Dynamic Programming
0-1 Knapsack Problem

Bruce Nan

Optimization Problems

- For most, the best known algorithm runs in exponential time.
- Some have quick Greedy or Dynamic Programming algorithms.

What is Dynamic Programming?

- Dynamic programming solves *optimization problems* by combining solutions to subproblems

- "Programming" refers to a tabular method with a series of choices, not "coding"

What is Dynamic Programming?

- A set of choices must be made to arrive at an optimal solution
- As choices are made, subproblems of the same form arise frequently
- The key is to *store* the solutions of subproblems to be *reused* in the future

A Sequence of 3 Steps

- A dynamic programming approach consists of a sequence of 3 steps
  1. Characterize the structure of an optimal solution
  2. Recursively define the value of an optimal solution
  3. Compute the value of an optimal solution in a bottom-up fashion

Elements of Dynamic Programming

- For dynamic programming to be applicable, an optimization problem must have:

1. *Optimal substructure*
   – An optimal solution to the problem contains within it optimal solutions to subproblems (but this may also mean a greedy strategy applies)
2. *Overlapping subproblems*
   – The space of subproblems must be small; i.e., the same subproblems are encountered over and over

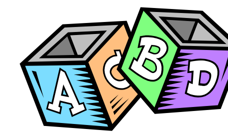
Elements of Dynamic Programming

- Dynamic programming uses optimal substructure from the bottom up:
  - *First* find optimal solutions to subproblems
  - *Then* choose which to use in optimal solution to problem.

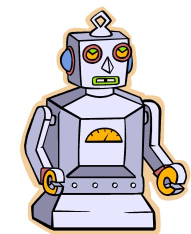
Knapsack Problem

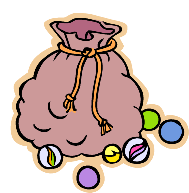
- 0-1 knapsack problem (easiest)
- Complete knapsack problem
- Multiple knapsack problem
- Mixed knapsack problem

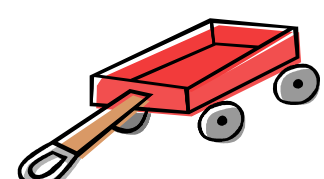
0-1 Knapsack Problem



Get as much value
as you can
into the knapsack







The (General) 0-1 Knapsack Problem

0-1 knapsack problem:

- n items.
- Item i is worth \$$v_i$, weighs w_i pounds.
- Find a most valuable subset of items with total weight $\leq W$.
- v_i, w_i and W are all integers.
- Have to either take an item or not take it - can’t take part of it.

Is there a greedy solution to this problem?

What are good greedy local choices?

- Select most valuable object?
- Select smallest object?
- Select object most valuable by weight?

Some example problem instances

Let $W =$ Capacity of knapsack = 10kg

Problem Instance 1:

$v_1 = \$60,\ w_1 = 6\text{kg}$

$v_2 = \$50,\ w_2 = 5\text{kg}$

$v_3 = \$50,\ w_3 = 5\text{kg}$

Problem Instance 2:

$v_1 = \$60,\ w_1 = 10\text{kg}$

$v_2 = \$50,\ w_2 = 9\text{kg}$

Problem Instance 3:

$v_1 = \$60,\ w_1 = 6\text{kg}$

$v_2 = \$40,\ w_2 = 5\text{kg}$

$v_3 = \$40,\ w_3 = 5\text{kg}$

- Select most valuable object?
- Select smallest object?
- Select object most valuable by weight?

All Fail!

Simplified 0-1 Knapsack Problem

- The general 0-1 knapsack problem cannot be solved by a greedy algorithm.
- What if we make the problem simpler:

Suppose $v_i = w_i$

- Can this simplified knapsack problem be solved by a greedy algorithm?
- No!

Some example problem instances

Let $W =$ Capacity of knapsack = 10kg

Problem Instance 1:

$v_1 = w_1 = 6$

$v_2 = w_2 = 5$

$v_3 = w_3 = 5$

Problem Instance 2:

$v_1 = w_1 = 10$

$v_2 = w_2 = 9$

- Select largest (most valuable) object?
- Select smallest object?

Both Fail!

