Dynamic Programming 101

Dynamic Programming 101

- DP = recursion (divide-n-conquer) + caching (overlapping subproblems)

Dynamic Programming 101

- DP = recursion (divide-n-conquer) + caching (overlapping subproblems)
- the simplest example is Fibonacci

Dynamic Programming 101

- DP = recursion (divide-n-conquer) + caching (overlapping subproblems)
- the simplest example is Fibonacci

$$f(n) = f(n-1) + f(n-2)$$
$$f(1) = f(2) = 1$$

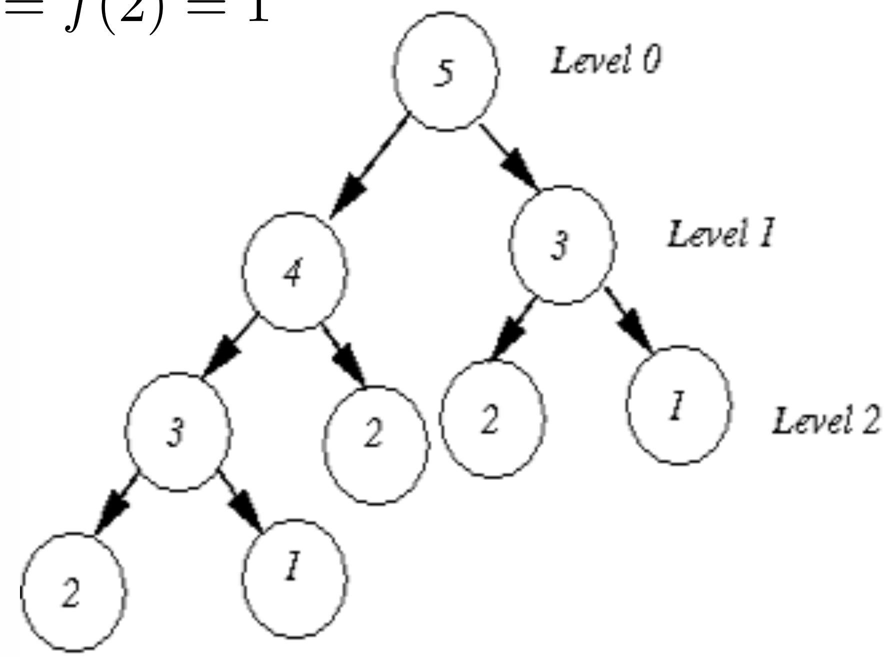
```
def fib(n):
    if n <= 2:
        return 1
    return fib(n-1) + fib(n-2)
```

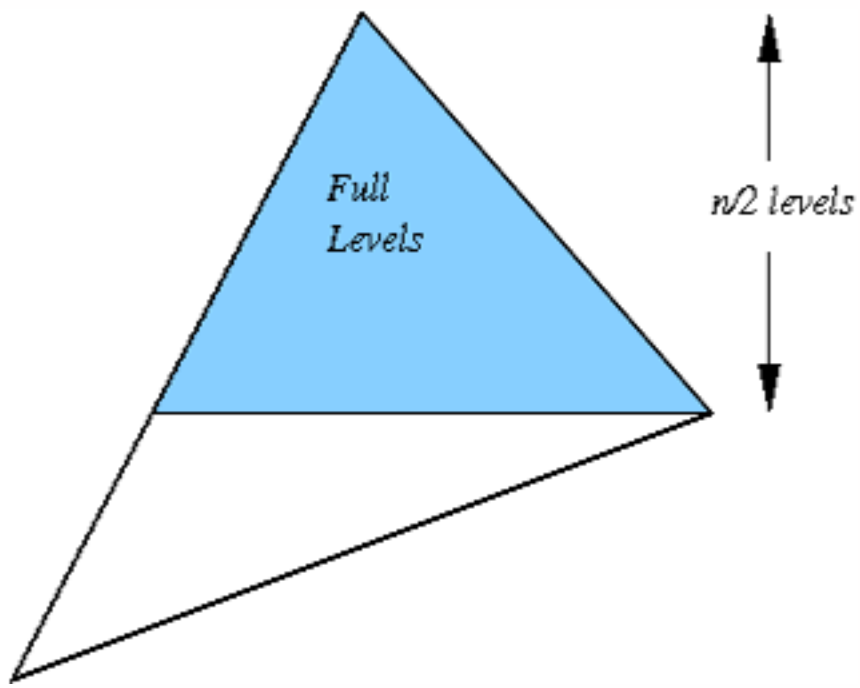
Dynamic Programming 101

- DP = recursion (divide-n-conquer) + caching (overlapping subproblems)
- the simplest example is Fibonacci

