Dynamic Programming

Agenda

- Contextualization of Dynamic Programming
- Main features
    - Subproblem overlapping
    - Principle of optimality
- Approaches
    - Memoization (Top-Down)
    - Tab (Bottom-up)

Context
Dynamic programming

- It is a powerful algorithm design technique

Context
Dynamic programming

- It is a powerful algorithm design technique
- Two perspectives on PD:
  - DP ≈ "careful brute force"
  - Using intelligently, one can reduce "exponential" problems to polynomials

Context
Dynamic programming

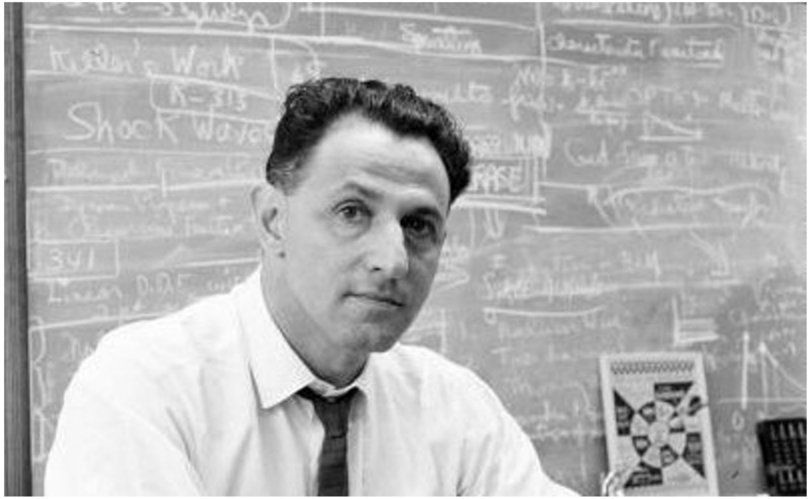
- It is a powerful algorithm design technique
- Two perspectives on PD:
  - DP ≈ "careful brute force”
  - Using intelligently, one can reduce "exponential" problems to polynomials
  - DP ≈ Recursion + "reuse"
  - We will be more precise throughout the class

Context
Dynamic programming

▷ Dynamic Programming?

Bellman, (1984) p. 159 explained that he invented the name “dynamic programming” to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who “had a pathological fear and hatred of the term, research.” He settled on “dynamic programming” because it would be difficult give it a “pejorative meaning” and because “It was something not even a Congressman could object to.

[John Rust 2006]
[https://editorialexpress.com/jrust/research/papers/dp.pdf]



Dr Richard Bellman

IEEE 1979 Medal

Contexto

Programação dinâmica

Dynamic Programming (DP)

- ▷ Dynamic Programming?

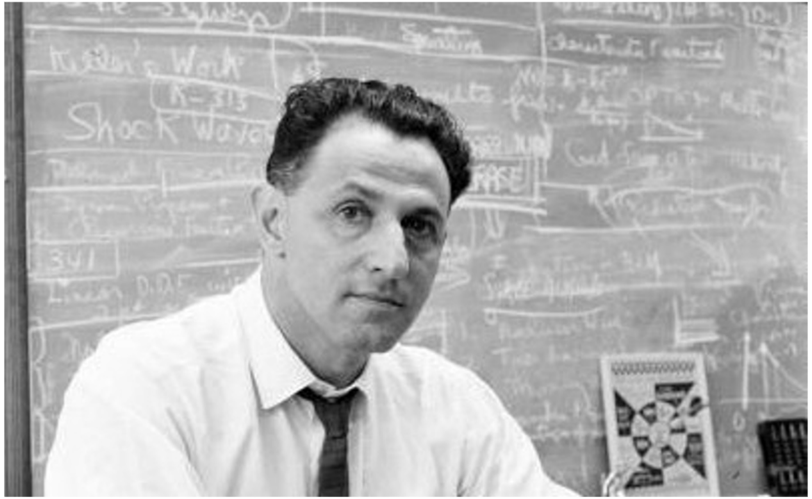
Something related to optimization

Something that won't give you problems

Bellman, (1984) p. 159 explained that he invented the name "dynamic programming" to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who "had a pathological fear and hatred of the term, research." He settled on "dynamic programming" because it would be difficult give it a "pejorative meaning" and because "It was something not even a Congressman could object to.

[John Rust 2006]

[https://editorialexpress.com/jrust/research/papers/dp.pdf]



Dr Richard Bellman

Dynamic programming

- Features
  - Overlapping problems (??)
  - Principle of optimality (??)

Fibonacci sequence

- Recurrence:
  - $F_n = F_{n-1} + F_{n-2}$
- Base case:
  - $F_1 = F_2 = 1$, or
  - $F_0 = F_1 = 1$

F_1	F_2	$\mathbf{F_3}$
1	1	**2**

Fibonacci sequence

- Recurrence:
  - $F_n = F_{n-1} + F_{n-2}$
- Base case:
  - $F_1 = F_2 = 1$, or
  - $F_0 = F_1 = 1$

F_1	F_2	F_3	$\mathbf{F_4}$
1	1	2	**3**

Fibonacci sequence

- Recurrence:
  - $F_n = F_{n-1} + F_{n-2}$
- Base case:
  - $F_1 = F_2 = 1$, or
  - $F_0 = F_1 = 1$

F_1	F_2	F_3	F_4	$\mathbf{F_5}$
1	1	2	3	**5**

Fibonacci sequence

- Recurrence:
  - $F_n = F_{n-1} + F_{n-2}$
- Base case:
  - $F_1 = F_2 = 1$, or
  - $F_0 = F_1 = 1$
- Goal:
  - Compute F_n

F_1	F_2	F_3	F_4	F_5	…	F_{n-2}	F_{n-1}	$\mathbf{F_n}$
1	1	2	3	5	…			

Fibonacci sequence
Naive solution

```
def fib(n):
    if n <= 2:
        f = 1
    else:
        f = fib(n-1) + fib(n-2)
    return f
```

Fibonacci sequence
Naive solution

```python
def fib(n):
  if n <= 2:
      f = 1
  else:
      f = fib(n-1) + fib(n-2)
  return f
```

Fibonacci sequence
Naive solution

```
def fib(n):
    if n <= 2:
        f = 1
    else:
        f = fib(n-1) + fib(n-2)
    return f
```

Fibonacci sequence
Naive solution

```
def fib(n):
    if n <= 2:
        f = 1
    else:
        f = fib(n-1) + fib(n-2)
    return f
```

- Does the algorithm work?
- Is it a good algorithm?

