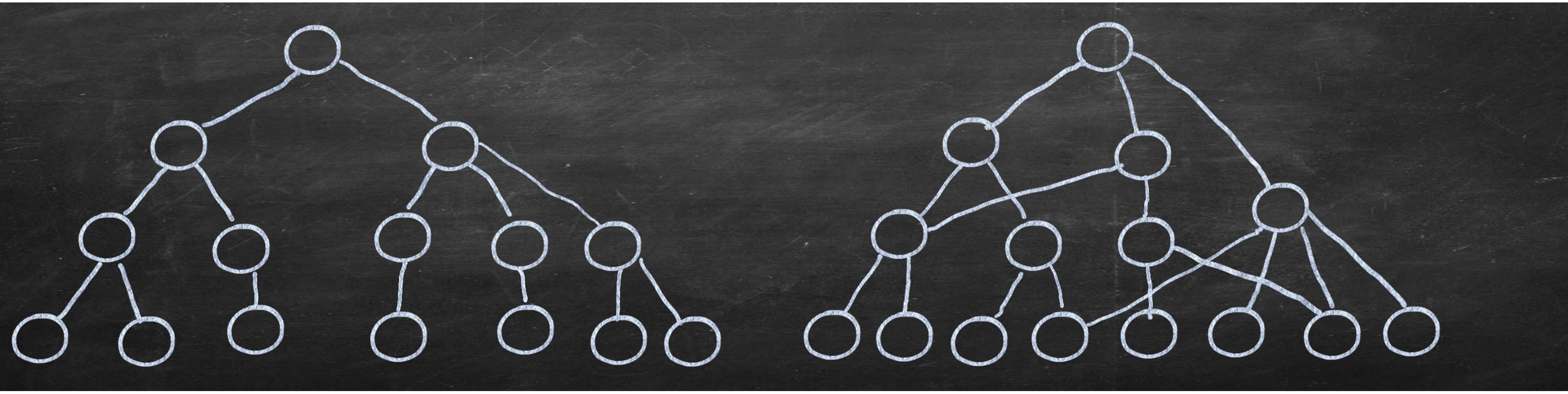


Dynamic Programming

Yannic Maus

Dynamic programming: Invented by Bellman ≈ 1950

Dynamic Programming is a very powerful algorithm design principle

invented by

Richard E. Bellman

Shortest path algorithm using dynamic programming named after him

Wikipedia on the name “dynamic programming”

> *I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, "Where did the name, dynamic programming, come from?" The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word "research". I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.*
>
> *—Richard Bellman, Eye of the Hurricane: An Autobiography (1984, page 159)*

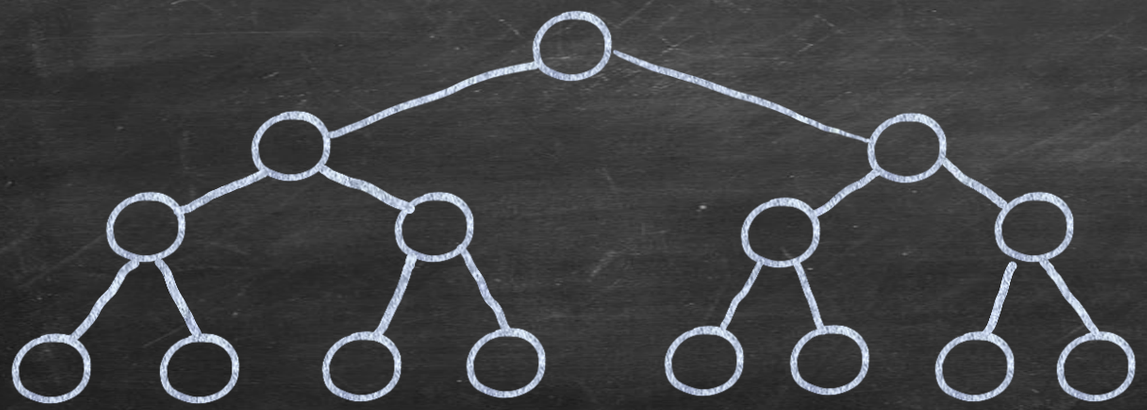
Recap: Divide & Conquer

Mergesort (divide & conquer)

MergeSort(A):

```
if length of A <= 1: return A
  middle = length of A / 2
  left = A[0 to middle - 1]; right = A[middle to end]
  left = MergeSort(left); right = MergeSort(right)
  return Merge(left, right)
```

recursion tree



base cases

Divide & Conquer:
Many non-overlapping subproblems

Runtime:
$T(n) = 2T(n/2) + O(n)$
→ Master Theorem $O(n \cdot \log n)$

Fibonacci numbers (toy problem for us)

Fibonacci numbers $F_n = F_{n-1} + F_{n-2}$ (recursive definition) $F_0 = F_1 = 1$ (base cases)

$F_2 = F_1 + F_0 = 2$

$F_3 = F_2 + F_1 = 3$

$F_4 = F_3 + F_2 = 5$

$F_5 = 8$

$F_6 = 13$

$F_7 = 21$

Fibonacci numbers grow very quickly (exponentially).

