1.

This problem could be solved by dynamic programming:

Initialize an array result with N+1 size.

Array d contains coin value.

Result[0] = 0

For x from 1 to N:

result[x] = max\_int

For c from 1 to k

if (x – d[c] >=0 && result[x-d[c]] + 1 < result[x])

result[x] = result[x-d[c]] + 1

print result[N]

Easy to see the time complexity is O(Nk)

2.

Dynamic programming:

Initialize array check[(m+1)\*(n+1)] with false value, where m is the size of A, n is the size of B.

If m+n different from C size, then the result is false, C is not an interleaving of A&B. Else:

Loop, Iterate check array, i=0-m(array A), j=0-n(array B):

if (i&j = 0) (mean sub-A&B strings are empty) then check[i,j] = true

else if (i=0 & B[j-1]=C[j-1]) then check[i,j]=check[i,j-1]

else if (j=0 & A[i-1]=C[i-1]) then check[i,j]=check[i-1,j]

else if (A match C & B not mach C) then check[i,j]=check[i-1,j]

else if (B match C & A not mach C) then check[i,j]=check[i,j-1]

else if (A&B match C character) then check[i,j] = check[i-1,j] | check[i,j-1]

Continue loop like that, the result is check[m][n]

Basically, create a table and store sub-problem result, the bottom problem is to determine C is interleaving when A/B is empty, then from that basis, compare every character to build a check table.

The final result is in the end of table, at MxN.

The time complexity is O(m\*n)

3.

This problem can be solved by dynamic programming.

Initialize an 3D array to store length of sub-sequence: L[m+1][n+1][p+1]

with m: sequence A length, n: sequence B length, p: sequence C length

If there is an empty string, then no common sub-sequence, else

Start 3 loops, i=0-m, j=0-n, k=0-p:

if (i,j,k = 0) then L[i,j,k] = 0

else if (A[i-1]=B[j-1]=C[k-1]) then L[i,j,k] = 1 + L[i-1,j-1,k-1]

else if (character of any 2 sequences not match) then

L[i,j,k] = max(L[i-1,j,k], L[i,j-1,k], L[i,j,k-1])

L[m,n,p] is the longest common sub-sequence length.

Index = L[min,p]

initialize i=m, j=n, k=p

while (i>0 & j>0 & k>0):

if A[i-1]=B[j-1]=C[k-1] then

i=i-1; j=j-1; k=k-1; index=index-1;

lcs[index]=A[i]

else if (L[i-1,j,k] > L[i,j-1,k] & L[i-1,j,k] > L[i,j,k-1]) then i=i-1

else if (L[i,j-1,k] > L[i-1,j,k] & L[i,j-1,k] > L[i,j,k-1]) then j=j-1

else if (L[i,j,k-1] > L[i,j-1,k] & L[i,j,k-1] > L[i-1,j,k]) then k=k-1

print lcs string

The main ideal is the same to 2 sequences. Find the longest common sub-sequence length of 3 sub-sequences, the continue to reach to the last element of 3 sequences.

The run-time is O(m\*n\*p)

4.

Solve this problem by simple dynamic programming

n=size of array A

TempMax=TrueMax=A[0]

for i=1:n-1

TempMax = max(A[i], TempMax+A[i])

TrueMax = max(TrueMax, TempMax)

print TrueMax

Easy to realize that: every time of loop, there are 2 choices, sum with current element then continue with previous sum or start with new sum. if current element larger than current sum, no need to stay at current sum, then start new sum.

The time complexity is O(n)

5.

initialize array result with size N+1

result[0]=result[1]=result[2]=1;

result[3]=2; result[4]=3;

for i=5:N

result[i] = result[i-1] + result[i-3] + result[i-5]

print result[N]

Consider number n=x1+x2+x3+...+xn. If the xn = 1 then remaining number sum = n-1..

. Similar with xn=3 or 5

6.

result(v, i) is the best value of i-node subtree rooted at v. Every vertex, we decide how many vertices to accept from the subtree at v in the left and right children.

Initialize:  
w is an array of weight

result(v,i) = w(v) + maxloop j from 1→i(result(v to left, j), result(v to right, i-j-1))

result(r, k) takes O(nk) for inner loop, O(k) to compute.

The time complexity is O(nk2)