

# Lecture 23

## Dynamic Programming / Memoization



EECS 281: Data Structures & Algorithms

# Dynamic Programming

Data Structures & Algorithms

# Dynamic Programming

- A primary advantage of a typical recursive algorithm is to divide the domain into *independent* sub-problems
- Some recursive problems do not divide into *independent* sub-problems
- Use Dynamic Programming (DP)
  - Bottom-up
  - Top-down

# Dynamic Programming

Dynamic Programming is applicable if:

- You have a large problem that can be broken into smaller subproblems
- The subproblems are NOT independent; the same subproblems recur in the course of solving the larger problem (hopefully many times)

# Dynamic Programming

- Reduce the run time of a recursive function
  - Change complexity class, often from  $O(c^n)$  to  $O(n^c)$
- Use memory to avoid computing values which have already been computed

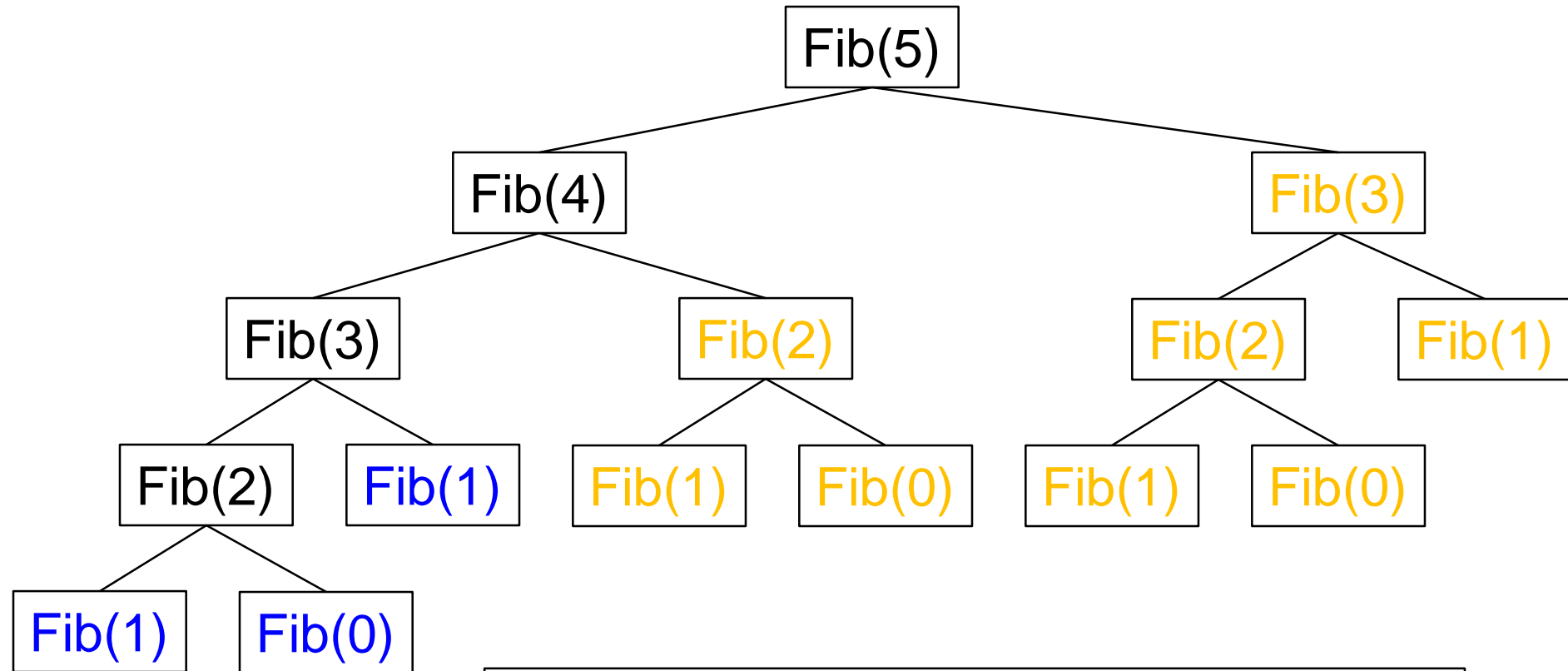
# Memoization

- This technique is called **memoization** (derived from the Latin word *memorandum* (to be remembered))
- Different word than “memorization” (although they are cognate words)
- This is how Dynamic Programming works

# Memoization

- Start with a recursive function and modify it
  - On exit:
    - Save the inputs and the result
  - On entry:
    - Check the inputs and determine if they have been seen before
    - If they have, retrieve the result from memory rather than recomputing it
- Often done with only minor code changes