Fibonacci numbers (toy problem for us)

Fibonacci numbers $F_n = F_{n-1} + F_{n-2}$ (recursive definition) $F_0 = F_1 = 1$ (base cases)

Naïve recursion:

```
function fibonacci(n)
      if n <= 1 return 1
      return fibonacci(n-1) + fibonacci(n-2)
```

Recurrence relation:

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

$\geq F_n \approx \varphi^n$

exponential runtime

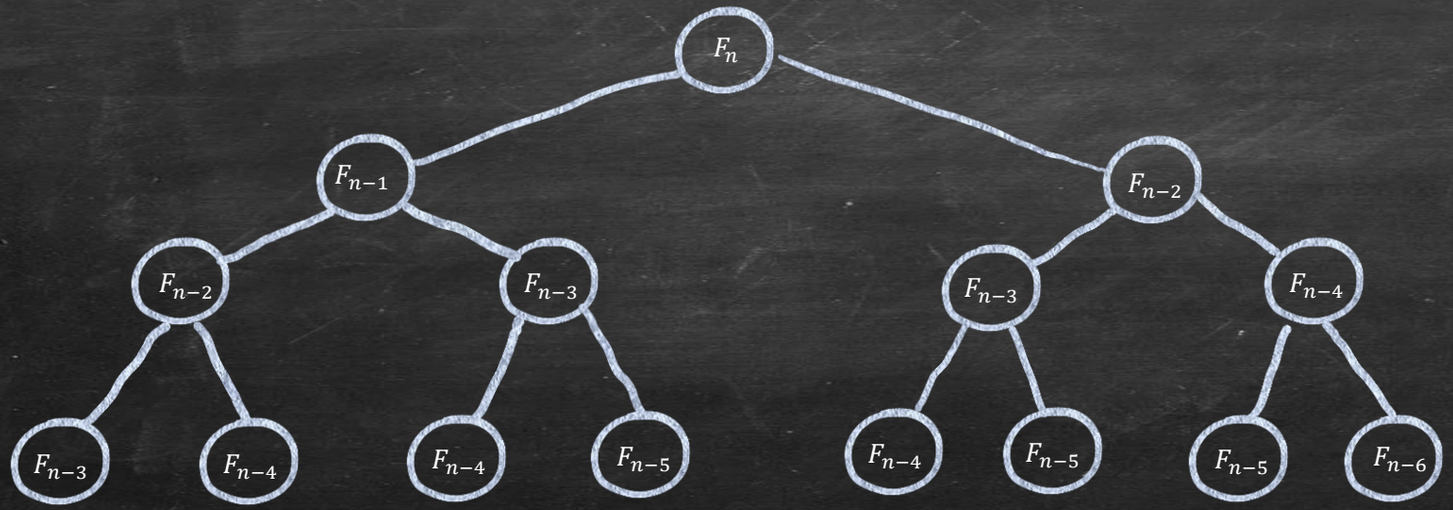
φ: golden ratio $\approx 1.618..$

Fibonacci numbers (toy problem for us)

Fibonacci numbers $F_n = F_{n-1} + F_{n-2}$ (recursive definition) $F_0 = F_1 = 1$ (base cases)

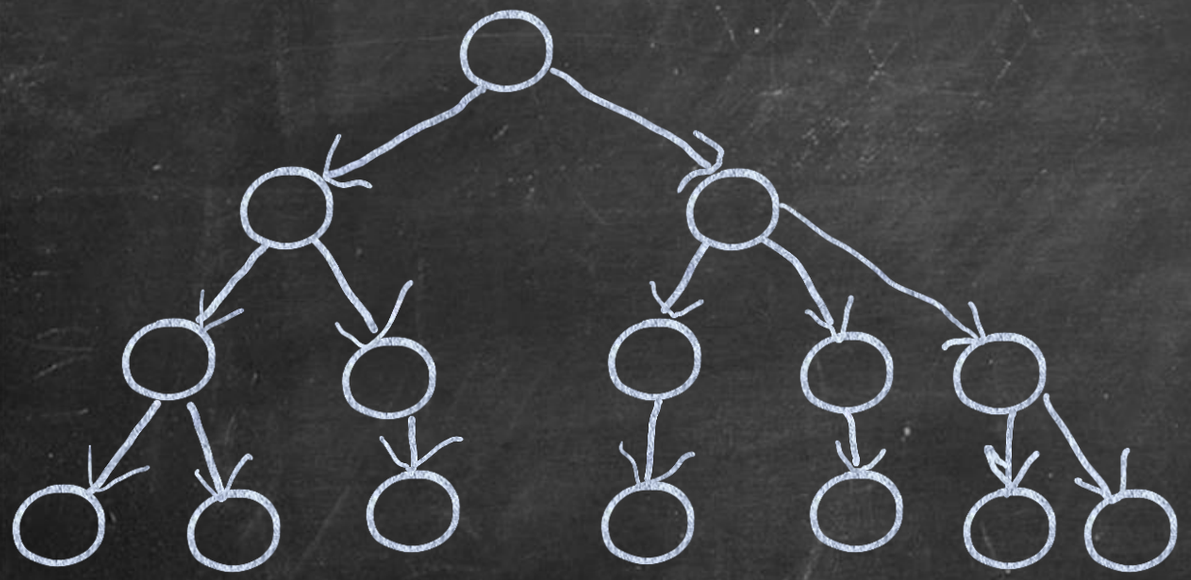
Naïve recursion:

```
function fibonacci(n)
      if n <= 1 return 1
      return fibonacci(n-1) + fibonacci(n-2)
```

