1.
(6 Points) Using the very definition of Big-Theta notation, prove that 2^{n+1} is $\theta(2^n)$. You must use the definition and find the constants in the definition to receive credit.

Solution:
Big-Theta notation is a means of representing a function's asymptotic behavior. This form, $\theta(x)$ represents the function's growth rate as x approaches infinity. Because we only care about the highest power of a variable in an asymptotic calculation, the second term of 2n+1 dominates at positive infinity. Big-Theta notation is often used to represent how long different algorithms take to run.

The Big Theta notation is used to identify the dominant term of a function as n approaches infinity. This analysis, $\theta(n)$ represents the function's growth rate as n approaches infinity. The function 2^{n+1} is (2^n) because as n approaches infinity, 2^{n+1} grows at the same rate as 2^n.

Big-Theta notation is called a tight bound because it represents the exact order of growth of the function. When a function's growth rate is within some constant multiplicative factor of another process, we say that the function's growth rate is within that bound.

2.
7 Points) Given that T(n) = 1 if n=1 and T(n) = T(n-1)+ n otherwise; show, by induction, that T(n) = n(n+1)/2. Show all three steps of your induction explicitly.

Solution:

Step 1: Check whether the given statement is true for n = 1. We have T(n) = n(n+1)/2.
Substituting n = 1, we get T(1) = 1 *(1 + 1) / 2 = 1

Step 2: Assume that the statement is True when n = k
So we are assuming that T(k) = k(k + 1) / 2

Step 3: Now we have to prove that the statement is true for n = k + 1

We will use the equation T(n) = T(n - 1) + n

Substituting n = k + 1 we get

T(k+ 1) = T(k + 1 - 1) + k + 1
= T(k) + k + 1

Since we know that T(k) = k(k + 1) / 2

T(k + 1) = (k (k + 1)) / 2 + (K + 1) = $(k^2 + 3k + 2) / 2$ = (k + 1)(k + 1 + 1) / 2

From the above statement it is proved that
T(n) = n * (n + 1) / 2

3.
(7 Points) Given the recurrence relation T(n) = 7 T(n/5) + 10n, for n>1; and T(1)=1. Find T(625).

Solution:
This is the process of repeating the loop or piece of code until the integer value is fulfilled.
This is commonly used to compute the values of a given integer without using a loop or iteration.

For the provided condition, these values were computed for each recursion call.

For n>1, T(625)=7T(125)+10n = 40551 + 6250 = 46801.

For n>1, T(125)=7T(25)+10n = 4543 + 1250 = 5793

For n>1, T(25)=7T(5)+10n = 399 + 250 = 649

For n>1, T(5)=7T(1)+10n = 57

As a result of the recursion, the result value is 46801.

4.

```
#include <stdlib.h>
#include <string.h>
// a structure to represent an edge in graph
struct Edge
{
    int src, dest;
};
// a structure to represent a graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;
    // graph is represented as an array of edges
    struct Edge* edge;
};
// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;
    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );
    return graph;
}
// A utility function to find the subset of an element i
```

```
int find(int parent[], int i)
{
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}
// A utility function to do union of two subsets
void Union(int parent[], int x, int y)
{
    int xset = find(parent, x);
    int yset = find(parent, y);
    if(xset!=yset){
       parent[xset] = yset;
    }
}
// The main function to check whether a given graph contains
// cycle or not
int isCycle( struct Graph* graph )
{
    // Allocate memory for creating V subsets
    int *parent = (int*) malloc( graph->V * sizeof(int) );
    // Initialize all subsets as single element sets
    memset(parent, -1, sizeof(int) * graph->V);
    // Iterate through all edges of graph, find subset of both
    // vertices of every edge, if both subsets are same, then
    // there is cycle in graph.
    for(int i = 0; i < graph->E; ++i)
    {
        int x = find(parent, graph->edge[i].src);
        int y = find(parent, graph->edge[i].dest);
        if (x == y)
            return 1;
        Union(parent, x, y);
    }
    return 0;
}
// Driver program to test above functions
int main()
{
    /* Let us create the following graph
         0
        |  \

        |    \

        1-----2 */
    int V = 3, E = 3;
    struct Graph* graph = createGraph(V, E);
```

```
    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    // add edge 1-2
    graph->edge[1].src = 1;
    graph->edge[1].dest = 2;
    // add edge 0-2
    graph->edge[2].src = 0;
    graph->edge[2].dest = 2;
    if (isCycle(graph))
        printf( "graph contains cycle" );
    else
        printf( "graph doesn't contain cycle" );
    return 0;
}
```

The time complexity of the above algorithm for union-find structure is O(logn) and the space complexity Is O(n).

5.
Solution:

We will be implementing 2 stacks s1 & s2 to implement the queue. Queue follows first in and first out Methodology where the elements are added first and will be removed first.

a) Let's calculate the time complexity of the dequeue operation. We know that the add operation takes O(1) Time when we implement a queue using two stacks. Let's say we enqueue 3 elements 1, 2, 3, and let's say We perform a dequeue operation. When we perform dequeue it should return 1 but stack follows first in Last out methodology. So the main question arises of how will be implemented.

When elements 1, 2, and 3 are added in stack s1 and when we perform the dequeue operation, we will pop all the elements from stack s1 and add it to s2. Then we pop the top element from s2 and Return it.

Once the element is returned we will again pop all the elements from stack s2 and push it to s1.

