Problem 1

Pseudocode:

KindOfIncreasingSubseq(input[ ]**):**

a. Initialize array OPT[length of input] [length of input] to 0
b. maxLen=0
c. for i to length of input:
    i. for j=i+1 to length of input:
        1. OPT[i] [j] = 2
        2. maxInside=-1
        3. for k=0 to i:
            a. if (input[k]+input[i]/2 < input[j]):
                i. maxInside= Max(OPT[k][i] and maxInside)
                ii. OPT[i][j]=1+maxInside
                iii. maxLen= Max(maxLen, 1+maxInside)
d. return maxLen;

What OPT[i][j]: length of the largest subsequence that can be formed using the first j elements, including j such that the average of each pair of elements before that is less than element at j.

Relation to smaller problems: OPT[i][j] = max({1+ max(all OPT[k][i])})

Here, j>i and i>k, and the condition is that the average of kth element and the element at i is less than the value of element at j, and if there is no such element the the OPT[i][j] is simply equal to 2.

Location of the solution to the original problem: Maximum value of all the OPT 2d array is the maximum length of the “kind-of-increasing” subsequence.

Running Time estimate:

- Line **a** is initializing the 2d array of size of the length of the input which is O(n^2)
- Line **b** is constant time, initializing a variable
- Line c contains a for loop from 0 to length, which will be O(n).
    - Inside this we have another for loop at line **(i)** which goes from i+1 to length, which is again O(n)
        - Inside this we have another for loop which goes from 0 to i, which is again O(n)
            - Inside this we have constant time tasks and assigning values to variables.

The complexity for line c after including 3 nested loops will be O(n^3)

So, the total running time for the problem will be given by:

O(n^2)+O(1)+O(n^3) which is just O(n^3)

Problem 2

Pseudocode:

1. **getMaxCost(**w1,w2,weights[ ], costs[ ], size**):**
    a. OPT[size][w1][w2] initialize it to 0
    b. for s1 from 0 to w1:
        i. for s2 from 0 to w2:
            1. for i from 1 to size:
                a. if s1>= weights[i] and s2>= weights[i]:
                    i. OPT[i][s2][s2]= Max(OPT[i-1][s1][s2],
                    ii. OPT[i-1][s1-weights[i]][s2]+ costs[i] ,
                    iii. OPT[i-1][s1][s2-weights[i]]+costs[i])
                b. else if s1>=weights[i]:
                    i. OPT[i][s2][s2]= Max(OPT[i-1][s1][s2],
                    ii. OPT[i-1][s1-weights[i]][s2]+ costs[i] ,
                c. else if s2>=weights[i]:
                    i. OPT[i][s2][s2]= Max(OPT[i-1][s1][s2],
                    ii. OPT[i-1][s1][s2-weights[i]]+costs[i])
                d. else
                    i. OPT[i][s2][s2]= OPT[i-1][s1][s2]
    c. result= OPT[size][w1][w2]
    d. i=size, j=w1, k=w2
    e. bag1[size]
    f. bag2[size]
    g. m=0, q=0
    h. for i from size to 0, conditions: result>0:
        i. if result==OPT[i-1][j][k]:
            1. continue
        ii. else
            1. result= result- costs[i]
            2. if j-weights[i] >=0:
                a. bag1[q]=i
                b. q++ // index for keeping track of items in bag1
                c. j=j-weights[i]
            3. else if k-weights[i] >= 0:
                a. bag2[m]=i
                b. m++ // index for keeping track of items in bag1
                c. k=k-weights[i]
    i. print bag1
    j. print bag2
    k. print OPT[size][w1][w2]

What OPT[i][w1][w2]: The maximum cost of a subset of first i items where the weight is w1 and w2 and items inserted in either w1 or w2 are indivisible.

Relation to smaller problems: There will be 4 conditions for the given problem:

- The item can be included in w1
  - Given that the item can fit in w1, then :
    - OPT[i][w1][w2]=MAX(OPT[i-1][w1][w2], OPT[i-1][w1-wi][w2]+costs at i)
- The item can be included in w2
  - Given that the item can fit in w2, then:
    - OPT[i][w1][w2]=MAX(OPT[i-1][w1][w2], OPT[i-1][w1][w2-wi]+costs at i)
- The item can be included in either of the 2 bags, i.e. w1 and w2
  - Given that the item can go in either bag, we need to check for the maximum profit/cost:
    - OPT[i][w1][w2]=MAX(OPT[i-1][w1][w2], OPT[i-1][w1-wi][w2]+costs at i, OPT[i-1][w1][w2-wi]+costs at i)
- The item is not included in either of them
  - Given that item does not fit any of the bags:
    - OPT[i][w1][w2]=OPT[i-1][w1][w2]

Location of the solution to the original problem: OPT [size][weight 1] [weight 2]

The items in the bag can be traced back using:
OPT[i-1][W1-wi][W2] +Ci>= OPT[i-1][W1][W2] (if in bag 1)or

OPT[i-1][W1][W2-wi] +Ci>= OPT[i-1][W1][W2] (if in bag 2)

Running Time estimate:

- Initializing the 3d OPT [size][w1][w2] which will be O(size*w1*w2)
- For loop 0 to w1 which will be O(w1)
  - For loop from 0 to w2 which will be O(w2)
    - For loop from 0 to size which will ne O(size)
      - Comparison operations, updating values which is contant time of O(1)
- Constant time operations from line c to h
- For loop from size to 0 which is O(size)
  - Costnat time operations inside that

Total runtime for the program will be: O(size*w1*w2) + O(size*w1*w2)+ O(size)

which is O(size*w1*w2).

