Question 1

a)Predict the output of the following program. What does the following fun() do in general?

```
#include<stdio.h>
int fun(int a, int b)
{
   if (b == 0)
       return 0;
   if (b % 2 == 0)
       return fun(a+a, b/2);
   return fun(a+a, b/2) + a;
}

int main()
{
  printf("%d", fun(4, 3));
  getchar();
  return 0;
}
```

solution
Output: 12
It calculates a*b (a multiplied by b).

b) In question 1, if we replace + with * and replace return 0 with return 1, then what does the changed function do? Following is the changed function.

```
#include<stdio.h>
int fun(int a, int b)
{
   if (b == 0)
       return 1;
   if (b % 2 == 0)
       return fun(a*a, b/2);
     return fun(a*a, b/2)*a;
}

int main()
{
```

```
  printf("%d", fun(4, 3));
  getchar();
  return 0;
}
```

solution
Output: 64
It calculates a^b (a raised to power b).

Question 2

Explain the functionality of the following function

a)

```
int fun1(int x, int y)
{
  if(x == 0)
    return y;
  else
    return fun1(x - 1,  x + y);
}
```

solution
The function fun() calculates and returns ((1 + 2 … + x-1 + x) +y) which is x(x+1)/2 + y. For example if x is 5 and y is 2, then fun should return 15 + 2 = 17.

b)

```
void fun(int x)
{
  if(x > 0)
  {
     fun(--x);
     printf("%d\t", x);
     fun(--x);
  }
}

int main()
{
  int a = 4;
```

```
  fun(a);
  getchar();
  return 0;
}
```

solution
Output: 0 1 2 0 3 0 1
Refer www.geeksforgeeks.org/practice-questions-for-recursion-set-4/amp/ question 1 for further hints

c)

```
int fun(int n)
 {
   if (n > 100)
     return n - 10;
   return fun(fun(n+11));
 }

int main()
{
  printf(" %d ", fun(99));
  getchar();
  return 0;
}
```

solution
Output: 91
fun(99) = fun(fun(110)) since 99 < 100
= fun(100) since 110 > 100
= fun(fun(111)) since 100 not < 100
= fun(101) since 111 > 100
= 91 since 101 > 100
Returned value of fun() is 91 for all integer arguments < 101.

d)

```
int fun(int count)
{
    printf("%d\n", count);
    if(count < 3)
    {
      fun(fun(fun(++count)));
    }
```

```
    return count;
}

int main()
{
    fun(1);
    return 0;
}
```

solution

Output:

```
1
2
3
3
3
3
3
```

The main() function calls fun(1). fun(1) prints “1” and calls fun(fun(fun(2))). fun(2) prints “2” and calls fun(fun(fun(3))). So the function call sequence becomes fun(fun(fun(fun(fun(3))))). fun(3) prints “3” and returns 3 (note that count is not incremented and no more functions are called as the if condition is not true for count 3). So the function call sequence reduces to fun(fun(fun(fun(3)))). fun(3) again prints “3” and returns 3. So the function call again reduces to fun(fun(fun(3))) which again prints “3” and reduces to fun(fun(3)). This continues and we get “3” printed 5 times on the screen.

Question 3

Consider the following Java method (assume that it will only be called with 0 ≤ k and k ≤ n):

```
int choose (int n, int k) {
        if ((k == 0) || (k == n)) return 1;
        return choose (n-1, k) + choose (n-1, k-1);}
```

Explain why the running time will not be polynomial in n in general. Describe (in code or in English) how to revise the algorithm to be polynomial time in n. Carefully determine and justify a big-O bound on the running time of your improved version.

solution
The recursion will call the method choose on pairs of arguments (i,j) with i ≤ n and j ≤ k, once it is originally called on (n,k). But there will be multiple calls to the same pairs of arguments -- for example, if n is even and the original call is to (n,n/2), then both the subcalls to (n-1,n/2) and (n-1,n/2-1) will result in separate calls to (n-2,n/2-1). Each of these two calls will result in two separate calls to (n-4,n/2-2), making four calls with those arguments. Similarly we get (at least) eight total calls to (n-6,n/2-3), 16 calls to (n-8,n/2-4), and eventually 2n/2 calls to (0,0) just from calls to (2,1).

Memoization would make only one call to each of these O(n2) pairs of arguments, and the processing of each pair of arguments requires only O(1) time if we exclude the time for the recursive calls. Thus the total time for the memoized algorithm (which records the value for each pair of arguments in a table the first time it is computed) is O(n2).

Similarly, we could fill out the table of results without recursion as follows:

```
int dpChoose (int n, int k) {
   int[] table = new int[n+1,k+1];
   for (int i=0; i < n+1; i++) {
      table [i,0] = 1;
      if (i <= k) table [i,i] = 1;}
   for (int j=0; j <= k; j++)
      for (int m=j+1; m <= n; m++)
         table [m,j] = table [m-1,j-1] + table [m-1,j];
   return table [n,k];}
```

This code's running time is dominated by the two nested loops on lines 6 and 7, which take O(n2) time.

