Problem 1

```
def minimum_gas_price(tank_capacity, distance, gas_location, gas_price):
    current_location = 0
    current_gas = tank_capacity
    total_cost = 0

    gas_location_price = []
    index = gas_location[0]

    for i in range(distance):
        if index < len(gas_location) and i == gas_location[index]:
            gas_location_price.append(gas_price[index])
            index += 1
        else:
            gas_location_price.append(-1)

    while current_location < distance:

        if current_location + current_gas >= distance - 1:
         # break if gas is enough.
            break
        if gas_location_price[current_location] == -1 or current_gas == tank_capacity:
         # move to next location if no gas station here or tank is full
            current_location += 1
            current_gas -= 1
            continue
        min_price = gas_location_price[current_location]
        min_gas_location = current_location

    #Following loop is to find the minimum gas to achieve to
    #another cheaper gas station
        for m in range(1, current_gas):
            if m + current_location < distance
                and gas_location_price[m + current_location] > 0:

                if min_price >= gas_location_price[m + current_location]:
                    min_price = gas_location_price[m + current_location]
                    min_gas_location = m + current_location
        if current_location == min_gas_location:
            total_cost += min_price
         #add only 1 unit gas each time then ckeck again whether can arrive
         #to next target location
            current_gas += 1
        else:
            current_gas = current_gas - (min_gas_location - current_location)
            current_location = min_gas_location

    return total_cost
```

The complexity is O(tank_capacity*distance)

Outer loop complexity is O(distance)

Inner loop complexity is O(tank_capacity)

Problem 2

```
def adjacentist(G)

    # define a dictionary to contain the adjacent list of each vertice
   neighbor = dict()

   for i in range(0, len(G.E))
    # x is the "from" node of each Edge and y is the "to" node of each Edge
    # assign adjacent node to corresponding list
    x = G.E.(i,0)
    y = G.E.(i,1)
        neighbor[x].append(y)


   return neighbor


def best_distance(G, current, t, distance, to_vertice)
    if current == t
        distance[current] = 0
        return (distance, to_vertice)
    else if to_vertice[current] < 0
        to_vertice[current] = 0
    #find the best distance with current location
        for vertx in neighbor[current]
            (distance, to_vertice) = best_distance(G, vertx, t, distance, to_vertice)
            if weights(current, vertx) + distance[vertx] > distance[current]
                distance[current] = weights(current, vertx) + distance[vertx]
                to_vertice[current] = vertx
    else
        return (distance, to_vertice)


    return (distance, to_vertice)


def longest_distance(G(V,E), s, t)

    distance     = []
    to_vertice   = []
    adjacentlist(G)
    for i in range(1,len(G.V))
        distance[i] = -infinity
        to_vertice[i] = -1
     (distance[i], to_vertice[i]) = best_distance(G, s, t, distance[i], to_vertice[i])

        print "The longest path weight is" distance[s]
        print "The longest path is" to_vertice[]
```

The complexity is O(V+E)

Find the neighbor is O(E) since we go through each edge only once.

It is difficult for me to code the adjacent-vertices list as O(E)at very beginning.

My method is really similar to the online resource. Online code considers the "no path condition".

Problem 3

```
def fibonacci(num):
    fib[0] = 0
    fib[1] = 1
    for i in range(2, num):
        fib[i] = fib[i - 1] + fib[i - 2]
    return fib[num]
```

The solution was almost the same as the solution provided online. The only difference is that fib[0] = 0 and fib[1] = 1 in my solution. But the online solution declares fib[0] = fib[1] = 1.

Problem 4

```
Def rod_cutting(p,c) :
        size = len(p)
        m = [0] * (size + 1)
    for i in range(1, size) :
                n = p[i]
         for j in range(1, i) :
                        n = max(n, p[i] + m[i-j]-c )
                        m[i] = n
    return m[size]
```

The solution was almost the same as the solution provided online. The difference is that size = len(p) in my solution instead of direct declaring. The online solution declares n directly. Also my solution uses m, n instead of r and q. For the inner loop, my solution makes j changes from 1 to i, but the online solution makes i change from 1 to j-1. The range is a little different.