Dynamic Programming Solution

- The General 0-1 Knapsack Problem can be solved by dynamic programming.

Let $W =$ capacity of knapsack (kg)

Let $(v_i, w_i) =$ value ($) and weight (kg) of item $i \in [1...n]$

Let $c[i,w] =$ value of optimal solution for knapsack of capacity w and items drawn from $[1...i]$

$$\text{Then } c\left[i,w\right] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c\left[i-1,\ w\right] & \text{if } w_i > w \\ \max\left(v_i + c[i-1,\ w-w_i], c[i-1,\ w]\right) & \text{if } i > 0 \text{ and } w_i \le w \end{cases}$$

Correctness

Let $W =$ capacity of knapsack (kg)

Let $(v_i, w_i) =$ value (\$) and weight (kg) of item $i \in [1...n]$

Let $c[i,w] =$ value of optimal solution for knapsack of capacity w and items drawn from $[1...i]$

$$\text{Then } c\left[i,w\right] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c\left[i-1,\ w\right] & \text{if } w_i > w \\ \max\left(v_i + c[i-1,\ w - w_i],\ c[i-1,\ w]\right) & \text{if } i > 0 \text{ and } w \geq w_i \end{cases}$$

Idea: $c[i-1, w] =$ value of optimal solution for capacity w and items drawn only from $[1...i-1]$

What happens when we are also allowed to consider item i?

Case 1. Optimal solution does not include item i.
Total value is the same as before.

Case 2. Optimal solution does include item i.
Total value is:

Value of item i

$+$ Value of optimal solution for remaining capacity of knapsack and allowable items

One of these must be true!

Bottom-Up Computation

Let $W =$ capacity of knapsack (kg)

Let $(v_i, w_i) =$ value (\$) and weight (kg) of item $i \in [1...n]$

Let $c[i,w] =$ value of optimal solution for knapsack of capacity w and items drawn from $[1...i]$

$$\text{Then } c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,\ w] & \text{if } w_i > w \\ \max\left(v_i + c[i-1,\ w - w_i], c[i-1,\ w]\right) & \text{if } i > 0 \text{ and } w \ge w_i \end{cases}$$

Need only ensure that $c[i-1, v]$ has been computed $\forall v \le w$

c[i,w]		*w*						
i	Allowed Items	*0*	*1*	*2*	…	*w*	…	*W*
0	*{}*							
1	*{1}*							
2	*{1 2}*							
…	…							
i	*{1 2 … i}*					*c[i,w]*		
…	…							
n	*{1 2 … n}*							

Example

Capacity $W = 6$

$$\text{Then } c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,\ w] & \text{if } w_i > w \\ \max\left(v_i + c[i-1,\ w - w_i], c[i-1,\ w]\right) & \text{if } i > 0 \text{ and } w \geq w_i \end{cases}$$

i	*v*	*w*
1	1	2
2	3	3
3	5	1
4	2	5
5	6	3
6	10	5

c[i,w]		*w*						
i	Allowed Items	*0*	*1*	*2*	*3*	*4*	*5*	*6*
0	*{}*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
1	*{1}*	*0*	*0*	*1*	*1*	*1*	*1*	*1*
2	*{1 2}*	*0*	*0*	*1*	*3*	*3*	*4*	*4*
3	*{1 2 3}*	*0*	*5*	*5*	*6*	*8*	*8*	*9*
4	*{1 2 3 4}*	*0*	*5*	*5*	*6*	*8*	*8*	*9*
5	*{1 2 3 4 5 }*	*0*	*5*	*5*	*6*	*11*	*11*	*12*
6	*{1 2 3 4 5 6 }*	*0*	*5*	*5*	*6*	*11*	*11*	*15*

