Lachlan Sinclair

6/26/2019

CS 325: Homework 1

1)

1. **f(n) is Ω(g(n)):** taking the limit of $n^{.75}/n^{.5}$ approaches infinity as n approaches infinity. There does not exist any constant that will allow g(n) to be greater for all n, therefor f(n) is omega to g(n).
2. **f(n) is Θ(g(n)):** $\log(n^2)$ which is equivalent to 2log(n) which grows at the same rate of ln, there for there exists constants c1 and c2 such that 0<=c1g(n)<=f(n)<=c2g(n), for all n>n0 where n0 is some nonnegative integer.
3. **f(n) is Ω(g(n)):** nlogn big O to n*sqrt(n) however n^2 is greater than n*sqrt(n) for all n>1, therefor f(n) is Ω(g(n)). More formally the limit of f(n)/g(n) approaches infinity as n approaches infinity.
4. **f(n) is O(g(n)):** e is about 2.7, therefor there does not exist constant such the f(n) would always be greater or equal to g(n). The limit of f(n)/g(n) approaches 0 as n approaches infinity.
5. **f(n) is Θ(g(n)):** The limit of f(n)/g(n)=2 as n approaches infinity, therefor f(n) is omega to g(n).
6. **f(n) is Ω(g(n)):** The limit of f(n)/g(n) approaches infinity as n approaches infinity. n! = 1*2*3*…*n-1*n, and n^n=n*n*n…*n*n. It is clear that n! grows less than n^n, making f(n) Ω(g(n)).

2)

1. f1((n) = Θ(g(n)) and f2 = Θ(g(n)) then f1(n)+f2(n) = Θ(g(n)).

   From the supposition we can assume there are positive constants c1, c2, c3, c4, n1, n2, n3 such that:
   0<=c1g(n)<=f1(n)<=c2g(n) for all n >n1
   0<=c3g(n)<=f2(g)<=c4g(n) for all n >n2

   Adding f1 and f2 produces:
   0<=(c1+c3)g(n)<=f1(n)+f2(g)<=(c2+c4)g(n) for all n >n3

   c1+c3 equals some positive constant, and so does c2+c4 therefor
   f1(n)+f2(n) = Θ(g(n)).

2. F1(n)=O(g(n)) and f2(n)=O(g(n)) then f1(n)=O(f2(n)).

   Proof by counter example: Suppose f1(n) = n^2, f2(n)=n, and g(n)=n^3.

   It is clear that f1(n)<=c1g(n). for some positive constant c1 for all n>n0.
   And that f2(n)=<c2g(n) for some positive constant c2 for all n>n1.

However there does not exist a constant c3 such that, f1(n)<=c3f2(n). Therefor the original supposition is false.

4)

a)

Insertion sort timed code:

```
/**************
*Author: Lachlan Sinclair
*Date: 6/26/2019
*Description: This program generates an array of size N filled with random numbers
*and then sorts them, then outputs the time in nanoseconds.
*insert.txt
****************/

#include <iostream>
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <ctime>

using namespace std;

const int N = 50000;

//prototype
void insertionSort(int[], int);

int main()
{

	int arr[N];
	timespec time1, time2, differ;
	int temp;
	srand(time(NULL));

	//generate the array of random numbers
	for (int i = 0; i < N; i++)
	{
		arr[i] = rand() % 10000;
	}


	//http://man7.org/linux/man-pages/man2/clock_gettime.2.html
	//set the feilds of the first timespec
	clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time1);

	insertionSort(arr, N);

	//set the feilds of the second timespec
	clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time2);
```

```
        //calculate the difference of the seonds and nanoseconds feilds
        differ.tv_nsec = time2.tv_nsec - time1.tv_nsec;
        differ.tv_sec = time2.tv_sec - time1.tv_sec;

        //output the results
        cout << "N: "<< N << endl;
        //cout << differ.tv_sec << endl;
        cout << "Time in nanoseconds: " << ((differ.tv_sec*1000000000)+differ.tv_nsec) <<
endl;

        return 0;
}

/****************
* Function: insertion sort
* Description: implements the insertion sort method to sort an array
* Input: Address of array, the size of the array
* Output: none
******************/
void insertionSort(int arr[], int size)
{
        for (int i = 1; i < size; i++)
        {
                int val = arr[i];
                int index = i - 1;
                while (index >= 0 && val < arr[index])
                {
                        arr[index + 1] = arr[index];
                        index-- ;
                }
                arr[index + 1] = val;
        }
    }
```