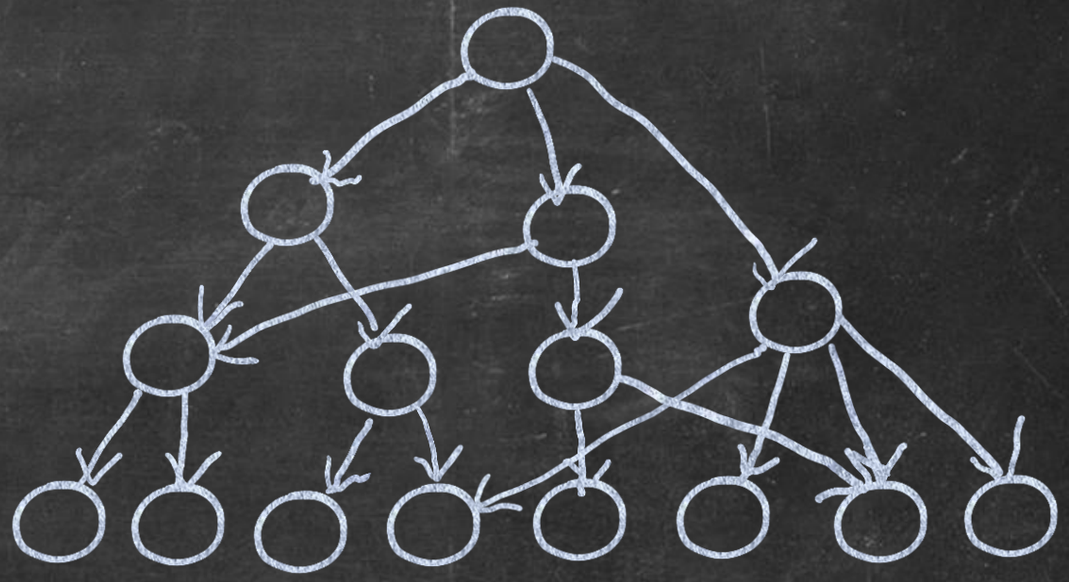


continue until base cases

Divide & Conquer



Independent subproblems



DAG (no cycles)

subproblems are used several times

→ Dynamic programming

Dynamic Programming

Memoization

Remember solutions to subproblems!

- Maintain a dictionary with solutions to subproblems
- **Recursive function:** If subproblem's solution is in dictionary or base case, **return it**,
  otherwise **compute (recursively) & store it**

DP = Recursion + Memoization

→ we only compute each subproblem once

Memoization: How to bound the runtime?

DP = Recursion + Memoization

→ we only compute each subproblem once

- **Subproblems**
- **Relate computation**
  How much time do we need to solve a SP knowing the solutions to other SPs in the recursion.

Runtime: $\sum_{S \text{ subproblem}}$ relate computation of S

(ignoring recursion overhead, e.g., stack etc.)

Fibonacci subproblems: F(0),F(1), … ,F(n)

Relate computation = adding two numbers

```
F(i)=F(i-1) + F(i-2)
```

Fibonacci numbers are large, use up to O(n) bits:

$O(n/w)$

Runtime:

- #subproblems $= n$

  ⇒ runtime $O\left(\#subproblems \cdot \frac{n}{w}\right) = O\left(\frac{n^2}{w}\right)$

assumption: wordsize w addition in $O(1)$ time

Dynamic programming

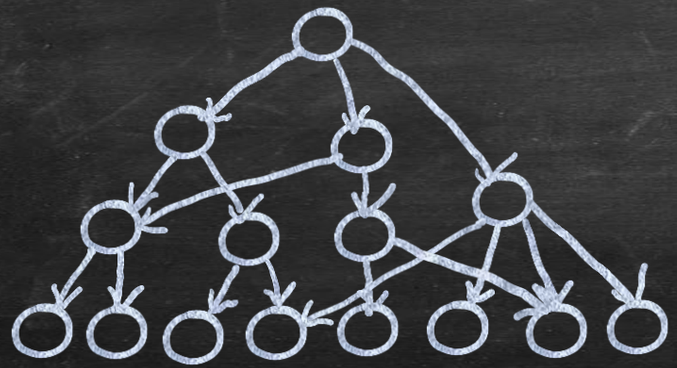
Dynamic Programming:

- **Subproblem definitions**
- **Original problem**
- **Base cases**
- **Recursive relation of subproblems**
- **Topological order**

Topological order:
*"ordering of the subproblems such that all subproblems of the recursive relation of subproblem S appear **before** S in the order"*

(all arrows go backwards)

recursive relation of subproblems provides you with a DAG (directed acyclic graph)



