Problem 1)

The Fibonacci series is a series of numbers in which there are two fixed numbers 0 and 1(0th Fibonacci number= 0 and 1st Fibonacci number = 1) and the next number in the series is calculated as the sum of the previous two numbers.

Series = 0, 1, 1, 2, 3, 5, 8, 13, ...

The recursive definition of this series is:

$f(n) = 0$ when $n = 0$

$f(n) = 1$ when $n = 1$

$f(n) = f(n-1) + f(n-2)$ when $n >= 2$

```
fibonacci(int n)

  zero = 0, first = 1

  if n==0  return 0
  else if n==1 return 1

  for i=2 to n
     nth = zero + first
     zero = first
     first = nth

  return nth
```

Problem 2)

(a) MaxPrice(n) = max(price[1] + MaxPrice(n-1), price[2] + MaxPrice(n-2), ...,price[n] + MaxPrice[0])
The equation simply comes from the definition of the problem. We will consider cutting from the rod of length n a piece of size i, and we will add the maximum price of a rod of size n-i with it. The value of i at which this sum is maximum is the best cut.

(b) The base case is MaxPrice[0] which is 0, as a rod of length 0 cannot have any price.

(c)

```
MaxPrice(pr,n)

    int best[n]
    best[0] = 0

    for i = 1 to n
        pr_i = -inf
        for j = 1 to i
            pr_i = max(q, pr[j]+best[j-i])
        best[j] = pr_i

    return best[n]
```

The running time for this algorithm comes from the loops in lines 3 to 7. The running time is 1+2+3+...+n-1 = $O(n^2)$.

Problem 3)
(a) If arr[n] = k, CanReach(n) ~ CanReach(n-k) = true. And if arr[m] = i(m <= n-k), CanReach(n-k) ~ CanReach(n-m) = true. And so on.
(b) n = 10. arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
(c) CanReach(arr[], n)

```
jumps = bool[n]
i, j
if n == 0 or arr[0] == 0
    return INT_MAX;
jumps[0] = 0

for i = 1 to n
    jumps[i] = 0
    for j = 0 to i
        if i <= j + arr[j] and jumps[j] != INT_MAX
            jumps[i] = jumps[i] or jumps[j] + 1
            break;

return jumps[n - 1]
```

Problem 4)

(a) I = 8: There are 5 ways: 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 (use 8 pennies), or 1 + 1 + 1 + 5 (3 pennies and 1 nickel), or 1 + 1 + 5 + 1 (2 penny, 1 nickel, and 1 penny), or 1 + 5 + 1 + 1 (1 penny, 1 nickel and 2 pennies), or 5 + 1 + 1 + 1 (1 nickel, and 3 pennies). i = 9: There are six ways: 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 (use 9 pennies), or 1 + 1 + 1 + 1 + 5 (4 pennies and 1 nickel), or 1 + 1 + 1 + 5 + 1 (3 penny, 1 nickel, and 1 penny), or 1 + 1 + 5 + 1 + 1 (2 pennies, 1 nickel and 2 pennies), or 1 + 5 + 1 + 1 + 1 (1 penny, 1 nickel, and 3 pennies), or 5 + 1 + 1 + 1 + 1(1 nickel, and 4 pennies).

(b) Let's say there is a N cents coin and the number of ways to make change for it needs to be calculated.
So, available coins are: 1,5,10. At every N there is a choice to take 1 coin and then count ways for N-1, to take 5 coin and then count ways for N-5 or to take 10 coin and then count ways for N-10.
So, recursive formula becomes:
count(N) = count(N-1) + count(N-2) + count(N-10)
Base case:
if(N==0)
return 1
if(N<0)
return 0

(c) In the above formula it can be seen that there is a lot of overlapping involved.
So, DP can be used for optimisation.
In DP a matrix can be maintained and values for previous N-1,N-5,N-10 can be taken from it.
So, pseudo code is:

```
dp[N+1] = {0}
dp[0]=1;
for i in range(1,N+1):
if(i-1>=0):
dp[i] = dp[i-1];
if(i-5>=0):
dp[i] = dp[i-5];
if(i-10>=0):
dp[i] = dp[i-10];
```

Time comlexity of the above code is O(3*N) because at for loop runs N times and it calculates for all the available coins 1,5 and 10