# Dynamic Programming I

# Fibonacci Numbers

Defined recursively by

$$f(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f(n-1) + f(n-2), & n \geq 2. \end{cases}$$

Problem: given k, compute f(k)

- (0 <= k <= 1,000,000)

$$f(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f(n-1) + f(n-2), & n \geq 2. \end{cases}$$

# Fibonacci Numbers

There are **at least** six different solutions.

Let’s code up the most naïve one.

$$f(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f(n-1) + f(n-2), & n \geq 2. \end{cases}$$

# Fibonacci Numbers

There are **at least** six different solutions.

Let’s code up the most naïve one.

```
int f(int n):
  if(n==0) return 0;
  if(n==1) return 1;
  return f(n-1) + f(n-2);
```

$$f(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f(n-1) + f(n-2), & n \geq 2. \end{cases}$$

# Fibonacci Numbers

There are **at least** six different solutions.

Let’s code up the most naïve one.

```
int f(int n):
  if(n==0) return 0;
  if(n==1) return 1;
  return f(n-1) + f(n-2);
```

what's the bug?

# Recursive Fibonacci Solution

Several issues with this solution:

1. *Stack overflow if n too large*

# Recursive Fibonacci Solution

Several issues with this solution:

1. *Stack overflow if n too large*

2. What’s the runtime?

# Recursive Fibonacci Solution

Several issues with this solution:

1. *Stack overflow if n too large*

2. What's the runtime? $O(2^n)$

# Recursive Fibonacci Solution

**Key Idea**:

why recompute f(k) a bunch of times?

store and reuse previous values.

# Recursive Fibonacci Solution

**Key Idea**:

why recompute f(k) a bunch of times?

store and reuse previous values.

```
int f(int n, hashset h):
  if(n == 0) return 0;
  if(n == 1) return 1;
  if(h.contains(n)) return h[n];
  return h[n] = f(n-1,h) + f(n-2,h);
```

# Recursive Fibonacci Solution

**Key Idea**:

why recompute f(k) a bunch of times?

store and reuse previous values.

What’s the running time?

What’s the space usage?

# Recursive Fibonacci Solution

**Key Idea**:

why recompute f(k) a bunch of times?

store and reuse previous values.

What's the running time?

What's the space usage? $O(n)$

$O(n)$

Called **memoization**: trade space for time

# Yet Another Solution

Iteratively **build a table** of partial sols:

| $f(0)$ | $f(1)$ | $f(2)$ | $f(3)$ | $f(4)$ | $f(5)$ |
|---|---|---|---|---|---|
| $0$ | $1$ | | | | |

# Yet Another Solution

Iteratively **build a table** of partial sols:

| $f(0)$ | $f(1)$ | $f(2)$ | $f(3)$ | $f(4)$ | $f(5)$ |
|---|---|---|---|---|---|
| 0 | 1 | | | | |

we have accumulated just enough information to compute this

# Yet Another Solution

Iteratively **build a table** of partial sols:

| $f(0)$ | $f(1)$ | $f(2)$ | $f(3)$ | $f(4)$ | $f(5)$ |
|---|---|---|---|---|---|
| $0$ | $1$ | $1$ | | | |

we have accumulated just enough
information to compute this

# Yet Another Solution

Iteratively **build a table** of partial sols:

| $f(0)$ | $f(1)$ | $f(2)$ | $f(3)$ | $f(4)$ | $f(5)$ |
|---|---|---|---|---|---|
| 0 | 1 | 1 | | | |

now "slide window right" and compute this

# Yet Another Solution

Iteratively **build a table** of partial sols:

| $f(0)$ | $f(1)$ | $f(2)$ | $f(3)$ | $f(4)$ | $f(5)$ |
|---|---|---|---|---|---|
| $0$ | $1$ | $1$ | $2$ | | |

now "slide window right" and compute this

# Yet Another Solution

```
int fib(int n):
  int table[n+1] = {0, 0, ….};
  table[0] = 0;
  table[1] = 1;
  for(int i=2; i<=n; i++)
    table[i] =
 table[i-1]+table[i-2];
  return table[n];
```

# Yet Another Solution

Iteratively **build a table** of partial sols

What’s the running time?
What’s the space usage?

# Yet Another Solution

Iteratively **build a table** of partial sols

What's the running time? $O(n)$
What's the space usage? $O(n)$

Called **dynamic programming**: trade space for time

# Dynamic Programming

Requires two things to be true:

1. Large problem can be recursively broken up into smaller sub-problems

# Dynamic Programming

Requires two things to be true:

1. Large problem can be recursively broken up into smaller sub-problems (has **optimal substructure**)

# Dynamic Programming

Requires two things to be true:

1. Large problem can be recursively broken up into smaller sub-problems (has **optimal substructure**)
2. Results from sub-problems are reused many times in solving the large problem

# Dynamic Programming

Dynamic programming (bottom-up) and memoization (top-down) are **equivalent**

- (but sometimes one approach is easier/ more natural)

# Dynamic Programming

Dynamic programming (bottom-up) and memoization (top-down) are **equivalent**

- (but sometimes one approach is easier/ more natural)

Hints to use dynamic programing:

- problem “smells exponential”

# Dynamic Programming

Dynamic programming (bottom-up) and memoization (top-down) are **equivalent**

- (but sometimes one approach is easier/ more natural)

Hints to use dynamic programing:

- problem “smells exponential”
- there are ways to reuse/prune/cull partial results

“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 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.”

