Fall 2016 – Dynamic programming

October 11, 2016
Problem – Making Change

- Task – buy a cup of coffee (say it costs 63 cents).
- You are given an unlimited number of coins of all types (neglect 50 cents and 1 dollar).
- Pay exact change.
- What is the combination of coins you’d use?

1 cent 5 cents 10 cents 25 cents
Logically, we want to minimize the number of coins.

The problem is then: Count change using the fewest number of coins – we have 1, 5, 10, 25 unit coins to work with.

The "greedy" part lies in the order: We want to use as many large-value coins to minimize the total number.

When counting 63 cents, use as many 25s as fit, $63 = 2(25) + 13$, then as many 10s as fit in the remainder: $63 = 2(25) + 1(10) + 3$, no 5’s fit, so we have $63 = 2(25) + 1(10) + 3(1)$, 6 coins.
Greedy Algorithms

- A greedy person grabs everything they can as soon as possible.
- Similarly a greedy algorithm makes locally optimized decisions that appear to be the best thing to do at each step.
- Example: Change-making greedy algorithm for “change” amount, given many coins of each size:
  - Loop until change == 0:
  - Find largest-valued coin less than change, use it.
  - change = change - coin-value;
The greedy method gives the optimal solution for US coinage.

With different coinage, the greedy algorithm doesn’t always find the optimal solution.

Example of a coinage with an additional 21 cent piece. Then $63 = 3(21)$, but greedy says use 2 25s, 1 10, and 3 1’s, a total of 6 coins.

The coin values need to be spread out enough to make greedy work.

But even some spread-out cases don’t work. Consider having pennies, dimes and quarters, but no nickels.

Then 30 by greedy uses 1 quarter and 5 pennies, ignoring the best solution of 3 dimes.
Greedy algorithms are very popular. They do not always guarantee the optimal solution but they are often simple and can be used as approximation algorithms when the exact solution is too hard. Sometimes they give the optimal solution, as with the US coins above. We will visit greedy algorithms later in the course. For now we need a method that guarantees optimality for any coin combination.
Example: change for 63 cents with coins = \{25, 10, 5, 1, 21\} no order required in array.

makeChange(63)
minCoins = 63
loop over j from 1 to 63/2 = 31
    thisCoins = makeChange(j) + makeChange(63-j)
    if thisCoins < minCoins
        minCoins = thisCoins
return minCoins

Lots and lots of redundant calls!
(Very bad) Recursive Solution

\[ T(n) = T(n-1) + T(n-2) + T(n-3) + \ldots + T(n/2) + \ldots \]

worse than \( T(n) = T(n-1) + T(n-2) \) the famous Fibonacci sequence discussed on pg. 242 (and hw1). Fibonacci is exponential, so this certainly is.
- We know we have 1, 5, 10, 21 and 25.
- Therefore, the optimal solution must be the minimum of the following:
  - 1 (A 1 cent) + optimal solution for 62.
  - 1 (A 5 cent) + optimal solution for 58.
  - 1 (A 10 cent) + optimal solution for 53.
  - 1 (A 21 cent) + optimal solution for 42.
  - 1 (A 25 cent) + optimal solution for 38.
  - This reduces the number of recursive calls drastically.
- Naive implementation still makes lots of redundant calls.
Dynamic Programming Implementation

- Idea – hold on to the fact that you only have to look at five previous solutions
- But instead of performing the same calculation over and over again, save pre-calculated results to an array.
- The answer to a large change depends only on results of smaller calculations, so we can calculate the optimal answer for all the smaller change and save it to an array.
- Then go over the array and minimize on:
  - $change(N) = \min_{k \in K} \{ change(N - k) + 1 \}$
  - For all $K$ types of coins, in our example $K = \{1, 5, 10, 21, 25\}$
- Runtime – $O(N \times K)$. 
public static void makeChange(int[] coins, int differentCoins, int maxChange, int[] coinsUsed, int[] lastCoin) {
    coinsUsed[0] = 0; lastCoin[0] = 1;
    for (int cents = 1; cents <= maxChange; cents++) {
        int minCoins = cents;
        int newCoin = 1;
        for (int j = 0; j < differentCoins; j++) {
            if (coins[j] > cents) // Cannot use coin j
                continue;
            if (coinsUsed[cents - coins[j]] + 1 < minCoins) {
                minCoins = coinsUsed[cents - coins[j]] + 1;
                newCoin = coins[j];
            }
        }
        coinsUsed[cents] = minCoins;
        lastCoin[cents] = newCoin;
    }
}
We define two arrays:

- One for the minimum number of coins for each of 1, 2, ..., n.
- The other is for the actual last coin that brought us there.
- This way we can backtrack and find the actual coin combination, not just the number of coins.
Dynamic Programming

- An algorithm design technique for optimization problems: often minimizing or maximizing.
- Notice that we are "reversing" our way of thinking about the problem.
- Instead of going top down – starting from the goal change and calculating smaller solutions on our way, we go bottom up – calculate small solutions independently of the big problem.
- This is a rather subtle but important distinction.
Dynamic Programming

- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not independent and may share subsubproblems,
- However, solution to one subproblem may not affect the solutions to other subproblems of the same problem. (More on this later.)
- DP reduces computation by solving subproblems in a bottom-up fashion.
- Storing solution to a subproblem the first time it is solved.
- Looking up the solution when subproblem is encountered again.
- **Key:** determine structure of optimal solutions
Dynamic Programming – Two Conditions

- One is the *optimal substructure property*: a solution contains within it the optimal solutions to subproblems – in this case, the minimum number of coins for smaller change.

- The second is the *overlapping subproblems*: There are only $O(n)$ distinct solutions, but they may appear multiple times on our way to solving the original problem.

- We only have to compute each subproblem once, and save the result so we can use it again.

- This is called *memoization*, which refers to the process of saving (i.e., making a “memo”) of an intermediate result so that it can be used again without recomputing it.

- Of course the words “memoize” and “memorize” are related etymologically, but they are different words, and you should not mix them up.
Longest Common Subsequence (LCS)

**Definition**

A *subsequence* of a sequence \( A = \{a_1, a_2, \ldots, a_n\} \) is a sequence \( B = \{b_1, b_2, \ldots, b_m\} \) (with \( m \leq n \)) such that

- Each \( b_i \) is an element of \( A \).
- If \( b_i \) occurs before \( b_j \) in \( B \) (i.e., if \( i < j \)) then it also occurs before \( b_j \) in \( A \).

We do *not* assume that the elements of \( B \) are consecutive elements of \( A \).

For example: “axdy” is a subsequence of “baxefdoym”

The “longest common subsequence” problem is simply this:

*Given two sequences* \( X = \{x_1, x_2, \ldots, x_m\} \) *and* \( Y = \{y_1, y_2, \ldots, y_n\} \) (note that the sequences may have different lengths), *find a subsequence common to both whose length is longest.*
This is part of a class of what are called *alignment problems*, which are extremely important in biology.

It can help us to compare genome sequences to deduce quite accurately how closely related different organisms are, and to infer the real “tree of life”.

Trees showing the evolutionary development of classes of organisms are called “phylogenetic trees”.

A lot of this kind of comparison amounts to finding common subsequences.
Try the obvious approach: list all the subsequences of $X$ and check each to see if it is a subsequence of $Y$, and pick the longest one that is.

There are $2^m$ subsequences of $X$. To check to see if a subsequence of $X$ is also a subsequence of $Y$ will take time $O(n)$. (Is this obvious?)

Picking the longest one an $O(1)$ job, since we can keep track as we proceed of the longest subsequence that we have found so far.

So the cost of this method is $O(n2^m)$.

That’s pretty awful, since the strings that we are concerned with in biology have hundreds or thousands of elements at least.
We have two strings, with possibly different lengths:
\[ X = \{x_1, x_2, \ldots, x_m\} \text{ and } Y = \{y_1, y_2, \ldots, y_n\} \]

A prefix of a string is an initial segment. So we define for each \( i \) less than or equal to the length of the string the prefix of length \( i \):
\[ X = \{x_1, x_2, \ldots, x_i\} \text{ and } Y = \{y_1, y_2, \ldots, y_i\} \]

A solution of our problem reflects itself in solutions of prefixes of \( X \) and \( Y \).

**Theorem**

Let \( Z = \{z_1, z_2, \ldots, z_k\} \) be any LCS of \( X \) and \( Y \).

1. If \( x_m = y_n \), then \( z_k = x_m = y_n \), and \( Z_{k-1} \) is an LCS of \( X_{m-1} \) and \( Y_{n-1} \).
2. If \( x_m \neq y_n \), then \( z_k \neq x_m \Rightarrow Z \) is an LCS of \( X_{m-1} \) and \( Y \).
3. If \( x_m \neq y_n \), then \( z_k \neq y_n \Rightarrow Z \) is an LCS of \( X \) and \( Y_{n-1} \).
**Corollary**

If \( x_m \neq y_n \), then either

- \( Z \) is an LCS of \( X_{m-1} \) and \( Y \), or
- \( Z \) is an LCS of \( X \) and \( Y_{n-1} \).