# Example: Fibonacci Sequence



Black: calculated for the first time

Blue: base case

Orange: recalculated + extra base case checks



# Fibonacci: Naïve Implementation

```
1  uint64_t findFib(uint32_t i) {  
2      if (i == 0 || i == 1)  
3          return i;  
4      return findFib(i - 1) + findFib(i - 2);  
5  } // findFib()
```

- Spectacularly inefficient recursive algorithm
- Exponential running time:  $O(1.618^N)$

# Fibonacci: Top-Down DP

```
1  #include <iostream>
2  using namespace std;
3
4  // fib(93) is the largest fib that fits in uint64_t
5  static const size_t MAX_FIB = 93;
6  uint64_t fib(uint32_t n);
7
8  int main() {
9      for (size_t i = 0; i <= MAX_FIB; ++i)
10         cout << "fib(" << i << "): " << fib(i) << endl;
11
12     return 0;
13 } // main()
```

# Fibonacci: Top-Down DP

```
15 uint64_t fib(uint32_t n) {
16     // Array of known Fibonacci numbers. Start out with 0, 1,
17     // and the rest get automatically initialized to 0.
18     // MAX_FIB + 1 used to account for 0-indexing
19     static uint64_t fibs[MAX_FIB + 1] = { 0, 1 };
20
21     // Doesn't fit in 64 bits, so don't even bother computing
22     if (n > MAX_FIB)
23         return 0;
24
25     // Is already in array, so look it up
26     if (fibs[n] > 0 || n == 0)
27         return fibs[n];
28
29     // Currently unknown, so calculate and store it for later
30     fibs[n] = fib(n - 1) + fib(n - 2);
31     return fibs[n];
32 } // fib()
```

# Fibonacci: Bottom-Up DP

```
1  uint64_t fibBU(uint32_t i) {  
2      uint64_t f[MAX_N];  
3      i = min(i, MAX_N - 1);  
4      f[0] = 0;  
5      f[1] = 1;  
6      for (size_t k = 2; k <= i; k++)  
7          f[k] = f[k - 1] + f[k - 2];  
8      return f[i];  
9  } // fibBU()
```

- Evaluate function by
  - Starting at smallest function value
  - Using previously computed values at each step to compute current value
- Must save all previously computed values
  - Note that the values in `f[]` grow exponentially

Simple technique has converted an exponential algorithm ( $O(1.618^N)$ ) to a linear one ( $O(n)$ )

# Example: Binomial Coefficient

Definition:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

For  $n \geq m \geq 0; n, m \in \mathbb{Z}$

# Binomial Coefficient

## Naïve Approach

- Solve for  $n!$  (iteratively or recursively)
  - $\text{fact}(n) = n * \text{fact}(n - 1)$
- Solve for  $(n - m)!$
- Solve for  $m!$
- Integer overflow is a major issue:
  - $13!$  causes overflow of a 32-bit integer
  - $21!$  causes overflow of a 64-bit integer
  - $35!$  causes overflow of a 128-bit integer

# Binomial Coefficient Rewritten

Base cases, for  $n \geq 0$ :

$$\binom{n}{0} = \binom{n}{n} = 1$$

Recursive definition, for  $n > m \geq 1$ :

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

# Binomial Coefficient: Top-Down

```
1  uint64_t biCoeffTD(uint32_t n, uint32_t m) {  
2      if (m == 0 || m == n)  
3          return 1;  
4      else  
5          return biCoeffTD(n - 1, m - 1)  
6                  + biCoeffTD(n - 1, m);  
7  } // biCoeffTD()
```

- Elegant? Yes.
- Efficient? NO!



# Binomial Coefficient: Top-Down

```
1  uint64_t binomHelper(uint32_t n, uint32_t m,
2                        vector<vector<uint64_t>> &memo) {
3      if (m == 0 || m == n) {
4          memo[m][n] = 1;
5          return 1;
6      } // if
7      if (memo[m][n] > 0)
8          return memo[m][n];
9      memo[m][n] = binomHelper(n - 1, m - 1, memo)
10                 + binomHelper(n - 1, m, memo);
11     return memo[m][n];
12 } // binomHelper()
13
14 uint64_t binomTD(unsigned int n, unsigned int m) {
15     vector<vector<uint64_t>> memo(m + 1, vector<uint64_t>(n + 1));
16     return binomHelper(n, m, memo);
17 } // binomTD()
```