Fibonacci sequence
Naive solution

```
def fib(n):
    if n <= 2:
        f = 1
    else:
        f = fib(n-1) + fib(n-2)
    return f
```

- Does the algorithm work?
  - Yes!
- Is it a good algorithm?
  - No!
  - Exponential time!!!

Fibonacci sequence
Naive solution

```python
def fib(n):
    if n <= 2:
        f = 1
    else:
        f = fib(n-1) + fib(n-2)
    return f
```

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

Fibonacci sequence
Naive solution

```
def fib(n):
    if n <= 2:
        f = 1
    else:
        f = fib(n-1) + fib(n-2)
    return f
```

$$T(n) = T(n-1) + T(n-2) + O(1) \geq F_n \approx \varphi^n$$

Fibonacci sequence
Naive solution

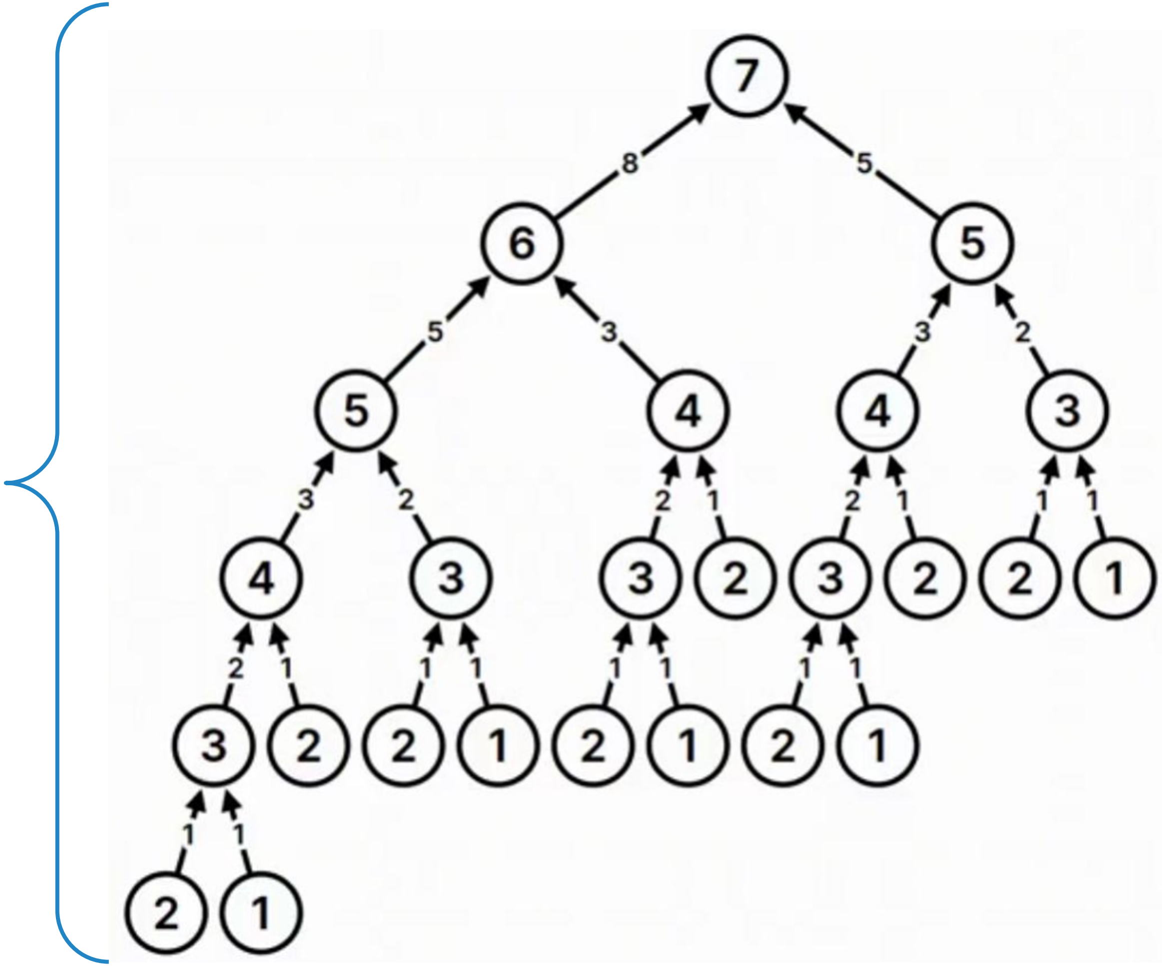
```python
def fib(n):
    if n <= 2:
        f = 1
    else:
        f = fib(n-1) + fib(n-2)
    return f
```

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + O(1) \geq F_n \approx \varphi^n \\
&\geq 2T(n-2) + O(1) \\
&\geq 2^{n/2}
\end{aligned}
$$

« ‹ › »

