Dynamic Programming

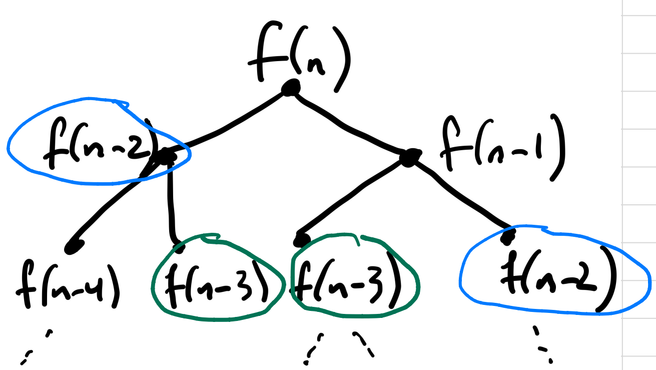
— "program" as in "exercise program", not "computer program"

Recall Fibonacci

recursive

$f(n)$
  if $n = 0$ **then**
    return 0
  elif $n = 1$ **then**
    return 1
  else
    return $f(n-1) + f(n-2)$
  fi

$T(n) = T(n-1) + T(n-2) + c$ so runtime grows like the Fibonacci numbers. BAD!



duplication!

iterative

$f(0) := 0$
$f(1) := 1$
for i **from** 2 **to** n **do**
  $f(i) := f(i-1) + f(i-2)$
od

$O(n)$ arithmetic operations. GOOD!
- an example of dynamic programming

Main idea of dynamic programming:

solve "subproblems" from smaller to larger (bottom up) storing solutions

Runtime: (# subproblems)
  $\times$ (time to solve one subproblem)

Text segmentation

Given a string of letters $A[1..n]$, $A[i] \in \{$A, B,. . ., Z$\}$, can you split into words?

Assume you have a test

$$\text{Word}[i,j] = \begin{cases} \text{True if } A[i..j] \text{ is a word} \\ \text{False otherwise} \end{cases}$$

where each call takes $O(1)$

e.g., THEMEMPTY splits into THEM EMPTY

Note: a greedy solution might try to find

the shortest word $A[1..i]$ (prefix) : THE MEMPTY wrong

or the longest word $A[1..i]$: THEME MPTY wrong

Can we do something like Fibonacci? Suppose we knew

$$\text{Split}[k] = \begin{cases} \text{True if } A[1..k] \text{ is splittable} \\ \text{False otherwise} \end{cases} \quad \text{for } k = 0..n-1$$

Can we then find Split$[n]$? Try Split$[j]$ **and** Word$[j+1,n]$ for all $j = 0..n-1$.

Claim: Split$[n]$ if and only if at least one j gives True. Why?

$\Leftarrow$ we have a way to split $A[1..n]$

$\Rightarrow$ if $A[1..n]$ is splittable, take $A[j+1..n]$ as last word

Resulting algorithm:

Split[0] := True
for k **from** 1 **to** n **do**
    Split[k] := False
    for j **from** 0 **to** $k-1$ **do**
        if Split[j] **and** Word[$j+1, k$] **then**
            Split[k] := True
        fi
    od
od

Runtime: $O(n^2)$

Ex. Show how to compute the actual split

Longest Increasing Subsequence

Given a sequence of numbers, $A[1..n]$, $A[i] \in \mathbb{N}$, find the longest increasing subsequence.

e.g., 5 2 1 4 3 1 6 9 2 increasing subsequence of length 4

Following previous approach, what if we set

$\text{LIS}[k]$ = length of longest increasing subsequence of $A[1..k]$?

This does not seem to give enough info to get $\text{LIS}[n]$ from previous $\text{LIS}[k]$'s.
$\rightarrow$ need to see if $A[n]$ is large enough to add to a previous sequence

Better Idea: Let $\text{LISe}[k]$ = length of longest increasing subsequence of $A[1..k]$ that ends with $A[k]$.