Merge sort timed code:

```
/**************
*Author: Lachlan Sinclair
*Date: 6/26/2019
*Description: This program generates an array of size N filled with random numbers
*and then sorts them, then outputs the time in nanoseconds.
*insert.txt
****************/

#include <iostream>
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <ctime>

using namespace std;

//prototypes
void mergeSort(int arr[], int size);
void merge(int first[], int second[], int lower, int middle, int upper);
void mergeSortHelper(int first[], int second[], int lower, int upper);
```

```
const int N = 50000;

/****************
* Function: mergeSort
* Description: copies the given array, then passes the orginal array address, and the new
array address
* to the recursive mergesort function
* Input: Address of array, size of the array
* Output: none
******************/
void mergeSort(int arr[], int size)
{
	//malloc a new array to allow for merging/spliting into one another
	int *temp = (int*)malloc(sizeof(int)*size);
	//copy the orginal array into the temp array
	for (int i = 0; i < size; i++)
	{
		temp[i] = arr[i];
	}
	//call the helper function
	mergeSortHelper(arr, temp, 0, size);
	//free the temporary array
	free(temp);
	temp = 0;
}

/****************
* Function: merge
* Description: Merges the two halfs of the given range into sorted order
* Input: Addresses of the two array's, the lower/middle/upper indexes of the given range
* Output: none
******************/
void merge(int first[], int second[], int lower, int middle, int upper)
{
	//temp variables to allow for incrementing through the two halfs of the array
subset
	int x = lower;
	int y = middle;

	//loop through the indicies of the given subset
	for (int index = lower; index < upper; index++)
	{
		//if there is still variables left in the upper half, and the lower half is
empty or it is less than the value in
		//the lower half, add the value from the upper half to the current position
in the subset
		if (y < upper && (x >= middle || second[y] <= second[x]))
		{
			first[index] = second[y];
			y++;
		}
		//else add the next value from the lower half
		else
		{
			first[index] = second[x];
			x++;
		}
	}
```

```
}

/****************
* Function: mergeSortHelper
* Description: The recursive mergesort function. Divides the array into halfs, until it 
is aof size one, it aleternates splitting and merging of the array
* into the two provided
* Input: Address of two arrays
* Output: none
*****************/
void mergeSortHelper(int first[], int second[], int lower, int upper)
{
	if (upper - lower < 2)
	{
		return;
	}
	int middle = (upper + lower) / 2;
	//alternate which array is getting merged to set up for merging the two halfs back 
together
	//start by spliting the first array into the second array(this insures merging 
occurs correctly)
	mergeSortHelper(second, first, lower, middle);
	mergeSortHelper(second, first, middle, upper);
	//merge the two sorted halfs together, the final call combined's two halfs from 
the temp array to the oringal array
	merge(first, second, lower, middle, upper);
}



int main()
{

	int arr[N];

	timespec time1, time2, differ;
	int temp;

	srand(time(NULL));

	//create the array of random numbers
	for (int i = 0; i < N; i++)
	{
		arr[i] = rand() % 10000;
	}

	//http://man7.org/linux/man-pages/man2/clock_gettime.2.html
	//set the feilds of the first timespec
	clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time1);

	mergeSort(arr, N);

	//set the feilds of the second timespec
	clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time2);

	//calculate the difference of the seonds and nanoseconds feilds
	differ.tv_nsec = time2.tv_nsec - time1.tv_nsec;
	differ.tv_sec = time2.tv_sec - time1.tv_sec;
```