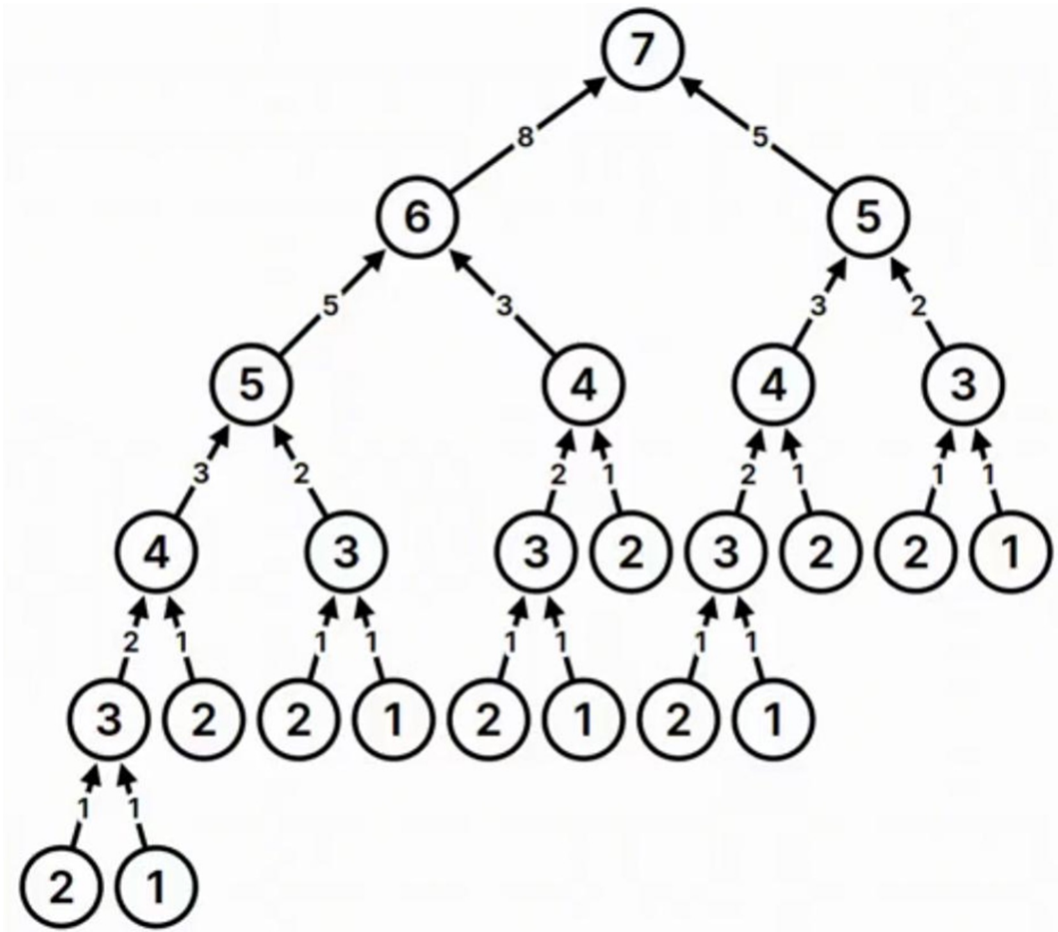
fn(7) starts running

Time ≈ # calls ≈ nodes



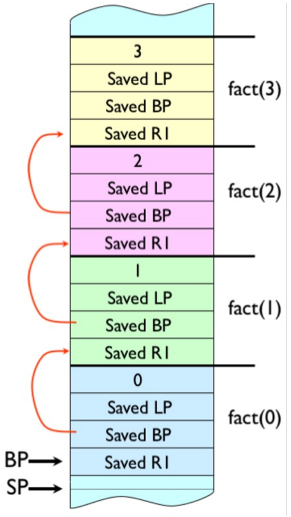
Time ≈ # calls ≈ nodes

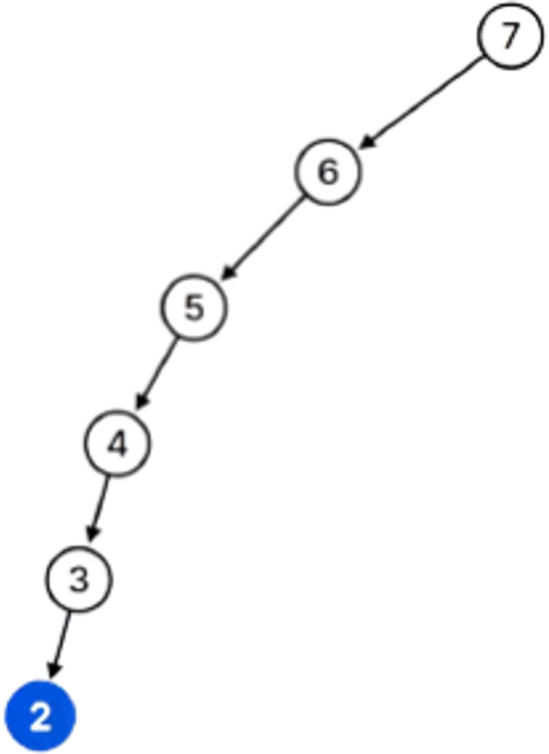
Space ≈ size of the longest path (root, leaf)



Fibonacci sequence
Naive solution

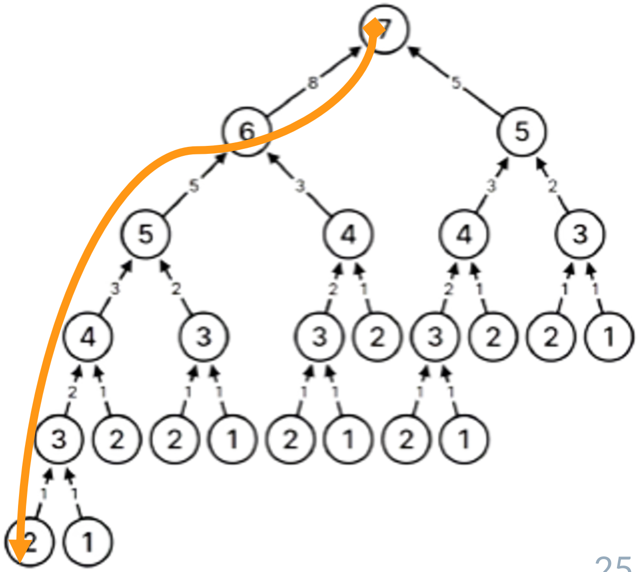
- We make n calls
- Calls are stored in the activation stack





Fibonacci sequence
Naive solution

```
fib(n):
    if n <= 2:
        f = 1
    else:
        f = fib(n-1) + f(n-2)
    return f
```