$$f(n) = f(n-1) + f(n-2)$$
$$f(1) = f(2) = 1$$

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



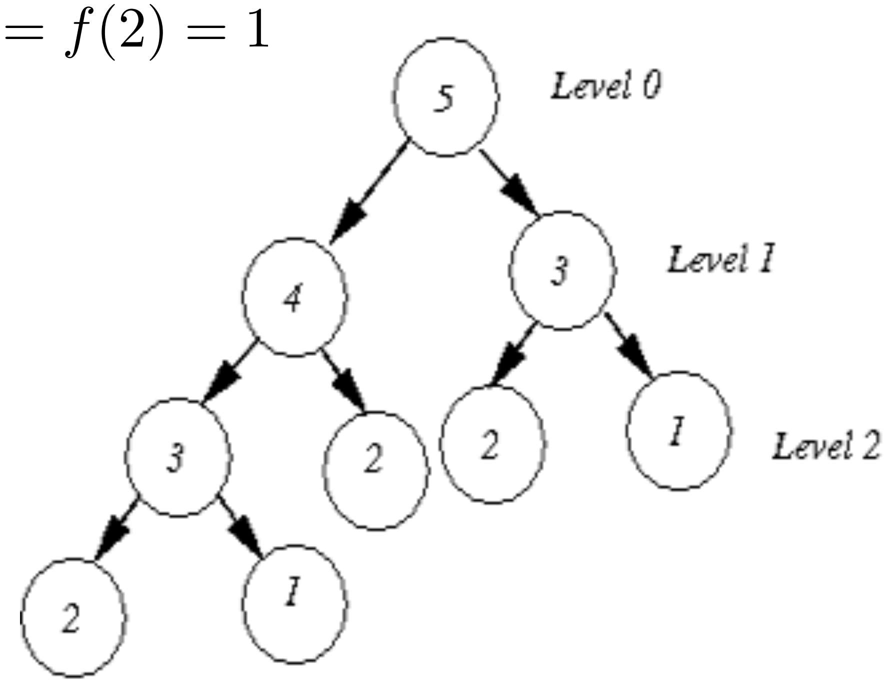


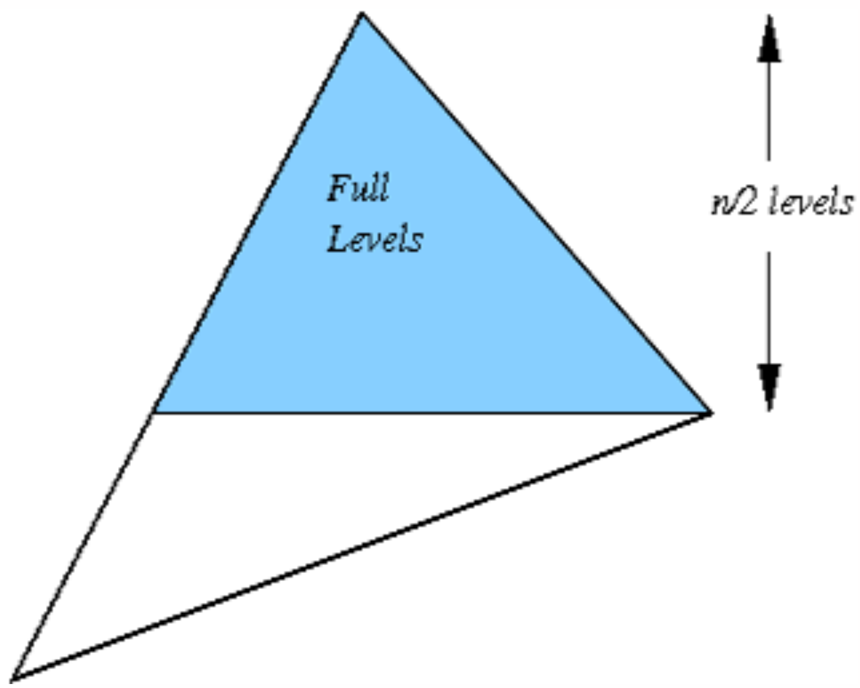
Dynamic Programming 101

- DP = recursion (divide-n-conquer) + caching (overlapping subproblems)
- the simplest example is Fibonacci