Question 4

True or false with justification: Let k be any positive integer constant greater than 1 . Then the recurrence T(n) = kT(n/k) + O(n), with T(1) = O(1), has the same big-O solution for any such k. (You may assume that T(n) is evaluated only when n is a power of k.)

solution
TRUE. We are familiar with the Mergesort recurrence, with k=2, and this results in T(n) = O(n log n). For any integer constant k, we will have:

T(n) = kT(n/k) + O(n)
T(n) ≤ kT(n/k) + cn (for some constant c)
T(n) ≤ k[kT(n/k2) + cn/k] + cn

$T(n) \leq k[k[kT(n/k3) + cn/k2] + cn/k] + cn$
...
$T(n) \leq k \log_k n T(1) + cn[1 + k/k + k2/k2 + ...]$
$T(n) \leq O(n) + \log_k n$ copies of cn
$T(n) = O(n \log n)$
As long as $k > 1$ the recurrence will stop this way, and as long as k is $O(1)$, $\log_k n = \Theta(\log n)$.

Question 5

Consider the following algorithm strangeSort, which sorts n Comparable items in a list A. assumption: the n items are all distinct. Otherwise, the algorithm below fails to terminate if given a list of two or more items that are all equal.:

1) If n ≤ 1, return A unchanged
2) For each item x in A, scan A and count how many other items in A are less than x
3) Put the items with counts less than n/2 in a list B
4) Put the other items in a list C
5) Recursively sort B and C using strangeSort
6) Append the sorted C to the sorted B and return the result

a)Prove by induction on n that strangeSort correctly sorts all lists of length n, with smaller items first.

solution
Clearly if $n \leq 1$ the list is already sorted, and thus line 1 returns the sorted version of the list as desired. Assume now that strangeSort sorts all lists of length n/2, with smaller items first. Consider the operation of strangeSort on a list of size n. In line 2, each item is assigned a number from 0 through n-1, and in line 3 the smallest n/2 items are put in B. The two lists B and C thus each have n/2 items, so by the inductive hypothesis the recursive calls sort B and C correctly. Since every item in B is smaller than every item in C, the append operation in line 6 creates a sorted list of n elements. So the inductive step is complete, assuming that n is a power of 2.

b) Formulate a recurrence for the running time T(n) of strangeSort on an input list of size n. Solve this recurrence to get the best possible big-O bound on T(n) -- you may assume if you like that n is a power of 2.

Line 1 means that $T(1) = O(1)$. Step 2 requires n scans of the entire list of n elements and so takes $O(n2)$ time. Lines 3 and 4 each move n/2 items and so take $O(n)$ time. Line 5 makes two calls to strangeSort with arguments of size n/2 and so takes time $2T(n/2)$. Line 6 also takes $O(n)$ time. The total time is thus $2T(n/2) + O(n2) + O(n) = 2T(n/2) + O(n2)$.

This recurrence was solved to T(n) = O(n2) in the book -- we repeat the derivation here:

T(n) = 2T(n/2) + O(n2)
T(n) ≤ 2T(n/2) + cn2 (for some c)
T(n) ≤ 4T(n/4) + 2c(n/2)2 + cn2
T(n) ≤ 8T(n/8) + 4c(n/4)2 + 2c(n/2)2 + cn2
...
T(n) <= nT(1) + cn2[1/n + 2/n + ... + 1/4 + 1/2 + 1]
T(n) ≤ O(n) + 2cn2
T(n) = O(n2)

Question 6

Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.
Example,
Given:
s1 = "aabcc",
s2 = "dbbca",

When s3 = "aadbbcbcac", return true.
When s3 = "aadbbbaccc", return false.

Solution:

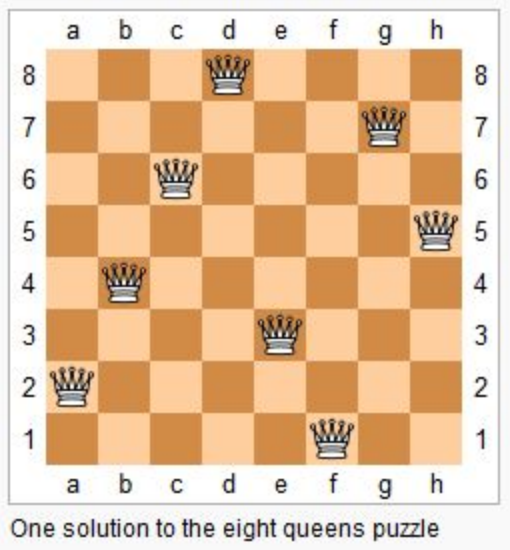
Talk about memoization

```
bool isInterleave(int index1, int index2, int index3) {
        if (index1 == s1.length() && index2 == s2.length()) {
            return index3 == s3.length();
        }
        if (index3 >= s3.length()) return false;
        if (memo[index1][index2][index3] != -1) return memo[index1][index2][index3];
        bool answer = false;
        if (index1 < s1.length() && s1[index1] == s3[index3]) answer |= isInterleave(index1 + 1,
index2, index3 + 1);
        if (index2 < s2.length() && s2[index2] == s3[index3]) answer |= isInterleave(index1,
index2 + 1, index3 + 1);
        return memo[index1][index2][index3] = answer;
    }
```

```
bool isInterleave(string S1, string S2, string S3) {
    s1 = S1;
    s2 = S2;
    s3 = S3;
    memset(memo, -1, sizeof(memo));
    if (S3.length() != S1.length() + S2.length()) return false;
    return isInterleave(0, 0, 0);
}
```

