

# Lecture 22

## Dynamic Programming / Memoization

EECS 281: Data Structures & Algorithms

# Dynamic Programming

- Advantage of typical recursive algorithm is to divide 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 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

- Motivation
  - Reduces the running time of a recursive function to be:
    - time required to evaluate the function
    - For all arguments the given argument
    - Treats the cost of a recursive call as a constant

# Saving intermediate results

- Idea: if the same functions repeatedly get called with the same inputs, **save the results** for these inputs
- Motivation: eliminate costly re-computation in any recursive program, given enough **space** to store values of the function for arguments smaller than the initial call
- Time / space tradeoff: use extra memory to reduce computation and save time

# Memoization

- Rewrite functions to:
  - On exit:
    - Save the inputs
    - Save the result
  - On entry:
    - Check the inputs and see if they have been seen before
    - If they have, just retrieve the result from memory rather than recomputing it
- Can often be done with only minor code changes

# Memoization

- This technique is called **memoization** (derived from the Latin word *memorandum* (to be remembered))
- Different word than “memo**r**ization” (although they are cognate words)
- Useful for “top down” dynamic programming

# Fibonacci Sequence

## Definition

- $F(0) = 0$ ;  $F(1) = 1$
- $F(n) = F(n - 1) + F(n - 2)$

Find  $F(3)$

- $F(3) = F(2) + F(1)$ ;  $F(2) = F(1) + F(0)$

$\Rightarrow$

- $F(3) = (F(1) + F(0)) + F(1) = (1 + 0) + 1$



# Recursive Implementation

## Fibonacci Sequence

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

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

# Dynamic Programming: Bottom-up

## Fibonacci Sequence

```
1  int findFibBU(int i) {
2      int f[MAX_N];
3      f[0] = 0;
4      f[1] = 1;
5      for (int k = 2; k <= i; k++)
6          f[k] = f[k - 1] + f[k - 2];
7      return f[i];
8  } // findFibBU()
```

- Evaluate any function by
  - Start at smallest function value
  - Use 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 exponential algorithm ( $O(1.618^N)$ ) to linear ( $O(n)$ )*

```
1 // Recursive, dynamic programming version
2 // of Fibonacci number calculation.
3
4 #include <cstdlib>
5 #include <iostream>
6 using namespace std;
7
8 unsigned long long fib(unsigned n);
9
10 int main() {
11     for (unsigned i = 0; i < 100; i++)
12         cout << "fib(" << i << "): " << fib(i) << endl;
13
14     cout << endl;
15     return 0;
16 } // main()
```

```

1 unsigned long long fib(unsigned n) {
2     // Largest Fibonacci number that fits in unsigned64 bits const      is fib(93)
3     unsigned SIZE = 94;
4
5     // Array of known Fibonacci numbers, start out with 0, 1,
6     // and the rest get automatically initialized to 0
7     static unsigned long long fibs[SIZE] = {      0, 1  };
8
9     // Doesn't fit in 64 bits,      so don't even bother computing
10    if (n >= SIZE)
11        return 0;
12
13    // Is already in array
14    if (fibs[n] > 0 || n == 0)
15        return fibs[n];
16
17    // Would fit in array, but      unknown at present time
18    fibs[n] = fib(n - 1) + fib(n - 2);
19    return fibs[n];
20 } // fib()

```

# Binomial Coefficient

Definition:

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

where  $n$  and  $m$  are non-negative integers

# Binomial Coefficient

## Naïve Approach

- Solve for  $n!$  recursively
  - $\text{fact}(n) = n * \text{fact}(n - 1)$
- Solve for  $(n - m)!$  recursively
- Solve for  $m!$  recursively
- Integer overflow is a major issue:
  - $13!$  cannot be represented by a 32-bit integer
  - $21!$  cannot be represented by a 64-bit integer
  - $35!$  cannot be represented by a 128-bit integer

# Binomial Coefficient Revisited

Recursive Definition:

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

Initial / boundary values:

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

for all non-negative integers  $n$

# Binomial Coefficient: Bottom Up

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

Note some cleverness in the algorithm

- Initialization of diagonal ( $i=j$ ) and first row ( $j=0$ ) in nested loop
- Only finding values for half of 2D array (i-loop)
- Clearly, algorithm calculates approx  $(nm)/2$  integers in  $c[i][j]$
- Therefore, is  $O(nm)$



# Binomial Coefficient: Top Down

```
1  int  biCoeffTD(int n, int    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()
```

- Think about it!