# Binomial Coefficient: Bottom-Up

```
1  uint64_t biCoeffBU(uint32_t n, uint32_t m) {
2      vector<vector<uint64_t>> c(m + 1, vector<uint64_t>(n + 1));
3      for (size_t i = 0; i <= m; ++i) {
4          for (size_t j = i; j <= n; ++j) {
5              if ((i == j) || (i == 0))
6                  c[i][j] = 1;
7              else
8                  c[i][j] = c[i - 1][j - 1]
9                          + c[i][j - 1];
10         } // for j
11     } // for i
12     return c[m][n];
13 } // biCoeffBU()
```

- Base cases: ( $i == j$ ) and ( $j == 0$ )
- Only finding values for half of 2D vector ( $j$  starts out equal to  $i$ )
- Clearly, algorithm calculates approx  $(nm)/2$  integers in vector  $c$
- Therefore, is  **$O(nm)$**

# General Approach: Top-Down

- Save known values as they are calculated
- Generally preferred because:
  - Mechanical transformation of ‘natural’ (recursive) problem solution
  - Order of computing subproblems takes care of itself
  - May not need to compute answers to all subproblems
- Adaptive
  - Only compute needed subproblems

# General Approach: Bottom-up

- Precompute values from base case *up* toward solution
- Loosely “non-adaptive”
  - May compute *all* smaller cases, needed or not
  - For example:
    - $g(n) = g(n - 2) + g(n - 4)$
    - Four base cases,  $g(0)$  through  $g(3)$
    - Given  $n$ , which values of  $g()$  are not needed?

# Nuances

- Time and space requirements for dynamic programming may become prohibitively large
  - Memo space used for either approach
  - Additional stack space used for top-down approach
  - Bottom-up approach can recycle/collapse previous memo steps
    - Fibonacci really only needs the previous 2 values, not all previous values)
    - With top-down you can't eliminate these, because you don't know order of evaluation

# Common Subproblems

- Input is  $x_1, x_2, \dots, x_n$ ; a subproblem is  $x_1, x_2, \dots, x_i$ 
  - Number of subproblems is  $O(n)$
  - Example: Fibonacci
- Input is  $x_1, x_2, \dots, x_n$ ; a subproblem is  $x_i, x_{i+1}, \dots, x_j$ 
  - Number of subproblems is  $O(n^2)$
  - Example: Pipe Welding and Positive Subset Sum (lab slides)

# More Common Subproblems

- Input is  $x_1, x_2, \dots, x_n$ , and  $y_1, y_2, \dots, y_m$ ; a subproblem is  $x_1, x_2, \dots, x_i$  and  $y_1, y_2, \dots, y_j$ 
  - Number of subproblems is  $O(nm)$
  - Example: Binomial Coefficient (earlier in slides), Edit Distance (final exam additional problems)
- Input is a rooted tree; a subproblem is a rooted subtree
  - See other classes

# Top-Down or Bottom-Up?

- For any problem we'll give you, either solution will work (top-down, bottom-up)
- Some problems the base cases are easy to see, but the ones just above that are hard
- Some problems it's hard to see which sub-problems need to be computed
- <https://www.quora.com/Can-every-problem-on-Dynamic-Programming-be-solved-by-both-top-down-and-bottom-up-approaches>



# Summary: Dynamic Programming

- Advanced technique for difficult problems
- Trading space (when ample) for time
- Applied when solution domains are dependent upon each other
- Bottom-up
  - Iteratively pre-compute all values ‘from the bottom’
- Top-down
  - Recursively compute and save needed values ‘from the top’