→ topological order exists

Fibonacci

Subproblems

$F(n)$ F_{n-2} F_{n-3} $F(2)$ $F(1)$ $F(0)$

Original Problem

$F(n)$

Base cases

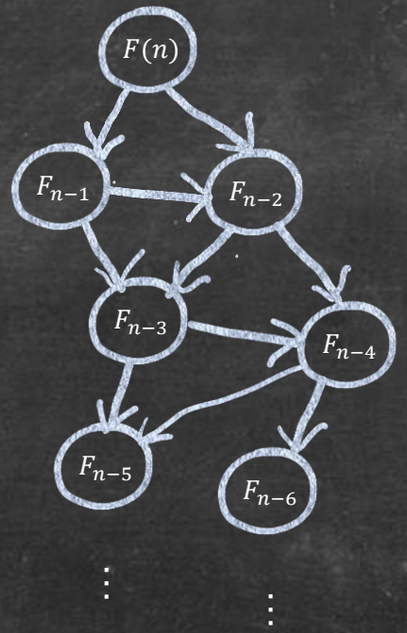
$F(0)$ $F(1)$

Recursive Relation

$F(i) = F(i-1) + F(i-2)$

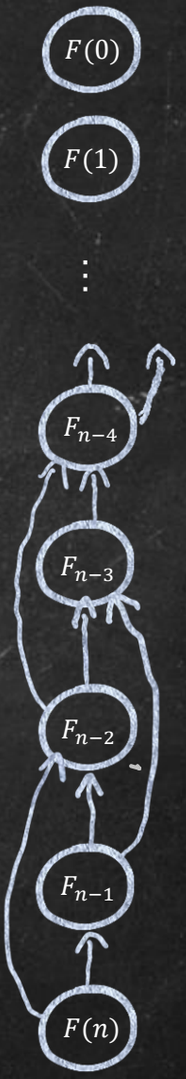
Formally: $F_i = F(i)$ (solution vs. subproblem)

Topological Ordering



DAG (no cycles)

"subproblems are used several times"



Topological order:

$F(0), F(1), F(2), F(3), \ldots, F(n)$

No recursion: Fibonacci iterative

You know: *Any recursive program can be written iteratively. For dynamic programming this is very helpful!*

Solve problems according to topological ordering

```
fib = new Array(n)
fib[0] = 1, fib[1] = 1 //base case
for i from 2 to n: //go through problems according to topological order
       fib[i] = fib[i-1] + fib[i-2]
return fib[n]
```

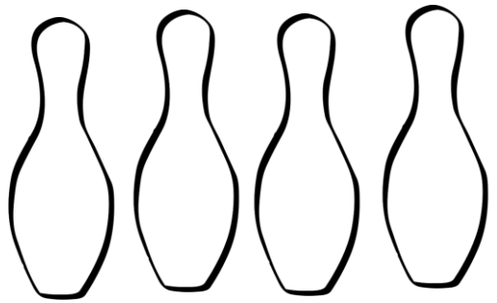
Runtime:

- Adding two numbers $O(n/w)$, $w =$ word size
- #subproblems $= n$

$\Rightarrow$ runtime $O\left(\#subproblems \cdot \frac{n}{w}\right) = O\left(\frac{n^2}{w}\right)$

Bowling

Toy problem for dynamic programming



*toy problem by Eric Demaine

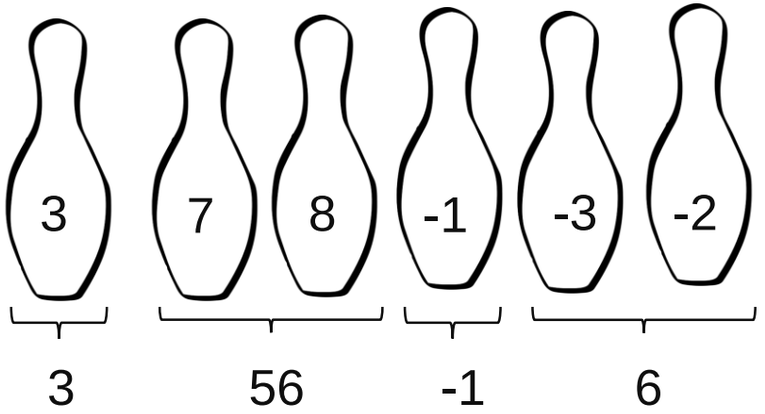
Bowling

Bowling (K):
Input:
sequence of $\boldsymbol{n}$ **bowling pins**, numbered with integers $\in [-K, K]$

You can either hit **single pins**, or **two adjacent pins**.
Throw as many balls as you want (you don't have to hit all pins)

Goal: Maximize your points

- hit a single pin with number x_i → x_i points
- hit two adjacent pins with numbers x and y → $x \cdot y$ points
- points of different balls add up



$= 65$ points

Question: How many points can you get?

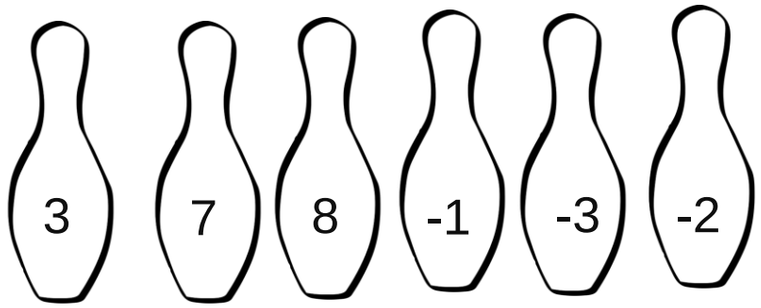
Exponentially (in n) many strategies

What are good subproblems?

