Dynamic Programming

Recall Fibonacci

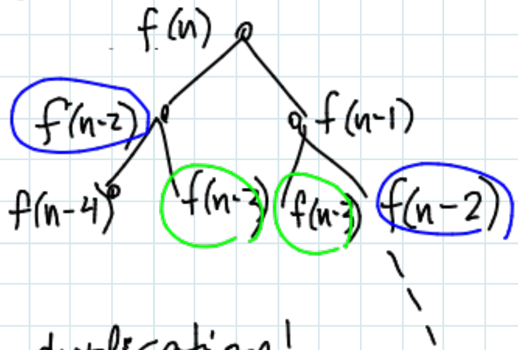
recursive

```
f(n)
   if n=0 return 0
   if n=1 return 1
   else return
      f(n-1) + f(n-2)
```

$T(n) = T(n-1) + T(n-2) + c$

so run time grows like the Fibonacci numbers

BAD!



duplication!

iterative

```
f(0) ← 0
f(1) ← 1
for i = 2 .. n
    f(i) ← f(i-1) + f(i-2)
```

$O(n)$ (assuming numbers are small enough)

GOOD!

– an example of dynamic programming.

Main idea of dynamic programming:
solve "subproblems" from smaller to larger (bottom up) storing solutions

Run time:
(# subproblems) × (time to solve one subproblem)

Weighted Interval Scheduling

Recall Interval Scheduling aka Activity Selection:

Given a set of intervals I, find a max size subset of disjoint intervals

Weighted Interval Scheduling – Given I and weight $w(i)$ for each $i \in I$, find set $S \subseteq I$ s.t. no two intervals in S overlap and maximize $\sum_{i \in S} w(i)$

e.g. you have preferences for certain activities.

Recall greedy alg.

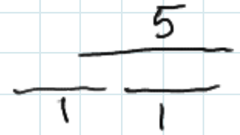
Order intervals $1, 2 \ldots, n$ by right endpt

recall

```
S ← ∅
for i = 1.. n
    if interval i is disjoint from those in S
    then S ← S ∪ {i}
end
```

This does not work for the weighted version

e.g.



What about greedy taking max weight first? No

Notation $OPT(I)$ – optimum set S

$w_{OPT}(I)$ – its weight

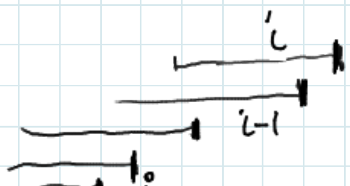
A general approach to finding OPT(I):
Consider one interval i. Either it is in OPT(I) or not
If $i \in$ OPT(I) then OPT(I) = $\{i\} \cup$ OPT(I')
I' = intervals disjoint from i
If $i \notin$ OPT(I) then OPT(I) = OPT($I - \{i\}$)
We want the max of these two possibilities

$$w_{OPT}(I) = \max\{ w_{OPT}(I - \{i\}),\ w(i) + w_{OPT}(I') \}$$

In general this does not give poly time

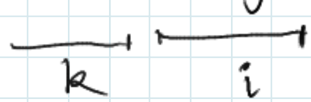
$$T(n) = 2T(n-1) + O(1) \quad \text{— exponential.}$$

Essentially, we may end up solving subproblems for each of the 2^n subsets of I.
However, if we order intervals $1..n$ by right endpt something nice happens



Intervals before i and disjoint from interval i are $1..j$ for some $j = p(i)$

Then $p(i)$ = largest index $k < i$
s.t. interval k is disjoint from interval i



[We'll see how to compute $p(i)$ soon]
Now subproblems can all be for subsets $1, .. i$
Let $M(i) = w_{OPT}(\{1, 2, .., i\})$

Then $M(i) = \max\{M(i-1),\ w(i) + M(p(i))\}$

How to compute $p(i)$:

we use sorted order $1 \cdots n$ by right endpoint

AND sorted order $l_1 \cdots l_n$ by left endpoint

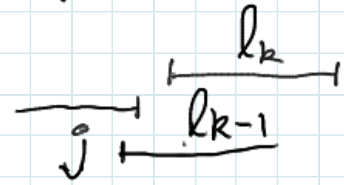
$j \leftarrow n$

for $k = n \cdots 1$

  while l_k overlaps j do $j \leftarrow j-1$

  $p(l_k) \leftarrow j$

end



Run time $\Theta(n)$ after sorting

Final algorithm

- sort intervals $1 \cdots n$ by right endpoint
- sort intervals by left endpoint
- compute $p(i)$ for all i

