Intro to Dynamic Programming
Duke COMPSCI 309s, Spring 2014

Siyang Chen
Introduction

“Those who cannot remember the past are condemned to repeat it”

– George Santayana
Introduction

*Dynamic programming* is a general technique which is useful solving various types of problems which have what is referred to as *optimal substructure*.
Introduction

*Dynamic programming* is a general technique which is useful solving various types of problems which have what is referred to as *optimal substructure*. Today we’re going to look at some concrete examples of these problems any dynamic programming techniques to solve them.
Introduction

Let's look at methods 1 and 2 from this article:
http://www.geeksforgeeks.org/
program-for-nth-fibonacci-number/
Now let’s go over this problem from last week’s problem set: http://codeforces.com/contest/379/problem/D
Now let’s go over this problem from last week’s problem set: http://codeforces.com/contest/379/problem/D

The problem is this: find some strings $s_1$ and $s_2$ of length $n$ and $m$, respectively, such that $s_k$ contains $x$ occurrences of the string AC. The strings are defined recursively as $s_i = s_{i-2} + s_{i-1}$.

- $k \in [3, 50]$, $x \in [0, 10^9]$, and $n, m \in [1, 100]$
First, we can try a simpler problem: given some $s_1$ and $s_2$, can we efficiently compute the number of occurrences of AC in $s_k$?
New Year Letter

First, we can try a simpler problem: given some $s_1$ and $s_2$, can we efficiently compute the number of occurrences of AC in $s_k$?

- The sequence is defined recursively, which strongly suggests some sort of memoisation / dynamic programming technique. The strings are too long to explicitly generate, but can we do something smarter?
First, we can try a simpler problem: given some $s_1$ and $s_2$, can we efficiently compute the number of occurrences of AC in $s_k$?

- The sequence is defined recursively, which strongly suggests some sort of memoisation / dynamic programming technique. The strings are too long to explicitly generate, but can we do something smarter?

- So what are our recursive variables?
New Year Letter

- Let’s start from something simple:

\[ \text{countAC}_i = \text{countAC}_{i-2} + \text{countAC}_{i-1} + (\quad) \]
Let’s start from something simple:

\[
\text{countAC}_i = \text{countAC}_{i-2} + \text{countAC}_{i-1} + (1 \text{ if } s_{i-2} \text{ ends with A and } s_{i-1} \text{ starts with C})
\]
Let’s start from something simple:

\[ \text{countAC}_i = \text{countAC}_{i-2} + \text{countAC}_{i-1} + (1 \text{ if } s_{i-2} \text{ ends with } A \text{ and } s_{i-1} \text{ starts with } C) \]

But \( s_{i-2} \) ends with \( A \) if and only if \( s_{i-3} \) ends with \( A \)
Let’s start from something simple:

\[ \text{countAC}_i = \text{countAC}_{i-2} + \text{countAC}_{i-1} \]
\[ + (1 \text{ if } s_{i-2} \text{ ends with } A \text{ and } s_{i-1} \text{ starts with } C) \]

- But \( s_{i-2} \) ends with \( A \) if and only if \( s_{i-3} \) ends with \( A \)
- And \( s_{i-1} \) starts with \( C \) if and only if \( s_{i-3} \) starts with \( C \)
New Year Letter

Idea: Let $startC_i$ and $endA_i$ denote whether string $i$ starts with $C$ or ends with $A$. Then

\[
\begin{align*}
  countAC_i &= countAC_{i-1} + 2 \cdot countAC_{i-2} + \left( endA_i \cdot startC_{i-1} \right),
  \quad \text{startC}_i = \text{startC}_{i-2} \\
  countAC_i &= countAC_{i-1} + 2 \cdot countAC_{i-2} + \left( endA_i \cdot startC_{i-1} \right),
  \quad \text{endA}_i = \text{endA}_{i-1}
\end{align*}
\]