Fibonacci sequence
Naive solution

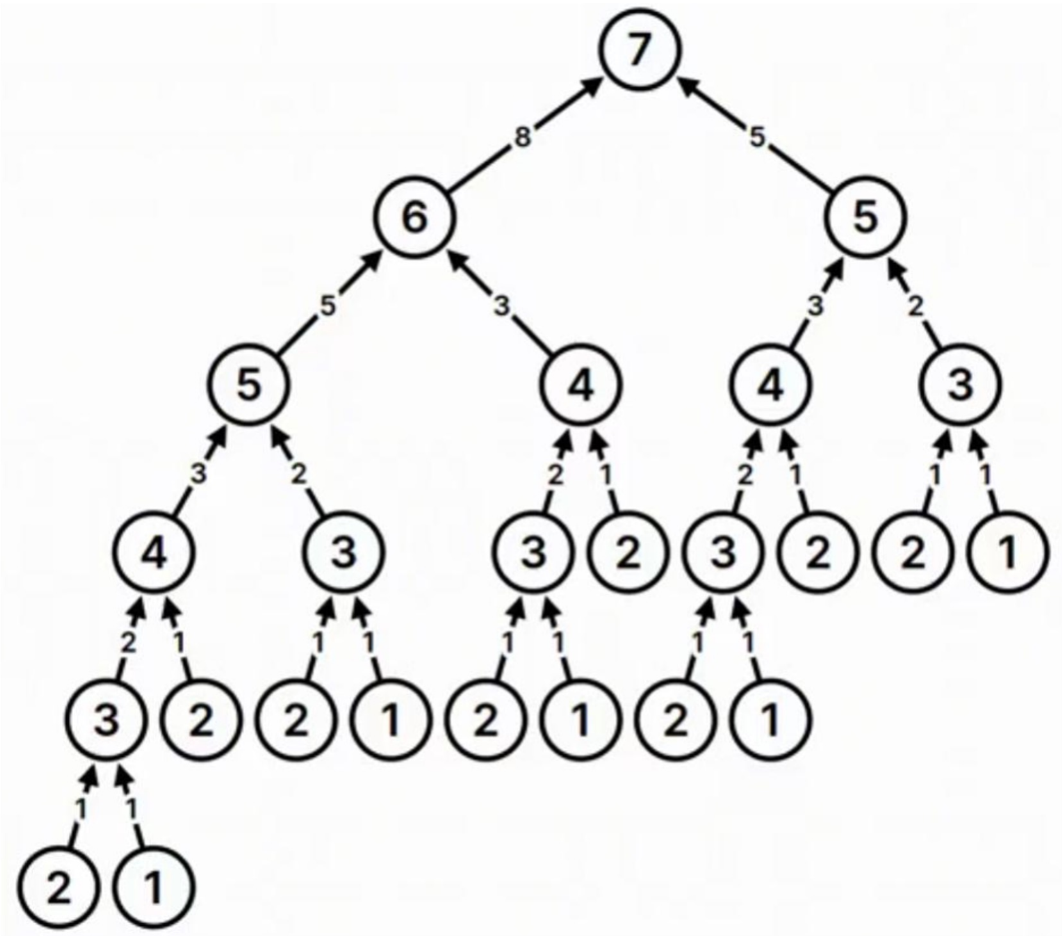
```
fib(n):
  if n <= 2:
      f = 1
  else:
      f = fib(n-1) + f(n-2)
  return f
```

