Algorithms - final assignment

Abhishek Rajak

(CRS1912)

Q1.

a.) Least common multiple(LCM): LCM of 2 positive numbers is the smallest number which is the multiple of both the numbers.

LCM(A,B) = (A*B)/GCD(A,B)

where GCD(A, B) is the greatest common divisor of two positive integers A and B.

Algorithm:

1. calculate gcd(greatest common divisor).
    a. if we subtract the smaller number from the bigger number the gcd of two numbers don't change, so we can repetitively subtract the smaller number from the bigger number until both the numbers become equal and that will be our GCD.
    b. repetitive subtraction is nothing but the division of two numbers, we will calculate the remainder and our algorithm stops when the remainder becomes 0.
2. divide product of A and B by gcd of A and B.

```
leastCommonMultiple ( A, B ) {

        lcm = A*B;
        while(B!=0){
                temp = a%B;
                A = B;
                B = temp;
        }
        gcd = A;
        lcm = lcm/gcd;

        return lcm;
}
```

b.) We are finding the LCM of two numbers a and b by dividing their product (a*b) by their GCD. So, the complexity of our algorithm depends on how efficiently we calculate the GCD of these 2 numbers.

In our algorithm apart from the while loop, all the other operations can be carried out in constant time. In each iteration of the while loop, we are reducing the larger number to the remainder of the two numbers, so there are two possibilities:

1. B is greater than or equal to half of A, and we will replace B with (A % B) and that will make B at most half of its previous value.
2. B is less than half of A, and we will replace A with B and that will make A at most half of its previous value, since B is less than A / 2.

So, in each iteration of the while loop, the algorithm is reducing at least one number to at least half less. So, after at most $\mathcal{O}(log_2 A)+\mathcal{O}(log_2 B)$ steps, we will get the gcd.

Hence, the final complexity of our algorithm is $\mathcal{O}(log_2 A + log_2 B)$ and since the input size N(number of bits in A and B) is equal to $log_2 A + log_2 B$, so the final complexity in terms of N is $\mathcal{O}(log_2 N)$.

c.) C code of the above algorithm:

```
#include<stdio.h>

long long int leastCommonMultiple (int a, int b ) {

        long long int lcm = a*b;
        while(b!=0){
                int temp = a%b;
                a = b;
                b = temp;
        }
        int gcd = a;
        lcm = lcm/gcd;

        return lcm;
}
```

```
int main(){

        int a = 15;
        int b = 24;

        long long int lcm = leastCommonMultiple(a,b);

        printf("%lld",lcm);

        return 0;
}
```

We have taken two numbers 15 and 24, and from the main function, we are calling the leastCommonMultiple() function which calculates the LCM of two numbers. We have taken care of the integer overflow by using long long data type for the lcm variable and the method type, as the product of two 32-bit integers can cause integer overflow.

Q2.

a.) We will use the direct method to calculate the Standard Deviation.

Algorithm to calculate the Standard Deviation using direct method:

1. Given an array of unsigned integers of size n.
2. Calculate the mean of the array by adding all the numbers and divide it by the n.
3. Replace each array element with its difference from the mean calculated in the previous step.
4. Replace each array element with its square, take care of the integer overflow.
5. Sum all the elements of the array and divide it with n.
6. Take the square root of the value obtained in step 4.

b.) As all the array elements are unsigned integers of size 4 bytes, so their summation for calculating mean would cause integer overflow, we can avoid this by declaring mean as double variable type.

In this algorithm based on the direct formula, we subtract the mean from each array element. Let's suppose the array elements are of order of 10^9 so the mean will also be around order of 10^9 i.e., the mean has around 9 significant figures. A double precision floating point number in C language contains about 15 significant figures, so we can retain about 6 figures in each calculation. Therefore our final result can be accurate to around 6 figures and we would lose 9 significant figures in subtracting off the mean.

c.) We can avoid the integer overflow by using double type variables for all the summations. We can use double also for the mean and standard deviation variable.

C code for the above algorithm:

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

double mean(int a[ ], int n){
        int i;
        double sum = 0.0;
        for( i = 0; i < n; i++ ) {
                sum = sum + a [ i ];
        }

        return sum/n;
}
```

```
double standardDeviation(int a[ ], int n) {

        // calling mean function to calculate the mean of the whole array
         double x = mean( a, n );

         int i;

         // replace each array element with its difference from the mean
         for( i = 0; i < n; i++) {

                  a [ i ] = a [ i ] - x;

         }


         double sum = 0.0;
         for( i = 0; i < n; i++) {

                  sum = sum + 1.0*a [ i ]*a [ i ];