Complexity to compute $countAC_k$? $O(k)$ time and memory.
New Year Letter

Idea: Let $startC_i$ and $endA_i$ denote whether string $i$ starts with $C$ or ends with $A$. Then

$$countAC_i = countAC_{i-2} + countAC_{i-1} + (endA_{i-2} \land startC_{i-1})$$
Idea: Let $startC_i$ and $endA_i$ denote whether string $i$ starts with $C$ or ends with $A$. Then

$$countAC_i = countAC_{i-2} + countAC_{i-1} + (endA_{i-2} \land startC_{i-1})$$

$$startC_i = startC_{i-2}$$
New Year Letter

Idea: Let $startC_i$ and $endA_i$ denote whether string $i$ starts with $C$ or ends with $A$. Then

$$countAC_i = countAC_{i-2} + countAC_{i-1} + (endA_{i-2} \land startC_{i-1})$$

$$startC_i = startC_{i-2}$$

$$endA_i = endA_{i-1}$$
Idea: Let $startC_i$ and $endA_i$ denote whether string $i$ starts with $C$ or ends with $A$. Then

$$countAC_i = countAC_{i-2} + countAC_{i-1} + (endA_{i-2} \land startC_{i-1})$$

$$startC_i = startC_{i-2}$$

$$endA_i = endA_{i-1}$$

Complexity to compute $countAC_k$?
Idea: Let $\text{start}C_i$ and $\text{end}A_i$ denote whether string $i$ starts with $C$ or ends with $A$. Then

$$\text{countAC}_i = \text{countAC}_{i-2} + \text{countAC}_{i-1} + (\text{end}A_{i-2} \land \text{start}C_{i-1})$$

$$\text{start}C_i = \text{start}C_{i-2}$$

$$\text{end}A_i = \text{end}A_{i-1}$$

Complexity to compute $\text{countAC}_k$? $O(k)$ time and memory.
New Year Letter

So let’s return to the original problem:
New Year Letter

So let’s return to the original problem:

- Note that $k$ is relatively small, as are $n$ and $m$. 
New Year Letter

So let’s return to the original problem:

- Note that $k$ is relatively small, as are $n$ and $m$.
- Let’s just search over all possible values of $countAC_i$, $startC_i$, and $endA_i$ for $i \in \{1, 2\}$
So let’s return to the original problem:

- Note that \( k \) is relatively small, as are \( n \) and \( m \).
- Let’s just search over all possible values of \( count_{AC_i} \), \( start_{C_i} \), and \( end_{A_i} \) for \( i \in \{1, 2\} \).
- We can compute \( count_{AC_k} \) for the fixed initial conditions and check if it’s equal to \( x \), the desired value.
So let’s return to the original problem:

- Note that $k$ is relatively small, as are $n$ and $m$.
- Let’s just search over all possible values of $\text{countAC}_i$, $\text{startC}_i$, and $\text{endA}_i$ for $i \in \{1, 2\}$
- We can compute $\text{countAC}_k$ for the fixed initial conditions and check if it’s equal to $x$, the desired value.
- Complexity?
New Year Letter

So let’s return to the original problem:

- Note that $k$ is relatively small, as are $n$ and $m$.
- Let’s just search over all possible values of $countAC_i$, $startC_i$, and $endA_i$ for $i \in \{1, 2\}$
- We can compute $countAC_k$ for the fixed initial conditions and check if it’s equal to $x$, the desired value.
- Complexity? $\approx 2^4 \times n \times m \times k = 4 \times 10^6$ operations.
New Year Letter

So let’s return to the original problem:

- Note that $k$ is relatively small, as are $n$ and $m$.
- Let’s just search over all possible values of $\text{countAC}_i$, $\text{startC}_i$, and $\text{endA}_i$ for $i \in \{1, 2\}$
- We can compute $\text{countAC}_k$ for the fixed initial conditions and check if it’s equal to $x$, the desired value.
- Complexity? $\approx 2^4 \times n \times m \times k = 4 \times 10^6$ operations.
- Example solution:
New Year Letter

So let’s return to the original problem:

- Note that \( k \) is relatively small, as are \( n \) and \( m \).
- Let’s just search over all possible values of \( countAC_i, startC_i, \) and \( endA_i \) for \( i \in \{1, 2\} \)
- We can compute \( countAC_k \) for the fixed initial conditions and check if it’s equal to \( x \), the desired value.
- Complexity? \( \approx 2^4 \times n \times m \times k = 4 \times 10^6 \) operations.
- Example solution:
  https://github.com/md143rbh7f/competitions/blob/master/codeforces/goodbye2013/Letter.java
Knapsack Variations

Another fundamental class of problems in dynamic programming is what I call the *knapsack variations*:
Knapsack Variations

Another fundamental class of problems in dynamic programming is what I call the knapsack variations:

▶ Given a set of coins with values \( \{c_1, \ldots, c_N\} \), is it possible to make the currency value \( X \)?
Knapsack Variations