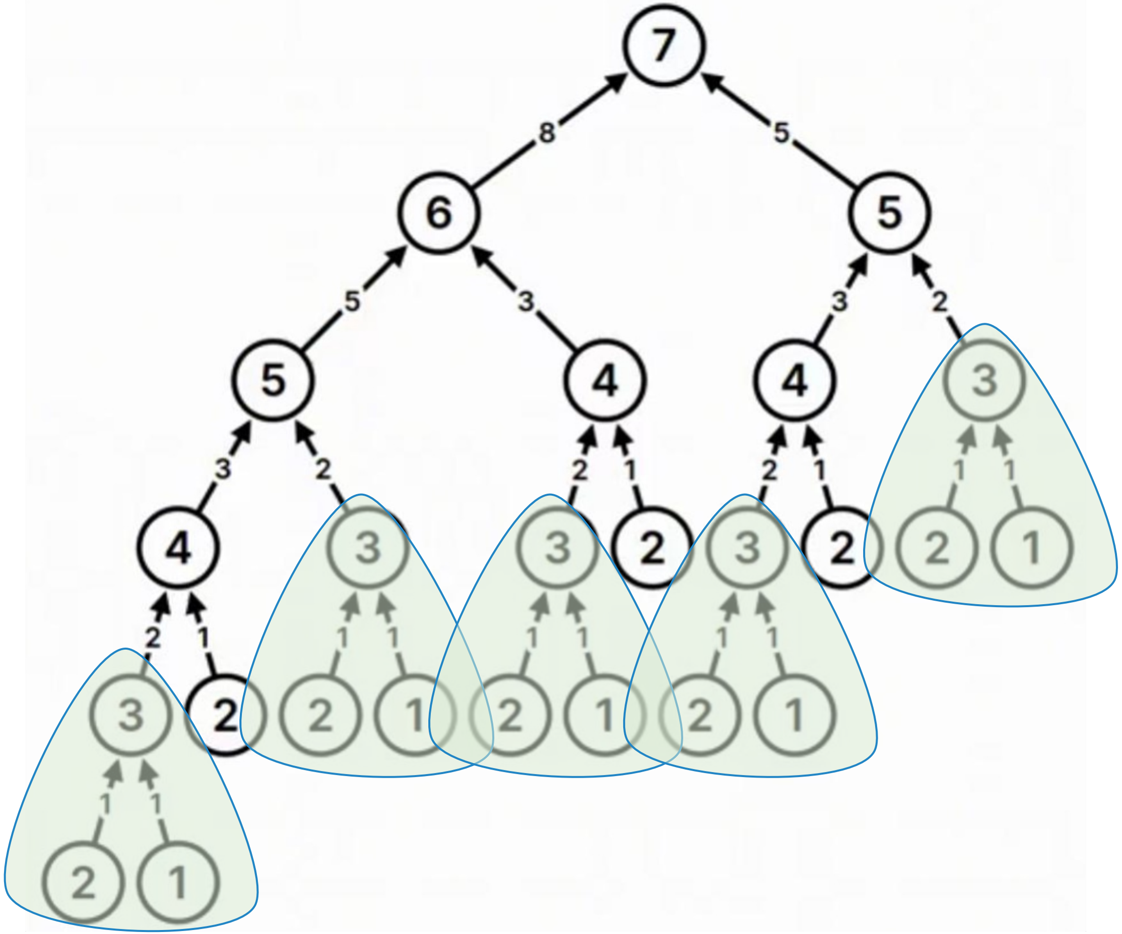
Time **$O(2^{n/2})$**

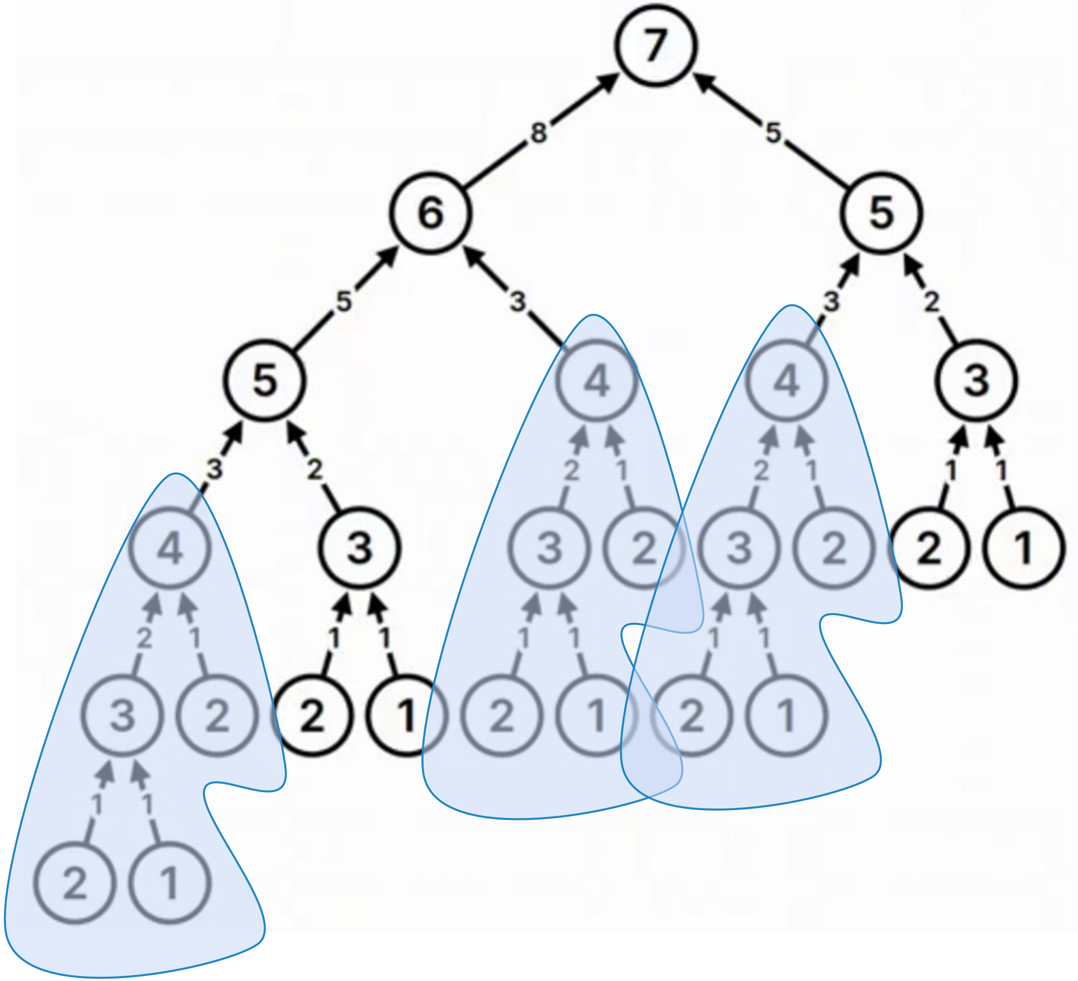
fib(50) ≈ **2^{50} steps**

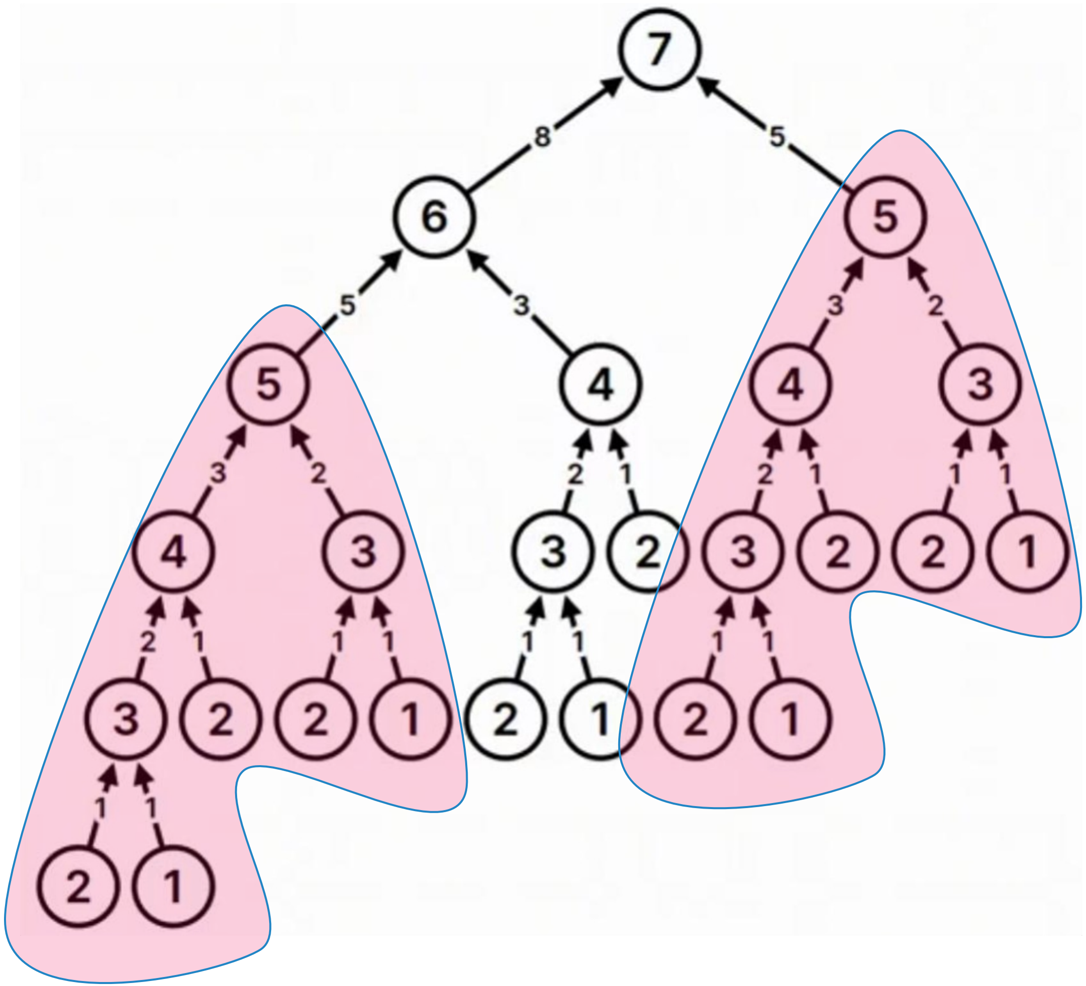
1.12e+15 =

1.125.899.906.842.624


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

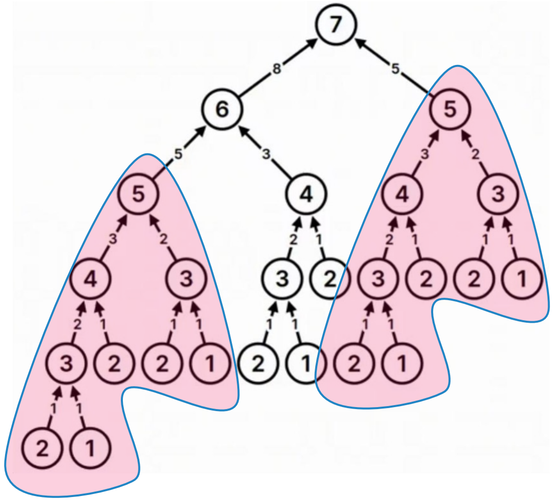

7
8
5
6
5
5
3
2
5
4
4
3
2
2
1
2
1
4
3
2
2
3
2
2
1
1
2
1
1
1
3
2
1
1
1
2
1
1
1
2
1
3
2
1
1
1
2
1

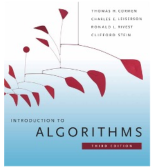




Dynamic programming

- Features
- Overlapping problems (✅)
- Principle of optimality (??)

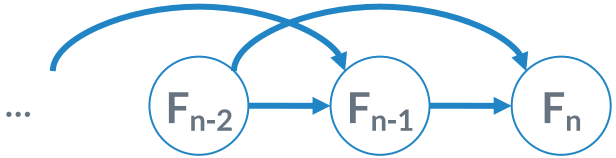




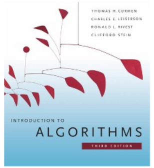
§15 e 27

The principle of optimality

- Optimal substructure:
  - "A problem has optimal substructure if the optimal solution can be built from optimal solutions to its subproblems."



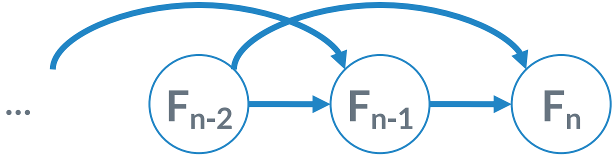
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.



§15 e 27

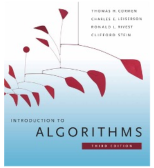
The principle of optimality