Question 7

The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other.



One solution to the eight queens puzzle

Given an integer n, return all distinct solutions to the n-queens puzzle.
Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.'both indicate a queen and an empty space respectively.
For example,
There exist two distinct solutions to the 4-queens puzzle:

```
[
 [".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
```

```
  ".Q.."]
]
```

Solution:

```
vector<vector<string> > solveNQueens(int n) {
        vector<vector<string> > solutions;
        vector<int> solution(n);
        solveNQueensImpl(0, solution, solutions);
        return solutions;
    }

    void solveNQueensImpl(int row, vector<int> &solution, vector<vector<string> > &solutions) {
        int n = solution.size();
        if (row == n) {
            solutions.push_back(solToStrings(solution));
            return;
        }
        // For each column...
        for (int j = 0; j < n; ++j) {
            // Skip if there is another queen in this column or diagonals
            if (isAvailable(solution, row, j)) {
                solution[row] = j;
                solveNQueensImpl(row + 1, solution, solutions);
            }
        }
    }

    bool isAvailable(const vector<int> &solution, int i, int j) {
        for (int k = 0; k < i; ++k) {
            if (j == solution[k] || i + j == k + solution[k] || i - j == k - solution[k]) return false;
        }
        return true;
    }

    vector<string> solToStrings(const vector<int>& sol) {
        int n = sol.size();
        vector<string> sol_strings(n);
        for (int i = 0; i < n; ++i) {
            sol_strings[i] = string(n, '.');
            sol_strings[i][sol[i]] = 'Q';
        }
```

```
    return sol_strings;
  }
```

Question 8

Given a string s, partition s such that every string of the partition is a palindrome.
Return all possible palindrome partitioning of s.
For example, given s = "aab",
Return

```
 [
   ["a","a","b"]
   ["aa","b"],
  ]
```

Solution:

```
bool isPalindrome(string &str, int s, int e) {
                while (s < e) {
                        if (str[s] != str[e])
                                return false;
                        s++;
                        e--;
                }
                return true;
        }

        void partitionHelper(int i, vector<string> &current, string &s, vector<vector<string> >
&ans) {

                if (i == s.length()) {
                        // we reached the end of the string.
                        // Valid sequence of strings found.
                        ans.push_back(current);
                        return;
                }

                for (int j = i; j < s.length(); ++j) {
                        if (isPalindrome(s, i, j)) {
                                current.push_back(s.substr(i, j - i + 1));
                                partitionHelper(j + 1, current, s, ans);
```

```
                    current.pop_back();
                }
            }
        }

        vector<vector<string> > partition(string s) {
            vector<vector<string> > ans;
            vector<string> current;
            partitionHelper(0, current, s, ans);
            return ans;
        }
```

Question 9:

The set [1,2,3,…,n] contains a total of n! unique permutations.
By listing and labeling all of the permutations in order,
We get the following sequence (ie, for n = 3) :
1. "123"
2. "132"
3. "213"
4. "231"
5. "312"
6. "321"

Given n and k, return the kth permutation sequence.
For example, given n = 3, k = 4, ans = "231"

Solution:

```
int factorial(int n) {
    if (n > 12) {
        // this overflows in int. So, its definitely greater than k
        // which is all we care about. So, we return INT_MAX which
        // is also greater than k.
        return INT_MAX;
    }
    // Can also store these values. But this is just < 12 iteration, so meh!
    int fact = 1;
    for (int i = 2; i <= n; i++) fact *= i;
    return fact;
```

```
}

string getPermutation(int k, vector<int> &candidateSet) {
    int n = candidateSet.size();
    if (n == 0) {
    return "";
    }
    if (k > factorial(n)) return ""; // invalid. Should never reach here.

    int f = factorial(n - 1);
    int pos = k / f;
    k %= f;
    string ch = to_string(candidateSet[pos]);
    // now remove the character ch from candidateSet.
    candidateSet.erase(candidateSet.begin() + pos);
    return ch + getPermutation(k, candidateSet);
}

string getPermutation(int n, int k) {
    vector<int> candidateSet;
    for (int i = 1; i <= n; i++) candidateSet.push_back(i);
    return getPermutation(k - 1, candidateSet);
}
```