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

$f(n) = f(n-1) + f(n-2)$
$f(1) = f(2) = 1$





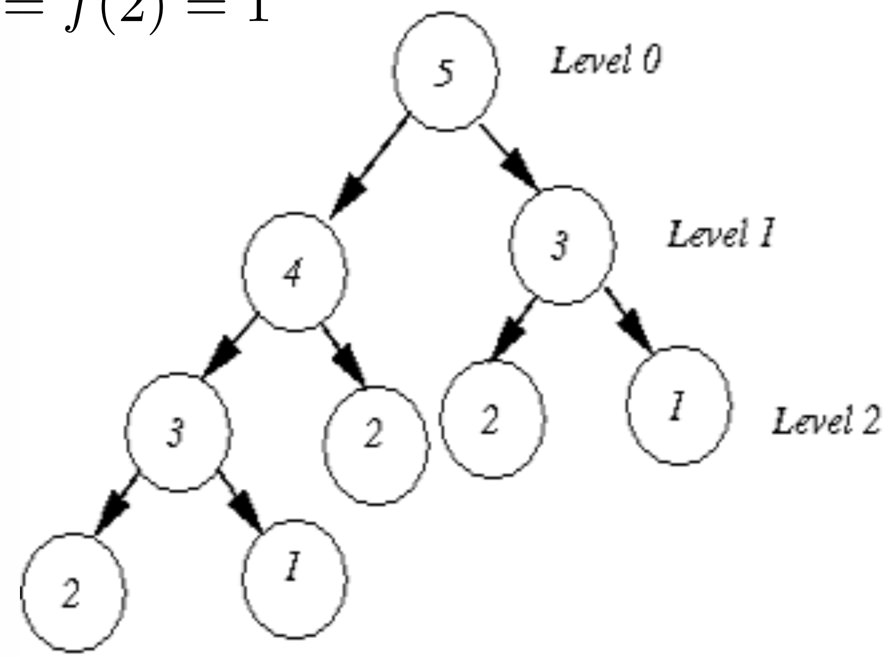
naive recursion without memoization: $O(1.618...^n)$

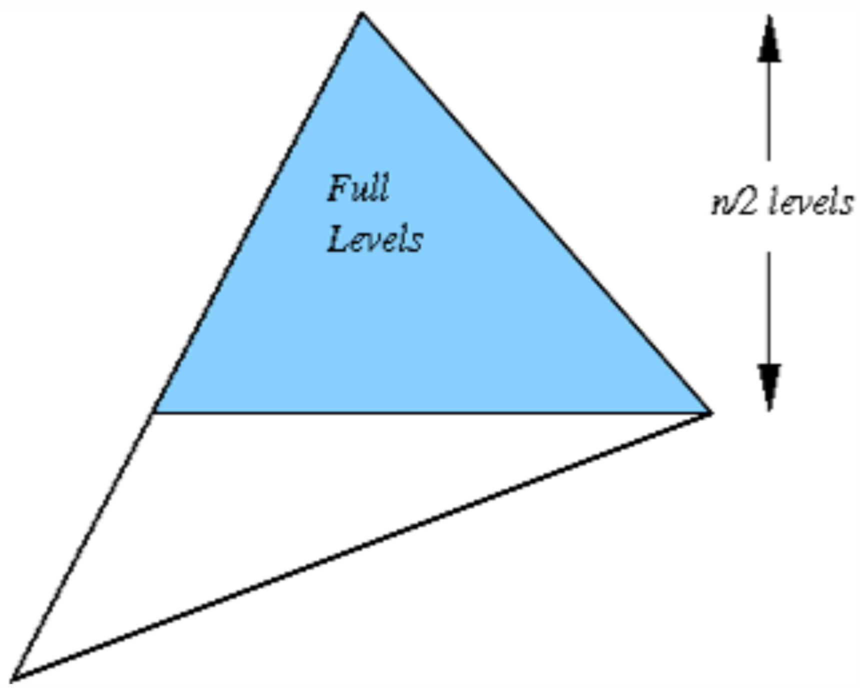
Dynamic Programming 101

- DP = recursion (divide-n-conquer) + caching (overlapping subproblems)
- the simplest example is Fibonacci