         }

         sum = sum/n;

         return sqrt(sum);
}

int main(){

         int a [ ] = { 25, 40, 38, 67, 92, 48, 73, 55, 63, 71 };

         int n = sizeof(a) / sizeof(int);

         double sd = standardDeviation( a, n );

         printf("%lf ", sd);

         return 0;
}
```

Q3.

a.) C program for merge sort:

```
#include<stdio.h>

void merge(int a[ ], int left, int mid, int right) {

        int n1, n2, i, j, k;

        n1 = mid - left + 1;            //      size of first subarray
        n2 = right - mid;               //      size of second subarray
        int temp [ n1 + n2 ];           //      temporary array to store current elements of a

        for( i = 0; i < n1; i++)  temp [ i ] = a [ left + i ];
        for( i = 0; i < n2; i++)  temp [ n1 + i ] = a [ mid + 1 + i ];

        // merging the elements back to the original array in sorted order by comparing them
        i = 0,j = n1,k = left;
        while(i < n1 && j < n1 + n2) {
                if(temp [ i ] <= temp [ j ]) {
                        a [ k++ ] = temp [ i ];
                        i++;
                }else {
                        a [ k++ ] = temp [ j ];
                        j++;
                }
        }

        // copying remaining elements if any that were left in first part of temporary subarray
        while(i < n1) {
                a[ k++ ] = temp [ i ];
                i++;
        }

        // copying remaining elements if any that were left in second part of temporary subarray
        while(j < n1 + n2) {
                a [ k++ ] = temp [ j ];
                j++;
        }

}
```

```
void mergeSort(int a[ ], int left, int right) {

        if(left < right) {

                int mid = (left + right)/2;

                // recursively call self till the array size reduces to 1 element
                mergeSort(a, left, mid);
                mergeSort(a, mid + 1, right);
                merge(a, left, mid, right);
        }
}


int main() {

        int n, i;

        int a[ ] = { 11, 79, 15, 60, 1, 100, 69, 31, 47 };
        n = sizeof(a) / sizeof(int);

        mergeSort(a, 0, n-1);

        for( i = 0; i < n; i++)             printf("%d ", a[ i ]);

        return 0;
}
```

time complexity: As in every step we are reducing the array into two arrays of half the size of initial array, so the time complexity can be represented with recursive equation:

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

following recurrence relation converges to the $\Theta(n\ log2\ n)$.

b.) Code to sort generated numbers in ascending order based on the previous code implementation of merge sort:

```c
#include <stdio.h>
#include <math.h>

void merge(int a[ ], int left, int mid, int right) {

	int n1, n2, i, j, k;

	n1 = mid - left + 1;		//	size of first subarray
	n2 = right - mid;		//	size of second subarray
	int temp [ n1 + n2 ];		//	temporary array to store current elements of a

	for( i = 0; i < n1; i++)  temp [ i ] = a [ left + i ];
	for( i = 0; i < n2; i++)  temp [ n1 + i ] = a [ mid + 1 + i ];

	// merging the elements back to the original array in sorted order by comparing them
	i = 0,j = n1,k = left;
	while(i < n1 && j < n1 + n2) {
		if(temp [ i ] <= temp [ j ]) {
			a [ k++ ] = temp [ i ];
			i++;
		}else {
			a [ k++ ] = temp [ j ];
			j++;
		}
	}
	// copying remaining elements if any that were left in first part of temporary subarray
	while(i < n1) {
		a[ k++ ] = temp [ i ];
		i++;
	}
	// copying remaining elements if any that were left in second part of temporary subarray
	while(j < n1 + n2) {
		a [ k++ ] = temp [ j ];
		j++;
	}
}

void mergeSort(int a[ ], int left, int right) {
	if(left < right){
		int mid = (left + right)/2;

		// recursively call self till the array size reduces to 1 element
		mergeSort(a, left, mid);
		mergeSort(a, mid + 1, right);
		merge(a, left, mid, right);
	}
}
```

```
int main( ) {

        int n, i, a [ 16 ], x [ 18 ];

        // utility to generate the numbers (birthday is 1st July 1990)
        x [ 0 ] = 1, x [ 1 ] = 1;
        for(i = 2; i < 18; i++) {
                x [ i ] = (int)pow(x [ i - 1 ] + 1, 2) % 97 + x [i - 2];
                a[ i - 2 ] = x[ i ];
        }

        n = sizeof(a) / sizeof(int);
        mergeSort(a, 0, n-1);

        for( i = 0; i < n; i++)                printf("%d ", a[ i ]);