Solving for the Items to Pack

$$\text{Then } c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,\ w] & \text{if } w_i > w \\ \max\left(v_i + c[i-1,\ w - w_i], c[i-1,\ w]\right) & \text{if } i > 0 \text{ and } w \geq w_i \end{cases}$$

i	*v*	*w*
1	*1*	*2*
2	*3*	*3*
3	*5*	*1*
4	*2*	*5*
5	*6*	*3*
6	*10*	*5*

c[i,w]		*w*						
i	Allowed Items	*0*	*1*	*2*	*3*	*4*	*5*	*6*
0	*{}*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
1	*{1}*	*0*	*0*	*1*	*1*	*1*	*1*	*1*
2	*{1 2}*	*0*	*0*	*1*	*3*	*3*	*4*	*4*
3	*{1 2 3}*	*0*	*5*	*5*	*6*	*8*	*8*	*9*
4	*{1 2 3 4}*	*0*	*5*	*5*	*6*	*8*	*8*	*9*
5	*{1 2 3 4 5}*	*0*	*5*	*5*	*6*	*11*	*11*	*12*
6	*{1 2 3 4 5 6}*	*0*	*5*	*5*	*6*	*11*	*11*	*15*

$i = n$

$w = W$

$\text{items} = \{\}$

loop for $i = n$ downto 1

  if $c[i,w] > c[i-1,w]$

    $\text{items} = \text{items} + \{i\}$

    $w = w - w_i$

Second Example

Capacity $W = 6$

$$\text{Then } c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,\ w] & \text{if } w_i > w \\ \max\left(v_i + c[i-1,\ w-w_i], c[i-1,\ w]\right) & \text{if } i > 0 \text{ and } w \ge w_i \end{cases}$$

i	v	w
1	1	2
2	4	3
3	2	1
4	5	4
5	4	3
6	2	3

c[i,w]		w						
i	Allowed Items	0	1	2	3	4	5	6
0	{}	0	0	0	0	0	0	0
1	{1}	0	0	1	1	1	1	1
2	{1 2}	0	0	1	4	4	5	5
3	{1 2 3}	0	2	2	4	6	6	7
4	{1 2 3 4}	0	2	2	4	6	7	7
5	{1 2 3 4 5 }	0	2	2	4	6	7	8
6	{1 2 3 4 5 6 }	0	2	2	4	6	7	8

Knapsack Problem: Running Time

- Running time $\Theta(n \times W)$. (cf. Making change $\Theta(d \times sum)$).
  - Not polynomial in input size!

Recall: Knapsack Problem Capacity $W = 6$

$$\text{Then } c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,\ w] & \text{if } w_i > w \\ \max(v_i + c[i-1,\ w - w_i], c[i-1,\ w]) & \text{if } i > 0 \text{ and } w \ge w_i \end{cases}$$

i	v	w
1	1	2
2	4	3
3	2	1
4	5	4
5	4	3
6	2	3

c[i,w]		w						
i	Allowed Items	0	1	2	3	4	5	6
0	{}	0	0	0	0	0	0	0
1	{1}	0	0	1	1	1	1	1
2	{1 2}	0	0	1	4	4	5	5
3	{1 2 3}	0	2	2	4	6	6	7
4	{1 2 3 4}	0	2	2	4	6	7	7
5	{1 2 3 4 5}	0	2	2	4	6	7	8
6	{1 2 3 4 5 6}	0	2	2	4	6	7	8

Observation from Last Day (Jonathon):
We could still implement this recurrence relation directly as a recursive program.

$$\text{Then } c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,\ w] & \text{if } w_i > w \\ \max\left(v_i + c[i-1,\ w-w_i], c[i-1,\ w]\right) & \text{if } i > 0 \text{ and } w \ge w_i \end{cases}$$

c[i,w]		*w*						
i	Allowed Items	*0*	*1*	*2*	*3*	*4*	*5*	*6*
0	*{}*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
1	*{1}*	*0*	*0*	*1*	*1*	*1*	*1*	*1*
2	*{1 2}*	*0*	*0*	*1*	*4*	*4*	*5*	*5*
3	*{1 2 3 }*	*0*	*2*	*2*	*4*	*6*	*6*	*7*
4	*{1 2 3 4 }*	*0*	*2*	*2*	*4*	*6*	*7*	*7*
5	*{1 2 3 4 5 }*	*0*	*2*	*2*	*4*	*6*	*7*	*8*
6	*{1 2 3 4 5 6 }*	*0*	*2*	*2*	*4*	*6*	*7*	*8*

Recall: Memoization in Optimization

- Remember the solutions for the subinstances

- If the same subinstance needs to be solved again, the same answer can be used.

