CS 255, Spring 2014, SJSU

Dynamic Programming

Fernando Lobo

Dynamic Programming

- Another algorithm design technique, like Divide and Conquer, and Greedy algorithms.
- The name *Dynamic Programming* is a bit misleading.
  - *Programming* ⇒ suggests computer programming.
  - *Dynamic* ⇒ suggests something that changes through time.
- The Dynamic Programming technique has not much to do with these things!

What is Dynamic Programming?

- It's a technique for solving problems.
- The idea is to solve subproblems and store their results.
- Those results are used for solving larger subproblems, and again we store their results.
- And so on until we are able to sove the complete problem.

Comparison with Divide and Conquer

Similarities

- To solve a problem we combine solutions of subproblems.

Differences

- D&C is efficient when the subproblems are distinct.
- If we have to solve the same subproblem over and over, D&C becomes inefficient.
- With Dynamic Programming, each subproblem is only solved once.

A very simple example

- A simple example: Compute the n^{th} element from the Fibonacci sequence.

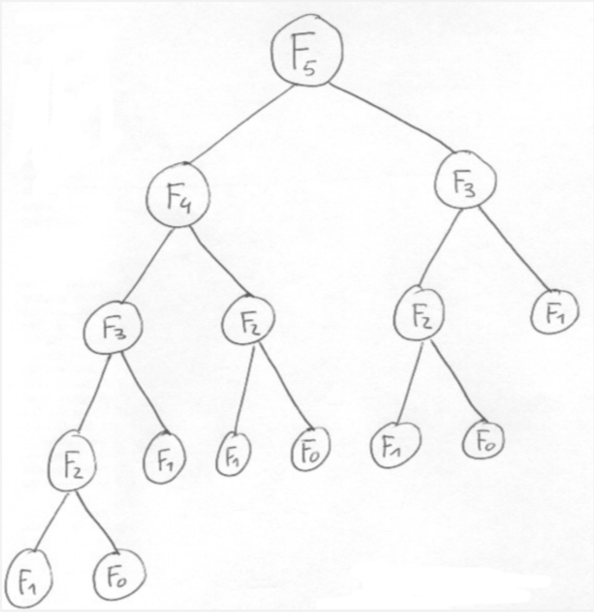
$$F_n = \begin{cases} 0 & , \text{se } n = 0 \\ 1 & , \text{se } n = 1 \\ F_{n-1} + F_{n-2} & , \text{se } n > 1 \end{cases}$$

Pseudocode

```
FIB-REC(n)
    if n == 0
        return 0
    if n == 1
        return 1
    return FIB-REC(n - 1) + FIB-REC(n - 2)
```

This algorithm is very bad. Why?

Let's see what happens with $n = 5$



Fibonacci: D&C Algorithm

- We are computing the same thing several times!
- We can prove that $F_{n+1}/F_n \approx \frac{1+\sqrt{5}}{2} \approx 1.62$ $\implies F_n > 1.6^n$
- What's the running time of the algorithm?
  - F_n is the sum of all the leaves of the tree.
  - $F_n > 1.6^n \implies$ tree has at least 1.6^n leaves.
- Algorithm's running time is $\Omega(1.6^n)$.
- Implement it and try running with increasing values of n.

Fibonacci: D&C Algorithm

- With $n = 5$ we compute:
  - $F_4 \rightarrow 1$ time
  - $F_3 \rightarrow 2$ times
  - $F_2 \rightarrow 3$ times
  - $F_1 \rightarrow 5$ times
  - $F_0 \rightarrow 3$ times
- A lot of unnecessary work. Each F_i should be computed once.
- We can do it with Dynamic Programming.

Fibonacci: Dynamic Programming algorithm

- The idea is to solve the problem bottom-up, starting with the base cases and storing the results in order to solve larger subproblems.

```
FIB-PD(n)
    F[0] = 0
    F[1] = 1
    for i = 2 to n
        F[i] = F[i − 1] + F[i − 2]
    return F[n]
```

- Time complexity: $\Theta(n)$.
- Space complexity: $\Theta(n)$.

Fibonacci: Dynamic Programming algorithm

- Space complexity can be reduced from $\Theta(n)$ to $\Theta(1)$ because to compute F_i we only need to keep the solutions of two subproblems: F_{i-1} and F_{i-2}.