Another fundamental class of problems in dynamic programming is what I call the *knapsack variations*:

- Given a set of coins with values \( \{c_1, \ldots, c_N\} \), is it possible to make the currency value \( X \)?
- Given a set of coins with values \( \{c_1, \ldots, c_N\} \), what is the smallest amount of coins I can use to make the currency value \( X \)?
Knapsack Variations

Another fundamental class of problems in dynamic programming is what I call the \textit{knapsack variations}:

- Given a set of coins with values \( \{c_1, \ldots, c_N\} \), is it possible to make the currency value \( X \)?
- Given a set of coins with values \( \{c_1, \ldots, c_N\} \), what is the smallest amount of coins I can use to make the currency value \( X \)?
- Given a set of items with values \( \{v_1, \ldots, v_N\} \) and weights \( \{w_1, \ldots, w_N\} \), what is the set of items \( S \) of largest total value \( \sum_{i \in S} v_i \) such that the total weight \( \sum_{i \in S} w_i \) is at most \( W \)?
Knapsack Variations

Another fundamental class of problems in dynamic programming is what I call the *knapsack variations*:

- Given a set of coins with values \(\{c_1, \ldots, c_N\}\), is it possible to make the currency value \(X\)?
- Given a set of coins with values \(\{c_1, \ldots, c_N\}\), what is the smallest amount of coins I can use to make the currency value \(X\)?
- Given a set of items with values \(\{v_1, \ldots, v_N\}\) and weights \(\{w_1, \ldots, w_N\}\), what is the set of items \(S\) of largest total value \(\sum_{i \in S} v_i\) such that the total weight \(\sum_{i \in S} w_i\) is at most \(W\)?

We proceed by solving the first problem (the change-making problem) and then extending it naturally to the subsequent two.
Making Change

*Given a set of coins with values \( \{c_1, \ldots, c_N\} \), is it possible to make the currency value \( X \) ?*
Making Change

Given a set of coins with values \{c_1, \ldots, c_N\}, is it possible to make the currency value \(X\)?

- We could try every subset of coins, but that’s \(O(2^N)\).
Making Change

Given a set of coins with values \( \{c_1, \ldots, c_N\} \), is it possible to make the currency value \( X \)?

- We could try every subset of coins, but that’s \( O(2^N) \).
- Assuming that \( C = \sum_i c_N \) is small (which is not always true), then we can define a recursive function \( f(i, j) \), which is true if and only if it’s possible to make currency value \( j \) with coins \( \{c_1, \ldots, c_i\} \).
Making Change

Now suppose we want to compute $f(i, j)$. We need to decide whether we want to include $c_i$ in our set of coins which sum up to $j$. 
Making Change

Now suppose we want to compute $f(i, j)$. We need to decide whether we want to include $c_i$ in our set of coins which sum up to $j$.

- If we don’t include $c_i$, then we must make value $j$ with the remaining coins $\{c_1, \ldots, c_{i-1}\}$. This is possible if and only if $f(i - 1, j)$ is true.
Making Change

Now suppose we want to compute $f(i, j)$. We need to decide whether we want to include $c_i$ in our set of coins which sum up to $j$.

- If we don’t include $c_i$, then we must make value $j$ with the remaining coins $\{c_1, \ldots, c_{i-1}\}$. This is possible if and only if $f(i - 1, j)$ is true.

- If we do include $c_i$, then we need to make value $j - c_i$ with coins $\{c_1, \ldots, c_{i-1}\}$. This is possible if and only if $f(i - 1, j - c_i)$ is true.
Making Change

Now suppose we want to compute \( f(i, j) \). We need to decide whether we want to include \( c_i \) in our set of coins which sum up to \( j \).

- If we don’t include \( c_i \), then we must make value \( j \) with the remaining coins \( \{c_1, \ldots, c_{i-1}\} \). This is possible if and only if \( f(i - 1, j) \) is true.

- If we do include \( c_i \), then we need to make value \( j - c_i \) with coins \( \{c_1, \ldots, c_{i-1}\} \). This is possible if and only if \( f(i - 1, j - c_i) \) is true.

Since those are the only two possibilities, then we can write

\[
f(i, j) = f(i - 1, j) \lor f(i - 1, j - c_i)
\]
Making Change

Using $f(i, j) = f(i - 1, j) \lor f(i - 1, j - c_i)$, we can write the following dynamic program to compute $f$ which uses $O(C)$ memory and $O(CN)$ time:

Let $f$ be an array filled with False.

