ASSIGNMENT 1 (RITIKA GOYAL) (301401516)

Ques1:

a) Insertion sort -> $8n^2$
Merge sort -> 64 * n * log(n)

Insertion sort runs faster when it takes less time:
$8n^2 < 64 \, n \log(n)$
$8n^2 - 64 \, n \log(n) < 0$
$8n(n - 8 \log(n)) < 0$

$8n < 0$ or $n-8 \log n<0$
(rejected as n can not be less than 0) $n< 8 \log(n)$

$=> n < 8 \log (n)$
By hit and trial method,
Insertion sort runs faster than merge sort when, $2 \leq n \leq 43$

b) Algorithm of running time $100n^2$ runs faster than algorithm of running time 2^n when:
$100 \, n^2 < 2^n$
When n = 14, $100 * 14^2 = 19600$ and $2^{14} = 16384$ since 19600<16384 is not true but
When n = 15, $100 * 15^2 = 22500$ and $2^{15} = 32768$ since 22500 < 32768 satisfies the equation and thus
By hit and trial, this equation satisfies when n = 15

Ques 2:

(online calculator was used from WolframAlpha to calculate the values of equations such as n*log(n)=10^6)

	1 second = $10^{6\ ms}$ µs	1 minute = 6×10^{7} µs	1 hour = 36×10^8 µs	1 day = 864×10^8 µs	1 month = 2592×10^9 µs	1 year = 31536×10^9
Log (n)	$2^{(10^6)}$	$2^{(6*10^7)}$	$2^{(36*10^8)}$	$2^{(864*10^8)}$	$2^{(2592*10^9)}$	$2^{(31536*10^9)}$
$\sqrt{n}$	10^{12}	36×10^{14}	1296×10^{16}	746496×10^{16}	671846×10^{18}	994519296×10^{18}
n	10^6	6×10^7	36×10^8	864×10^8	2592×10^9	31536×10^9
n log (n)	62746	2801420	133378000	275515×10^4	718709×10^5	797634×10^6
n^2	10^3	7745	6×10^4	293938	1609968	5615692
n^3	100	391	1532	4420	13736	31593
2^n	19	25.84 = 25	31.745 = 31	36.33 = 36	41.24 = 41	44
n!	9	11	12	13	15	16

Ques 3:

Input: A sequence of n numbers A = (a1, a2, .., an) and a value v.

Output: An index i such that v = A[i] or the special value nil if v is not in A

<u>Pseudo code for linear search:</u>

```
For i = 0 to (n-1) do
        if A[i] = v  then return i;
        i++;
return nil;
```

<u>loop invariant</u>: In the beginning of each iteration of for loop, the subarray A[0 … i-1] do not have value v.

To prove that the algorithm is correct, we need to prove the initialization, maintenance and termination of loop invariant.

<u>Initialization</u>: In starting, A is empty and does not have any element so v is not in A, the invariant is true.

<u>Maintenance</u>: In every iteration, it is checked if A[i] = v. if it is equal, then the index i is returned and loop is terminated. If it do not contain v, the subarray A[0 … i-1] will not contain v before next iteration, satisfying loop invariant.

<u>Termination</u>: The loop terminates when v is found in array A at index i v is not found after traversing the array and returns NIL.

Ques 4:

<u>Inputs</u>: Array A and B contains n-bit binary integer

A[0] …. A[n-1] (total length = n)

B[0] ….. B[n-1] (total length = n)

<u>Output</u>: The sum of integers (from A and B) in binary form is added and stored in C

C[0] ….. C[n] (total length = n+1)

Pseudo code:

```
carry = 0;
i = n-1;
while i ≥ 0
       C[i+1] = (A[i] + B[i] + carry) mod 2;
       carry = (A[i] + B[i] + carry) / 2;
       i--;
C[i] = carry;
```

Ques 5:

Binary Search: Input: sorted array A[0 … n-1] , size of array and a value v

Pseudo code for Binary Search:

```
start = 0;
end = size-1;
while start ≤ end
        mid = (start + end) / 2;
        if A[mid] = v   then return v
         else if  v  < A[mid]  then end = mid -1
          else  start = mid +1
return nil;
```

Wort case: When v is not in A.