You only want "few" subproblems

Optimally:
- polynomial number of subproblems
- polynomially fast relate computation

Good ideas for **"sequence problems" (of numbers/letters)**
- prefix subproblems: S[0,i] $O(n)$
- suffix subproblems, S[i,n] $O(n)$
- substring subproblems, S[i,j] $O(n^2)$



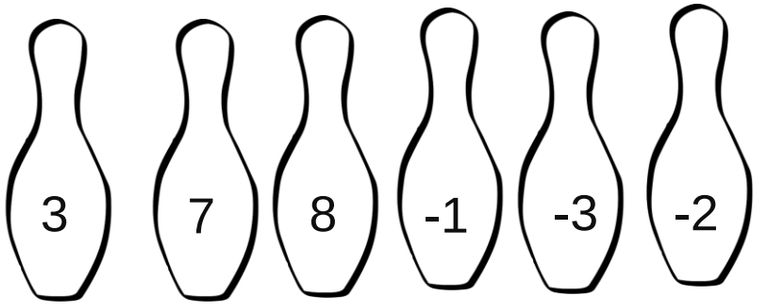
You do not want subsequence problems (too many exist).

Solution to the subproblem:
- optimal value ("think of points")
- solvable? Yes/no answer

Bowling as dynamic program

Input: sequence of n **bowling pins**, numb. w. integers $\in [-K, K]$
Goal: Maximize your points

- hit a single pin with number x_i → x_i points
- hit two adjacent pins with numbers x and y → $x \cdot y$ points
- points of different balls add up



- **Subproblems:** $S[i]$: Optimal number of points that we can get with the first i pins (prefix problem)
- **Original problem:** $S[n]$
- **Base case:** $S[0] = 0$ points (empty prefix)

Relation: if $i \geq 2$: $S[i] = \max\{S[i-1], S[i-1] + x_i, S[i-2] + x_{i-1} \cdot x_i\}$

if $i = 1$: $S[i] = \max\{S[i-1],\ S[i-1] + x_i\}$

- **Topological order:** S[0], S[1], S[2], …, S[n]

Pseudocode for Bowling

```
S[0]=0, S[1]=max{0, x_1}     //base case
FOR i=2 to n
      S[i]= max{S[i − 1], S[i − 1] + x_i, S[i − 2] + x_{i−1} · x_i}
Return S[n]
```

“for a single pin, we go through all options”

local bruteforce

Runtime: $\sum_{S}$ relate computation of $S = O(n) \cdot O(1) = O(n)$

subproblem

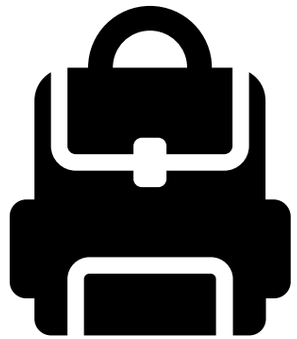
A linear time algorithm for the bowling problem!

Remark: This only computes the **value** of optimal points, **not a** bowling **strategy.**

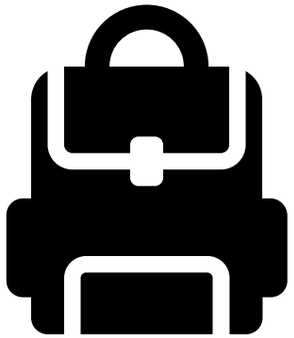
(more about how to get a strategy later)

Knapsack

German: “Rucksackproblem”



Knapsack



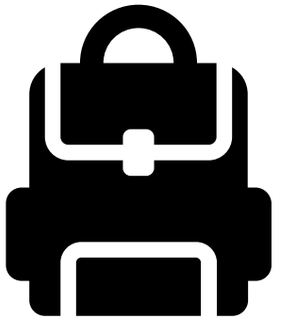
Knapsack
Input:
n items: i-th item has weight $\boldsymbol{w_i}$ and value $\boldsymbol{v_i}$
W: the weight capacity of the knapsack.

Question: What's the maximum value of items you can pack such that their weights' sum does not exceed W?

Knapsack

Knapsack
Input:
n items: i-th item has weight $\boldsymbol{w_i}$ and value $\boldsymbol{v_i}$
W: the weight capacity of the knapsack.

Question: What's the maximum value of items you can pack such that their weights' sum does not exceed W?



Example:

$W = 14$

$w_1 = 4, v_1 = 7$

$w_2 = 10, v_2 = 15$

$w_3 = 3\ , v_3 = 4$

$w_4 = 2\ , v_4 = 8$

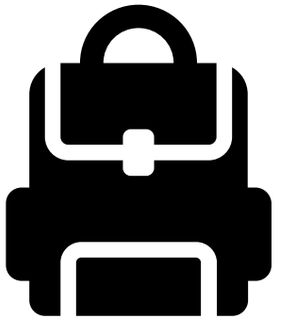
$w_4 = 6\ , v_4 = 11$

$\sum w_i = 14$ $\sum v_i = 22$

Knapsack

Knapsack
Input:
n items: i-th item has weight $\boldsymbol{w_i}$ and value $\boldsymbol{v_i}$
W: the weight capacity of the knapsack.