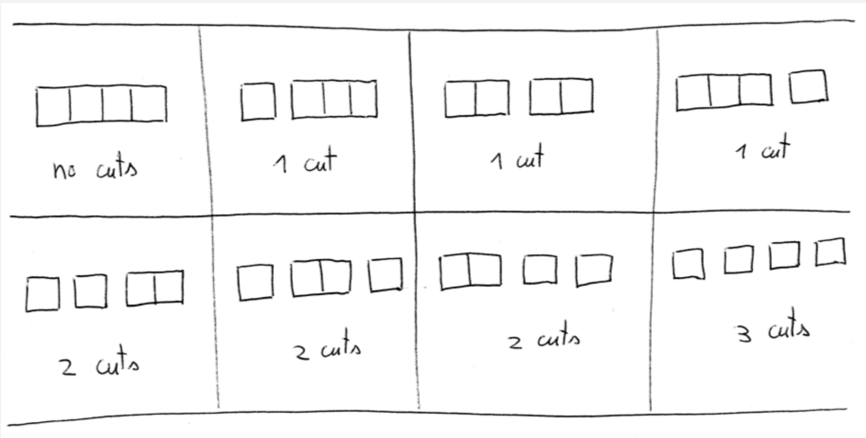
```
FIB-PD-V2(n)
    if n == 0
        return 0
    if n == 1
        return 1
    back2 = 0
    back1 = 1
    for i = 2 to n
        next = back1 + back2
        back2 = back1
        back1 = next
    return next
```


Another example: Rod cutting

- Problem: Given a rod of length n and a table of prices p_i for for pieces of length i ($i = 1, 2, .. n$), determine the maximum revenue r_n that can be obtained by cutting the rod into pieces and selling them. Assume cuts are free and rod lengths are integers.
- Example: $n = 4$

length i	1	2	3	4
price p_i	1	5	8	9

8 ways of cutting it



- Best solution: cut into two pieces of size 2. Total revenue is $5 + 5 = 10$.
- For each $i = 1 \mathinner{.\,.} n-1$, either cut or not cut $\implies 2^{n-1}$ ways of cutting the rod.

Exploiting problem structure

- Let's try to define the optimal solution in terms of optimal solutions of subproblems.
- Let r_i be the maximum revenue for a rod of length i.
- Then r_n will be the maximum of:
  - p_n
  - $r_1 + r_{n-1}$
  - $r_2 + r_{n-2}$
  - ...
  - $r_{n-1} + r_1$

Simpler decomposition

- Every optimal solution must have a leftmost piece (potentially of size n in case there's no cuts).
- Total revenue will be the cost of that piece plus the cost of the best revenue obtainable but cutting the remaining piece.
- r_n is the maximum of:
  - $p_1 + r_{n-1}$
  - $p_2 + r_{n-2}$
  - ...
  - $p_n + r_0$

Pseudocode

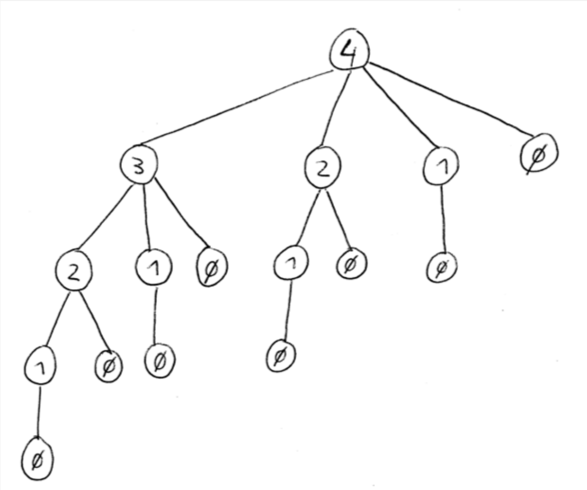
CUT-ROD(p, n)

```
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

- Very inefficient, just like the Fibonacci example.

Recursion tree of CUT-ROD with $n = 4$

- Computes the same subproblems over and over.
- Running time: $\Theta(2^n)$.



Dynamic programming approach

- Solve each subproblem once and store the result for further use.
- Do a bottom-up approach: solve smaller subproblems first.
- When we need to solve a larger subproblem, we use the pre-computed results of smaller subproblems.

Dynamic programming algorithm

BOTTOM-UP-CUT-ROD(p, n)