Memoization

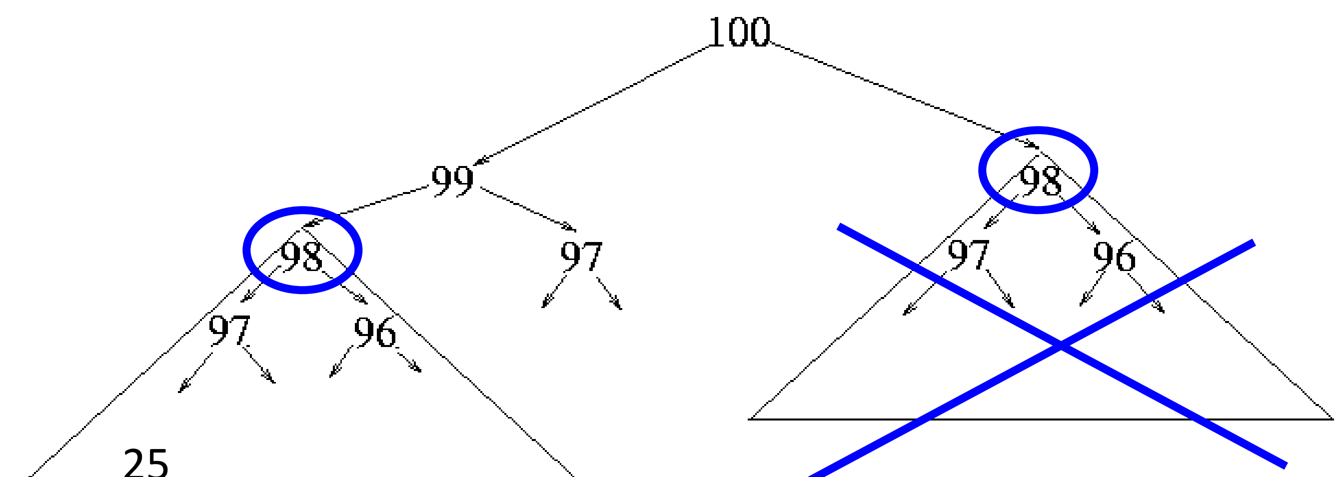
algorithm $Fib(n)$

$\langle \boldsymbol{pre-cond} \rangle$: n is a positive integer.

$\langle \boldsymbol{post-cond} \rangle$: The output is the n Fibonacci number.

Memoization reduces the complexity from exponential to linear!

```
begin
    ⟨saved, fib⟩ = Get(n)
    if( saved ) then
        result( fib )
    end if
    if( n = 0 or n = 1 ) then
        fib = n
    else
        fib = Fib(n − 1) + Fib(n − 2)
    end if
    Save(n, fib)
    result( fib )
end algorithm
```



From Memoization to Dynamic Programming

- Determine the set of subinstances that need to be solved.
- Instead of recursing from top to bottom, solve each of the required subinstances in smallest to largest order, storing results along the way.

backup

Example 1

- Fibonacci numbers are defined by:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \text{ for } i \geq 2.$$

Fibonacci Example

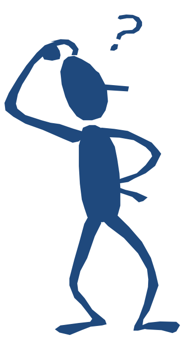
algorithm $Fib(n)$

$\langle \boldsymbol{pre-cond} \rangle$**:** n is a positive integer.

$\langle \boldsymbol{post-cond} \rangle$**:** The output is the n Fibonacci number.

```
begin
    if( n = 0 or n = 1 ) then
        result( n )
    else
        result( Fib(n − 1) + Fib(n − 2) )
    end if
end algorithm
```

Time?