# Dynamic Programming

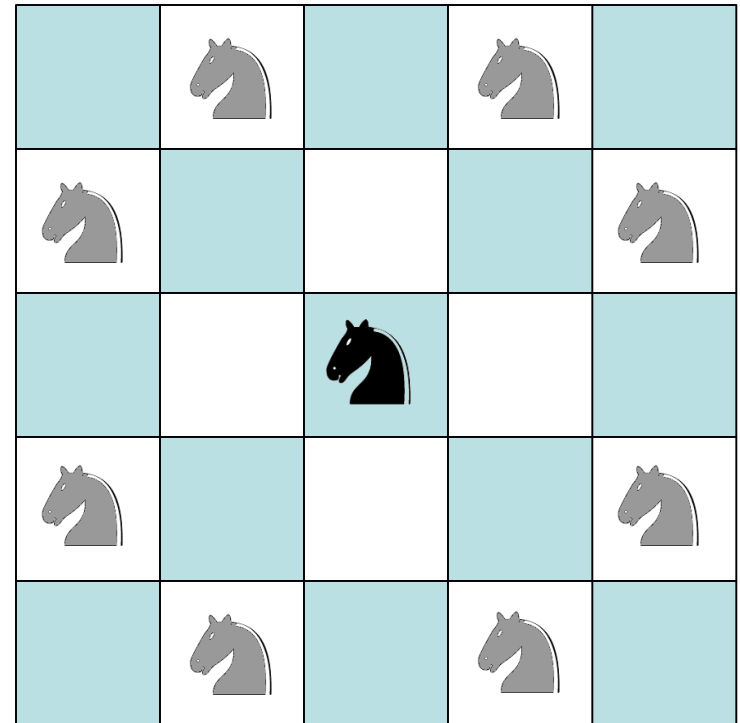
Data Structures & Algorithms

# DP: Knight Moves

Data Structures & Algorithms

# Knight Moves

The Knight travels in an “L” fashion, moving 2 spaces in any direction, turning 90° left or right and moving 1 space



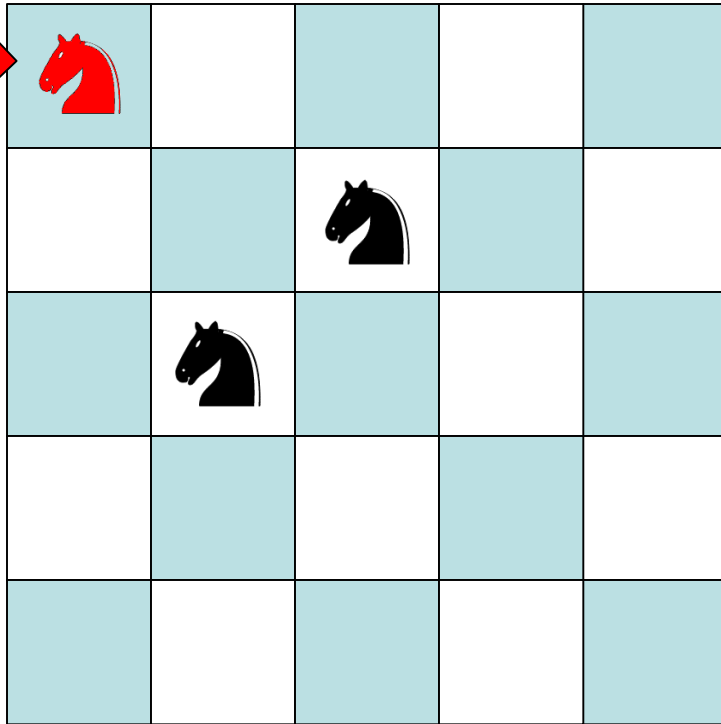
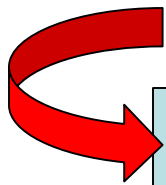
# Knight Moves

- Moving from the top-left corner to bottom-right corner of a standard 8x8 board can be done in a minimum of 6 moves
- Q: How could we determine the number of different ways it could be done, in exactly 6 moves?
- A: Use Dynamic Programming, with a 3D memo to calculate number of routes to all possible locations after 6 moves.