```
Let r[0 . . n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

r:

0	1	2	3	4
$\emptyset$	1	5	8	10

$\max(p_1 + r_\emptyset) = 1$

$\max(p_1 + r_1, p_2 + r_\emptyset)$
$= \max(1+1, 5+\emptyset) = 5$

$\max(p_1 + r_2, p_2 + r_1, p_3 + r_\emptyset)$
$= \max(1+5, 5+1, 8+\emptyset) = 8$

$\max(p_1 + r_3, p_2 + r_2, p_3 + r_1, p_4 + r_\emptyset)$
$= \max(1+8, 5+5, 8+1, 9+\emptyset) = 10$

Running time

- Two nested for loops depending on n, and constant time in each iteration. Running time is $\Theta(n^2)$.
- We reduced from exponential to polynomial time.

Reconstructing the solution

- Algorithm returned the value of the optimal solution, not the solution itself (where to cut).
- Can obtain the solution with a little modification: $s[i]$ saves the first cut point for a rod of length i.

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  Let $r[0..n]$ and $s[0..n]$ be new arrays
  $r[0] = 0$
  for $j = 1$ **to** n
    $q = -\infty$
    for $i = 1$ **to** j
      $q = \max(q, p[i] + r[j-i])$
      $s[j] = i$
    $r[j] = q$
  return r and s

Reconstructing the solution

i	0	1	2	3	4
$r[i]$	0	1	5	8	10
$s[i]$	0	1	2	3	2

PRINT-CUT-ROD-SOLUTION(p, n)
  $(r, s) =$ EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  while $n > 0$
    print $s[n]$
    $n = n - s[n]$

Memoization

- A technique similar to Dynamic Programming.
- Keeps the algorithm in a recursive (*top-down*) form.
- The idea is to flag unsolved subproblems with "Unknown".
- Then, to solve a subproblem we first verify if it has already been solved.
  - If yes, we simply return the solution previously stored.
  - If not, we solve the subproblem and store its solution for further use.
- Each subproblem is only solved once.

Memoized version of Cut-Rod

$\textsc{Memoized-Cut-Rod}(p, n)$

Let $r[0\,..\,n]$ be a new array
for $i = 0$ **to** n
    $r[i] = -\infty$
return $\textsc{Memoized-Cut-Rod-Aux}(p, n, r)$

$\textsc{Memoized-Cut-Rod-Aux}(p, n, r)$

if $r[n] \geq 0$
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ **to** n
        $q = \max(q, p[i] + \textsc{Memoized-Cut-Rod-Aux}(p, n-i, r))$
$r[n] = q$
return q

Running time

- Each subproblem is only solved once.
- Subproblems have sizes $0, 1, \ldots, n$, and require a for loop over its size.
- Running time is also $\Theta(n^2)$.

Longest Common Subsequence (LCS)

- Given two sequences, $X = x_1x_2 \ldots x_m$ and $Y = y_1y_2 \ldots y_n$, find a common subsequence between X and Y which is as long as possible.
- Example:
  - $X =$ p a c i f i c
  - $Y =$ a t l a n t i c
  - $\text{LCS}(X, Y) =$ a i c

Brute force algorithm

- Generate all subsequences of X and for each one check if it is also a subsequence of Y, keeping aside the longest common subsequence found so far.
- Running time?
  - $\Theta(2^m) \rightarrow$ to generate all subsequences of X.
  - $\Theta(n) \rightarrow$ to check if a subsequence of X is a subsequence of Y.
  - Total: $\Theta(n\ 2^m)$
  - Exponential. Very bad!

Can we apply dynamic programming?

- If yes we should be able to define the problem recursively in terms of subproblems.
- The number of subproblems has to be relatively small (polynomial in n and m) so that dynamic programming can be useful.
- Once we define the problem in terms of subproblems, we can solve it bottom-up starting with the base cases and storing the solutions as we move to larger and larger subproblems.