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

$$f(n) = f(n-1) + f(n-2)$$
$$f(1) = f(2) = 1$$





naive recursion without memoization: $O(1.618...^n)$

DP1: top-down with memoization: $O(n)$

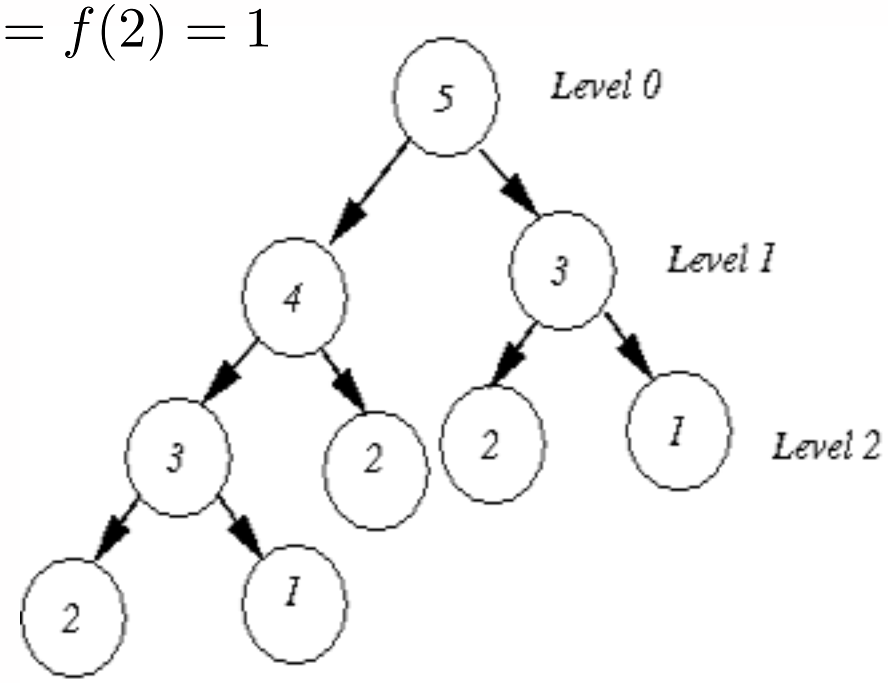
```
fibs={1:1, 2:1}
def fib1(n):
    if n not in fibs:
        fibs[n] = fib1(n-1) + fib1(n-2)
    return fibs[n]
```

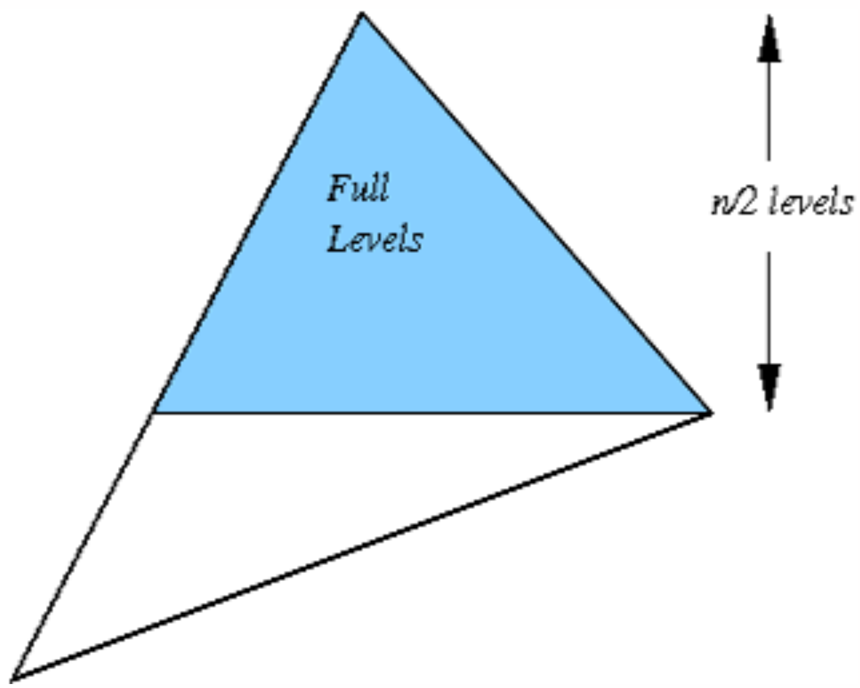
Dynamic Programming 101

- DP = recursion (divide-n-conquer) + caching (overlapping subproblems)
- the simplest example is Fibonacci

$$f(n) = f(n-1) + f(n-2)$$
$$f(1) = f(2) = 1$$

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





naive recursion without memoization: $O(1.618...^n)$

DP2: bottom-up: $O(n)$

```
def fib0(n):
    a, b = 1, 1
    for i in range(3, n+1):
        a, b = a+b, a
    return a
```

