Task 1

The largest amount of money that can be picked up is \$22 — the \$7, \$10 coin and \$12 coins.

The best solution for n coins is the maximum value out of (a) the best solution for n-1 coins, and (b) the best solution for n-2 coins, plus the value of the nth coin. (Since the nth coin is not adjacent to any coins in the solution for n-2 coins, it is always allowable to add its value to the solution for n-2 coins.) The recurrence relation is:

$F(n) = max\{F(n-1), F(n-2) + V_n\}$

where $F(n)$ is the solution for n coins, and V_n is the value of the n^{th} coin.

For the example coin values given, the solutions for $n = 0 \ldots 5$ would be as follows:

n	0	1	2	3	4	5
$F(n)$	0	7	7	17	19	22

Task 2

The number of ways of getting to a certain point on the grid can be found by finding the number of ways to reach the point immediately below it, and adding the number of ways of reaching the point immediately to the left of it. If one of those points was fenced off, then the number of ways of reaching it is 0, so it contributes nothing to the sum, which is what we need.

The left and bottom edges of the grid are initialised to 1, while the grid locations inside the fenced off regions are initialised to 0 (this overrides the bounday condition, so that a fenced region on an edge will still be initialised to 0).

Algorithm 1 gives one possible solution for solving an n $\times$ m board.

Algorithm 1 gridPaths(G[0..n-1][0..m-1])

```
INPUT: A table G representing the city. We assume that the table is
initially filled with 0s, and the location of the fenced regions is indicated
by -1s in those squares.
OUTPUT: The number of paths from the bottom left to the top right.
ASSUMPTIONS: Start at (n-1,0) and can only move right and up. Note
that the top left corner is (0,0). Also, assuming no fenced regions are on
the bottom or left side. Finally, we assume that there is at least one way
to reach the top right hand corner by only moving up or right.
i ← n-1
while i>-1 do
   j ← 0
   while j<m do
      if i=n-1 or j=0 then
         G[i,j] ← 1                          # if we are on the bottom or left, set to 1
      else
         if G[i,j]=-1 then
            G[i,j]=0                         # if the cell is fenced, mark 0
         else
            G[i,j] ← G[i+1,j]+G[i,j-1]       # otherwise, evaluate the cell
         end if
      end if
      j←j+1
   end while
   i←i-1
end while
return G[0,m-1]                              # the value in the top right is the solution
```

For the example board given, the table G would be filled out as follows:

1	1	4	11	11	17
1	0	3	7	0	6
1	2	3	4	5	6
1	1	1	1	1	1

Each coordinate in this table represents the number of shortest paths from the bottom left to that point.

			Capacity of knapsack																	
Item No	Item Weight	Item Value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	4	0	0	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
2	4	5	0	0	0	4	5	5	5	9	9	9	9	9	9	9	9	9	9	9
3	7	10	0	0	0	4	5	5	5	10	10	10	14	15	15	15	19	19	19	19
4	8	11	0	0	0	4	5	5	5	10	11	11	14	15	16	16	19	21	21	21
5	9	13	0	0	0	4	5	5	5	10	11	13	14	15	17	18	19	21	23	24

Figure 1: Knapsack problem: no duplicates allowed.

Task 3

Figure 1 shows the table generated when solving the given example case. This table is generated by following the dynamic programming algorithm for knapsack, given in lectures.

Task 4

In order to change the edit distance algorithm to use different costs for the different operations, we need to first allow the algorithm to take those costs as inputs. We need to change the boundary conditions so they are multiples of the insertion cost and deletion cost (before they were multiples of 1). Lastly, we need to change the update rule so that it adds the costs for insertion, deletion and substitution, rather than using 1.