-Bellman

# The *Hades* Problem

You are playing a game with 10 levels. Each level you can use 10 different strategies. You start with 100 health.

Strategy s on level l:

- Takes $t_{l,s}$ seconds
- Costs $h_{l,s}$ health

What's the quickest you can beat the game without dying?

# Example (3 levels, 3 strats, 5 hp)

| health cost | time |
|---|---|
| 0 hp | 100 s |
| 2 hp | 80 s |
| 4 hp | 30 s |

level 1

| health cost | time |
|---|---|
| 1 hp | 90 s |
| 1 hp | 80 s |
| 3 hp | 70 s |

level 2

| health cost | time |
|---|---|
| 0 hp | 120 s |
| 1 hp | 70 s |
| 2 hp | 50 s |

level 3

# Example (3 levels, 3 strats, 5 hp)

| health cost | time |
|---|---|
| 0 hp | 100 s |
| 2 hp | 80 s |
| 4 hp | 30 s |

level 1

| health cost | time |
|---|---|
| 1 hp | 90 s |
| 1 hp | 80 s |
| 3 hp | 70 s |

level 2

| health cost | time |
|---|---|
| 0 hp | 120 s |
| 1 hp | 70 s |
| 2 hp | 50 s |

level 3

one possible run: 100+80+50 = 230 s. 2 hp left over

optimal?

# The *Hades* Problem

Naïve solution: try all 10^10 sets of strategies (no chance)

**What is the optimal substructure?**

# The *Hades* Problem

Thought process:

> "I can beat level L with different amounts of health left. For each amount of health, I care about the **fastest** I can get to the end of level L with at least that amount of health. I don't care about slower strategies that also get me there with the same amount of health."

# The *Hades* Problem

Naïve solution: try all 10^10 sets of strategies (no chance)

**What is the optimal substructure?**

Compute f(L, h): the fastest you can beat level L with at least h health.

# Example (3 levels, 3 strats, 5 hp)

| health cost | time |
|---|---|
| 0 hp | 100 s |
| 2 hp | 80 s |
| 4 hp | 30 s |

| health cost | time |
|---|---|
| 1 hp | 90 s |
| 1 hp | 80 s |
| 3 hp | 70 s |

| health cost | time |
|---|---|
| 0 hp | 120 s |
| 1 hp | 70 s |
| 2 hp | 50 s |

l

| h | | | |
|---|---|---|---|
| 0 | 30 | | |
| 1 | 30 | | |
| 2 | 80 | | |
| 3 | 80 | | |
| 4 | 80 | | |
| 5 | 100 | | |

# Example (3 levels, 3 strats, 5 hp)

| health cost | time |
|---|---|
| 0 hp | 100 s |
| 2 hp | 80 s |
| 4 hp | 30 s |

| health cost | time |
|---|---|
| 1 hp | 90 s |
| 1 hp | 80 s |
| 3 hp | 70 s |

| health cost | time |
|---|---|
| 0 hp | 120 s |
| 1 hp | 70 s |
| 2 hp | 50 s |

l

| h | | | |
|---|---|---|---|
| 0 | 30 | ? | |
| 1 | 30 | | |
| 2 | 80 | | |
| 3 | 80 | | |
| 4 | 80 | | |
| 5 | 100 | | |

strategy 1: 120 s

# Example (3 levels, 3 strats, 5 hp)

| health cost | time |
|---|---|
| 0 hp | 100 s |
| 2 hp | 80 s |
| 4 hp | 30 s |

| health cost | time |
|---|---|
| 1 hp | 90 s |
| 1 hp | 80 s |
| 3 hp | 70 s |

| health cost | time |
|---|---|
| 0 hp | 120 s |
| 1 hp | 70 s |
| 2 hp | 50 s |

l

| h | | | |
|---|---|---|---|
| 0 | 30 | ? | |
| 1 | 30 | | |
| 2 | 80 | | |
| 3 | 80 | | |
| 4 | 80 | | |
| 5 | 100 | | |

strategy 1: 120 s

strategy 2: 110 s

# Example (3 levels, 3 strats, 5 hp)

| health cost | time |
|---|---|
| 0 hp | 100 s |
| 2 hp | 80 s |
| 4 hp | 30 s |

| health cost | time |
|---|---|
| 1 hp | 90 s |
| 1 hp | 80 s |
| 3 hp | 70 s |

| health cost | time |
|---|---|
| 0 hp | 120 s |
| 1 hp | 70 s |
| 2 hp | 50 s |

l

| h | | | |
|---|---|---|---|
| 0 | 30 | 110 | |
| 1 | 30 | | |
| 2 | 80 | | |
| 3 | 80 | | |
| 4 | 80 | | |
| 5 | 100 | | |

strategy 1: 120 s
strategy 2: 110 s
strategy 3: 150 s

# The *Hades* Problem

Compute f(L, h): the fastest you can beat level L with at least h health.