Question: What's the maximum value of items you can pack such that their weights' sum does not exceed W?



Example:

$W = 14$

$w_1 = 4, v_1 = 7$

$w_2 = 10, v_2 = 15$

$w_3 = 3\ , v_3 = 4$

$w_4 = 2\ , v_4 = 8$

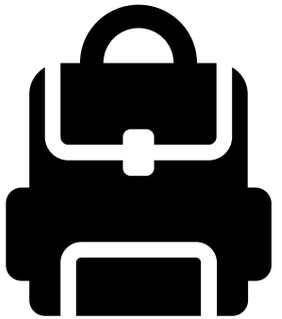
$w_4 = 6\ , v_4 = 11$

$\sum w_i = 9$ $\sum v_i = 19$

Knapsack

Knapsack
Input:
n items: i-th item has weight $\boldsymbol{w_i}$ and value $\boldsymbol{v_i}$
W: the weight capacity of the knapsack.

Question: What's the maximum value of items you can pack such that their weights' sum does not exceed W?



Example:

$W = 14$

$w_1 = 4, v_1 = 7$

$w_2 = 10, v_2 = 15$

$w_3 = 3\ , v_3 = 4$

$w_4 = 2\ , v_4 = 8$

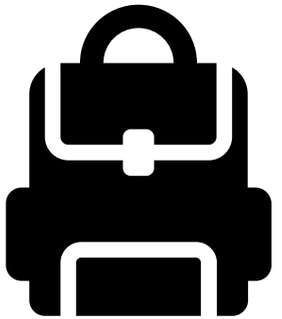
$w_4 = 6\ , v_4 = 11$

$\sum w_i = 11 \quad \sum v_i = 23$

Knapsack

Knapsack
Input:
n items: i-th item has weight $\boldsymbol{w_i}$ and value $\boldsymbol{v_i}$
W: the weight capacity of the knapsack.

Question: What's the maximum value of items you can pack such that their weights' sum does not exceed W?



Example:

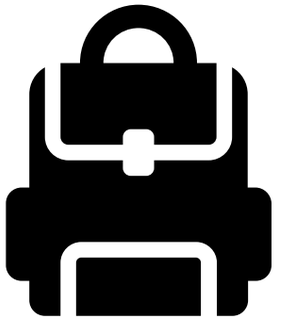
$W = 286$

Exercise: Greedy packing does not give an optimal solution.

Knapsack

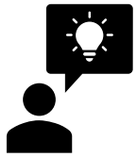
Knapsack
Input:
n items: i-th item has weight $\boldsymbol{w_i}$ and value $\boldsymbol{v_i}$
W: the weight capacity of the knapsack.

Question: What's the maximum value of items you can pack such that their weights' sum does not exceed W?



Dynamic Programming for knapsack (first try)

Which subproblems?

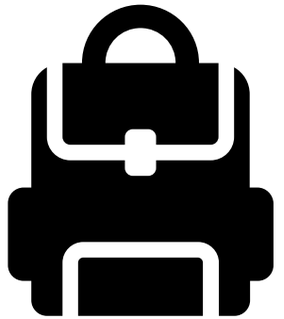


n items are like a sequence. Maybe prefix subproblems?

Knapsack

Knapsack
Input:
n items: i-th item has weight $\boldsymbol{w_i}$ and value $\boldsymbol{v_i}$
W: the weight capacity of the knapsack.

Question: What's the maximum value of items you can pack such that their weights' sum does not exceed W?



Dynamic Programming for knapsack (first try)

Subproblems: $S[i]$ prefix: best value we can get with first i items

Relate: Which choice do we have for the i-th item? Take it or leave it

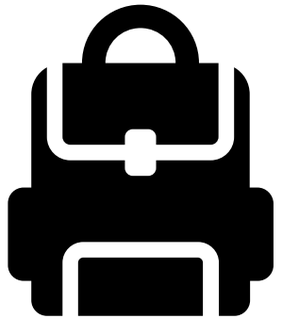
$$S[i] = \max\{S[i-1], S[i-1] + v_i\}$$

(does not take the weights into account)

Knapsack

Knapsack
Input:
n items: i-th item has weight $\boldsymbol{w_i}$ and value $\boldsymbol{v_i}$
W: the weight capacity of the knapsack.

Question: What's the maximum value of items you can pack such that their weights' sum does not exceed W?



Dynamic Programming for knapsack (~~first try~~)

Subproblems: $S[i, x]$, $0 \leq i \leq n$, $0 \leq x \leq W$:
best value we can get with first i items with "using" capacity $\leq x$

Original problem: $S[n, W]$ (all items, full capacity)

Base case: $S[0, x] = 0$ for $0 \leq x \leq W$; $S[i, 0] = 0$ for $0 \leq i \leq n$

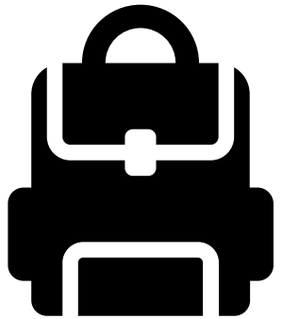
Relate: $S[i, x] =$ **if** $w_i \leq x$: $\max\{S[i-1, x], S[i-1, x-w_i] + v_i\}$ **else**: $S[i-1, x]$

Topological order: `For i=0 to n: For x=0 to W:` $S[i, x]$

Knapsack

Knapsack
Input:
n items: i-th item has weight $\boldsymbol{w_i}$ and value $\boldsymbol{v_i}$
W: the weight capacity of the knapsack.

