Part 1)
Code :

```
def optimumOrder (timeList, weightList) :
    jobOrder =[]
    for i in range(0,len(timeList)) :
        list =[(i+1)]
        list.append(float(weightList[i])/timeList[i])
        jobOrder.append(list.copy())
    size =len(jobOrder)
    for i in range(0,size) :
        max=i
        for j in range(i+1,size) :
            if jobOrder[max][1] < jobOrder[j][1] :
                max =j
        temp =jobOrder[i]
        jobOrder[i] =jobOrder[max]
        jobOrder[max] =temp
    sum =0
    finish =0
    for i in range(0,size) :
        jobList =jobOrder.pop(0)
        job =jobList[0]-1
        time =timeList[job]
        weight =weightList[job]
        finish =finish+time
        sum =sum +finish*weight
        jobOrder.append(job+1)
    result =[jobOrder,sum]
    return result
```

optimumOrder function will take timeList and weightList that contains time and weight information for the corresponding job. Our aim is to minimize the weighted sum of completion time which is $\sum ni = Wi \cdot Ci$. To get the minimum value, we need to prioritize the bigger weighted job so that they don't increase the summation. But if we just look at the weight of the job and sort them in decreasing order, it does not give us the optimum solution because some of them can spend too much time comparing to others. We need to look at the ratio of the weight/time for that job so that bigger weighted job with smaller time will be the first one. We sort them in decreasing order of weight/time. The biggest one will be the first job to do.

Time complexity for optimumOrder function:
Function first creates a list that includes job's index and ratio weight/time for that job. To create this list for n jobs is O(n) times. Then function does selective sort for this list which is O(n^2)
and eventually calculates the weighted sum of completion time which is O(n)
so T(n) = O(n)+O(n^2)+O(n) = O(n^2) times

Part 2)
a) Show that the following algorithm does not correctly solve this problem by giving an instance which it does not return the correct answer.

```
for i= 1 to n
  if N i < S i then
    Output “NY in Month i”
  else
    Output “SF in Month i”
  end
```

This algorithm only checks the location costs, but there is a moving cost when we change the location. We need to add this moving cost to the calculation to get the correct result.
For instance, for moving cost = 10, NY =[10, 20, 10], SF =[20, 15, 20] .
This algorithm will give the result =[NY, SF, NY] and cost of this plan is 55. But correct optimum solution is [NY, NY, NY] and cost of this plan is 40.

b)
Code :

```
def optimalPlan (NY,SF,move) :
    for i in range(1,size+1) :
        if optN[i-1] < (optS[i-1]+move) :
            optN.append(NY[i-1]+optN[i-1])
            optNPath.append("NF")
        else :
            optN.append(NY[i-1]+optS[i-1]+move)
            optNPath =optSPath[0:i-1].copy()
            optNPath.append("NF")
        if optS[i-1] < optN[i-1]+move :
            optS.append(SF[i-1]+optS[i-1])
            optSPath.append("SF")
        else :
            optS.append(SF[i-1]+optN[i-1]+move)
            optSPath =optNPath[0:i-1].copy()
            optSPath.append("SF")
```

optimalPlan function takes New York cost list, San Francisco cost list and moving cost. To find the optimum plan for i. month we need to find minimum cost of (i-1). month, Since We have two option New York and San Francisco, we need to check which one will be the minimum. First we look for the New York, if i. month ends in New York we need to check where (i-1). month will be minimum, it can be New York + i. month New York cost or it can be San Francisco + moving cost + i. month New York cost. Same calculation is done for the San Francisco. We start from the 1. month and save the optimum solution for it and keep looking for the others months.
Optimum plan for i. month that ends in New York is OptN[i] = NY[i] + min(OptN[i-1]), optS[i-1]+move)
For i. month we are looking for (i-1). month optimum plan, it can be NY to NY or SF to NY, if it is NY to NY we add NY cost for i. month but if it is San Francisco we add moving cost additionally.
Optimum plan for i. month that ends in San Francisco is OptS[i] = SF[i] + min(OptS[i-1]), optN[i-1]+move)
Same calculation is done for San Francisco.
At the end of calculation we we look at the minimum cost for i. month. It can be either New York or San Francisco and return the one that gives minimum cost.

Time complexity for this algorithm:
For n. month, we need to calculate from 0 to (n-1). month’s optimum plan. In the loop for this calculation we are only doing if else and adding operations. We can assume that they are done at O(1) time and we have to run the loop n times then this algorithms runs at O(n) times.