$f(0) = True$

For $i$ from 1 to $N$:

For $j$ from $C$ to 0:

$f(j) = f(j) \lor f(j - c_i)$

Return $f(C)$

At the end $f(j)$ is true if and only if we can make value $j$ with all our coins.

Exercise for the reader: why do we need to run the inner loop backwards?
Making Change

Using $f(i, j) = f(i - 1, j) \lor f(i - 1, j - c_i)$, we can write the following dynamic program to compute $f$ which uses $O(C)$ memory and $O(CN)$ time:

- Let $f$ be an array filled with $False$. 

Making Change

Using \( f(i, j) = f(i - 1, j) \lor f(i - 1, j - c_i) \), we can write the following dynamic program to compute \( f \) which uses \( O(C) \) memory and \( O(CN) \) time:

- Let \( f \) be an array filled with \( False \).
- \( f(0) \leftarrow True \)
Making Change

Using \( f(i, j) = f(i - 1, j) \lor f(i - 1, j - c_i) \), we can write the following dynamic program to compute \( f \) which uses \( O(C) \) memory and \( O(CN) \) time:

- Let \( f \) be an array filled with \( False \).
- \( f(0) \leftarrow True \)
- For \( i \) from 1 to \( N \):
Making Change

Using \( f(i, j) = f(i - 1, j) \lor f(i - 1, j - c_i) \), we can write the following dynamic program to compute \( f \) which uses \( O(C) \) memory and \( O(CN) \) time:

- Let \( f \) be an array filled with \( False \).
- \( f(0) \leftarrow True \)
- For \( i \) from 1 to \( N \):
  - For \( j \) from \( C \) to 0:
Making Change

Using \( f(i, j) = f(i - 1, j) \lor f(i - 1, j - c_i) \), we can write the following dynamic program to compute \( f \) which uses \( O(C) \) memory and \( O(CN) \) time:

- Let \( f \) be an array filled with \( False \).
- \( f(0) \leftarrow True \)
- For \( i \) from 1 to \( N \):
  - For \( j \) from \( C \) to 0:
    - \( f(j) \leftarrow f(j) \lor f(j - c_i) \)

At the end, \( f(j) \) is true if and only if we can make value \( j \) with all our coins.

Exercise for the reader: why do we need to run the inner loop backwards?
Making Change

Using \( f(i, j) = f(i - 1, j) \lor f(i - 1, j - c_i) \), we can write the following dynamic program to compute \( f \) which uses \( O(C) \) memory and \( O(CN) \) time:

- Let \( f \) be an array filled with \( True \).
- \( f(0) \leftarrow True \)
- For \( i \) from 1 to \( N \):
  - For \( j \) from \( C \) to 0:
    - \( f(j) \leftarrow f(j) \lor f(j - c_i) \)
- Return \( f(C) \)
Making Change

Using \( f(i, j) = f(i - 1, j) \lor f(i - 1, j - c_i) \), we can write the following dynamic program to compute \( f \) which uses \( O(C) \) memory and \( O(CN) \) time:

- Let \( f \) be an array filled with \( False \).
- \( f(0) \leftarrow True \)
- For \( i \) from 1 to \( N \):
  - For \( j \) from \( C \) to 0:
    - \( f(j) \leftarrow f(j) \lor f(j - c_i) \)
- Return \( f(C) \)

At the end \( f(j) \) is true if and only if we can make value \( j \) with all our coins.
Making Change

Using \( f(i, j) = f(i - 1, j) \lor f(i - 1, j - c_i) \), we can write the following dynamic program to compute \( f \) which uses \( O(C) \) memory and \( O(CN) \) time:

1. Let \( f \) be an array filled with \( False \).
2. \( f(0) \leftarrow True \)
3. For \( i \) from 1 to \( N \):
   - For \( j \) from \( C \) to 0:
     - \( f(j) \leftarrow f(j) \lor f(j - c_i) \)
4. Return \( f(C) \)

At the end \( f(j) \) is true if and only if we can make value \( j \) with all our coins.

Exercise for the reader: why do we need to run the inner loop backwards?
Making Change

Let’s consider a variation: Find the minimum number of coins needed to make $C$. 
Let’s consider a variation: Find the minimum number of coins needed to make $C$.

