(1) A pirate has hidden gold on all of the integer points in the positive orthant of the plane, gi,j ounces of gold at point (i, j). You must start at point (1, 1) and, moving only either up or right one unit, collect as much gold as possible on your way to point (n, n).

Answer:

<u>Recurrence Relation</u>

$$\begin{cases} g_{xy} & x=y=1 \\ & \text{if } 1<=x<=n \\ \text{Max(getmax(x+1,y), getmax(x,y+1)} + g_{xy} & \text{if } 1<=x<=n \end{cases}$$

<u>Unmemoized Code</u>

```
int getmax(int x, int y, int n)
{
if(x==1 && y==1){
          return gxy;
          }
else{
          int right = 0;
          int top = 0;
          if(x<n){
                    right = getmax(x+1,y);
                              }
          if(y<n){
                    top = getmax(x,y+1);
                              }
          if(right > top){
                    return gxy + right;
                                                  }
          else{
                    return gxy + top;
                    }

     }
}
```

<u>Memoized Iterative Code</u>

```
int getmax(int x, int y, int n)
{
if(x==1 && y==1){
        return $g_{xy}$;
        }
else{
        int right = 0;
        int top = 0;
        if(x<n){
                right = getmax(x+1,y);
                        }
        if(y<n){
                top = getmax(x,y+1);
                        }
        if(right > top){
                $g_{xy}$ = $g_{xy}$ + right;                    // Memoized
                return $g_{xy}$;
                                        }
        else{
                $g_{xy}$  = $g_{xy}$ + top;                     // Memoized
                return $g_{xy}$;
                }

    }
}
```

<u>Time Complexity:</u>

Minimum recursive calls needed for a value to be computed is 2. Therefore, time required for computation will be O(n^2)

<u>Sequence of Steps (Path)</u>

```
pathseq(x,y,g,n)

{
if(x<n && y<n)
{

   if(x==1 and y ==1){
          Print (x,y,"*" j)              //* indicates path
```

```
         }

        if(g[x-1][y] > g[x][y-1]){

        Print(x-1,y,"*");                    //* indicates path
        update x;
        }

        if(g[x][y-1] > g[x-1][y]){

        Print(x,y-1,"*");                    //* indicates path
        update y;
        }

        }
}
```

(2) Given a string of letters, you must split it into as few strings as possible such that each string is its own reversal. For example, the string MADAMIMADAM can be split into 11 separate characters, or it can be split into 3 strings MADAM I MADAM, but because it is its own reversal, the minimum number of strings is just 1, the string itself.

Answer:

<u>Recurrence Relation:</u>

$$\begin{cases} \text{String[i] == String[n]} & \text{If } i < j-1 \\ \text{True} & \text{If } i=j \\ \text{False} & \text{If } i != j \end{cases}$$

<u>Umemoized Code:</u>

```
public class Palindrome
{
public void SplitAtPalindrome(string string,int n){
    int stringLength = string.length();
    if(n<stringLength){
        int splitLength = stringLength - n;
        String leftSubString = string.substring(0,splitLength);
        if(checkPalindrome(leftSubString)){
            String rightSubString = string.substring(splitlengt,stringLength);
            if(checkPalindrome(leftSubString)){
                System.out.println(leftSubString+""+rightSubString);
            }else{
                System.out.println(leftSubString+"");
                SplitAtPalindrome(rightSubString,0);
            }
        }else{
            SplitAtPalindrome(string,n+1);
        }
    }
}

public boolean checkPalindrome(String string){
i=0;
j=string.length() - 1;
char[] charString = string.toCharArray();
while(i<j){
    if(charString[i]==charString[j]){
        i++;
```

```
                    j--;
          }else {
                    return false;
          }
 }
          return true;
}
}
```

Memoized Iterative Code

```
public class Palindrome

Private HashMap<String, Boolean> recentStrings = new HasMap<>();            // Memoized
public void SplitAtPalindrome(string string,int n){
          int stringLength = string.length();
          if(n<stringLength){
                    int splitLength = stringLength - n;
                    String leftSubString = string.substring(0,splitLength);
                    if(checkPalindrome(leftSubString)){
                              String rightSubString = string.substring(splitlengt,stringLength);
                              if(checkPalindrome(leftSubString)){
                                        System.out.println(leftSubString+""+rightSubString);
                              }else{
                                        System.out.println(leftSubString+"");
                                        SplitAtPalindrome(rightSubString,0);
                              }
                    }else{
                              SplitAtPalindrome(string,n+1);
                    }
          }
}

public Boolean checkPalindrome(String string){
if(recentStrings.get(string) == null){
          i=0;
          j=string.length() - 1;
          char[] charString = string.toCharArray();
          while(i<j){
                    if(charString[i]==charString[j]){
                              i++;
                              j--;
                    }else {
```

```
                    recentStrings.put(string,false);           // Memoized
                    return false;
                }
 }
        recentStrings.put(string,true);                          // Memoized
        return true;
}
}
```

<u>Time Complexity:</u>

Minimum partitions needed for Palindrome partitioning is 2. Therefore time required for computation would be O (n^2)