$M(0) = 0$

for $i = 1 \cdots n$

  $M(i) \leftarrow \max\{M(i-1),\ w(i) + M(p(i))\}$

end

$M[0 \cdots n]$ is an array we are filling in

final answer: $M(n)$

Runtime $\underbrace{\Theta(n \log n)}_{\text{sort}} + \underbrace{\Theta(n)}_{p(\cdot)} + O(n \cdot c)$

\#subproblems

time per subproblem

How to compute the actual subset

Recursive fn OPT(i)

  if $i = 0$ return ϕ

  else if $M(i-1) \geq w(i) + M(p(i))$

    return OPT($i-1$)

  else return $\{i\} \cup$ OPT($p(i)$)

end

Summary

- a general idea to find opt. subset: solve subproblems where one element is <u>in</u> or <u>out</u>
  Exponential in general; can sometimes be efficient
- key ideas of dynamic programming
  identify subproblems (not too many) and an order of solving them s.t. each subproblem can be solved by combining a few previously solved subproblems

Maximum Common Subsequence

Recall pattern matching from CS 240
Given a long string T and short string P
find occurrences of P in T
Useful in grep, find, etc.
Also useful: given two long strings find longest common subsequence

TARMAC
CATAMARAN

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

Given strings $x_1 \cdots x_m$
$y_1 \cdots y_n$

Let $M(i,j)$ = length of longest common subsequence of $x_1 \cdots x_i$ and $y_1 \cdots y_j$

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

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

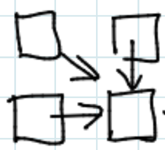
$$M(i,j) = \max \begin{cases} 1 + M(i-1, j-1) & \text{if } x_i = y_j \\ M(i-1, j) \\ M(i, j-1) \end{cases}$$

$M(i,0) = 0$
$M(0,j) = 0$

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					9
		∅	C	A	T	A	M	A	R	A	N
x	∅	0	0	0	0	0	- .				
1	T	0	$\boxed{0}$	$\boxed{0}$	$\boxed{1}$	$\boxed{0}$					
2	A	0	$\boxed{0}$	$\boxed{1}$	$\boxed{1}$	$\boxed{2}$					
3	R	0	$\boxed{0}$								
	M	0									
	A	.									
6	C	.									

M matrix



to fill this entry we look at 3 other entries

$\boxed{4}$ $M(6,9)$

Ex. Fill in the table.

Ex. Write out the pseudo-code

for $i = 0 .. m$ $M(i,0) \leftarrow 0$

for $j = 0 .. n$ $M(0,j) \leftarrow 0$

for $i = 1 .. m$

  for $j = 1 .. n$ } note: this was wrong in class.

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

Note that this is a correct ordering of i, j

In fact, if $x_i = y_j$ we can use the first choice (no need to check max of other two choices)

Ex. prove this.

Runtime: $O(\underbrace{n \cdot m}_{\text{\# subproblems}} \cdot \underbrace{c}_{\text{time to solve one subproblem}})$

\# subproblems

time to solve one subproblem
(compare 3 possibilities)

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

$OPT(i,j)$
- if $M(i,j) = M(i-1,j)$ then $OPT(i-1,j)$
- else if $M(i,j) = M(i,j-1)$ then $OPT(i,j-1)$
- else -- we must have matched i and j
  - output i,j
  - $OPT(i-1,j-1)$

OR we can record, when we fill $M(i,j)$, where the max came from

Next day: more sophisticated "edit" distance between strings.

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 = sort L.

<u>Claim</u> longest increasing subsequence of L = max common subsequence of L and S

So get $O(n^2)$ to find longest increasing subsequence (there is a more clever $O(n \log n)$ algorithm. — see wikipedia)