Optimal substructure

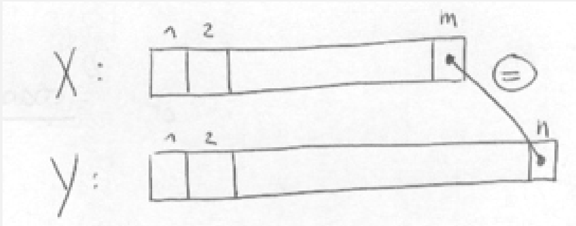
- Let's look at prefixes of X and Y.
- Let X_i be the prefix of the the first i elements of X.
- Example: $X = \mathrm{p\ a\ c\ i\ f\ i\ c}$
  - $X_4 = \mathrm{p\ a\ c\ i}$
  - $X_0 = \emptyset$
  - $X_7 = \mathrm{p\ a\ c\ i\ f\ i\ c}$

Optimal substructure

- Let $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$.
- Let $Z = z_1 z_2 \dots z_k$ be a LCS between X and Y.
- Three cases:
  1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is a LCS between X_{m-1} and Y_{n-1}.
  2. If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is a LCS between X_{m-1} and Y_n.
  3. If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is a LCS between X_m and Y_{n-1}.

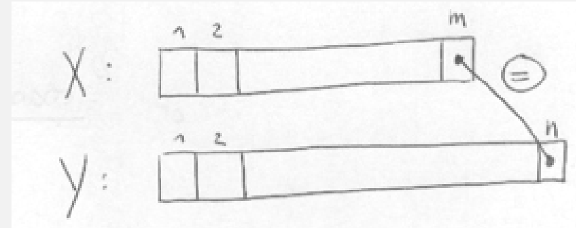
Proof of case 1

Case 1: If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is a LCS between X_{m-1} and Y_{n-1}.



- We need to prove that $z_k = x_m = y_n$. Suppose that's not true. Then the subsequence $Z' = z_1 z_2 \dots z_k x_m$ is a common subsequence between X and Y and has length $k+1$.
  - $\Rightarrow$ Contradicts Z being a LCS between X and Y.

Proof of case 1



- We now need to prove that Z_{k-1} is a LCS between X_{m-1} and Y_{n-1}. Suppose there is a subsequence W common to X_{m-1} and Y_{n-1} which is longer than Z_{k-1}.
  - $\Rightarrow$ length of $W \geq k$.
- The subsuquence $W' = W \mid\mid x_m$ is common to X and Y and has length $\geq k+1$.
  - Contradicts Z being a LCS between X and Y.

Proof of cases 2 and 3

Case 2: If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is a LCS between X_{m-1} and Y_n.

- Suppose the is a subsequence W common to X_{m-1} and Y_n with length $> k$. Then W is a common subsequence between X and Y.
  - $\Longrightarrow$ Contradicts Z being a LCS between X and Y.

Case 3: If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is a LCS between X_m and Y_{n-1}.

- Proof of case 3 is analogous to case 2.

In summary

We can define LCS(X_m, Y_n) in terms of subproblems.

$$\text{LCS}(X_m, Y_n) = \begin{cases} \emptyset & \text{, if } m = 0 \text{ or } n = 0 \\ \text{LCS}(X_{m-1}, Y_{n-1}) \mid\mid x_m & \text{, if } x_m = y_n \\ \text{LCS}(X_{m-1}, Y_n) \text{ or} & \\ \text{LCS}(X_m, Y_{n-1}) & \text{, or } x_m \neq y_n \end{cases}$$

Length of LCS(X, Y)

- Let us try to solve a simpler problem: Obtain $|\text{LCS}(X, Y)| \rightarrow$ the length of LCS(X, Y)
- Let $c[i, j] = |\text{LCS}(X_i, Y_j)|$
- We want to obtain $c[m, n]$

Recursive definition of $c[i, j]$

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

Recursive algorithm

```
LCS-LENGTH-REC(X, Y, i, j)
   if i == 0 or j == 0
       return 0
   elseif X[i] == Y[j]
       return LCS-LENGTH-REC(X, Y, i − 1, j − 1) + 1
   else a = LCS-LENGTH-REC(X, Y, i − 1, j)
       b = LCS-LENGTH-REC(X, Y, i, j − 1)
       return max(a, b)
```

- Initial call: LCS-LENGTH-REC(X, Y, m, n)
- Similarly to FIB-REC, the tree gives rise to many repeated subproblems.
- The algorithm's running time is exponential. But the number of distinct subproblems is $= m \cdot n$.