        return 0;
}
```

Q6.

a.) Longest Common Subsequence(LCS): Longest common subsequence is a subsequence of maximum length that is present in all the sequences of the given set and in which the characters appear in the same relative order as in the original sequences. Unlike substring, subsequence doesn't have to be made up of consecutive characters.

Consider a set S with two sequences "INDI" and "INSTI", the longest common subsequence of this set S will be "INI" (INDI, INSTI). Let's find out how?

Length of subsequence	String "INDI"	String "INSTI"
1	"I", "N", "D"	"I", "N", "S", "T"
2	"IN", "ID", "II", "ND", "NI", "DI"	"IN", "IS", "IT", "II", "NS", "NT", "NI", "ST", "SI", "TI"
3	"IND", "INI", "NDI"	"INS", "INT", "INI", "NST", "NSI", "STI"
4	"INDI"	"INST", "INSI", "ISTI", "NSTI"
5	-	"INSTI"

From table we can clearly see that the common subsequences of length 1 are "I" and "N", of length 2 are "IN", "II" and "NI", of length 3 are "INI" and there are no subsequences of length 4 and 5, so the answer is "INI".

b.) Algorithm to calculate LCS using dynamic programming paradigm as the problem contains overlapping substructures.

1. n = length(X), m = length(Y)
2. Initialize a 2D array dp of size n*m with 0.
3. for each index i in X:
   for each index j in Y:
   if the last character of both sequences are equal i.e., X[i-1] == Y[j-1] then:
   $dp[i][j] = dp[i-1][j-1] + 1$
   else if the last character till now is not equal then we can take the maximum value obtained by removing the last character from both the sequences:
   $dp[i][j] = max(dp[i][j-1], dp[i-1][j])$
4. The length of LCS of sequences X and Y will be equal to the value of dp[n][m].

5. To obtain the LCS we need to backtrack.
    a. Initialize a char array lcs[ ] of size dp[n][m]+1;
    b. Start from the bottom-right corner and one by one store characters in the end in lcs[ ].
        i = n, j = m, k = 0;
    c. If the current character X[i] and Y[j] are equal then this element is part of LCS and should be added to the lcs[ ] array and decrement i and j by 1.

```
if X[ i ] == Y[ j ]:
        lcs [ k ] = X[ i ];
        i = i - 1;
        j = j - 1 ;
```

    d. If it is not the same, then we need to move in the direction where the value of dp is maximum, so if dp[i][j-1] > dp[i-1][j] then decrement j and goto to step c and repeat and if dp[i-1][j]>dp[i][j-1] then decrement i and goto step c and repeat.

```
if dp[ i ][ j - 1 ] > dp[ i - 1][ j ]:
        j = j - 1;
else :
        i = i - 1;
```

    e. repeat the steps c and d until i and j are greater than 0.
6. LCS of sequences X and Y will be lcs [ ].
7. end.

c.) Time complexity: As we are calculating the length of subsequence for each character of X with each character of Y, so this part will contribute a factor of O(n*m) to the total time complexity.

To print the LCS we are backtracking the 2d array dp of size n*m so this will part will also contribute a factor of O(n*m) to the total time complexity.

Hence, the total time complexity of this algorithm will be O(n*m) where n is length of X and m is length of Y, which is way better than the naive approach that calculates all the 2^n subsequences of X and match them with Y making the time complexity of naive algorithm to be O(m* 2^n).

Q7.

a.) Greedy algorithms work by choosing the optimal solution at each step, that is each step on adding to our solution provides us an immediate benefit. This solution doesn't have to be the most optimal answer for the whole problem, but this solution is locally optimal. At each step we make some decision, usually to include or exclude some particular element from our solution; we never backtrack or change our mind.

b.) We will be finding the minimum spanning tree with the help of Kruskal Algorithm. Kruskal algorithm greedily chooses edges at each step which have less weight such that it does not form a cycle.

Problem Statement: Let there be an undirected weighted graph G with number of vertices V and number of edges E where weight on every edge is denoted by array W. Find the minimum spanning tree for graph G.

Algorithm:

1. Sort the edges of the graph on the basis of their weight.
2. Start adding edges one by one to the MST based starting from the edge with the smallest weight. Take the smallest edge and increment the index for the next iteration.
    a. If including this edge doesn't cause a cycle then, include it in the result using the disjoint set union and increment the index of the result for the next edge.
    b. Else discard the next edge.
3. Repeat Step 2 until the edge with the largest weight is encountered.

C++ code of the above algorithm:

```
#include <stdio.h>
#include <vector>
#include <algorithm>

using namespace std;

struct edge {
        int source, destination, weight;
};

bool customComparator( edge const& a, edge const& b ) {
        return a.weight < b.weight;
}

vector < edge > e(20), MST;
vector < int > parent(10), rank(10);
```

```
void make ( int u ) {
	parent [u] = u;
	rank [ u ] = 0;
}

int find ( int u ) {
	if ( u == parent [ u ] ) {
		return u;
	}
	return parent [ u ] = find( parent[ u ] );
}

void customUnion ( int u, int v ) {

	u = find ( u );
	v = find ( v );

	if ( u != v ) {
		if ( rank[ u ] < rank [ v ] ) {
			swap( u, v);
		}
		parent [ v ] = u;

		if( rank [ u ] == rank [ v ] ) {
			rank [ u ] = rank [ u ] + 1;
		}
	}
}

void addEdge(int i, int u, int v, int w) {
	e[ i ].source = u;
	e[ i ].destination = v;
	e[ i ].weight = w;
}

int main( ) {

	int V=10, E=20, minCost = 0, u, v, w;

	addEdge(0, 0, 1, 4);
	addEdge(1, 0, 7, 8);
	addEdge(2, 1, 2, 8);
	addEdge(3, 1, 7, 11);
	addEdge(4, 2, 3, 7);
	addEdge(5, 2, 8, 2);
	addEdge(6, 2, 5, 4);
	addEdge(7, 3, 4, 9);
```

```
        addEdge(8, 3, 5, 14);
        addEdge(9, 4, 5, 10);
        addEdge(10, 5, 6, 2);
        addEdge(11, 6, 7, 1);
        addEdge(12, 6, 8, 6);
        addEdge(13, 7, 8, 7);
        addEdge(14, 7, 9, 1);
        addEdge(15, 8, 9, 12);
        addEdge(16, 9, 6, 6);
        addEdge(17, 9, 1, 2);
        addEdge(18, 0, 4, 13);
        addEdge(19, 6, 3, 6);

        for( int i = 0; i < V; i++) {
                make( i );
        }

        sort( e.begin(), e.end(), customComparator);

        for ( int i = 0; i < E; i++) {

                u = e [ i ].source;
                v = e [ i ].destination;
                w = e [ i ].weight;

                if ( find( u ) != find(v) ) {

                        minCost += w;
                        MST.push_back( e [ i ] );
                        customUnion( u, v );
                }
        }

        printf("Minimum cost of the spanning tree: %d\n",minCost);
        for( unsigned int i = 0; i < MST.size(); i++ ) {
                printf("%d --- %d = %d\n",MST[i].source, MST[i].destination, MST[i].weight);
        }

        return 0;
}
```

c.) Proof of correctness:

Let G be a connected weighted graph and running Kruskal's algorithm on graph G, outputs MST, a minimum weight spanning tree.

MST is a spanning tree because there are no cycles as we have always rejected the edge that can form a cycle in our output. We will prove that the MST cannot be disconnected by contradiction.

Suppose that MST is not connected then, MST has two or more connected components. But, as G is connected, then there must be at least an edge to connect these components in G, and not in MST. The first of these edges (in sorted order on the basis of weight) would have been included in MST because it could not have created a cycle, which contradicts the definition of MST. So, MST must be connected and contain all vertices of G.

Hence, it is proved that the MST is a spanning tree since neither it contains cycles nor it is connected.

Now, we will prove that it is also the minimum cost spanning tree using contradiction as well.

Suppose MST is not minimal then there must exist a minimum spanning tree MST' with edges that are not in MST.

Now consider an edge E which was the first edge to be added to MST by Kruskal's algorithm and it is not in MST'. Then MST'+{E} must contain a cycle, and since MST does not contain cycles, one of the edges in this cycle must not be in MST. Let this edge be E'. Therefore MST" = MST + {E} - {E'} is also a spanning tree.

Since MST is a minimum spanning tree, then weight(E) should be greater than or equal to weight of E'. But, as E was chosen by Kruskal's algorithm, it must be of minimal weight, therefore weight of E should be less than or equal to weight of E'.
Now, only way this is possible if the weight of edge E is equal to weight of E'. Hence, MST" is also a minimum spanning tree, but it has one more edge in common with MST than MST' does, but as per our assumption MST' has the most number of edges in common with MST, so we arrive at a contradiction. Hence, we can conclude that MST is minimal.

d.) Time Complexity: In our implementation of Kruskal's algorithm, most time consuming operation is sorting which is of order $O(ElogE)$. We have used a disjoint-set data structure to keep the track of vertices in which components they belong. We placed each vertex into its own disjoint set, which performs $O(V)$ operations.

In the worst case scenario, we need to iterate through all the edges, and for each edge we need to do two 'find' operations and possibly one union. Our implementation of disjoint-set data structure performs $O(E)$ operations in $O(E\ log\ V)$ time.

So, the total time complexity will be $O(E\ log\ E\ +\ E\ log\ V)$ but, as the maximum number of edges can be V^2.

Therefore,

$$E \leq V^2$$
$$log\ E \leq log\ V^2$$
$$logE \leq 2logV$$
$$logE \simeq logV$$

So, the total time complexity of our algorithm is $O(Elog\ V)\ or\ O(ElogV)$.