Fibonacci Example

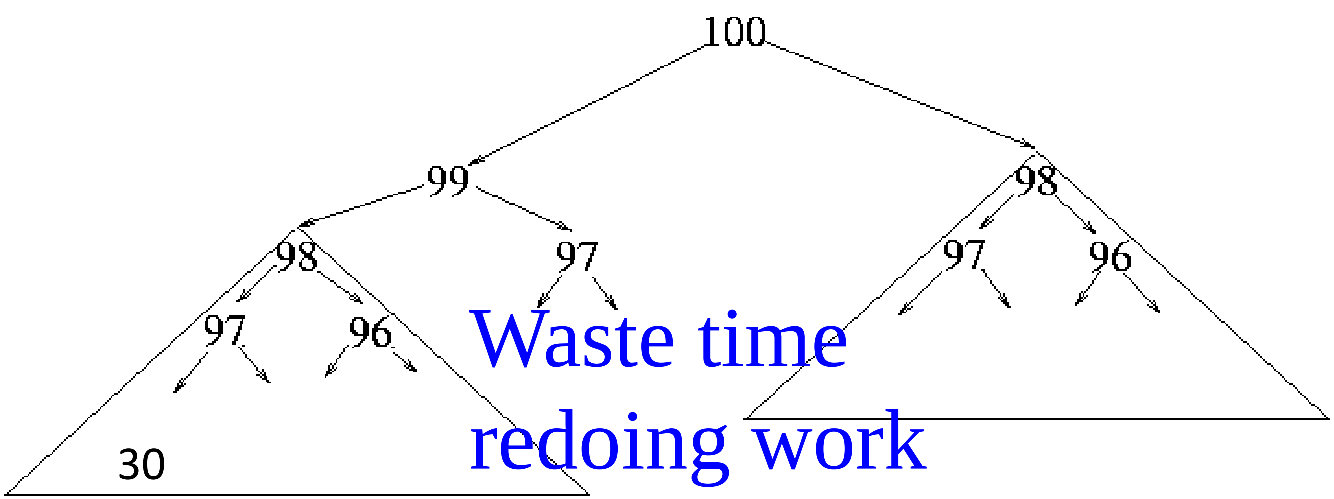
algorithm $Fib(n)$

$\langle \boldsymbol{pre-cond} \rangle$: n is a positive integer.

$\langle \boldsymbol{post-cond} \rangle$: The output is the n Fibonacci number.

```
begin
    if( n = 0 or n = 1 ) then
        result( n )
    else
        result( Fib(n − 1) + Fib(n − 2) )
    end if
end algorithm
```

Time:
Exponential



Waste time redoing work

Memoization

Definition: An algorithmic technique which saves (memoizes) a computed answer for later reuse, rather than recomputing the answer.

•Memo functions were invented by Professor Donald Michie of Edinburgh University.

•The idea was further developed by Robin Popplestone in his Pop2 language.

•This same principle is found at the hardware level in computer architectures which use a cache to store recently accessed memory locations.

Memoization in Optimization

- Remember the solutions for the subinstances
- If the same subinstance needs to be solved again, the same answer can be used.

Memoization

algorithm $Fib(n)$

$\langle$***pre* − *cond***$\rangle$**:** n is a positive integer.

$\langle$***post* − *cond***$\rangle$**:** The output is the n Fibonacci number.

begin

 $\langle saved, fib\rangle = Get(n)$

 if($saved$) then

  result(fib)