In Binary Search, after each iteration of the loop, the number of elements to compare with v becomes half.
Running time, $T(n) = c + T(n/2)$

$$= \Theta(1) + \Theta(\log n)$$
$$= \Theta(\log n)$$

Ques6:

A	B	O	o	Ω	ω	Θ
$\log^k n$	n^{ε}	YES	YES	NO	NO	NO
n^k	c^n	YES	YES	NO	NO	NO
2^n	$2^{n/2}$	N0	NO	YES	YES	NO
$n^{\log c}$	$c^{\log n}$	YES	NO	YES	NO	YES
$\log(n!)$	$\log(n^n)$	YES	NO	YES	NO	YES

Ques7:

N (number of elements)	10	20	50	100	200	500	1000
Brute force (time in sec)	3×10^{-6}	3×10^{-6}	9×10^{-6}	2.8×10^{-5}	0.000106	0.000662	0.001752
Divide and conquer (time in sec)	6×10^{-6}	9×10^{-6}	2.3×10^{-5}	4.4×10^{-5}	8.6×10^{-5}	0.000219	0.000217

The minimum n_o that for every $n \geq n_0$, the divide-and-conquer algorithm runs faster than brute-force algorithm is 100.

Source Code:

```
#include <iostream>
#include <ctime>
#include <limits.h>
using namespace std;

//the function returns the array of three elements A[0] = maximum sum, A[1]=left index, A[2]=right index
int* brute_force_sub_array(int A[],int size)
{
	 int max = A[0];
	int sum = 0;
	 int left =0;
	 int right =0 ;

	for(int i=0;i<size; i++)
	{
		sum =0;
		for(int j=i; j<size;j++)
		{
			sum = sum + A[j];
			if(sum>max)
			{
				max = sum;
				left = i;
				right = j;
			}
		}
	}
	int * return_array = new int[3];
	return_array[0]= max;
	return_array[1] = left;
	return_array[2] = right;
	return return_array;
}

//the function returns the array of three elements A[0] = maximum sum, A[1]=left index, A[2]=right index
int* find_max_crossing_subarray(int A[], int left, int mid, int right)
{

	int return_left =0;
	int return_right =0;

	int left_sum = -1000000000;
	int sum =0;
	for(int i=mid; i>=left;i--)
	{
```

```
        sum = sum + A[i];
        if(sum>left_sum)
        {
            left_sum= sum;
            return_left =i;
        }

    }

    int right_sum = -1000000000;
    sum =0;
    for(int j = mid+1; j<= right; j++)
    {
        sum = sum + A[j];
        if(sum>right_sum)
        {
            right_sum = sum;
            return_right = j;
        }
    }

    int * return_array = new int[3];
    return_array[0]= left_sum + right_sum;
    return_array[1] = return_left;
    return_array[2] = return_right;
    return return_array;

}

int* divide_conquer_sub_array (int A[], int left, int right)
{
    if(left == right)
    {
        int* return_array = new int[3];
        return_array[0]= A[left];
        return_array[1] = left;
        return_array[2] = right;
        return return_array;
    }
    else
    {
        int mid = (left + right)/2;
        int* left_part = divide_conquer_sub_array(A, left,mid);
        int* right_part = divide_conquer_sub_array(A,mid+1, right);
        int* centre_part = find_max_crossing_subarray(A,left,mid,right);
```

```
		if(left_part[0]>= right_part[0] && left_part[0]>=centre_part[0])
		{
			delete [] right_part;
			delete[] centre_part;
			return left_part;
		}
		else if (right_part[0]>= left_part[0] && right_part[0]>= centre_part[0])
		{
			delete [] centre_part;
			delete [] left_part;
			return right_part;
		}
		else
		{
			delete [] left_part;
			delete [] right_part;
			return centre_part;
		}
	}
}



int main() {
	srand(time(NULL));
	int Array_n[] = {10,20,50,100,200,500,1000};

	for(int i =0 ; i<7; i++)
	{
		cout<<"FOR N = "<< Array_n[i] <<" :"<<endl;
		int* array_A = new int[Array_n[i]];

		for(int j =0 ; j<Array_n[i]; j++)
		{
			array_A[j] =  rand()%200 + (-100);
			//cout<<array_A[j]<<" ";
		}

		cout<<endl;

		//Brute Force
		clock_t time_brute_force = clock();
		int* brute_force_result = brute_force_sub_array( array_A,Array_n[i]);
		time_brute_force = clock() - time_brute_force;

		cout<<"BRUTE FORCE:"<<endl;
```

```
			cout<<"TIME (Brute Force) = "<< (float) time_brute_force / CLOCKS_PER_SEC<<" seconds"<<endl;
			cout<<"max sum = "<<brute_force_result[0]<<endl;
			cout<<"left index = "<<brute_force_result[1]<<endl;
			cout<<"right index = "<<brute_force_result[2]<<endl<<endl;
			delete [] brute_force_result;

			//Divide and Conquer
			clock_t time_divide_conquer = clock();
			int* divide_conquer_result = divide_conquer_sub_array(array_A, 0, Array_n[i]-1);
			time_divide_conquer = clock() - time_divide_conquer;

			cout<<"DIVIDE AND CONQUER"<<endl;
			cout<<"TIME (Divide and Conquer) = "<< (float) time_divide_conquer / CLOCKS_PER_SEC<<" seconds"<<endl;
			cout<<"max sum = "<<divide_conquer_result[0]<<endl;
			cout<<"left index = "<<divide_conquer_result[1]<<endl;
			cout<<"right index = "<<divide_conquer_result[2]<<endl<<endl;;
			delete[] divide_conquer_result;

			delete [] array_A;
	}
	return 0;
}
```

