Dynamic programming

April 9, 2019
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.
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, but sometimes they don’t.
We need a method that guarantees optimality for any coin combination.
First, let’s think of a recursive way to solve the problem (don’t worry about runtime at the moment).
(Very bad) Recursive Solution

Example: change for 63 cents with coins = \{25, 10, 5, 1, 21\} no order required in array.

\begin{verbatim}
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
\end{verbatim}

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 earlier on in the course. 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 a 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.
LCS – Naive approach

- 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.
Recursive Algorithm

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:

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

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
Overlapping Subproblems

- 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 $i \leftarrow 1 \ldots m$ do
2: \hspace{1em} $c[i, 0] \leftarrow 0$
3: end for
4: for $j \leftarrow 0 \ldots n$ do
5: \hspace{1em} $c[0, j] \leftarrow 0$
6: end for
7: for $i \leftarrow 1 \ldots m$ do
8: \hspace{1em} for $j \leftarrow 1 \ldots n$ do
9: \hspace{2em} if $x_i == y_j$ then
10: \hspace{3em} $c[i, j] \leftarrow c[i - 1, j - 1] + 1; b[i, j] \leftarrow \text{"\\"}\\\text{"}"
11: \hspace{2em} else
12: \hspace{3em} if $c[i - 1, j] \geq c[i, j - 1]$ then
13: \hspace{4em} $c[i, j] \leftarrow c[i - 1, j]; b[i, j] \leftarrow \text{"\uparrow"}$
14: \hspace{3em} else
15: \hspace{4em} $c[i, j] \leftarrow c[i, j - 1]; b[i, j] \leftarrow \text{"\<"}$
16: \hspace{2em} end if
17: \hspace{em} end if
18: end for
19: end for
20: return $c$ and $b$
## LCS Table – Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x&lt;sub&gt;i&lt;/sub&gt;</td>
<td>0</td>
<td>0</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>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Explanation

The table above is an example of the Longest Common Subsequence (LCS) problem, where we attempt to find the longest subsequence common to both input sequences, x and y. Each cell in the table represents the length of the longest common subsequence up to that point. The arrows at the edges of the matrix indicate the direction of the optimal path to find the LCS:

- **Diagonal Up (↑)**: The characters at the same position in both sequences are part of the LCS.
- **Left (←)**: The LCS is extended by the character from the first sequence.
- **Up (↑)**: The LCS is extended by the character from the second sequence.

The final length of the LCS is indicated by the number in the last cell of the table.
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 $b[i, j] == \text{↖}$ then
5. PrintLCS$(b, X, i - 1, j - 1)$
6. PRINT $x_i$
7. else
8. if $b[i, j] == \text{↑}$ then
9. PrintLCS$(b, X, i - 1, j)$
10. else
11. PrintLCS$(b, X, i, j - 1)$
12. end if
13. end if
What Makes Dynamic Programming Work?

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.
Remember the interval scheduling problem?

The weighted problem goes as follows:

Job \( j \) starts at \( s_j \), finishes at \( f_j \), and has weight \( w_j > 0 \).

Two jobs are compatible if they don’t overlap.

Goal: find **max-weight** subset of mutually compatible jobs.
Recall earliest finish-time first:
- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.
- Greedy algorithm is correct if all weights are 1.
- However, it fails spectacularly for weighted version.
- Say, weight of b is 999 and weights of a and h is 1 each.
Convention: Jobs are in ascending order of finish time: 
\[ f_1 \leq f_2 \leq \cdots \leq f_n. \]
- \( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).
- Example: \( p(8) = 1, p(7) = 3, p(2) = 0. \)
- \( i \) is leftmost interval that ends before \( j \) begins. By convention, \( p(j) \) and all the jobs before it are compatible with \( j \).
Dynamic programming: Binary Choice

- Def. $OPT(j) = \max \text{ weight of any subset of mutually compatible jobs for subproblem consisting only of jobs } 1, 2, \ldots, j$.
- Goal: $OPT(n) = \max \text{ weight of any subset of mutually compatible jobs}$.
- For each job $j$ we have two options:
  - Case 1: $OPT(j)$ does not select job $j$.
    - Then there is an optimal solution consisting of remaining jobs $1, 2, \ldots, j - 1$.
  - Case 2. $OPT(j)$ selects job $j$.
    - Then we collect profit $w_j$, and the remaining jobs must include only remaining compatible jobs $1, 2, \ldots, p(j)$. 

Nurit Haspel
CS310 - Advanced Data Structures and Algorithms
Recursive Formula

\[ OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max(OPT(j - 1), w_j + OPT(p(j))) & \text{otherwise}
\end{cases} \]

Algorithm 3  Brute-force\((n, s_1 \ldots s_n, f_1 \ldots f_n, w_1 \ldots w_n)\)

Sort jobs by finish time
Compute \(p[1], p[2], \ldots, p[n]\) via binary search
\textbf{return}  Compute-Opt(n)

Algorithm 4  (Compute-Opt(j))

\begin{verbatim}
if (j = 0) then
    return  0
else
    return  \max Compute - Opt(j - 1), w_j + Compute - Opt(p[j])
end if
\end{verbatim}
Recursive algorithm is spectacularly slow because of overlapping subproblems, which makes it an exponential-time algorithm.

In the example the number of recursive calls for a family of ”layered” instances grows like Fibonacci sequence (for every $j > 2$, $p(j) = j-2$).
Dynamic Programming Solution

- Cache result of subproblem $j$ in $M[j]$.
- Use $M[j]$ to avoid solving subproblem $j$ more than once.

**Algorithm 5** TOP-DOWN$(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \cdots \leq f_n$

Compute $p[1], p[2], \ldots, p[n]$ via binary search

$M[0] \leftarrow 0$

return M-COMPUTE-OPT$(n)$

**Algorithm 6** M-COMPUTE-OPT$(j)$

if (M[$j$] is uninitialized) then
  
  $M[j] \leftarrow \max(M\text{-COMPUTE\text{-}OPT}(j-1), w_j + M\text{-COMPUTE\text{-}OPT}(p[j]))$

end if

return M[$j$]
Dynamic Programming Solution Analysis

- Memoized version of algorithm takes $O(n \log n)$ time.
- Sort by finish time: $O(n \log n)$.
- Compute $p[j]$ for each $j$: $O(n \log n)$ via binary search.
- M-COMPUTE-OPT(j): each invocation takes $O(1)$ time and either returns an initialized value $M[j]$ or initializes $M[j]$ and makes two "recursive" calls.
- By the time we get to $j$ both $j-1$ and $p[j]$ have already been initialized, so these calls just return an existing value.
- At most two "recursive" calls per $j$.
- Overall running time of M-COMPUTE-OPT(n) is $O(n)$. 
Q: DP algorithm computes optimal value. How to find optimal solution?
A: Make a second pass by calling FIND-SOLUTION(n).

Algorithm 7 FIND-SOLUTION(n)

if (j = 0) then
    return Φ
else
    if (wj + M[p[j]] > M[j − 1]) then
        return {j}∪FIND-SOLUTION(p[j])
    else
        return FIND-SOLUTION(j-1)
    end if
end if

Number of recursive calls ≤ n, so runtime is O(n)
Bottom-up dynamic programming. Unwind recursion.

The bottom-up version takes $O(n \log n)$ time.

---

**Algorithm 8** BOTTOM-UP($n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \cdots \leq f_n$

Compute $p[1], p[2], \ldots, p[n]$ via binary search

$M[0] \leftarrow 0$

for $j = 1$ to $n$ do

$M[j] \leftarrow \max\{M[j - 1], w_j + M[p[j]]\}$

end for