General remarks

- We learn about dynamic programming.

Reading from CLRS for week 8

- Chapter 15: We cover Section 15.2, 15.3, 15.4.
- Section 15.1 is a nice introduction, so it would be good if you could also read this.
- We also treat Section 25.2.

When greedy algorithms fail: making change

Suppose we want to solve the Making Change problem of paying 9 pence with 1, 4 and 6 pence coins.

- The greedy algorithm gives $6+1+1+1$ rather than the optimal $4+4+1$.
- So a new idea is required.
- We need a more systematic way of searching for a solution.
- While we want to avoid (if possible) searching through all possible combinations.

Perhaps we can solve very simple problems, and then proceed recursively:

- If 0 pence is to be returned, we just use zero coins (whatever the coins are).
- If we just have one coin, then the solution is also clear.

Dealing with two coins

Now assume that we have two coins with values $d_1 \neq d_2$, and we have to pay the sum of $N$ pence. Consider an optimal solution $a_1 \cdot d_1 + a_2 \cdot d_2$ (assuming a solution exists at all!), using $a_1 + a_2$ coins.

- Either we use coin $d_2$ or not.
- That is, either $a_2 > 0$ or $a_2 = 0$.
- In the second case, only one coin is left and we are done.
- So assume $a_2 > 0$.
- We know $d_2$ is used at least once.
- Now for the amount $N - d_2$ we know that $a_1 \cdot d_1 + (a_2 - 1) \cdot d_2$ is an optimal solution!
- Since if there would be a solution using fewer coins, then by using one more coin we would obtain a better solution for the original problem.
Making change: the general idea

We arrive at the idea of a general strategy, to solve the problem with coins $d_1, \ldots, d_n$ and amount $N$ to be payed:

- Look how we do if we use only coins $d_1, \ldots, d_{n-1}$.
- Look how we do if we use $d_n$ once, decreasing $N$ to $n - d_n$.
- Compare the two possibilities, and choose the better.

This scheme is to be applied recursively. Note the following:

- In the first case we have a simpler case, since we use fewer coins.
- In the second case we also have a simpler case, since the amount is decreased (though we do not use fewer coins).

So for the recursion basis we must make sure that in “both directions” we end up in a case we can solve directly.

Bookkeeping for making change

To summarise we fill out the table as follows:

- Clearly $c[i, 0] = 0$ for every $i$.
- Also, for every $j$, $c[1, j] = \begin{cases} j \text{ div } d_1 & \text{if } j \mod d_1 = 0, \\ \infty & \text{otherwise.} \end{cases}$

(Whenever we cannot make change for amount $j$ using coins $d_1, \ldots, d_i$, we let $c[i, j] = \infty$.)

- For $c[i, j]$ ($i > 1, j > 0$), we either:
  - pay $j$ pence using only coins $d_1, \ldots, d_{i-1}$: $c[i, j] \leq c[i-1, j]$  
  - or use (at least) one coin $d_i$, and reduce the problem to that of paying $j - d_i$: $c[i, j] \leq 1 + c[i, j - d_i]$.

- As we want to minimise the number of coins, we choose the better of these two options:

  $$c[i, j] = \min\left(c[i-1, j], 1 + c[i, j-d_i]\right).$$

Making change: the general structure

So we can solve the general Making Change problem as follows:

- In order to pay the sum of $N$ pence using $n$ distinct coins $(d_1, d_2, \ldots, d_n)$, we set up an $n \times (N+1)$ table $c[1 \ldots n, 0 \ldots N]$.
- In this table, $c[i, j]$ will hold the minimum number of coins required to pay the amount $j$ using only coins $d_1, \ldots, d_i$.

  *(If no arrangement of such coins makes up $j$ pence, then we shall have $c[i, j] = \infty$.)

- The solution will then be contained in $c[n, N]$.

The table is filled out recursively according to the case-distinction “use last coin or not”.