- Optimal substructure:
  - "A problem has optimal substructure if the optimal solution can be built from optimal solutions to its subproblems."
- In other words:
  - We can solve bigger problems using smaller instance solutions of the same problem!



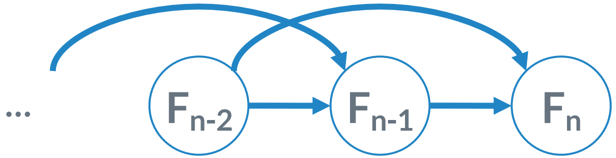
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.

The principle of optimality



§15 e 27

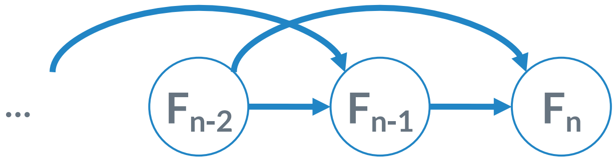
- Dependence on subproblems
  - Must form DAG (Directed Acyclic Graph)
  - If it has cycles, the PD algorithm can execute infinitely



Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.

Dynamic programming

- Features
- Overlapping problems (✅)
- Principle of optimality (✅)
- The dependencies of the subproblems must be acyclic (DAG!)



Pause and Reflect

Why?

Dynamic programming

- By using smartly one can reduce "exponential" problems to polynomials

How?

Prob. must have 2 characteristics

- *Overlapping problems (✅)*
- *Principle of optimality (✅)*

What?