Algorithm 2 editDistance(s[0..n-1],t[0..m-1], inscost, delcost, subcost)

```
INPUT: Two strings, s and t, and the cost of insertions, deletions, and substitutions.
OUTPUT: Edit distance between s and t
ASSUMPTIONS: -
distance ← makeTable(0,n+1,m+1)
i ← 1
while (i ≤ n) do
    distance[i,0] ← i×delcost
    i ← i+1
end while
j ← 1
while (j ≤ m) do
    distance[0,j] ← j×inscost
    j ← j+1
end while
i ← 1
while (i ≤ n) do
    j ← 1
    while (j ≤ m) do
        diff ← 0
        if (s[i-1] ≠ t[j-1]) then
            diff ← subcost
        end if
        distance[i,j] ← min(distance[i-1,j]+delcost, distance[i,j-1]+inscost,
        distance[i-1,j-1]+diff)
        j ← j+1
    end while
    i ← i+1
end while
return distance[n,m]
```

Task 5

Part 1.

We can solve this problem recursively by noting that the binary coefficient (n,k) is the kth number in the nth row of pascals triangle (where the top row is considered the zeroeth row). In this way, we compute binCoeff(n,k) by adding binCoeff(n-1,k) and binCoeff(n-1,k-1). The base cases are when k=0 or k=n (the edges of the trianle, where the values are all 1) and when k>n ("outside" Pascal's triangle. Another way of thinking of this base case is that one cannot choose k items from n items when k>n).

Algorithm 3 binCoeffRec(n,k)

```
INPUT: n, k
OUTPUT: nCk
ASSUMPTIONS: n, k are positive integers
if (k > n) then
  return 0
else
  if (k = 0 OR k = n) then
    return 1
  else
    return binCoeffRec(n-1,k-1) + binCoeffRec(n-1,k)
  end if
end if
```

Part 2.

The Dynamic programming version is very similar to the recursive version. The difference is that values are stored, so that the same value does not need to be computed many times.

Algorithm 4 binCoeffDyn(n,k)

```
INPUT: n, k
OUTPUT: nCk
ASSUMPTIONS: n, k are positive integers
BC ← makeTable(0,n+1,k+1)
for (i=0..n) do
   BC[n,0] ← 1          # If k=0, solution is 1; so we can initialise the first row of the table.
end for
for (i=0..n) do
   for (j=1..k) do
      if (k > n) then
         BC[n,k] ← 0
      else
         if (k = n) then
            BC[n,k] ← 1
         else
            BC[n,k] ← BC[n-1,k-1] + BC[n-1,k]
         end if
      end if
   end for
end for
return BC[n,k]
```

Task 6

Algorithm 5 gives the modified algorithm solving the problem for when duplicate items are allowed. The only alteration required is in the initialisation of *valueIncludingI*, the best possible solution including the new item. This value now involves looking at the *current* row in the table, rather than the previous row, since the new item can be included more than once.

			Capacity of knapsack																	
Item No	Item Weight	Item Value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	4	0	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
2	4	5	0	0	0	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
3	7	10	0	0	0	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
4	8	11	0	0	0	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
5	9	13	0	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24

Figure 2: Knapsack problem: duplicates allowed.

Algorithm 5 knapsackWithDuplicates(weights[0..N-1], values[0..N-1], capacity)

```
INPUT: List of item weights, list of item values, knapsack capacity.
OUTPUT: Maximum value of items you can carry.
ASSUMPTIONS: Duplicates allowed — i.e., there is an unlimited supply of each item type.
MaxValue ← makeTable(0, N+1, capacity+1)
i ← 1
while (i ≤ N) do
    j ← 1
    while (j ≤ capacity) do
        MaxValue[i,j] ← MaxValue[i-1, j]
        if weights[i-1]≤j then
            valueIncludingI ← values[i-1] + MaxValue[i, j-weights[i-1]]
            if MaxValue[i-1, j] < valueIncludingI) then
                MaxValue[i,j] ← valueIncludingI
            end if
        end if
        j ← j+1
    end while
    i ← i+1
end while
return MaxValue[N, capacity]
```

Figure 2 shows Algorithm 5 running on the case solved in task 3.