Question 1

(1): Apply similarly Dijkstra's algorithm == each node is just the result of multiplying of the cost and keeping the largest instead of adding the cost and keeping the smallest values.

```
INIT(G, s)
        For (each v in V)
                d[v] <-  0; //adjusted
                                        π  [v] -> NULL ;
                d[s] <- 1; //adjusted
RELAX(u,v)
              If  d[v] < d[u]  *r(u,v)
                d[v] <- d[u] *r(u, v)

                                π  [v] <-u
Dijkstra_algorithm(G,s)
              INIT(G,s)
              Set <- Max-queue
              Max_queue <- V[G]
                    While max-queue is not equal to Null Set
              u <- Extract-Max(max_queue)
              Set <- Set U *{u}
                    For(v in Adj[u])
              RELAX(u,v);
```

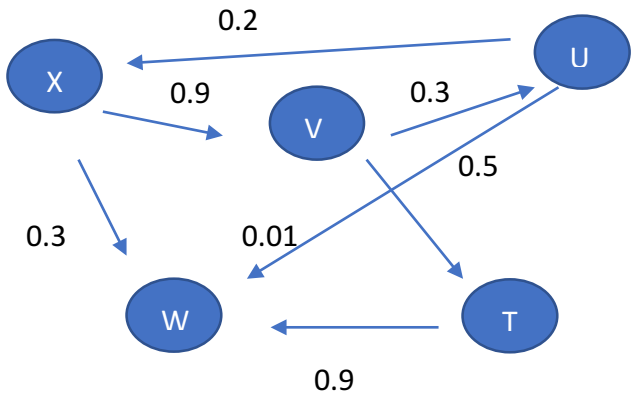
(2)Note:

Set:= set of discovered nodes. Max-queue is the max-priority queue key based on the d[v] values. Time complexity: INIT(G,s) is O(v)

Time complexity: building the max-queue is O(E log V).

Total time complexity is O(E log V).

(3)



Node	Most Reliable Path	Reliability Value

X	U -> X	0.2
V	U -> X -> V	0.18
T	U -> X -> V -> T	0.09
W	U -> X -> V -> T -> W	0.081

Question 2

Answer:
YES, we can use Dijkstra algorithm to solve this problem.
Assuming there is no shorter way to go to that particular node through another non-visited node, we can conclude that the nodes we have visited are the exact shortest path. No matter which edge we will add in the future, it will make the weights larger. However, if we add the negative weights, there could exist some shorter path than the shortest we have assumed. Therefore, it only the edges leaving S may have the negative weights and we do not need to worry about having even shorter path for others are guaranteed to be positive.

Question 3
(a)
A. Select the activity that ends first
B. Once selected, deleted all incompatible rides
C. Repeat the algorithm for remaining rides

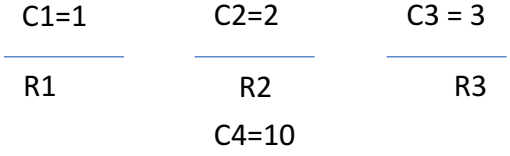
1. Array_S and Array_E based on the end times =>>O(nlogn)
2. Recursive –Selection –Activity(Array_S, Array_E, k, n) =>> O(n)

```
m = k + 1;
while (m <= n) && (Array_S[m] < Array_E[k])
        m++;
if (m <= n)
        return {Rm} U Recursive –Selection –Activity(Array_S, Array_E, k, n)
```

3.Else

```
return null set
```

Analyse:
Total time complexity is O(n logn).
Selecting the activity that ends first, once selected an activity, repeating the algorithm for the remaining activates, then we can get a global optimal.
(2)



R4

In THIS CASE, Alex's result will be {R1, R2, R3}, but Mark's will be {R4}

(c)
Following:

1. First sort jobs according to finish time.
2. Apply following recursive process.

```
//arr[] is array of n rides
findmaximumCredit(arr[], n)
{
        a) if (n ==1) return arr[0]
        b) Return the maximum of following two credits
                (i)  Maximum credit by excluding current
                     ride(Findmaximumcredit(arr, n-1))
                (ii) Maximum credit by including the current ride
}
```

Here are the code in my thought.

```
//a ride has start time , end time and credits
struct ride
{
        startTime, endTime, Credit
}

//A function that is used for sorting events based on the end time
bool rideComparataor (ride 1, ride2 )
{
        return (ride1.end < ride2.end)
}

//find the latest ride (in sorted array) that doesn't conflict with the ride[i]
int latestNoConflict(ride arr[], int i)
{
        for (int j = i-1; j >= 0; j--)
        {
                if(arr[j].end <= arr[i].start)
                        return j;
        }
        return -1;
}
```

```
int findMaxCredit(ride arr[], int n)
{
	//sort jobs according to end time
	sort(arr, arr+n, rideComparator);

	//create an array to store solutions of sun problems. Table[i] stores the credit for rides till arr[i](includeing arr[i])
	int *table = new int[n];
	table[0] = arr[0].credit

	//fill entries in [] using recursive property
	for(int i = 1; i < n; i ++)
	{
		//find credit including the current ride
		int includeCredit = arr[i].credit
		int latest = latestNoConflict( arr, i);
		if (latest != -1)
			includeCredit += table[latest];
			//store maximum of including and excluding
			table[i] = max(includeCredit, table[i-1]);
	}
		//store result and free dynamic memory allocated for table[]
			int result = table[n-1];
			delete[] table;
		return result;


}

int main()
{
	ride arr[] = {{…}, {…}, {…}}
	int n = sizeof(arr)/sizeof(arr[0]);
	findMaxCredit(arr, n);
	return 0;
}
```

time complexity of the above is O(n^2)

Update: in the latestNoConflict can use the Binary Search instead of liner search

Which -> **O(nlogn) -> final**

Space complexity of the above is O(n)

Question4:

4

	G	T	A	A	T	C	A	T	T	T	A	A
G	0	2	4	6	8	10	12	14	16	18	20	22
A	2	2	2	4	6	8	10	12	14	16	18	20
T	4	2	4	4	4	6	8	10	12	14	16	18
T	6	4	4	6	4	6	8	8	10	12	14	16
A	8	6	4	4	6	6	6	8	10	12	12	14
C	10	8	6	6	6	6	8	8	10	12	14	14
T	12	10	8	8	6	8	8	8	8	10	12	14
G	14	12	10	10	8	7	9	10	10	10	12	14
A	16	14	12	10	10	9	7	9	11	12	10	12
T	18	16	14	12	10	11	9	7	9	11	12	12
A	20	18	16	14	12	12	11	9	9	11	11	12

S1 : G T A A T C A T T T A T A

S2: G A T T A C T G A T A

Delete T -> cost 2
Delete A -> cost 2
Align(C, T) -> cost 2
Align(T,C) -> cost 2
Align(T, G) -> cost 2
Insert T -> cost 2

Total cost : 12