The making-change algorithm

```plaintext
MAKING-CHANGE(N, d[1..n]) //Running time $O(nN)$
1  for i = 1 to n
2    c[i, 0] = 0
3  for j = 1 to N
4    if (j mod d_i) = 0
5      c[i, j] = j div d_i
6    else c[i, j] = \infty
7  for i = 2 to n
8    for j = 1 to N
9      if j < d_i
10         c[i, j] = c[i-1, j]
11      else c[i, j] = min(c[i-1, j], 1 + c[i, j-d_i])
12  return c
```
The making-change algorithm (continued)

Example: Paying 9 pence using 6, 1 and 4 pence coins (order irrelevant).

<table>
<thead>
<tr>
<th>Amount</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1 = 6</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>d2 = 1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>d3 = 4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Determining an optimal solution

The algorithm Making-Change tells us that $c[3, 9] = 3$ coins are sufficient to make up 9 pence — but which three coins to use?

The following algorithm will answer that, retracing the solution through the $c$-table (precomputed by Making-Change). The output is an array $a_1, \ldots, a_n$ of natural numbers $\geq 0$ such that

$$a_1 \cdot d_1 + \cdots + a_n \cdot d_n = N$$

and such that $a_1 + \cdots + a_n$ is minimal.

```
PAY-OUT(c[1..n, 0..N], d[1..n])
1   for (i = 1; i <= n; ++i) a[i] = 0;
2     i = n; j = N;
3   while j > 0
4      if (i = 1) a[1] = c[1][j]; j = 0
5      else if (c[i, j] = c[i - 1, j]) i = i - 1
6      else a[i] = a[i] + 1; j = j - d[i]
7   return a
```

Complexity of finding an optimal solution

- This algorithm involves stepping back (at most) $n$ rows, and making $c[n, N]$ jumps to the left.
- Hence it runs in time $O(n + N)$.
- Thus it is a negligible addition to the $O(nN)$ algorithm Making-Change.

Of course, we could have computed the array $a_1, \ldots, a_n$ right away directly in Making-Change — we don’t need array $c$ to be completely computed. As an exercise, modify the code of Making-Change to do so.

Why the greedy algorithm fails

As with problems which can be solved by greedy algorithms, Making-Change has the

Optimal Substructure Property: An optimal solution to the problem contains optimal solutions to subproblems.

However, the greedy-choice property fails. We now have to consider many potential solutions, which requires added bookkeeping: we need to remember past decisions, and also build solutions from the bottom up.

We do have something though, namely the

Overlapping Subproblems Property: The space of subproblems is small, so an otherwise-obvious top-down, divide-and-conquer recursive algorithm would solve the same subproblems over and over.
Dynamic programming

The presence of

- **Optimal Substructure Property** and
- **Overlapping Subproblems Property**

characterises **Dynamic Programming**.

With dynamic programming, we take a natural recursive definition, and instead of computing it in a top-down fashion, we compute it bottom-up, exploiting the overlapping subproblems property by only solving each subproblem once.

For example, a top-down algorithm for computing Fibonacci numbers from their definition:

\[ F_0 = 0 \quad F_1 = 1 \quad F_n = F_{n-1} + F_{n-2} \]

would run in exponential time, while a bottom-up algorithm, computing \( F_0 , F_1 , F_2 , F_3 , F_4 , \ldots , F_{n-1} , F_n \) would run in linear time.

### Digraphs

In terms of digraphs we have the following natural formulation of the all-pairs shortest path problem:

Given a digraph \( G \), where every edge \( e \in E(G) \) is labelled by a positive real number \( w_e \in \mathbb{R}_{>0} \), compute the distance matrix, which contains for all pairs of vertices the distance between them.

Note the relation to the previous formulation:

- The vertices of \( G \) are the numbers \( 1, \ldots , n \).
- There is an edge between vertices \( i, j \) iff \( 0 < d_{i,j} < \infty \).
- For an edge \((i, j)\) we have \( w_{(i,j)} = d_{i,j} \).

BFS solves the problem, when all \( w_e = 1 \), and only the **single-source shortest path problem**.