 end if

 if($n = 0$ or $n = 1$) then

  $fib = n$

 else

  $fib = Fib(n-1) + Fib(n-2)$

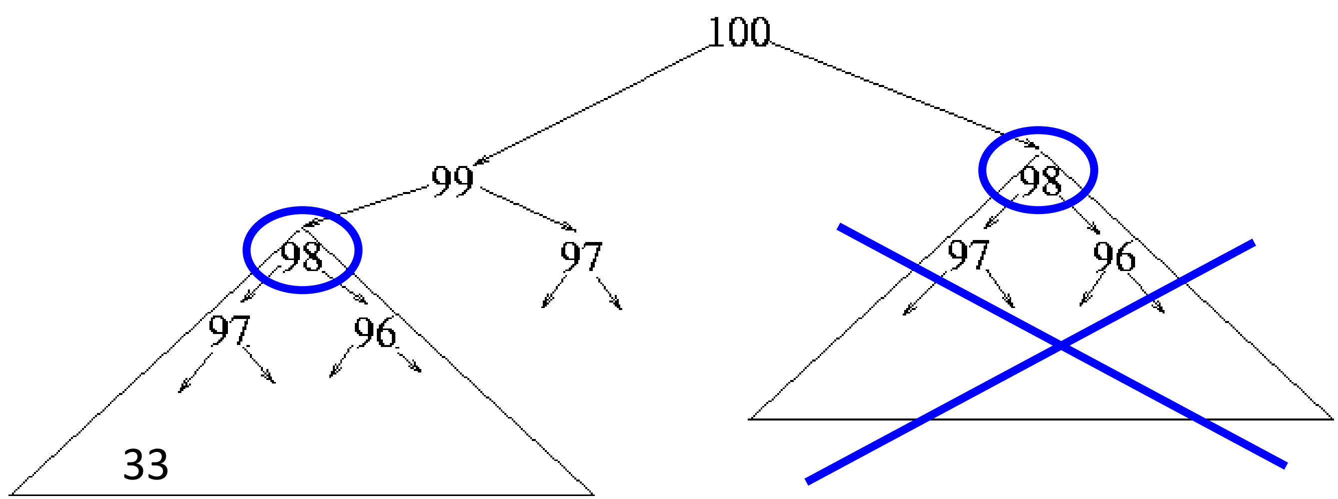
 end if

 $Save(n, fib)$

 result(fib)

end algorithm

Memoization reduces the complexity from exponential to linear!



From Memoization to Dynamic Programming

- Determine the set of subinstances that need to be solved.
- Instead of recursing from top to bottom, solve each of the required subinstances in smallest to largest order, storing results along the way.

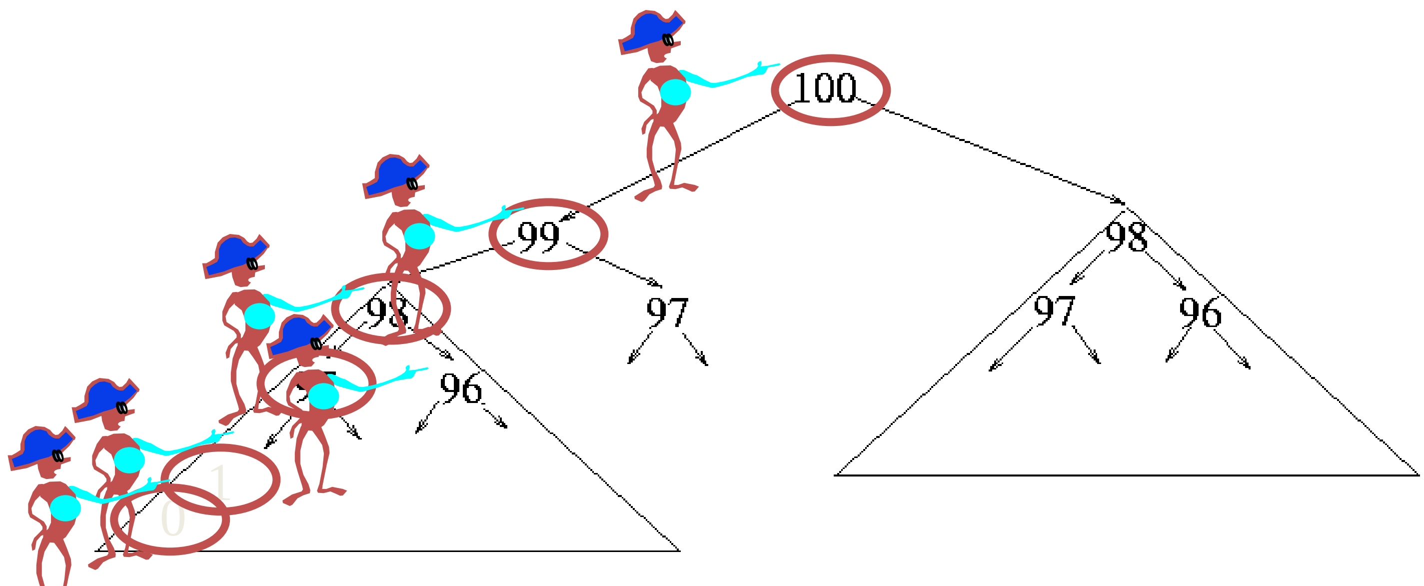
Dynamic Programming

First determine the complete set of subinstances

{100, 99, 98,…, 0}

Compute them in an order such that no friend must wait.