- Let $f$ be an array filled with $\infty$.
- $f(0) \leftarrow 0$
Making Change

Let’s consider a variation: Find the minimum number of coins needed to make $C$.

- Let $f$ be an array filled with $\infty$.
- $f(0) \leftarrow 0$
- For $i$ from 1 to $N$:
Making Change

Let’s consider a variation: Find the minimum number of coins needed to make \( C \).

- Let \( f \) be an array filled with \( \infty \).
- \( f(0) \leftarrow 0 \)
- For \( i \) from 1 to \( N \):
  - For \( j \) from \( C \) to 0:
    - For \( j \) from \( C \) to 0:
Making Change

Let’s consider a variation: Find the minimum number of coins needed to make $C$.

- Let $f$ be an array filled with $\infty$.
- $f(0) \leftarrow 0$
- For $i$ from 1 to $N$:
  - For $j$ from $C$ to 0:
    - $f(j) \leftarrow \min\{f(j), f(j - c_i) + 1\}$
Let's consider a variation: Find the minimum number of coins needed to make $C$.

- Let $f$ be an array filled with $\infty$.
- $f(0) \leftarrow 0$
- For $i$ from 1 to $N$:
  - For $j$ from $C$ to 0:
    - $f(j) \leftarrow \min\{f(j), f(j - c_i) + 1\}$
- Return $f(C)$
A final variation, the knapsack problem: finding the maximum value $\sum_{i \in S} v_i$ such that $\sum_{i \in S} w_i \leq W$ for some $W$. 
A final variation, the knapsack problem: finding the maximum value $\sum_{i \in S} v_i$ such that $\sum_{i \in S} w_i \leq W$ for some $W$.

- Let $f$ be an array filled with $-\infty$.
- $f(0) \leftarrow 0$
Knapsack

A final variation, the knapsack problem: finding the maximum value \( \sum_{i \in S} v_i \) such that \( \sum_{i \in S} w_i \leq W \) for some \( W \).

- Let \( f \) be an array filled with \(-\infty\).
- \( f(0) \leftarrow 0 \)
- For \( i \) from 1 to \( N \):
Knapsack

A final variation, the knapsack problem: finding the maximum value \( \sum_{i \in S} v_i \) such that \( \sum_{i \in S} w_i \leq W \) for some \( W \).

- Let \( f \) be an array filled with \(-\infty\).
- \( f(0) \leftarrow 0 \)
- For \( i \) from 1 to \( N \):
  - For \( j \) from \( W \) to 0:
A final variation, the knapsack problem: finding the maximum value \( \sum_{i \in S} v_i \) such that \( \sum_{i \in S} w_i \leq W \) for some \( W \).

- Let \( f \) be an array filled with \(-\infty\).
- \( f(0) \leftarrow 0 \)
- For \( i \) from 1 to \( N \):
  - For \( j \) from \( W \) to 0:
    - \( f(j) \leftarrow \max\{f(j), f(j - w_i) + v_i\} \)
Knapsack

A final variation, the knapsack problem: finding the maximum value $\sum_{i \in S} v_i$ such that $\sum_{i \in S} w_i \leq W$ for some $W$.

- Let $f$ be an array filled with $-\infty$.
- $f(0) \leftarrow 0$
- For $i$ from 1 to $N$:
  - For $j$ from $W$ to 0:
    - $f(j) \leftarrow \max\{f(j), f(j - w_i) + v_i\}$
- Return $\max_j \{f(j)\}$
Final Notes

A lot of dynamic programming problems can be solved by cleverly applying the techniques covered in this presentation.
Eternal Recurrence

The greatest weight. – What, if some day or night a demon were to steal after you into your loneliest loneliness and say to you: “This life as you now live it and have lived it, you will have to live once more and innumerable times more; and there will be nothing new in it, but every pain and every joy and every thought and sigh and everything unutterably small or great in your life will have to return to you, all in the same succession and sequence – even this spider and this moonlight between the trees, and even this moment and I myself. The eternal hourglass of existence is turned upside down again and again, and you with it, speck of dust!” Would you not throw yourself down and gnash your teeth and curse the demon who spoke thus? Or have you once experienced a tremendous moment when you would have answered him: “You are a god and never have I heard anything more divine”? If this thought gained possession of you, it would change you as you are, or perhaps crush you.

– Friedrich Nietzsche