Thus, the LCS problem has the **optimal substructure property**: in this case, to subproblems constructed from prefixes of the original data.
Let $c[i, j]$ be the length of the LCS of $X_i$ and $Y_j$. Based on The optimal substructure theorem, we can write the following recurrence:

$$
c[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
\max\{c[i - 1, j], c[i, j - 1]\} + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\
\max\{c[i - 1, j], c[i, j - 1]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j 
\end{cases}
$$

The optimal substructure property allows us to write down an elegant recursive algorithm.

However, the cost is still far too great – we can see that there are $\Omega(2^{\min\{m, n\}})$ nodes in the tree, which is still a killer.
Recursive Algorithm
There are only $O(mn)$ distinct nodes, but many nodes appear multiple times.

We only have to compute each subproblem once, and save the result so we can use it again (memoization).
Algorithm 1 LCSLength(X,Y,m,n)

1: for i ← 1...m do
2:     c[i,0] ← 0
3: end for
4: for j ← 0...n do
5:     c[0,j] ← 0
6: end for
7: for i ← 1...m do
8:     for j ← 1...n do
9:         if x_i == y_j then
10:             c[i,j] ← c[i-1,j-1] + 1; b[i,j] ← “↖”
11:         else
12:             if c[i-1,j] ≥ c[i,j-1] then
13:                 c[i,j] ← c[i-1,j]; b[i,j] ← “↑”
14:             else
15:                 c[i,j] ← c[i,j-1]; b[i,j] ← “←”
16:         end if
17:     end if
18: end for
19: end for
20: return c and b
### LCS Table – Example

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><em>j</em></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td><em>i</em></td>
<td><em>y</em></td>
<td>B</td>
<td>D</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td><em>x</em></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

The diagram illustrates the Longest Common Subsequence (LCS) table for two sequences *x* and *y*. The table is filled with the lengths of common subsequences at each position, with arrows indicating the direction of the maximum length found.
Constructing the Actual LCS

Just backtrack from $c[m, n]$ following the arrows:

---

**Algorithm 2** PrintLCS(b, X, i, j)

1: if $i = 0$ or $j = 0$ then
2:     return
3: end if
4: if $b[i, j] == \sim\text{↖}$ then
5:     PrintLCS(b, X, $i - 1$, $j - 1$)
6:     PRINT $x_i$
7: else
8:     if $b[i, j] == \sim\text{↑}$ then
9:         PrintLCS(b, X, $i - 1$, $j$)
10:     else
11:         PrintLCS(b, X, $i$, $j - 1$)
12:     end if
13: end if
It is important to understand the two properties of this problem that made it possible for use of dynamic programming:

- **Optimal substructure:** subproblems are just “smaller versions” of the main problem.

- **Finding the LCS of two substrings** could be reduced to the problem of finding the LCS of shorter substrings.

- **This property** enables us to write a recursive algorithm to solve the problem, but this recursion is much too expensive – typically, it has an exponential cost.

- **Overlapping subproblems:** This is what saves us: The same subproblem is encountered many times, so we can just solve each subproblem once and “memoize” the result.

- **In the current problem,** that memoization cut down the cost from exponential to quadratic, a dramatic improvement.
Another famous example is the sequence of binomial coefficients.

Coefficients of the powers of the series generated by

\[(x + y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} \cdot\]

See for example the binomial expansion of

\[(x + y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4\]

They can also be generated by Pascal’s triangle:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Nurit Haspel
CS310 - Advanced Data Structures and Algorithms
Binomial Coefficients

- Start a new row with 1’s on the edges.
- Each number is the sum of the two closest above it.
- The row number is $N$, and the entries are $k=0$, $k=1$, $\ldots$, $k=N$ across a row, so for example

\[
\begin{align*}
C(4,0) &= 1, & C(4,1) &= 4, & C(4,2) &= 6, & C(4,3) &= 4, & C(4,4) &= 1.
\end{align*}
\]
Also, \( C(N, k) = \) number of ways to choose a set of \( k \) objects from \( N \). (read \( N \) choose \( k \))

Ex. \( C(4, 2) = 6 \) The 2-sets of 4 numbers are the 6 sets: 
\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}

Base cases: \( C(N, 0) = 1, C(N, N) = 1 \)

To choose \( k \) objects from \( N \), set one object \( x \) aside and find all the ways of choosing \( k \) objects from the remaining \( N-1 \).

These are all the sets we want that don’t include \( x \), \( C(N-1, k) \) in number.