The time complexity of the above algorithm is O(2N) which is equivalent to O(N) and the space complexity Is O(N).

b)
Assuming the add, copying an element operation from one stack to another takes one cyber dollar each, We charge three cyber dollars for adding
to the stack, saving two cyber dollars for future copying the elements from one stack to another

In the worst case, let's say n add operations are used. So the total cyber dollar which will be used is 3n cyber dollars and for the n deque operations, we will be charged n cyber dollars. Since is given that we are performing one Dequeue operation, so

the worst case of a single dequeue operation will become O(1).

6.
Consider an n by n matrix M whose elements are 0's and 1's such that in any row, all the 1's come before any 0's in that row. Assuming A is already in memory, describe an efficient algorithm for finding the row of M that contains the most of 1's.
What is the running time of the algorithm?

Solution:

Algorithm: FindMaximumRow(int A[][], int row, int column)
Input: A matrix A of size n * n, variables row and column which represents the length of rows and columns in the matrix.We are using 0 based indexing
Output: Returning the value of the row with the maximum number of 1's

```
i ← 0
j ← 0
row ← 0

while(i < row && j < column) {

        if(A[i][j] == 0) then i ← i +1
        else {
                while((j + 1) < column && A[j+ 1] != 0)
                        j++;
                row = i
        }
}

return row;
```

The time complexity of the above algorithm is O(n + n)
which is equivalent to O(n) and the space complexity of the above algorithm is O(1).

7.
(16 Points) You are given two sequences A and B of n numbers each, possibly containing duplicates. Describe an efficient algorithm for determining if A and B contain the same set of numbers, possibly in different orders. What is the running time of this algorithm?

Solution:

We will be using the HashSet data structure to check if there are the same set of numbers in different orders.

Algorithm: checkIsEqual(int[] A, int[]B)
Input: Two sequences A and B
Output: Returning true if A & B contain same set of numbers

```
Set s1
Set s2
for( i ← 0 to A.length - 1)
        s1.add(A[i]);

for( i ← 0 to A.length - 1)
        s2.add(B[i]);

for each element in the set s1
        if( set2 does not contain element) then
                return false;

Return true;
```

Since we are iterating over all the elements of Sequence A & B the time complexity of the above algorithm Is O(n) + O(m) where n is the length of the sequence A and m is the length of the sequence B and The space complexity of the above algorithm is O(n) + O(m).

8.
(16 Points) Let A and B be two sequences of n integers each, in the range [1, n 4].Given an integer x, describe an O(n)–time algorithm for determining if there is an integer a in A and an integer b in B such that x=a+b.

Solution:
We will be the HashSet data structure to solve this problem.

#Define the algorithm.
Algorithm: find_seq(A, B, n, x):
Input: Two sequences A & B. A variable n which denotes the size
Output: Return true if x = any element in A + any element in B

```
  #Defining the hashset in the memory storing the elements of A
 Set A_set
  for( i ← 0 to n - 1)
     A_set.insert(A[i])
  for j in range (0, n):
     if A_set.contains(x - B[i]):
        return true
  #Else not found output will be false
  return false
```

Since we are iterating over all the elements of A & B the time complexity of the above algorithm is O(n + n) which is equivalent to O(n) and the space complexity of the above algorithm is O(n).

9.

Solution:

Assume you have n items with equal weights in a fractional knapsack. Assume their earnings are p1, p2, p3,... pn. Due to the sorting of the profit-weight ratio list, fractional knapsack typically requires O(nlogn) time. If the selection algorithm, commonly known as the median finding procedure, is used, this problem can be solved in linear time O(n).

The following are the steps to solving the knapsack problem in linear time:

1)Dividing the array of profit weight ratio (p/w) into [n/5] elements. While making one more group that will contain the remaining n mod 5 elements of the list. Thus, there will be a total[n/5] +1 groups.

2)Find medians of all groups using insertion sort. It will take linear time to sort the lists of maximum 5 elements. It will result into [n/5] +1 medians.

3)We need to find medians of all medians. Partition the array of elements into three parts:P1, P2, and P3 such that:

Pl = {pi/wi, if pi/wi>m, for 1<=i<=n}

Pl = {pi/wi, if pi/wi=m, for 1<=i<=n}

Pl = {pi/wi, if pi/wi<m, for 1<=i<=n}

So, weight of w1 of first partition will be,

$W1 = \sum i \in P1\ wi$

weight of second partition W2, will be as follows:

$W2 = \sum i \in P2\ wi$

Weight of second partition, W2

$W3 = \sum i \in P3\ wi$

4)P1 now comprises the elements with a higher profit-weight ratio than the elements in P2 and P3. Choosing elements from P1 first will optimize profits.

5)If W1 is more than W, recurse on P1 until one partition has elements with weight sums equal to or less than the total weight.

For example,

If W2=7, W=5, we cannot fill the elements in the knapsack since the list is not sorted.

Thus, call the selection process recursively until the weight of the partition is less than or equal to W, at which point the entire partition can be filled in the knapsack.

6)If W1 < W, place all of the partition P1's elements in the knapsack.

If W2>(W-W1) /After filling the knapsack with partition W /remaining capacity is W-W1,

Recurse on P2 and return the stuff to fill the backpack.

7)If W2(W-W1), place all partition P2 elements in the knapsack. Recurse on P3 and return the goods to fill the knapsack if W3> (W-W1-W2).

8)If W3 < (W-W1-W2) add all elements of the partition P3 into the knapsack return knapsack.

Therefore, the algorithm will run in O(n) time complexity because of the following three reasons:

1) To sort n elements, no sorting method is used. As a result, it will not take O (n logn) time.
2) The selection procedure is called iteratively in the complete process and runs in O(n) time.
3) if-else comparisons are applied, these comparisons take O (1) time.

Therefore, the fractional knapsack problem can be solved in O(n) time.