Large problem: $f(100, 1)$

Recursive sub-problems:

$$f(L, h) = \min_{s=1,\ldots,10} f(L-1, h + h_{L,s}) + t_{L,s}$$

# The *Hades* Problem

Compute f(L, h): the fastest you can beat level L with at least h health.

Large problem: $f(100, 1)$

Recursive sub-problems:

$$f(L, h) = \min_{s=1,\ldots,10} f(L-1, h + h_{L,s}) + t_{L,s}$$

"first get to level L-1 with at least $h + h_{L,s}$ health. Then use level L strategy s."

# The *Hades* Problem

Compute f(L, h): the fastest you can beat level L with at least h health.

**Implementation notes:**

- probably easiest to make the table for f size (# levels + 1) * (maxhealth + 1)
  - allows for an easy “level 0” base case

# The *Hades* Problem

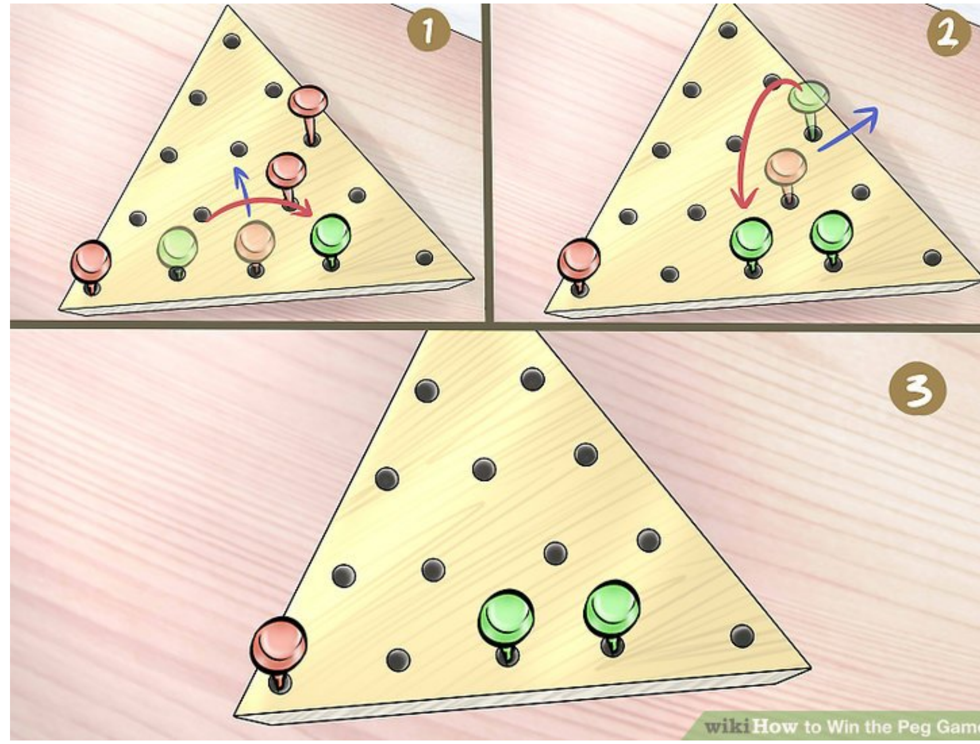
Compute f(L, h): the fastest you can beat level L with at least h health.

**Implementation notes:**

- probably easiest to make the table for f size (# levels + 1) * (maxhealth + 1)
  - allows for an easy “level 0” base case
- need to deal with “impossible” entries in the table (including the final answer)

```
int Hades():
int f[11][101] = {0,…};
for(int L=1; L<=10; L++)
  for(int h=0; h<=100; h++)
    int best = infinity;
    for(int s=0; s<10; s++)
      int hneeded = h+hpcost[L][s];
      if(hneeded <= 100)
        stime = f[L-1][hneeded] + t[L][s];
        best = min(best, stime);
    f[L][h] = best;
return f[10][1];
```

# The Peg Game




Given starting board state, what is the fewest number of pegs possible after a legal sequence of jump?

Has obvious recursive subproblems. Can dynamic programming be used? How?

# Concern #1

Requires two things to be true:

1. Large problem can be recursively broken up into smaller sub-problems (has **optimal substructure**)
2. **Results from sub-problems are reused many times in solving the large problem**

# Concern #1

Requires two things to be true:

1. Large problem can be recursively broken up into smaller sub-problems (has **optimal substructure**)
2. **Results from sub-problems are reused many times in solving the large problem**

Ok, because many different move sequences
can end at the same peg state

# Concern #2

Aren't there still exponential many subproblems we need to examine?

(Board with n holes a p pegs has n choose p possible configurations)

# Concern #2

Aren't there still exponential many subproblems we need to examine?

(Board with n holes a p pegs has n choose p possible configurations)

Only a few such configurations are reachable from the starting state

- Memoization **much easier** than DP