The sets that do include \( x \) also need \( k-1 \) other objects from the other \( N-1 \), \( C(N-1, k-1) \) in number.
Recursion: $C(N, k) = C(N-1, k) + C(N-1, k-1)$ This is just the sum rule of Pascal’s triangle. If we write a recursive function:

```python
combo(N, k):
    if (k == 0) return 1
    if (k == N) return 1
    return combo(N-1, k) + combo(N-1, k-1)
```

Note the double recursion, without halving the $N$ value, so dangerous recursion.
We get exponential $T(N)$

$T(N, k) = T(N-1, k) + T(N-1, k-1) - 2$ terms in $N-1$
$= T(N-2, k) + ...$ $4$ terms in $N-2$
$= ...$ some of these hit base cases and stop
If we save and reuse values, it’s much faster.

In other words, use Pascal’s triangle to generate all the coefficients.

One way: set up a table and use it for each N in turn.

\[
\begin{align*}
C[1][0] &= 1 \\
C[1][1] &= 1 \\
\text{for } n \text{ up to } N \\
\quad &\text{for } k \text{ up to } n \\
\quad &\quad C[n][k] = C[n-1][k] + C[n-1][k-1]
\end{align*}
\]

\(O(1)\) to fill each spot in \(N\times N\) array, so \(O(N^2)\)
Another approach: set up Map from \((N, k)\) to value.

if \(N\) and \(k\) both ints, long key = \(N + (\text{long})k \gg 32\)

Case of classic dynamic programming, saving partial results along the way.
Map Approach to Binomial Coefficients

combo(N, k):
    val = M.get(key(N,k))
    if (val != null) return val
    if (k == 0) val = 1
    if (k == N) val = 1
    else val = combo(N-1, k) + combo(N-1, k-1)
    M.put(key(N, k), val)
    return val

once this recursion reaches a cell, fills it in, so work bounded by number of cells below (N, k), which is $< N^2$. 
Given a sequence of integers \((A_1, A_2, ..., A_N)\), possibly negative.

Identify the subsequence \((A_i, ..., A_j)\) that corresponds to the maximum value of \(\sum_{i}^{j} A_k\)

Naive approach is cubic (examine all \(O(N^2)\) sequences and sum each one).

Use a divide-and-conquer algorithm.
Sample input is \{4, -3, 5, -2, -1, 2, 6, -2\}

3 possible cases:
1. in the first half
2. in the second half
3. begins in the first half and ends in the second half

For case 3: \(\text{sum} = \text{sum }1^{st} + \text{sum }2^{nd}\)

<table>
<thead>
<tr>
<th>First half</th>
<th>Second half</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>4  5  -2</td>
<td>-1  2  6</td>
<td></td>
</tr>
<tr>
<td>4*  3  -2</td>
<td>-1  1  7*</td>
<td>Running sums</td>
</tr>
</tbody>
</table>

Running sum from the center
(* denotes maximum for each half)
Case 3 is solved in linear time.

Apply case 3’s strategy to solve case 1 and 2

Summary:

- Recursively compute the max subsequence sum in the first half
- Recursively compute the max subsequence sum in the second half
- Compute, via 2 consecutive loops, the max subsequence sum that begins in the first half but ends in the second half
- Choose the largest of the 3 sums
private static int maxSumRec( int[] a, int left, int right ) {
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = (left + right) / 2;
    if(left == right) return a[left] > 0 ? a[left] : 0; // base
    int maxLeftSum = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );
    for( int i = center; i >= left; i-- ) {
        leftBorderSum += a[i];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }
    for( int i = center + 1; i <= right; i++ ) {
        rightBorderSum += a[i];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }
    return max3(maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSum);
}
Let’s look at index j.

The maximum contiguous subsequence ending at j (denoted \text{MaxSum}(j)) either extends a previous maximum subsequence (ending at j-1) or starts a new sum.

The former happens if \text{MaxSum}(j-1) is positive.

The latter happens if \text{MaxSum}(j-1) is non-positive.
Therefore a dynamic programming solution for $Max(j)$ is:

$$Max(j) = \max\{Max(j - 1) + a[j], a[j]\}$$

(constant for each $j$, considering that $Max(j-1)$ was already computed).

The overall solution to the problem is $\max_j Max(j)$.

- $T(1) = O(1)$ – the maximum sum for $a[0]$ is $\max\{a[0], 0\}$.
- $T(N) = T(N-1) + O(1)$ – maximizing over two $O(1)$ expressions.
- $T(N) = O(N)$. 