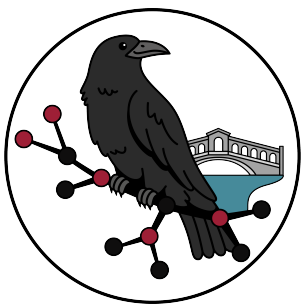
Dynamic Programming

CPSCP – Collaborative Problem Solving and Competitive Programming

Ruben Becker



RAVEN – Research on Algorithms Venice

Naive Recursion

Staircase problem: 1 or 2 steps at a time – # ways to reach step n?

Naive Recursion

Staircase problem: 1 or 2 steps at a time – # ways to reach step n?

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

```
int f(int n) {
  if (n <= 1) return 1;
  return f(n-1)+f(n-2);
}
```

Naive Recursion

Staircase problem: 1 or 2 steps at a time – # ways to reach step n?

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

```
int f(int n) {
  if (n <= 1) return 1;
  return f(n-1)+f(n-2);
}
```

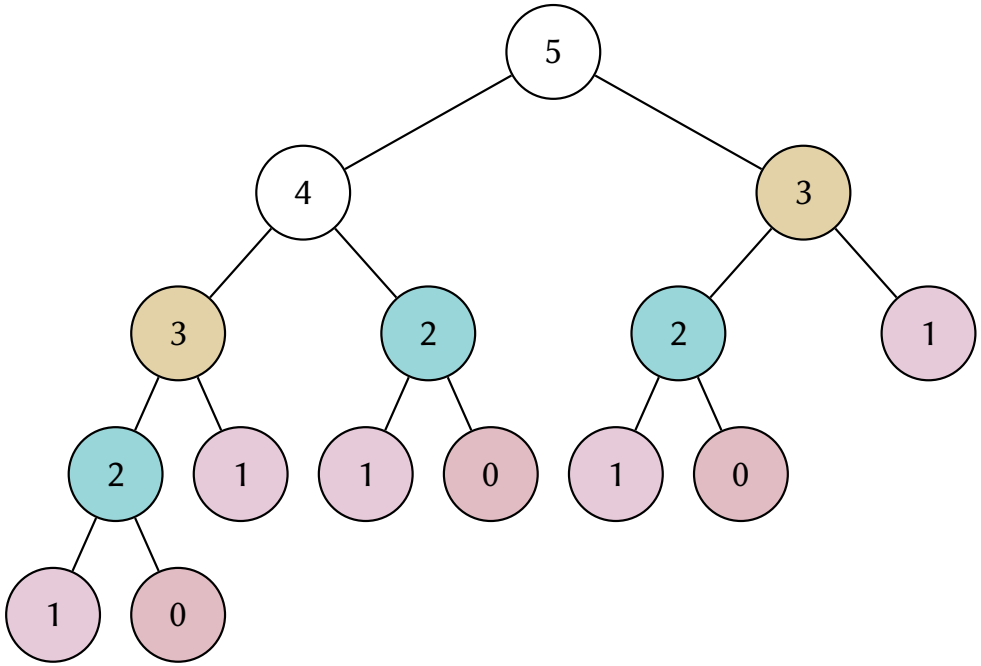
$\Rightarrow$ $T(n) =$
$T(n-1)+T(n-2)+O(1) =$
$O(\varphi^n)$ – **exponential**

Naive Recursion

Staircase problem: 1 or 2 steps at a time – # ways to reach step n?

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

```
int f(int n) {
  if (n <= 1) return 1;
  return f(n-1)+f(n-2);
}
```