Fibonacci sequence Problem

- $F_n = F_{n-1} + F_{n-2}$

Memoization
A dynamic programming technique

- Remember & reuse previously computed problem solutions

Memoization
A dynamic programming technique

- Remember & reuse previously computed problem solutions
  - Maintains a "dictionary"
  - Subproblems → solutions

```
memo {
 Subp₁: val₁,
 Subp₂: val₂,
  ...   : ...
 Subpₙ: valₙ
}
```

Memoization
A dynamic programming technique

- Remember & reuse previously computed problem solutions
  - Maintains a "dictionary"
  - Subproblems → solutions
- Recursive calls either:
  - Return a stored solution or
  - Compute and store a solution

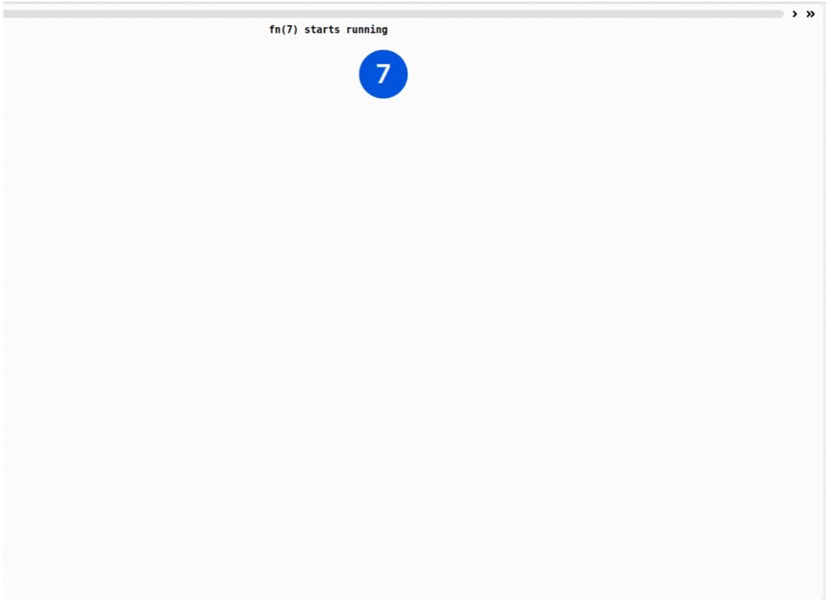
```
memo {
 Subp1: val1,
 Subp2: val2,
  ...    : ...
 Subpn: valn
}
```

Fibonacci sequence
Solution using Memoization

```
memo = {}
def fib(n):
    if n in memo: return memo[n]
    if n <= 2:
        f = 1
    else:
        f = fib(n-1) + f(n-2)
    memo[n] = f
    return f
```

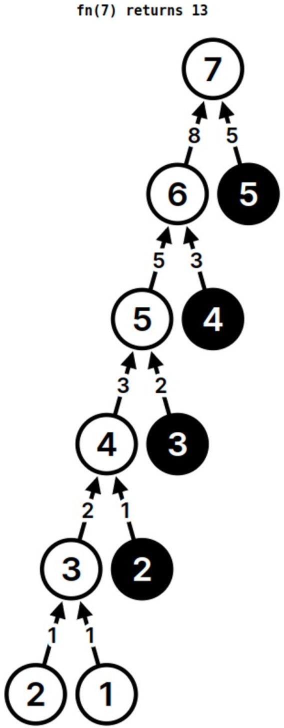
Fibonacci sequence
Solution using Memoization

```
memo = {}
def fib(n):
    if n in memo: return memo[n]
    if n <= 2:
        f = 1
    else:
        f = fib(n-1) + f(n-2)
    memo[n] = f
    return f
```



Fibonacci sequence
Solution using Memoization

```
memo = {}
def fib(n):
    if n in memo: return memo[n]
    if n <= 2:
        f = 1
    else:
        f = fib(n-1) + f(n-2)
    memo[n] = f
    return f
```



Fibonacci sequence
Solution using Memoization

```
memo = {}
def fib(n):
    if n in memo: return memo[n]
    if n <= 2:
        f = 1
    else:
        f = fib(n-1) + f(n-2)
    memo[n] = f
    return f
```

- Does fib(k) once for each k
- Runtime O(n)
    - Only n no 'memorized' calls
    - O(1) time per call
        - Ignore recursion

Memoization
A dynamic programming technique

- **The cost to compute each solution is paid only once**

Memoization
A dynamic programming technique

- **The cost to compute each solution is paid only once**
- **The cost of DP with memoization:**

$$Time \leq \sum_{subproblems} Nonrecursive\ work$$

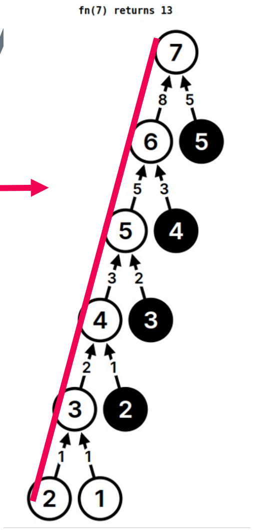
Memoization
A dynamic programming technique

- The cost to compute each solution is paid only once
- The cost of DP with memoization:
-

$$Time \le \sum_{subproblems} \text{Non-recursive work}$$

$$\le \boxed{\#\text{subproblems}} \times \boxed{\text{non-recursive work}}$$

O(1)



Memoization
A dynamic programming technique

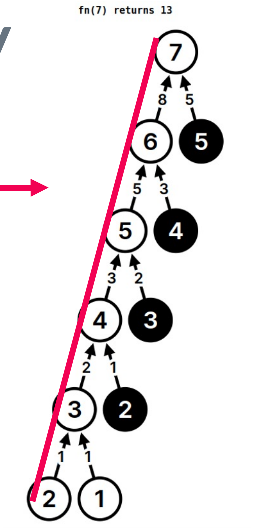
- The cost to compute each solution is paid only once
- The cost of DP with memoization:
-

$$Time \leq \sum_{subproblems} \text{Non-recursive work}$$

$$\leq \boxed{\#\text{subproblems}} \times \boxed{\text{non-recursive work}}$$

$$\leq \text{O(n)}$$

O(1)



Context
Dynamic programming

- Second perspective on PD:
  - DP ≈ Recursion + ”recycling"

Context
Dynamic programming

- Second perspective on PD:
  - DP ≈ Recursion + "reuse"
    - Memoization ("remind") & reuse solutions to subproblems that help solve the original problem

Bottom-Up
ANOTHER dynamic programming technique

```
def fib_botton_up(n):
  memo[0] = memo[1] = 1
  for i in range(2,n+1):
    memo[i] = memo[i-1] + memo[i-2]
  return memo[n]
```

F_1	F_2
1	1

Bottom-Up
ANOTHER dynamic programming technique

```
def fib_button_up(n):
  memo[0] = memo[1] = 1
  for i in range(2,n+1)
    memo[i] = memo[i-1] + memo[i-2]
  return memo[n]
```