Problem 3

Pseudocode:

1. **canBeBuilt(c1, c2, c3, n, input):**
    - a. totalLengthTobeBuilt= sum of all the integers in input
    - b. cracks[n-1] list of integer, that will store the crack points
    - c. q=0
    - d. prev=0
    - e. for i from n-1 to 0:
        - i. cracks[q]+=input[i]+prev+1
        - ii. prev+=input[i]
        - iii. q++
    - f. q=0
    - g. OPT[totalLengthToBeBuilt +1][c1+1][c2+1] [c3+1] of type Boolean
    - h. for l from 0 to totalLengthToBeBuilt:
        - i. for i from c1 to 0:
            - 1. for j from c2 to 0:
                - a. for k from c3 to 0:
                    - i. if l==0
                        - 1. OPT[l][i][j][k]= true
                        - 2. Continue
                    - ii. if (l>=1 and i>=1 and OPT[l-1][i-1][j][k]) or l>=2 and j>=1 and OPT[l-2][i][j-1][k]) or l>=3 and k>=1 and OPT[l-3][i][j][k-1]):
                        - 1. OPT[l][i][j][k]=true
                    - iii. If q< cracks.length and cracks[q]=l:
                        - 1. OPT[l][i][j][k]=false
        - ii. If q<cracks.length and l>cracks[q]:
            - 1. q++
    - i. return OPT[totalLengthToBeBuilt][c1][c2][c3]

What OPT[l][i][j][k]: Whether the bricks of length 1,2,3 of quantity can i,j,k respectively can be placed on top of the given bricks of total length l input of one row.

Relation to smaller problems: Following is the relation:

- if l==0, the given OPT[l][i][j][k]= true for all the values
- if any of the following condition are true, the OPT[l][i][j][k] should be:
    - OPT[l-1][i-1][j][k] // when l>= 1 and i >=1
    - OPT[l-2][I][j-1][k] // when l >=2 and j>= 1
    - OPT[l-3][I][j][k-1] // when l >=3 and k>=1
- If l is any of the cracks in the input array, then the OPT[l][i][j][k]=false for all the cases.

Location of the solution to the original problem: The value of the OPT[totalLengthOfAllBricksInRow][c1][c2][c3] will decide whether or not the wall can be built on top a row of bricks.

Running Time estimate:

- Getting the total length of the row, is O(n)
- Filling in the cracks array is O(n)
- Initializing the OPT[totalLength+1][c1+1][c2+1][c3+1] will be O(totalLength*c1*c2*c3)
- For loop from 0 to totalLength which is O(totalLength)
    - For loop from c1 to 0 which is O(c1)
        - For loop from c2 to 0 which is O(c2)
            - For loop from c3 to 0 which is O(c3)
                - Constant time operations which is O(1)
    - Updating the value of q which is again O(1)

Total runtime of the program will be given by O(n)+ O(totalLength*c1*c2*c3)+ O(totalLength*c1*c2*c3) which is same as :
O(totalLength*c1*c2*c3), According to the given problem statement, the maximum length of 1 number will be 100, so the worst case, total length would be 100*n, n is the number of bricks

So the total runtime estimate will be: O(n*c1*c2*c3)

Problem 4

Pseudocode:

1. **getMinimumCost(input[ ]):**
   a. n = length of input
   b. initialize a arrays m[ ] [ ] and s[ ] [ ] of size n
   c. j=0, q=0
   d. for d=1 to n-1:
      i. for i=1 to n-d
         1. j=i+d
         2. min= -infinity
         3. for k=i to j-1:
            a. q=m[i][k]+m[k+1][j]+input[i-1] * input [k] * input[j]
            b. if q < min:
            c. min=q
            d. s[i][j] =k
         4. m[i][j]=min
   e. cost = m[1][n-1]
   f. printOptimat(s, 1, len of s -1)

2. **printOptimal**(s[ ] [ ], i, j):
   a. if i==j:
      i. print("A" + i)
   b. else:
      i. print("(")
      ii. printOptimal(s, i, s[i][j])
      iii. print(" x ")
      iv. printOptimal(s, s[i][j]+1, j)
      v. print(")")

What OPT[i][j]: minimum numbers of multiplications needed to multiply matrix Ai through Aj, where A is the input array of the size of the matrices.

Relation to smaller problems:

OPT[i][j]= {0 when i=j,

Minimum of (OPT[i][k] + OPT[k+1][j] + Ai*Ak*Aj) when i<j }

Location of the solution to the original problem: OPT[1, n] n represents the length of the input.

Running Time estimate:

- For getMinimumCost()
  - Initialzing the 2d array m[][] and s[][] is 2*O(n^2)
  - For loop from 1 to length which is O(n)
    - Another for loop from 1 to len – d which is again O(n)
      - Another loop from i to j which is again O(n)
        - Constant time operations for updating the variables
- For print optimal()
  - We have recursive calls to the function to print the paranthesis and format the solution, and depth of the tree is same as n, and the complexity should be O(n).

Total running time will be given by: 2*O(n^2) + O(n^3)+O(n) which is : O(n^3)