtimes were slightly similar to what I was expecting compared to the theoretical runtimes. When the size is doubled (n), it correlates to doubling the runtime for the programs, but between the two, insertion performed merge sort.

Extra Credit

https://www.random.org/integers/ was used to generate list of random integers. From there I would take the list generated and run those numbers in addition to reversing the algorithms to sort the numbers from largest to smallest for a complete worst case scenario. Additionally, for best case scenario, I used a small list of integers that was almost completely sorted or if not, sorted already. In the worst case, the time increased substantially for both merge and insert as the number of integers increased. In the best case scenario, the times were not much different from one another, being only on average .003 seconds different for merge sort and .006 for the insertion sort.