# Nuances

- Bottom-up dynamic programming can be used any time Top-down dynamic programming is used
- Time and space requirements for dynamic programming may become prohibitively large

# General Approach: Bottom-up

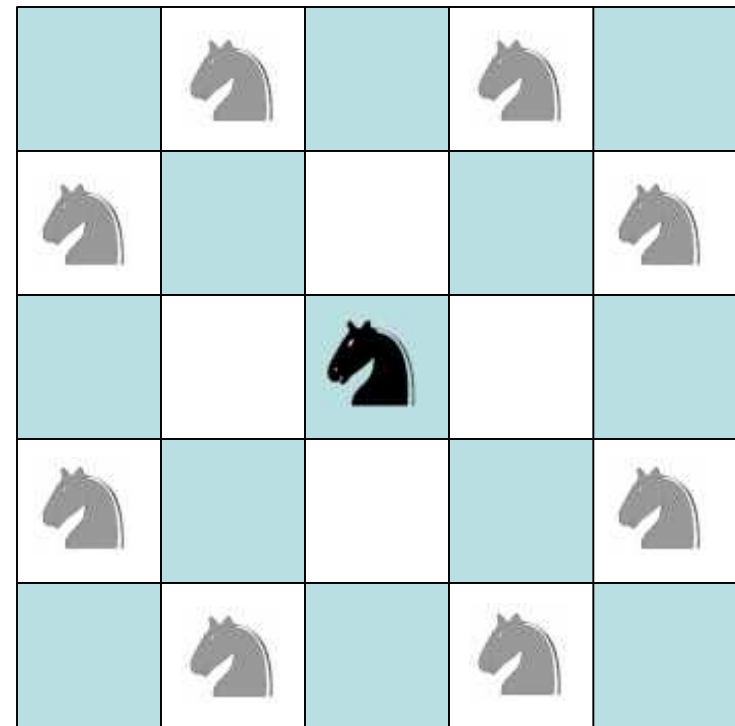
- Precompute values from base case *up* toward solution
- Loosely “non-adaptive”
  - will compute *all* smaller cases, needed or not

# 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 subcases

# Knight Moves

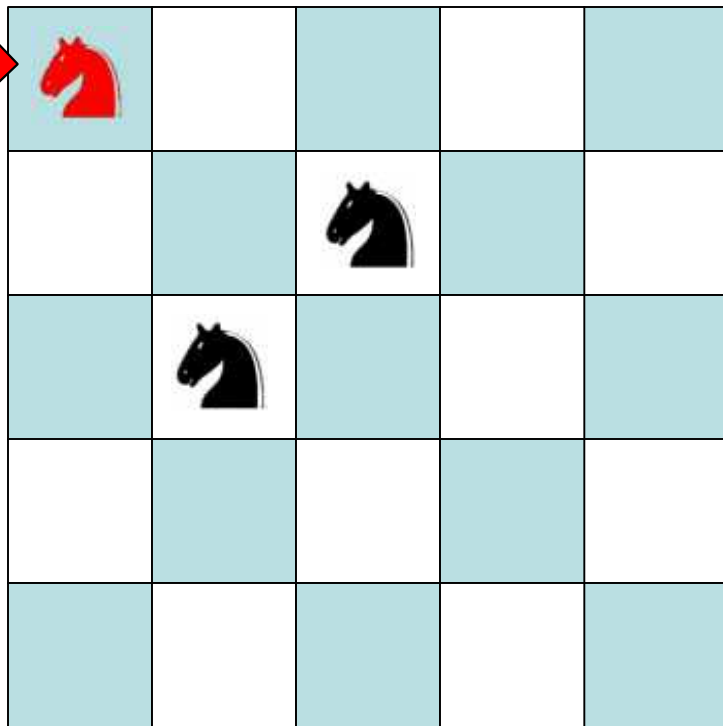
The Knight travels in an “L” fashion, moving 2 spaces in any direction, turning 90° left or right and moving 1 space