Ques8:

a) T(n) = 2T(n/4) +1

Comparing the above equation with T(n) = a T(n/b) + f(n), we get f(n) = 1, a =2, b=4

Solving $n^{(\log_b a)} = n^{(\log_2 4)} = n^{1/2} = \sqrt{n}$

Since f(n) = 1 grows slower than $n^{(\log_b a)} = \sqrt{n}$, Case 1 is used.

So, T(n) is $\Theta(n^{(\log_b a)}) = \Theta(\sqrt{n})$.

b) $T(n) = 2T(n/4) + \sqrt{n}$

Comparing the above equation with T(n) = a T(n/b) + f(n), we get $f(n) = \sqrt{n}$, a =2, b=4

Solving $n^{(\log_b a)} = n^{(\log_2 4)} = n^{1/2} = \sqrt{n}$

Since $f(n) = \sqrt{n}$ is equal to $n^{(\log_b a)} = \sqrt{n}$, Case 2 is used.

So, T(n) is $\Theta(n^{(\log_b a)} \log(n)) = \Theta(\sqrt{n} \log (n))$.

c) T(n) = 2T(n/4) + n

Comparing the above equation with T(n) = a T(n/b) + f(n), we get f(n) = n , a =2, b=4

Solving $n^{(\log_b a)} = n^{(\log_2 4)} = n^{1/2} = \sqrt{n}$

Since f(n) = n grows faster than $n^{(\log_b a)} = \sqrt{n}$, Case 3 is used.

So, T(n) is $\Theta(f(n)) = \Theta(n)$.

d) $T(n) = 2T(n/4) + n^2$

Comparing the above equation with $T(n) = a\,T(n/b) + f(n)$, we get $f(n) = n^2$, a =2, b=4

Solving $n^{(\log_b a)} = n^{(\log_2 4)} = n^{1/2} = \sqrt{n}$

Since $f(n) = n^2$ grows faster than $n^{(\log_b a)} = \sqrt{n}$, Case 3 is used.

So, T(n) is $\Theta(f(n)) = \Theta(n^2)$.