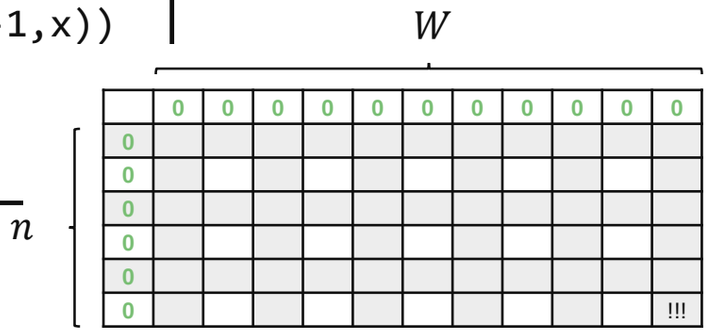
Question: What's the maximum value of items you can pack such that their weights' sum does not exceed W?



Pseudocode

```
For i from 0 to n:
For x from 0 to W:
        If i == 0 or x == 0: S(i,x) = 0
        else if: w_{i-1} <= x: S(i,x)= max(v_{i-1} + S(i-1,x - w_{i-1}), S(i-1,x))
        else: S(i,x) = S(i-1,x)

Return S(n,W)
```



Knapsack in Action

$W = 10$

W

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0										
2	0										
3	0										
4	0										
5	0										

$w_1 = 2, v_1 = 3$

$w_2 = 3, v_2 = 4$

$w_3 = 4\ , v_3 = 5$

$w_4 = 5\ , v_4 = 6$

$w_5 = 6\ , v_4 = 7$

Knapsack in Action

$W = 10$

$S[i,x] =$ **if** $w_i \leq x$: $\max\{S[i-1,x], S[i-1,x-w_i] + v_i\}$ **else**: $S[i-1,x]$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3	3
2	0										
3	0										
4	0										
5	0										

$w_1 = 2, v_1 = 3$

$w_2 = 3, v_2 = 4$

$w_3 = 4\ , v_3 = 5$

$w_4 = 5\ , v_4 = 6$

$w_5 = 6\ , v_4 = 7$

Knapsack in Action

$W = 10$

$S[i,x] =$ **if** $w_i \leq x$: $\max\{S[i-1,x], S[i-1,x-w_i] + v_i\}$ **else**: $S[i-1,x]$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7	7	7
3	0										
4	0										
5	0										

$w_1 = 2, v_1 = 3$

$w_2 = 3, v_2 = 4$

$w_3 = 4\ , v_3 = 5$

$w_4 = 5\ , v_4 = 6$

$w_5 = 6\ , v_4 = 7$

+4

+4

Knapsack in Action

$W = 10$

$S[i,x] =$ **if** $w_i \leq x$: $\max\{S[i-1,x], S[i-1,x-w_i] + v_i\}$ **else**: $S[i-1,x]$

$w_1 = 2, v_1 = 3$

$w_2 = 3, v_2 = 4$

$w_3 = 4\ , v_3 = 5$

$w_4 = 5\ , v_4 = 6$

$w_5 = 6\ , v_4 = 7$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7	7	7
3	0	0	3	4	5	7	8	9	9	12	12
4	0										
5	0										

+5 +5

Knapsack in Action

$W = 10$

$$S[i,x] = \textbf{if } w_i \leq x\textbf{: } \max\{\boldsymbol{S[i-1,x]}, \boldsymbol{S[i-1,x-w_i]} + \boldsymbol{v_i}\} \textbf{ else: } S[i-1,x]$$

$w_1 = 2, v_1 = 3$

$w_2 = 3, v_2 = 4$

$w_3 = 4\ , v_3 = 5$

$w_4 = 5\ , v_4 = 6$

$w_5 = 6\ , v_4 = 7$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7	7	7
3	0	0	3	4	5	7	8	9	9	12	12
4	0	0	3	4	5	7	8	9	10	11	13
5	0										

Knapsack in Action

$W = 10$

$S[i, x] =$ **if** $w_i \leq x$: $\max\{S[i-1, x], S[i-1, x-w_i] + v_i\}$ **else**: $S[i-1, x]$

$w_1 = 2, v_1 = 3$

$w_2 = 3, v_2 = 4$

$w_3 = 4\ , v_3 = 5$

$w_4 = 5\ , v_4 = 6$

$w_5 = 6\ , v_4 = 7$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7	7	7
3	0	0	3	4	5	7	8	9	9	12	12
4	0	0	3	4	5	7	8	9	10	11	13
5	0	0	3	4	5	7	8	9	10	11	13