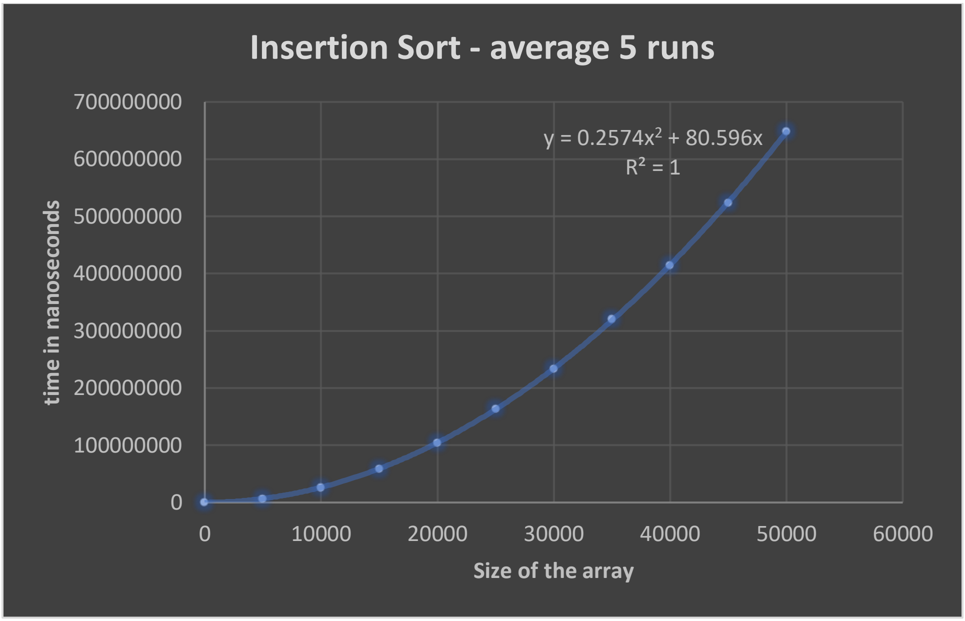
```
	//output the results
	cout << "N: " << N << endl;
	//cout << differ.tv_sec << endl;
	cout << "Time in nanoseconds: " << ((differ.tv_sec * 1000000000) + differ.tv_nsec)
<< endl;

	return 0;
}
```

b)

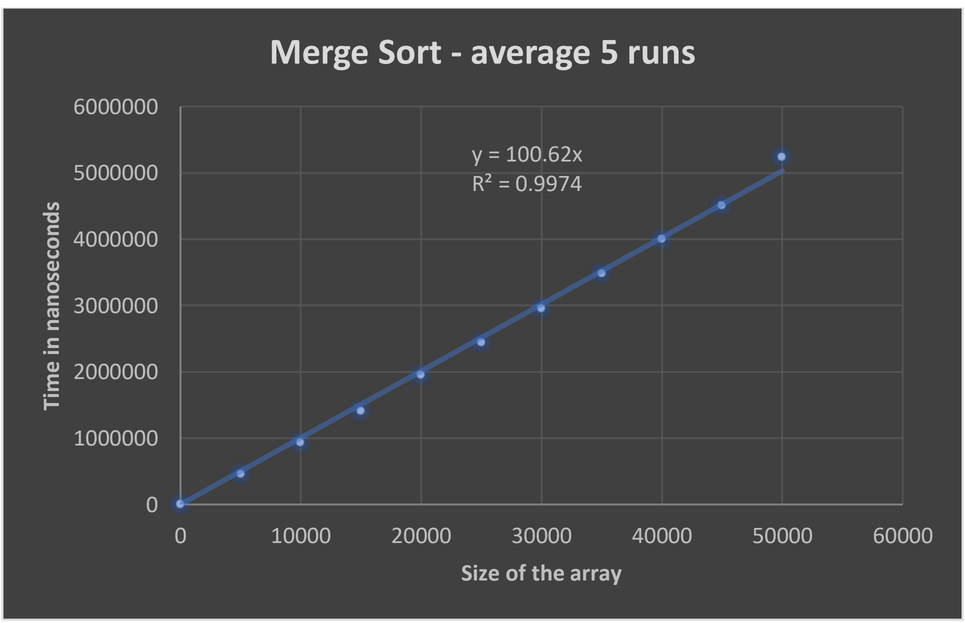
MergeSort						
N	trial 1	trial 2	trial 3	trial 4	trial 5	Average
0						0
5000	448279	457967	476115	448188	469111	459932
10000	943138	916883	912316	960949	960027	938662.6
15000	1398178	1399513	1395784	1455832	1408138	1411489
20000	1922630	1910917	1996724	1992600	1901622	1944898.6
25000	2422759	2431795	2425869	2512516	2423162	2443220.2
30000	2966791	2945618	2957086	2937114	2960497	2953421.2
35000	3463796	3484978	3486337	3494186	3512193	3488298
40000	4012025	4010576	4004838	4010034	3970930	4001680.6
45000	4516411	4472086	4496950	4521055	4524127	4506125.8
50000	5240903	5075710	5797268	5066025	5024848	5240950.8

InsertionSort						
N	trial 1	trial 2	trial 3	trial 4	trial 5	Average
0						0
5000	6533945	6516649	6442082	6631803	6575855	6540066.8
10000	25952578	25992502	25721008	25978014	25880850	25904990.4
15000	57896161	58428079	58348038	58352412	57928288	58190595.6
20000	103079821	103800059	103761754	108317984	102878086	104367540.8
25000	160510839	161477074	160793479	173351008	163543575	163935195
30000	231615210	233470100	232497451	234021755	237236470	233768197.2
35000	337297698	316979532	315402369	315613609	316359518	320330545.2
40000	414718375	413684332	414114508	416100498	412436265	414210795.6
45000	522942648	532548017	520339097	519825388	520127527	523156535.4
50000	646950664	647597656	654956594	646050800	645904833	648292109.4

c)

