1. Consider Problem 1 from Lecture 5 in Jeff Erickson’s notes.

(a) Since 91 × 4 + 52 = 416, we can make change for 416 using 5 bills. However, the greedy algorithm uses 365, 28, 13, 7, 1, 1, 1 which is 7 bills.

(b) As mentioned in the hint, the optimal change for \( k' \) using denominations in \( D[1..j] \), we either use a bill with denomination \( D[j] \) or we don’t.

This means that the subproblem \( C(k', j) \) can be expressed in terms of two subproblems \( C(k' - D[j], j) \) which denotes the case that we use a bill with denomination \( D[j] \) and \( C(k', j - 1) \) which denotes the case that we don’t use the denomination \( D[j] \).

We just need to be careful in the case when the value of the bill \( D[j] \) is larger than the amount that we need change for \( k' \). In this case we simply consider bills with lesser denomination to construct our solution. The recurrence relation is written below –

\[
C(k', j) = \begin{cases} 
0, & \text{if } k' = 0 \\
k', & \text{if } j = 1 \\
C(k', j - 1), & \text{if } D[j] > k' \\
\min\{1 + C(k' - D[j], j), C(k', j - 1)\}, & \text{otherwise}
\end{cases}
\]

(c) In this part we will use a 2-dimensional \((k + 1) \times 8\) table in which the table-slot \( \text{Table}[k', j] \) is filled with \( C(k', j) \). Note that according to the recurrence relation form part (b), the subproblem \( C(k', j) \) depends on \( C(k', j - 1) \) and \( C(k' - D[j], j) \). Therefore the entry \( \text{Table}[k', j] \) can be calculated if we know what \( \text{Table}[k', j - 1] \) and \( \text{Table}[k' - D[j], j] \) are. This means that we need to fill \( \text{Table} \) from top to bottom and left to right. The following function finds and returns the fewest number of bills needed to make change for \( k \) Dream Dollars, when the denominations come from \( D[1..8] \) –

```java
// Base cases
for j ← 1 to 8 do
    Table[0, j] ← 0
for k' ← 1 to k do
    Table[k', 1] ← k'

// Recursive cases
for k' ← 2 to k do
    for j ← 1 to 8 do
        if D[j] > k' then
            Table[k', j] ← Table[k', j - 1]
        else if 1 + Table[k' - D[j], j] < Table[k', j - 1] then
            Table[k', j] ← 1 + Table[k' - D[j], j]
        else
            Table[k', j] ← Table[k', j - 1]
    return Table[k, 8]
```

(d) If we look at the intuition for the recurrence relation then we realize that if \( \text{Table}[k', j] = \text{Table}[k', j - 1] \) then we did not pick a bill of denomination \( D[j] \) and if \( \text{Table}[k', j] = 1 + \text{Table}[k' - D[j], j] \) then we did pick a bill of denomination \( D[j] \). The following recursive function returns the optimal list of bills –
OptimalChange\( (k, D[1, \ldots, j], \text{Table}) \):

- if \( k = 0 \) then return \([\,]\) // the empty list
- if \( j = 1 \) then return \([1 \times k]\) // a list containing \( k \) 1’s
- if Table\([k, j]\] = Table\([k, j - 1]\) then return OptimalChange\((k, D[1, \ldots, j - 1], \text{Table})\)
- if Table\([k, j]\] = 1 + Table\([k - D[j], j]\) then return \([D[j]] + \text{OptimalChange}(k - D[j], D[1, \ldots, j], \text{Table})\)

We can also implement this function as an iteration through the array Table. Note that both implementations are equivalent in that they will provide the same solution.

\[
k' \leftarrow k \\
j \leftarrow 8 \\
S \leftarrow \text{an empty list} \\
\text{while } k' > 0 \text{ do} \\
\quad \text{if Table}[k', j] = \text{Table}[k', j - 1] \text{ then} \\
\quad \quad j \leftarrow j - 1 \\
\quad \text{if Table}[k', j] = 1 + \text{Table}[k' - D[j], j] \text{ then} \\
\quad \quad k' \leftarrow k' - D[j] \\
\quad S \leftarrow S + [D[j]] \\
\text{return } S
\]

2. You are given a an array \( A[1..n] \) of numbers (which can be positive, 0 or negative). You need to design an algorithm that finds a contiguous subsequence of \( A \) with largest sum. (This is just a restatement of Problem 2(a) in Jeff Erickson’s Lecture 5.) For example, given the array \([-6, 12, -7, 0, 14, -7, 5] \), the contiguous subsequence \([12, -7, 0, 14] \) has the largest sum.

(a) For \( S(1, \cdot) \) we have the additional constraint that the contiguous subsequence must contain \( A[j] \). This means that we can either have just \( A[j] \) as the subsequence or tag along a contiguous subsequence that contains \( A[j - 1] \). This thought process will lead to the following recurrence relation –

\[
S(1, j) = \begin{cases} 
0, & \text{if } j = 0 \\
\max\{S(1, j - 1) + A[j], A[j]\} & \text{if } j > 0
\end{cases}
\]

And similarly, for \( S(2, \cdot) \), we don’t have this additional constraint, so we can also consider the possibility of not having \( A[j] \) as part of the contiguous subsequence. Note that we need \( S(1, \cdot) \) to ensure that the subproblems that we are solving produce contiguous subsequences as the solution. The recurrence relation is –

\[
S(2, j) = \begin{cases} 
0, & \text{if } j = 0 \\
\max\{S(2, j - 1), S(1, j - 1) + A[j], A[j]\} & \text{if } j > 0
\end{cases}
\]

(b) In this part, we will use a 2-dimensional \( 2 \times (n + 1) \) table in which the table-slot Table\([i, j]\) is filled with \( S(i, j) \), where \( i \in \{1, 2\} \) and \( 0 \leq j \leq n \). Note that in order to fill Table\([1, j]\), we need the entry Table\([1, j - 1]\) to be filled and to fill Table\([2, j]\), we need the entries Table\([1, j - 1]\) and Table\([2, j - 1]\). This means that if we fill column \( j - 1 \) before filling column \( j \) then we should be good.
// Base cases
Table[1, 0] ← 0
Table[2, 0] ← 0

// Recursive cases
for j ← 1 to n do
        Table[1, j] ← Table[1, j - 1] + A[j]
    else
        Table[1, j] ← A[j]
    Table[2, j] ← max{Table[2, j - 1], Table[1, j - 1] + A[j], A[j]}
return Table[2, n]

(c) The following function takes as input A and Table (filled out using the function in (b)) and returns the optimal contiguous subsequence from A[1..n] –

OptimalSequence(i, A[1, .., j], Table):
    if j = 0 then
        return [ ] // the empty list
    if i = 2 then
        if Table[2, j] = A[j] then
            return [A[j]]
        if Table[2, j] = Table[2, j - 1] then
            return OptimalSequence(2, A[1, .., j - 1], Table)
        if Table[2, j] = Table[1, j - 1] + A[j] then
            return OptimalSequence(1, A[1, .., j - 1], Table) + [A[j]]
    if i = 1 then
        if Table[1, j] = A[j] then
            return [A[j]]
        if Table[1, j] = Table[1, j - 1] + A[j] then
            return OptimalSequence(1, A[1, .., j - 1], Table) + [A[j]]

To get optimal subsequence, we call OptimalSequence(2, A[1, .., n], Table).