Knapsack: Recovering a solution

$W = 10$

Optimal solution: item 1 item 2 item 4 $\sum v_i = 13$

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	add item 1		3	3	3	3	3	3	3	3	3
2	0	0	3		4	7	7	7	7	7	7
3	0	0	3	4	5	7	8	9	9	12	12
4	0	0	3	4	5	7	8			11	13
5	0	0	3	4	5	7	8	9	10	11	13

add item 2

add item 4

$w_1 = 2, v_1 = 3$

$w_2 = 3, v_2 = 4$

$w_3 = 4\ , v_3 = 5$

$w_4 = 5\ , v_4 = 6$

$w_5 = 6\ , v_4 = 7$

Knapsack: Recovering a solution

$W = 10$

Optimal solution: item 1 | item 2 | item 4 $\sum v_i = 13$

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	add item 1		3	3	3	3	3	3	3	3	3
2	0	0	3		4	7	7	7	7	7	7
3	0	0	3	4	5	7	8	9	9	12	12
4	0	0	3	4	5	7	8			11	13
5	0	0	3	4	5	7	8	9	10	11	13

add item 2

add item 4

$w_1 = 2, v_1 = 3$

$w_2 = 3, v_2 = 4$

$w_3 = 4\ , v_3 = 5$

$w_4 = 5\ , v_4 = 6$

$w_5 = 6\ , v_4 = 7$

Knapsack: Runtime

Pseudocode for knapsack

```
For i from 0 to n:
For x from 0 to W:
        If i == 0 or x == 0: S(i,x) = 0
        else if: w_{i-1} <= x: S(i,x)= max(v_{i-1} + S(i-1,x - w_{i-1}), S(i-1,x))
        else: S(i,x) = S(i-1,x)

Return S(n,W)
```

Runtime:
Relate computation: $O(1)$
#subproblems: $O(n \cdot W)$

Is this polynomial time?

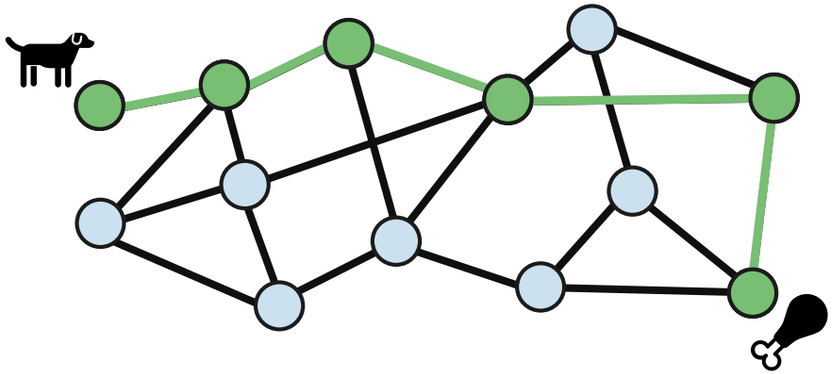
No. W can be exponential in the input.

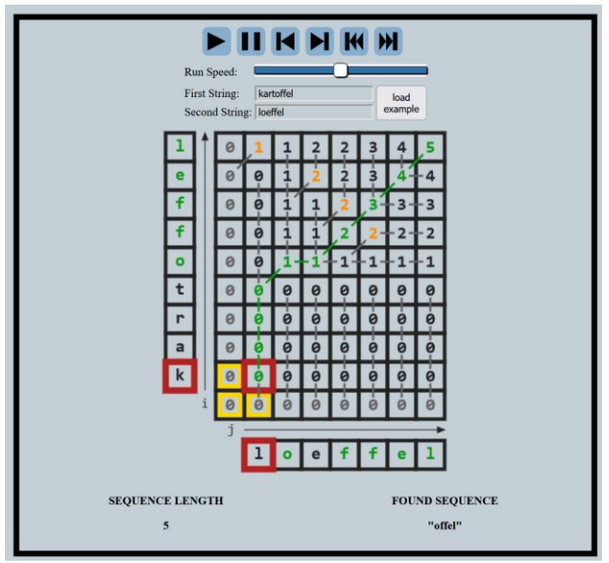
Knapsack is **NP-complete.**

[Karp '72]

Conclusion

- Dynamic programming is a very **powerful design principle**
  - It can be used to solve various problems: many knapsack variants, bin packing problems, shortest paths, many string algorithms, matrix multiplication orders, …





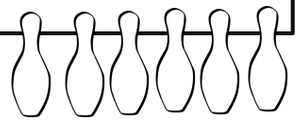
http://animalgo.ist.tugraz.at/

A few tips

A few tips for designing dynamic progamming algorithms:

- Subproblems, recursive relation, base cases, original problems, topological order → pseudocode comes for free
- **Difficult part:** Which subproblems should I use? What is their recursive relation?
    - tips for subproblems: Suffixes, Prefixes, Substrings. Which question are you interested in?
        - Optimal number of points?
        - Is this subproblem solvable at all? Yes/No
    - tips for recursive relation:
        - Ask a question, e.g., which choice do I have with the i-th item?
- Pseudocode is often just a few lines
- Arranging your subproblems in a table is often helpful
- Recovering a solution: remember how you got to your optimal value

```
S[0]=0, S[1]=max{0, x_1}    //base case
FOR i=2 to n
      S[i]= max{S[i − 1], S[i − 1] + x_i, S[i − 2] + x_{i−1} · x_i}
Return S[n]
```



	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7	7	7
3	0	0	3	4	5	7	8	9	9	12	12
4	0	0	3	4	5	7	8	9	10	11	13
5	0	0	3	4	5	7	8	9	10	11	13