F_1	F_2	$\mathbf{F_3}$
1	1	**2**

Bottom-Up
ANOTHER dynamic programming technique

```
def fib_button_up(n):
  memo[0] = memo[1] = 1
  for i in range(2,n+1)
    memo[i] = memo[i-1] + memo[i-2]
  return memo[n]
```

F_1	F_2	F_3	$\mathbf{F_4}$
1	1	2	**3**

Bottom-Up
ANOTHER dynamic programming technique

```
def fib_button_up(n):
  memo[0] = memo[1] = 1
  for i in range(2,n+1)
    memo[i] = memo[i-1] + memo[i-2]
  return memo[n]
```

F_1	F_2	F_3	F_4	$\mathbf{F_5}$
1	1	2	3	**5**

Bottom-Up
ANOTHER dynamic programming technique

```python
def fib_button_up(n):
  memo[0] = memo[1] = 1
  for i in range(2,n+1)
    memo[i] = memo[i-1] + memo[i-2]
  return memo[n]
```

F_1	F_2	F_3	F_4	F_5	…	F_{n-2}	F_{n-1}	$\mathbf{F_n}$
1	1	2	3	5	…			

Bottom-Up
ANOTHER dynamic programming technique

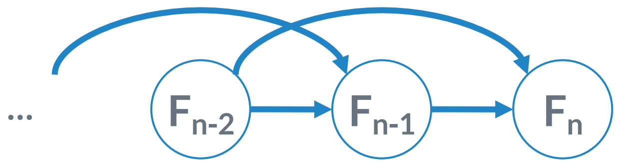
- Does the same computation as the memoized version

```
def fib_button_up(n):
  memo[0] = memo[1] = 1
  for i in range(2,n+1)
    memo[i] = memo[i-1] + memo[i-2]
  return memo[n]
```

Bottom-Up
ANOTHER dynamic programming technique

```
def fib_button_up(n):
  memo[0] = memo[1] = 1
  for i in range(2,n+1)
    memo[i] = memo[i-1] + memo[i-2]
  return memo[n]
```

- Does the same computation as the memoized version
- Topological ordering of subproblem dependencies (form a DAG!)



Bottom-Up
ANOTHER dynamic programming technique

```
def fib_button_up(n):
  memo[0] = memo[1] = 1
  for i in range(2,n+1)
    memo[i] = memo[i-1] + memo[i-2]
  return memo[n]
```

O(n)

- In practice it is faster
  - There is no recursion
- The analysis is more obvious

Bottom-Up
ANOTHER dynamic programming technique

```
def fib_button_up(n):
  memo[0] = memo[1] = 1
  for i in range(2,n+1):
    memo[i] = memo[i-1] + memo[i-2]
  return memo[n]
```

- In practice it is faster
  - There is no recursion
- The analysis is more obvious
- Can save space
  - We can remember only the last 2 fibs
    - Space O(1)

Bottom-Up
ANOTHER dynamic programming technique

```
def fib_button_up(n):
  memo[0] = memo[1] = 1
  for i in range(2,n+1)
    memo[i] = memo[i-1] + memo[i-2]
  return memo[n]
```

- In practice it is faster
  - There is no recursion
- The analysis is more obvious
- Can save space
  - We can remember only the last 2 fibs
    - Space O(1)

There is an implementation of the seq. Time cost Fibonacci O(lg n) via a different technique!

https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/

Generic algorithms
Top-Down and Bottom-Up

```python
memo = {}
def fib(n):
    if n in memo: return memo[n]
    if n <= 2:
        f = 1
    else:
        f = fib(n-1) + f(n-2)
    memo[n] = f
    return f
```

```python
memo = {}
def f(subprob):
    if subprob not in memo:
        Memo[subprob] = base or recurrence
    return memo[n]
```

Generic algorithms
Top-Down and Bottom-Up

```
def fib_button_up(n):
  memo[1] = momo[2] = 1
  for i in range(2,n+1)
    memo[i] = memo[i-1] + memo[i-2]
  return memo[n]
```

```
def f(subprob):
  Base case
  for subprob:
    memo[subprob] = REC relation.
  original return
```

Comparison between PD techniques: memoization (top-down) and tabulation (bottom-up)

	Tabulation (bottom-up)	Memoization (Top-Down)
Speed	Fast. Directly accesses dependent solutions directly from the table	Slow. Due to multiple recursive calls and returns
Solution for subprob.	If all subproblems must be solved at least once, DP using Bottom-up usually performs better than top-down DP	If not all subproblems in the subproblem space need to be solved, the solution using memoization has the advantage of solving only the necessary subproblems
Memo filling	Starts from the first entry. The other entries are filled in one by one.	The table is populated on demand, that is, not all entries are necessarily populated.
Code	It can become complex when you have multiple conditions	Typically less complicated and drawn directly from recurrence.

Algorithmic paradigms so far

Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

Recursive/ Divide-and-conquer. Break up a problem into *independent* subproblems, solve each subproblem, and combine solutions to form solution to original problem.

- subproblems are defined by their smaller size
- the input of the subproblems is not the same, only the size

Dynamic programming. Break problem into a series of *reusable* subproblems, and build up solutions to larger and larger subproblems.

- subproblems are defined both by size and content
  - the outcome of each subproblem (as specified by their input) is reused multiple times

Dynamic Programming Applications

Areas.

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ….

Some famous dynamic programming algorithms.

- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

6.4 Knapsack Problem

Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy: repeatedly add item with maximum ratio v_i / w_i.
Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

Dynamic Programming: False Start

Def. OPT(i) = max profit subset of items 1, …, i.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 }

- Case 2: OPT selects item i.
  - accepting item i does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before i, we don't even know if we have enough room for i

Conclusion. Need more sub-problems!

Dynamic Programming: Adding a New Variable

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

- Case 2: OPT selects item i.
  - new weight limit = w – w_i
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i) \,\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array.

```
Input: n, W, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Problem: Running Time

Running time. $\Theta(n\ W)$.
- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

Knapsack approximation algorithm. There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]