$\Rightarrow$ $T(n) =$
$T(n-1)+T(n-2)+O(1) =$
$O(\varphi^n)$ – **exponential**

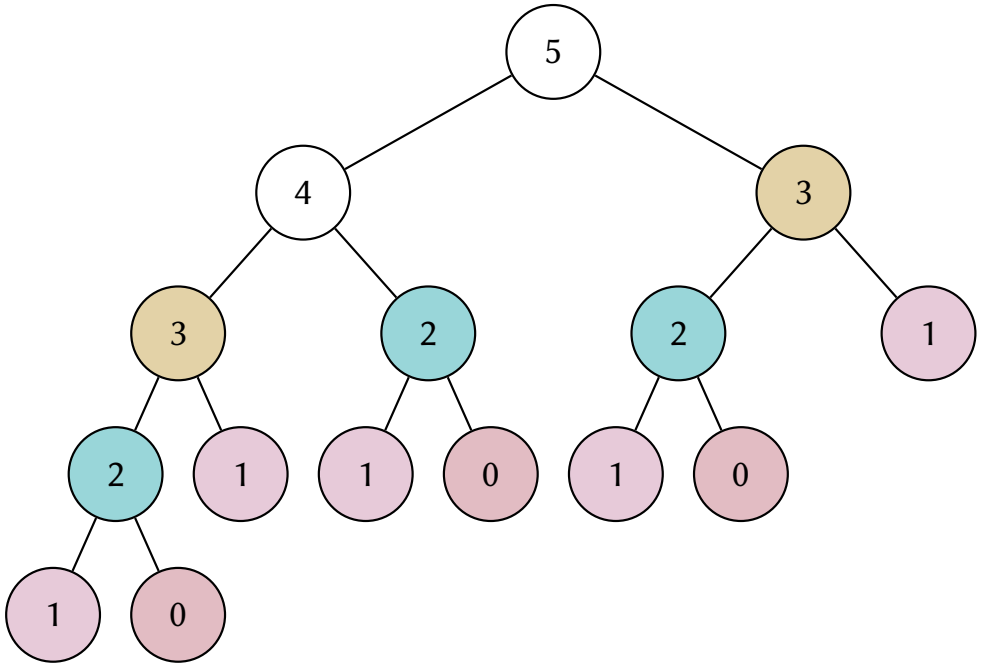
Naive Recursion

Staircase problem: 1 or 2 steps at a time – # ways to reach step n?

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

```
int f(int n) {
  if (n <= 1) return 1;
  return f(n-1)+f(n-2);
}
```

$\Rightarrow$ $T(n) =$
$T(n-1)+T(n-2)+O(1) =$
$O(\varphi^n)$ – **exponential**



same-colored nodes = same argument, recomputed

Dynamic Programming

Key observation

Exponentially many calls, but only $n+1$ **distinct** arguments: $0, \ldots, n$.

Dynamic Programming

Key observation

Exponentially many calls, but only $n+1$ **distinct** arguments: $0, \ldots, n$.

- we have **overlapping subproblems** (unlike, e.g., merge sort, where in general every subarray is different)

$\Rightarrow$ **DP: recursion + remember what you already solved**

```
int memo[MAXN];
bool seen[MAXN];

int f(int n) {
  if (n <= 1) return 1;
  if (seen[n]) return memo[n];
  seen[n] = true;
  return memo[n] =
    f(n-1) + f(n-2);
}
```

Dynamic Programming

Key observation

Exponentially many calls, but only $n+1$ **distinct** arguments: $0, \ldots, n$.

- we have **overlapping subproblems** (unlike, e.g., merge sort, where in general every subarray is different)

$\Rightarrow$ **DP: recursion + remember what you already solved**

```
int memo[MAXN];
bool seen[MAXN];

int f(int n) {
  if (n <= 1) return 1;
  if (seen[n]) return memo[n];
  seen[n] = true;
  return memo[n] =
    f(n-1) + f(n-2);
}
```

$\Rightarrow$ each value computed **once**

$\Rightarrow$ $O(n)$ time, $O(n)$ space

Top-Down vs. Bottom-Up

Memoization (top-down)

```
int f(int n) {
  if (n <= 1) return 1;
  if (seen[n]) return memo[n];
  seen[n] = true;
  return memo[n] =
    f(n-1) + f(n-2);
}
```

Top-Down vs. Bottom-Up

Memoization (top-down)

```
int f(int n) {
  if (n <= 1) return 1;
  if (seen[n]) return memo[n];
  seen[n] = true;
  return memo[n] =
    f(n-1) + f(n-2);
}
```

Tabulation (bottom-up)

```
f[0] = f[1] = 1;
for (int i = 2; i <= n; i++)
  f[i] = f[i-1]+f[i-2];
```

Top-Down vs. Bottom-Up

Memoization (top-down)

```
int f(int n) {
  if (n <= 1) return 1;
  if (seen[n]) return memo[n];
  seen[n] = true;
  return memo[n] =
    f(n-1) + f(n-2);
}
```

Tabulation (bottom-up)

```
f[0] = f[1] = 1;
for (int i = 2; i <= n; i++)
  f[i] = f[i-1]+f[i-2];
```

- both $O(n)$ time
- tabulation avoids recursion overhead and can drop to $O(1)$ space

Another Example: Longest Common Subsequence

Problem: length of the LCS of s_1 (len. n), s_2 (len. m).