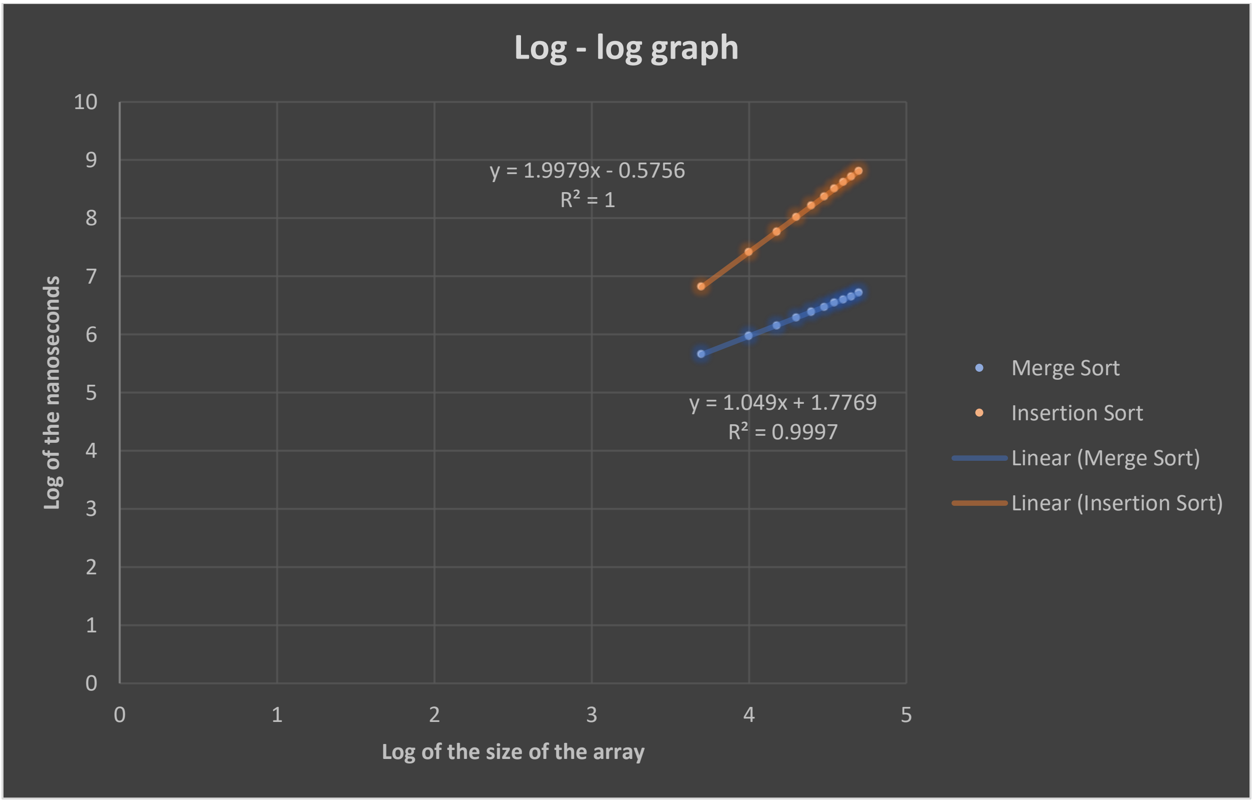


For Insertion sort a polynomial of degree 2 was the best fit for the data, which was expected.



For Merge sort a linear equation was the best fit, this was also expected.

d) The coefficients for the log – log graph ended up being extremely close to the expected values with quite good R values.



e) My experimental data's runtime was extremely close to the expected theoretical runtime. My insertion sort program produced data that followed a trendline of a polynomial with a degree of two. These results are exactly what the theoretical run time is expected to be.

For the Merge sort the data also closely matched the theoretical run time. While it is hard to tell since it returned a linear (n) runtime, this is expected the outer n value in n*log(n) is so much larger than the log(n) value.

For both methods the coefficients in my log- log graph also exactly matched the expected theoretical values.

I did find it odd however that my insertion sort produced a better R^2 value than my merge sort. I had expected merge sort to be more stable since its best case runtime and worse case runtime are exactly the same. This was likely do to something going on behind the scenes of the server.