DP1: top-down with memoization: $O(n)$

```
fibs={1:1, 2:1}
def fib1(n):
    if n not in fibs:
        fibs[n] = fib1(n-1) + fib1(n-2)
    return fibs[n]
```

Number of Bitstrings

Number of Bitstrings

- number of n-bit strings that do not have 00 as a substring

Number of Bitstrings

- number of n-bit strings that do not have 00 as a substring
    - e.g. n=1: 0, 1; n=2: 01, 10, 11; n=3: 010, 011, 101, 110, 111

Number of Bitstrings

- number of n-bit strings that do not have 00 as a substring
  - e.g. n=1: 0, 1; n=2: 01, 10, 11; n=3: 010, 011, 101, 110, 111
  - what about n=0?

Number of Bitstrings

- number of n-bit strings that do not have 00 as a substring
  - e.g. n=1: 0, 1; n=2: 01, 10, 11; n=3: 010, 011, 101, 110, 111
  - what about n=0?
  - first bit "1" followed by $f(n-1)$ substrings

Number of Bitstrings

- number of n-bit strings that do not have 00 as a substring
  - e.g. n=1: 0, 1; n=2: 01, 10, 11; n=3: 010, 011, 101, 110, 111
  - what about n=0?
  - first bit "1" followed by $f(n-1)$ substrings
  - first two bits "01" followed by $f(n-2)$ substrings

Number of Bitstrings

- number of n-bit strings that do not have 00 as a substring
  - e.g. n=1: 0, 1; n=2: 01, 10, 11; n=3: 010, 011, 101, 110, 111
  - what about n=0?
  - first bit "1" followed by $f(n-1)$ substrings
  - first two bits "01" followed by $f(n-2)$ substrings

$$f(n) = f(n-1) + f(n-2)$$

Number of Bitstrings

- number of n-bit strings that do not have 00 as a substring
  - e.g. n=1: 0, 1; n=2: 01, 10, 11; n=3: 010, 011, 101, 110, 111
  - what about n=0?
  - first bit "1" followed by $f(n-1)$ substrings
  - first two bits "01" followed by $f(n-2)$ substrings

$$f(n) = f(n-1) + f(n-2)$$

$f(1)=2$, $f(0)=1$

Max Independent Set

Max Independent Set

- max weighted independent set on a linear-chain graph

Max Independent Set

- max weighted independent set on a linear-chain graph
  - e.g. 7 -- 2 -- 3 -- 5 -- 8

Max Independent Set

- max weighted independent set on a linear-chain graph
  - e.g.   7 -- 2 -- 3 -- 5 -- 8
  - subproblem: $f(n)$ -- max independent set for a[1]..a[n]

Max Independent Set

- max weighted independent set on a linear-chain graph
  - e.g. 7 -- 2 -- 3 -- 5 -- 8
  - subproblem: $f(n)$ -- max independent set for a[1]..a[n]

    $$f(n) = \max\{f(n-1),\ \ f(n-2) + a[n]\}$$

Max Independent Set

- max weighted independent set on a linear-chain graph
  - e.g. 7 -- 2 -- 3 -- 5 -- 8
  - subproblem: $f(n)$ -- max independent set for a[1]..a[n]

    $$f(n) = \max\{f(n-1),\ \ f(n-2) + a[n]\}$$

    $f(0)=0$; $f(1)=a[1]$?

Max Independent Set

- max weighted independent set on a linear-chain graph
  - e.g. 7 -- 2 -- 3 -- 5 -- 8
  - subproblem: $f(n)$ -- max independent set for a[1]..a[n]

    $$f(n) = \max\{f(n-1),\ \ f(n-2) + a[n]\}$$

    $f(0)$=0; $f(1)$=a[1]?   better: $f(0)$=0; $f(-1)$=0

Summary

- Dynamic Programming = divide-n-conquer + overlapping
  - “distributivity” of work: a*c+b*c+a*d+b*d = (a+b)*(c+d)
- two implementation styles
  - 1. recursive top-down + memoization
  - 2. bottom-up
  - also need backtracking for recovering best solution
- three steps in solving a DP problem
  - define the subproblem
  - recursive formula
  - base cases