$\mathrm{lcs}(i, j)$ = LCS of prefixes $s_1[1..i]$, $s_2[1..j]$:

Another Example: Longest Common Subsequence

Problem: length of the LCS of s_1 (len. n), s_2 (len. m).
$\mathrm{lcs}(i, j)$ = LCS of prefixes $s_1[1..i]$, $s_2[1..j]$:

Example

$s_1 =$ **A**B**CD**, $s_2 =$ **AC**B**D**

$\Rightarrow$ lcs = ACD (length 3)

Another Example: Longest Common Subsequence

Problem: length of the LCS of s_1 (len. n), s_2 (len. m).

$\mathrm{lcs}(i,j)$ = LCS of prefixes $s_1[1..i]$, $s_2[1..j]$:

$$\mathrm{lcs}(i,j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ 1 + \mathrm{lcs}(i-1, j-1) & i,j>0,\ s_1[i] = s_2[j] \\ \max\big(\mathrm{lcs}(i-1,j), \mathrm{lcs}(i,j-1)\big) & i,j>0,\ s_1[i] \neq s_2[j] \end{cases}$$

Example

$s_1 = \mathbf{A}\mathrm{B}\mathbf{CD}$, $s_2 = \mathbf{AC}\mathrm{B}\mathbf{D}$

$\Rightarrow$ lcs = ACD (length 3)

Another Example: Longest Common Subsequence

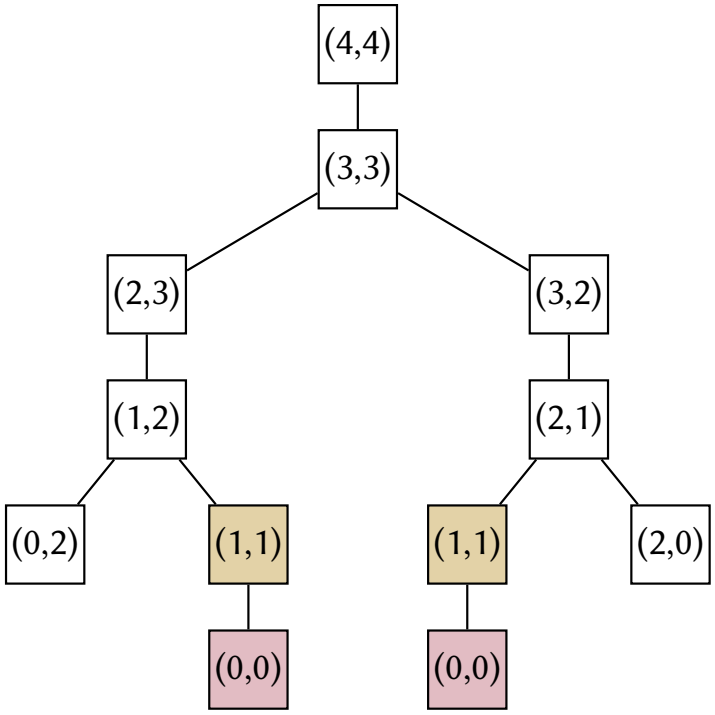
Problem: length of the LCS of s_1 (len. n), s_2 (len. m).
$\mathrm{lcs}(i,j)$ = LCS of prefixes $s_1[1..i]$, $s_2[1..j]$:

$$\mathrm{lcs}(i,j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ 1 + \mathrm{lcs}(i-1, j-1) & i,j>0,\ s_1[i] = s_2[j] \\ \max\left(\mathrm{lcs}(i-1,j), \mathrm{lcs}(i,j-1)\right) & i,j>0,\ s_1[i] \neq s_2[j] \end{cases}$$

- state is now a **pair** (i,j); naive recursion: $O(2^{n+m})$

Example

$s_1 =$ **A**B**CD**, $s_2 =$ **AC**B**D**
$\Rightarrow$ lcs = ACD (length 3)