Algorithm

$\text{LISe}[1] := 1$
for k **from** 2 **to** n **do**
  $\text{LISe}[k] := 1$
  for j **from** 1 **to** $k-1$ **do**
    if $A[k] > A[j]$ **then**
      $\text{LISe}[k] := \max\{\text{LISe}[k], \text{LISe}[j] + 1\}$
    fi
  od
od

Ex. Argue correctness

Runtime $O(n^2)$

Example

$A =$

1	2	3	4	5	6	7	8	9
5	2	1	4	3	1	6	9	2

$LISe =$

1	2	3	4	5	6	7	8	9
1	1	1	2	2	1	3	4	2

coming from
(value of j used)

1	2	3	4	5	6	7	8	9
0	0	0	2	2	0	4	7	3

this array allows us to recover the seq.

Run time: $O(n^2)$

How do we get the final answer?

maximum entry in LISe

OR

add dummy entry $A[n+1] = +\infty$ and return $\text{LISe}[n+1] - 1$

Note: there is an $O(n \log n)$ time algorithm

Longest Common Subsequence

Recall pattern matching from CS 240:
Given a long string T and short pattern P find occurrences of P in T.

Useful in grep, find, etc.

Also useful: given two long strings find longest common subsequence

$x =$ T A R M A C

$y =$ C A T A M A R A N

Note that we can skip letters in both strings, but must preserve ordering.

Given strings $x_1 \ldots x_n$ and $y_1 \ldots y_m$:

Let $M(i, j)$ = length of longest common subsequence of $x_1 \cdots x_{i-1} x_i$ and $y_1 \cdots y_{j-1} y_j$.

How can we solve this subproblem based on solutions to "smaller" subproblems?

Choices: match $x_i = y_j$, skip x_i, skip y_j

$$
\begin{aligned}
M(i,0) &= 0 \\
M(0,j) &= 0 \\
M(i,j) &= \max \begin{cases} 1 + M(i-1, j-1) & \text{if } x_i = y_j \\ M(i-1, j) \\ M(i, j-1) \end{cases}
\end{aligned}
$$

Solve subproblems in any order with $M(i-1, j-1), M(i-1, j), M(i, j-1)$ before $M(i, j)$

	y	1	2	3	4					9
	$\emptyset$	C	A	T	A	M	A	R	A	N
x $\emptyset$	0	0	0	0	0	0				
1 T	0	$\boxed{0}$	$\boxed{0}$	$\boxed{1}$	$\boxed{1}$					
2 A	0	$\boxed{0}$	$\boxed{1}$	$\boxed{1}$	$\boxed{2}$					
3 R	0	$\boxed{0}$								
M	0						$\boxed{}$	$\boxed{}$		
A							$\boxed{}$	$\boxed{}$		
6 C										$\boxed{4}$ M(6,9)

1 red arrow entering each box gives optimum for that box

M matrix

to fill this entry we look at 3 other entries

for $i = 0..n$: $M(i,0) := 0$
for $j = 0..m$: $M(0,j) := 0$
for $i = 1..n$
  for $j = 1..m$

$$M(i,j) := \max \begin{cases} 1 + M(i-1, j-1) & \text{if } x_i = y_j \\ M(i-1, j) \\ M(i, j-1) \end{cases}$$

Note that this is a correct ordering of i and j.
In fact, if $x_i = y_j$ we can use the first choice (no need to check max of other two choices).

Runtime: $O(n \cdot m \quad \cdot \quad c)$

subproblems

time to solve one subproblem (compare 3 possibilities)

To find the actual max. common subsequence: work backwards from $M(n, m)$.
$\rightarrow$ Call OPT(n, m).

OPT(i, j) — recursive routine
if $i = 0$ **or** $j = 0$ **then** done **fi**
if $M(i, j) = M(i-1, j)$ **then**
OPT$(i-1, j)$
elif $M(i, j) = M(i, j-1)$ **then**
OPT$(i, j-1)$
else — we must have matched i and j
output i, j
OPT$(i-1, j-1)$
fi

Or we can record, when we fill $M(i, j)$, where the max comes from.

Next day: more sophisticated “edit” distance between strings.

Maximum common subsequence solves

Longest increasing subsequence

L	5 2 9 6 3 7 4
S	2 3 4 5 6 7 9

increasing subsequence of length 3

$S = \text{sort } L$

Claim: Longest increasing subsequence of L = maximum common subsequence of L and S.