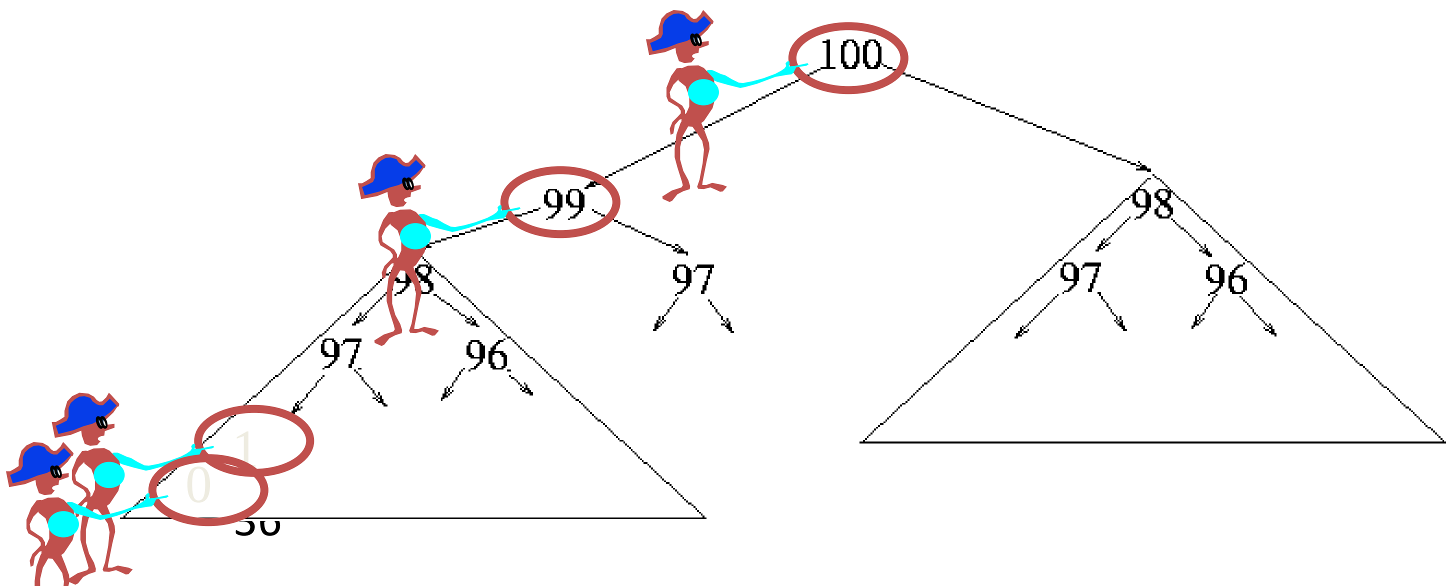
Smallest to largest



Dynamic Programming

Fill out a table containing
an optimal solution for each subinstance.

0	1	1	2	3	5			2.19×10^{20}	3.54×10^{20}
0,	1,	2,	3,	4,	5,	….		99,	100



Dynamic Programming

algorithm $Fib(n)$

$\langle \boldsymbol{pre-cond} \rangle$: n is a positive integer.

$\langle \boldsymbol{post-cond} \rangle$: The output is the n Fibonacci number.

```
begin
    table[0..n] fib
    fib[0] = 0
    fib[1] = 1
    loop i = 2..n
        fib[i] = fib[i − 1] + fib[i − 2]
    end loop
    result( fib[n] )
end algorithm
```

Time Complexity?

Linear!

Dynamic Programming vs Divide-and-Conquer

- Recall the divide-and-conquer approach
  - Partition the problem into independent subproblems
  - Solve the subproblems recursively
  - Combine solutions of subproblems
  - e.g., mergesort, quicksort

- This contrasts with the dynamic programming approach

Dynamic Programming vs Divide-and-Conquer

- Dynamic programming is applicable when *subproblems are not independent*
  - i.e., subproblems share subsubproblems
  - Solve every subsubproblem only once and store the answer for use when it reappears

- A divide-and-conquer approach will do more work than necessary