### All-pairs shortest path

**Problem**: Calculating the shortest route between any two cities from a given set of cities \( 1, 2, \ldots , n \).

**Input**: A matrix \( d_{i,j} (1 \leq i, j \leq n) \) of nonnegative values indicating the length of the direct route from \( i \) to \( j \).

Note: \( d_{i,i} = 0 \) for all \( i \); and if there is no direct route from \( i \) to \( j \), then \( d_{i,j} = \infty \).

**Output**: A shortest distance matrix \( s_{i,j} \) indicating the length of the shortest route from \( i \) to \( j \).

We shall give a recursive definition for \( s \), which can be computed by a dynamic-programming algorithm.

### Example

The distance matrix is

\[
\begin{pmatrix}
0 & 1 & 2.5 & \infty & 1.5 & 2 \\
\infty & 0 & 2 & \infty & \infty & \infty \\
\infty & \infty & 0 & \infty & \infty & \infty \\
1 & 2 & 3.5 & 0 & 2.5 & 3 \\
\infty & \infty & 1 & \infty & 0 & 0.5 \\
\infty & \infty & 0.5 & \infty & \infty & 0 \\
\end{pmatrix}
\]
The Floyd-Warshall algorithm

Let \( s^{(k)}_{i,j} \) denote the shortest distance from \( i \) to \( j \) which only passes through cities \( 1, 2, \ldots, k \) (besides \( i,j \)). A recursive definition for \( s^{(k)}_{i,j} \) is given as follows.

\[
s^{(k)}_{i,j} = \begin{cases} 
  d_{i,j} & \text{if } k = 0, \\
  \min \left( s^{(k-1)}_{i,j}, s^{(k-1)}_{i,k} + s^{(k-1)}_{k,j} \right) & \text{if } k > 0.
\end{cases}
\]

**Floyd-Warshall (d, n)**

1. \( s^{(0)} = d \)
2. for \( k = 1 \) to \( n \)
   3. for \( i = 1 \) to \( n \)
      4. for \( j = 1 \) to \( n \)
         5. \( s^{(k)}_{i,j} = \min \left( s^{(k-1)}_{i,j}, s^{(k-1)}_{i,k} + s^{(k-1)}_{k,j} \right) \)

This algorithm runs in \( O(n^3) \) time and space. However, we can safely remove the superscripts from \( s \) (can you see why?), and achieve \( O(n^2) \) space.

**Example**

\[
\begin{array}{c|c|c|c|c}
 \pi^{(0)} & s^{(0)} & \pi^{(1)} & s^{(1)} \\
\hline
 1 & 0/\text{NIL} & 8/1 & 4/1 & \infty/\text{NIL} \\
 2 & 8/2 & 0/\text{NIL} & 3/2 & 4/2 \\
 3 & 4/3 & 3/3 & 0/\text{NIL} & \infty/\text{NIL} \\
 4 & \infty/\text{NIL} & 4/4 & \infty/\text{NIL} & 0/\text{NIL} \\
\end{array}
\]

**Constructing shortest paths**

To construct the shortest paths, we maintain a predecessor matrix \( \pi_{i,j} \) in which \( \pi_{i,j} \) denotes the predecessor of \( j \) on some shortest path from \( i \) to \( j \). (If \( i = j \) or there is no such path, then \( \pi_{i,j} = \text{NIL} \).) The final algorithm for computing \( s \) and \( \pi \):

**Floyd-Warshall (d, n)**

1. \( s = d \)
2. for \( i = 1 \) to \( n \)
   3. for \( j = 1 \) to \( n \)
      4. if \( i=j \) or \( d_{i,j} = \infty \)
         5. \( \pi_{i,j} = \text{NIL} \)
      6. else \( \pi_{i,j} = i \)
     7. for \( k = 1 \) to \( n \)
        8. for \( i = 1 \) to \( n \)
           9. for \( j = 1 \) to \( n \)
              10. \( x = s_{i,k} + s_{k,j} \)
                 11. if \( x < s_{i,j} \)
                    12. \( s_{i,j} = x; \pi_{i,j} = \pi_{k,j} \)