$(1,1)$ and $(0,0)$ each recomputed

Another Example: Longest Common Subsequence

Problem: length of the LCS of s_1 (len. n), s_2 (len. m).
$\text{lcs}(i,j)$ = LCS of prefixes $s_1[1..i]$, $s_2[1..j]$:

$$\text{lcs}(i,j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ 1 + \text{lcs}(i-1, j-1) & i,j>0,\ s_1[i] = s_2[j] \\ \max\big(\text{lcs}(i-1,j), \text{lcs}(i,j-1)\big) & i,j>0,\ s_1[i] \neq s_2[j] \end{cases}$$

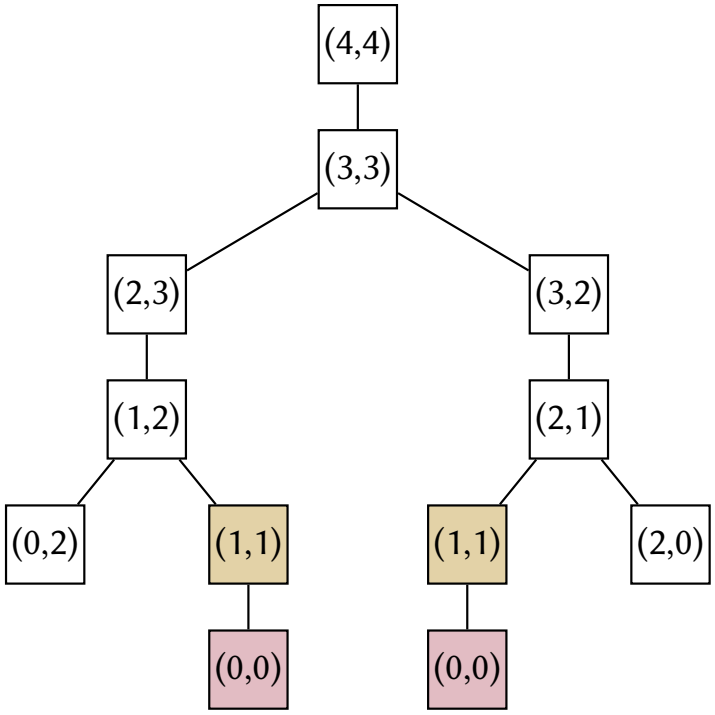
- state is now a **pair** (i,j); naive recursion: $O(2^{n+m})$

		A	C	B	D
	0	0	0	0	0
A	0	1	1	1	1
B	0	1	1	2	2
C	0	1	2	2	2
D	0	1	2	2	**3**

dp filled bottom-up: no state recomputed

Example

s_1 = **A**B**CD**, s_2 = **AC**B**D**
$\Rightarrow$ lcs = ACD (length 3)



$(1,1)$ and $(0,0)$ each recomputed

Conclusion

	Fibonacci	**LCS**
naive	$O(\varphi^n)$	$O(2^{n+m})$
# states	$n+1$	$(n+1)(m+1)$
DP	$O(n)$	$O(nm)$

Conclusion

	Fibonacci	**LCS**
naive	$O(\varphi^n)$	$O(2^{n+m})$
# states	$n+1$	$(n+1)(m+1)$
DP	$O(n)$	$O(nm)$

Take-away

- DP usually gives exponential speed-up w.r.t. simple recursion/brute-force
- unlike greedy, DP tries all options, not just most promising/locally best one

Conclusion

	Fibonacci	**LCS**
naive	$O(\varphi^n)$	$O(2^{n+m})$
# states	$n+1$	$(n+1)(m+1)$
DP	$O(n)$	$O(nm)$

Take-away

- DP usually gives exponential speed-up w.r.t. simple recursion/brute-force
- unlike greedy, DP tries all options, not just most promising/locally best one

Recipe

1. **State**: Which parameters describe subproblem?
   E.g., number/pair of indices
2. **Recurrence**: write how one state results from previous states
3. **Overlap?** Yes $\rightarrow$ DP.
   No $\rightarrow$ Simple Recursion.
4. **Store & Reuse**: Either by memoization (top-down) or Tabulation (bottom-up)

Your Task

- https://open.kattis.com/problems/knapsack
- https://open.kattis.com/problems/trainsorting