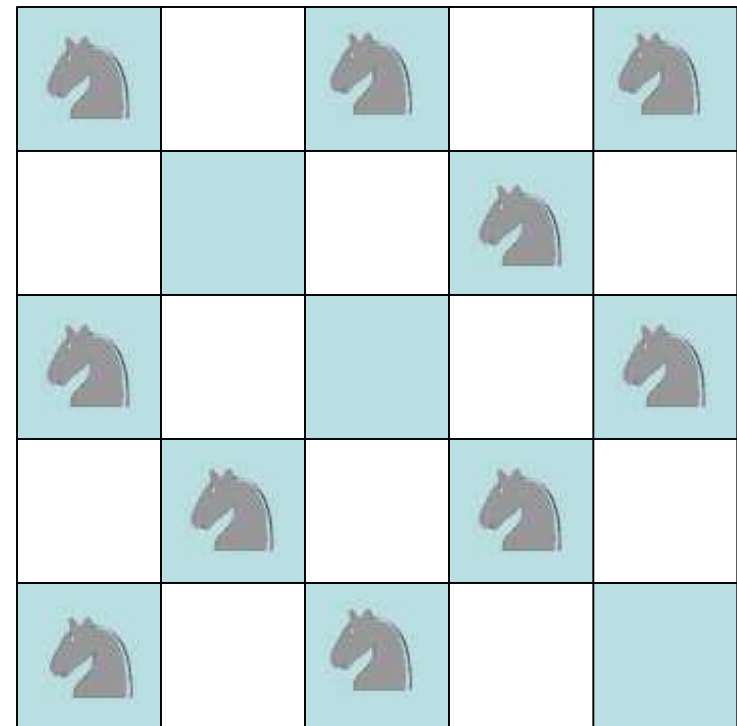
# Knight Moves

- 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







 Possible after 1 move

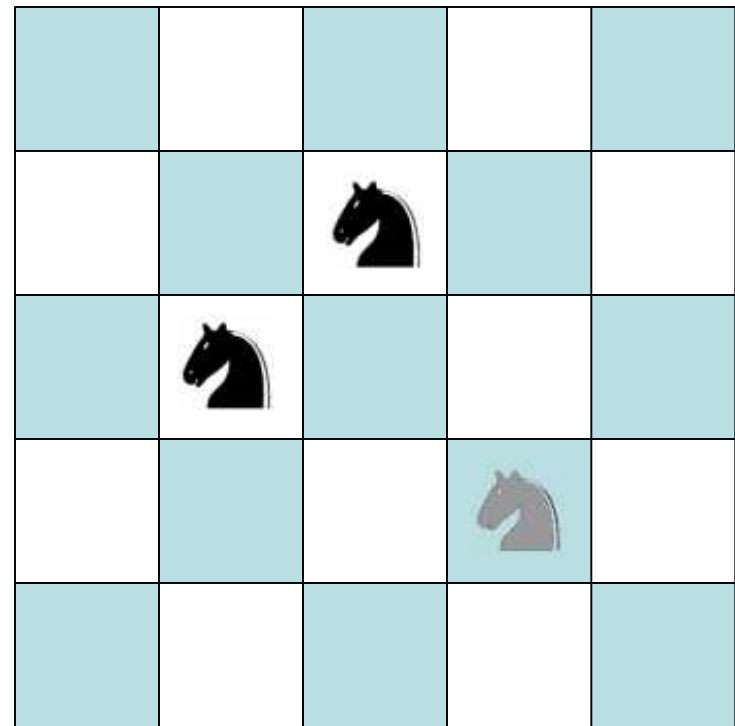


 Possible after 2 moves

# 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


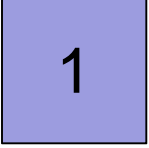
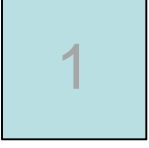
Memo[2]  
(w/ Memo[1]  
in gray)

<b>2</b>		<b>1</b>		<b>1</b>			
		<i>1</i>	<b>1</b>				
<b>1</b>	<i>1</i>			<b>1</b>			
	<b>1</b>		<b>2</b>				
<b>1</b>		<b>1</b>					

Memo[2] only


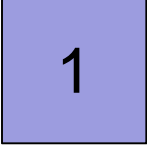
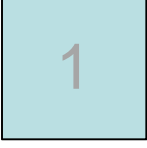
2		1		1			
			1				
1				1			
	1		2				
1		1					

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		<b><u>108</u></b>

# Dynamic Programming 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?
- Was this method top-down or bottom-up?
- Describe the opposite method

# When Good Approaches Go Bad

- When the number of possible function values is too high...
- More than a minor annoyance
- No good solution is known, and it is quite possible that no known solution exists

# 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’