We can dynamic programming

```
LCS-LENGTH-DP(X, Y)
   m = X.length
   n = Y.length
   for i = 1 to m
       c[i, 0] = 0
   for j = 0 to n
       c[0, j] = 0
   for i = 1 to m
       for j = 1 to n
           if X[i] == Y[j]
               c[i, j] = c[i − 1, j − 1] + 1
           elseif c[i − 1, j] ≥ c[i, j − 1]
               c[i, j] = c[i − 1, j]
           else c[i, j] = c[i, j − 1]
   return c[m, n]
```

Example

		∅	1 a	2 t	3 l	4 a	5 n	6 t	7 i	8 c
∅		∅	∅	∅	∅	∅	∅	∅	∅	∅
1	p	∅	∅	∅	∅	∅	∅	∅	∅	∅
2	a	∅	1	1	1	1	1	1	1	1
3	c	∅	1	1	1	1	1	1	1	2
4	i	∅	1	1	1	1	1	1	2	2
5	f	∅	1	1	1	1	1	1	2	2
6	i	∅	1	1	1	1	1	1	2	2
7	c	∅	1	1	1	1	1	1	2	3

- $c[i,j]$ is filled row by row, from left to right.

How to obtain the LCS itself?

- Our algorithm only gets the length of the LCS.
- The idea is to change the code of LCS-LENGTH-DP so that each time we obtain a given $c[i,j]$, we record how it was obtained.
- This allows us to reconstruct the solution.

Modified pseudocode

```
LCS-LENGTH-DP-V2(X, Y)
   ⋮
   for i = 1 to m
       for j = 1 to n
           if X[i] == Y[j]
               c[i, j] = c[i − 1, j − 1] + 1
               b[i, j] = "↖"
           elseif c[i − 1, j] ≥ c[i, j − 1]
               c[i, j] = c[i − 1, j]
               b[i, j] = "↑"
           else c[i, j] = c[i, j − 1]
               b[i, j] = "←"
   return c[m, n] , b
```


Example

- The arrows ↑, ←, and ↖ are stored in $b[i,j]$.
- $b[i,j]$ indicates the subproblem chosen to obtain $c[i,j]$.

- Having matrix b filled in, we can easily obtain a LCS between X and Y.
- Initial call: $\textsc{Print-LCS}(b, X, m, n)$

$\textsc{Print-LCS}(b, X, i, j)$
```
if i == 0 or j == 0
    return // Do nothing
if b[i,j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print X[i]
elseif b[i,j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else
    PRINT-LCS(b, X, i, j − 1)
```


Complexity

- Running time is $\Theta(m \cdot n)$
- Again, dynamic programming allowed us to reduce from exponential to polinomial complexity.
- Your textbook has more examples.

Memoized version of LCS-Length

$\textsc{LCS-Length-Memoized}(X, Y)$
```
m = X.length
n = Y.length
for i = 0 to m
    for j = 0 to n
        c[i,j] = UNKNOWN
return M-LCS-LENGTH(X, Y, m, n)
```


$\textsc{M-LCS-Length}(X, Y, i, j)$
```
if c[i,j] == UNKNOWN
    if i == 0 or j == 0
        c[i,j] = 0
    elseif X[i] == Y[j]
        c[i,j] = M-LCS-LENGTH(X, Y, i − 1, j − 1) + 1
    else a = M-LCS-LENGTH(X, Y, i − 1, j)
        b = M-LCS-LENGTH(X, Y, i, j − 1)
        c[i,j] = max(a, b)
return c[i,j]
```

How to apply dynamic programming?

To apply *dynamic programming* or *memoization* to solve a problem we need to do 4 things:

1. Caracterize the structure of an optimal solution.
2. Define the value of an optimal solution recursively in terms of optimal solutions of subproblems.
3. Compute the value of the optimal solution bottom-up (in case of dynamic programming) or top-down (in case of memoization).
4. Obtain the optimal solution from the information previously computed and stored in step 3.