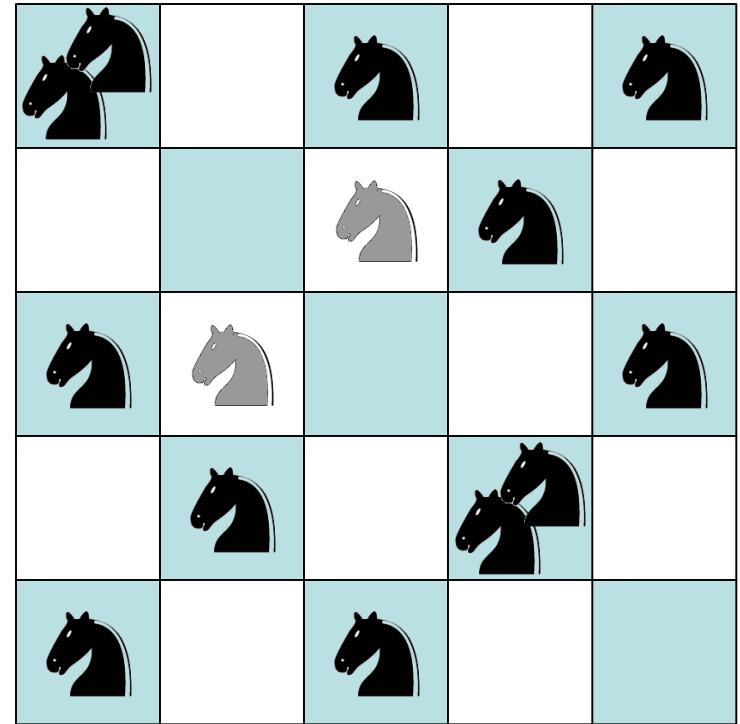
# Knight Moves

Top-Left Corner

Start here







Possible after 1 move

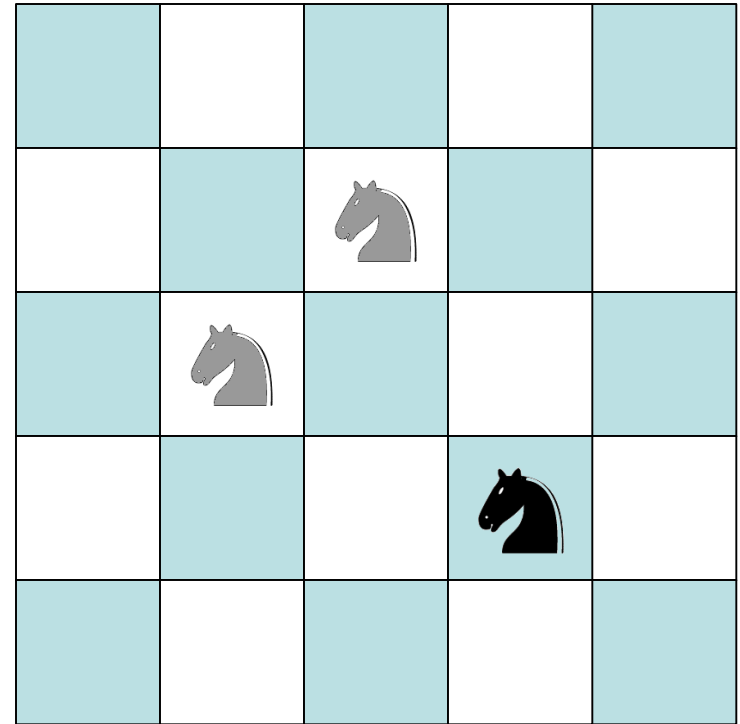


Possible after 2 moves

# Knight Moves

Overlapping sub-problem:

-  is reachable after moving to either 
- Calculating a partial path from  to the final destination could be used in both complete solutions through 



# Knight Moves

Memo[0]

1				

Memo[1]

1*				
		1		
	1			

\* *gray* numbers  
are previous  
memo

(View of upper-left 5x5 sections of a full 8x8 board)




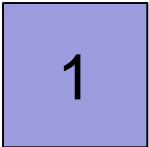

Memo[2]  
(w/ Memo[1]  
in gray)

2		1		1			
		1	1				
1	1			1			
	1		2				
1		1					

Memo[2] only


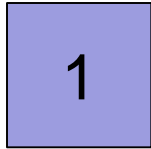
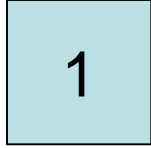
2		1		1			
			1				
1				1			
	1		2				
1		1					

Calculating  
Memo[3]:  
Location  
(1, 2)

-  is Memo[3]
-  is Memo[2]
-  is Memo[2]  
(unused)

2		1		1			
		8	1				
1				1			
	1		2				
1		1					

Calculating  
Memo[3]:  
Location  
(2, 1)

 is Memo[3]  
 is Memo[2]  
 is Memo[2]  
(unused)

2		1		1			
			1				
1	8			1			
	1		2				
1		1					

Memo[3]  
(w/ Memo[2]  
in gray)

2	2	1	1	1	2		
2		8	1	3		2	
1	8		4	1	4		
1	1	4	2	2		1	
1	3	1	2		3		
2		4		3			
	2		1				

Memo[4] only

16		17		18		7	
	10		22		8		7
17		16		23		10	
	22		36		14		9
18		23		18		9	
	8		14		6		4
7		10		9		6	
	7		9		4		

Memo[5] only

	55		57		62		18
55		132		100		64	
	132		120		126		38
57		120		108		66	
	100		108		111		36
62		126		111		54	
	64		66		54		19
18		38		36		19	

Memo[6] only

264		407		442		264	
	354		603		407		254
407		640		720		435	
	603		938		641		357
442		720		732		456	
	407		641		458		250
264		435		456		294	
	254		357		250		<u>108</u>



# Knight Moves Wrap-up

- What is the time complexity for this solution?
- Generalize time complexity given a board size (N) and number of moves (M)
- What is the memory complexity?
- How could the memory footprint be reduced?
- Describe the top-down method

# DP: Knight Moves

Data Structures & Algorithms