Questions for the day

- How can *we* develop efficient algorithms like MergeSort and QuickSort?
- What are some challenges and shortcomings of this approach?

Introduction to algorithm design

William Hendrix

Lecture 8

Outline

- Algorithm design strategies
  - Brute force
  - Divide-and-conquer
  - Dynamic programming

Algorithm design strategies

- "Templates" for creating algorithms
- Need to be adapted to specific problem
- Give you more tools for approaching difficult problems

- **Strategy 0:** brute force
  - A.k.a., exhaustive search
  - Test all possibilities for the solution
  - Report the correct/best solution

Brute force example

- Brute force sorting

Input: *data*: array to be sorted
Input: n: length of *data*
Output: permutation of *data* such that
$data[1] \leq \ldots \leq data[n]$

Algorithm: BruteSort

repeat
    Find the next permutation of *data* to test
    if *data* is sorted **then**
        return *data*
    end
until no more permutations of *data* are left
error *data* cannot be sorted

Analysis: brute force

Pros

- Applicable to most problems
- *Always* gets the correct/optimum answer
- Easy to design and describe

Cons

- Almost always slowest solution
- Often infeasible
- Exponential or factorial number of tests are impractical for most realistic problems

Divide-and-conquer

- **Goal**
  - Reduce complexity of high-complexity algorithms

- **Outline**
  - Divide large problems into one or more subproblems of roughly the same size
    - E.g., split array into 2 halves, 3 thirds, etc.
  - Solve subproblems via recursion
  - Combine solutions to subproblems into solution for full problem
  - Solve small problems directly (*base case*)

- **Intuition**
  - If combining solutions is easier than solving directly, divide-and-conquer solution may be faster

Divide-and-conquer examples

- General strategy
  - Divide problem into equal parts
  - Solve subproblems recursively
  - Combine solutions
  - Solve small instances directly

- MergeSort
  - Split array in half
  - Recursively sort each half — $2T(n/2)$
  - Merge sorted arrays — $\Theta(n)$
  - Base case: 0 or 1 element

$$T(n) = 2T(n/2) + \Theta(n)$$
$$T(n) = \Theta(n \lg n)$$

Searching

- **Input**
  - *data*: sorted array of length *n*
  - *t*: target value
- **Output:** index *i* such that *data[i]* = *t*, or -1 if $t \notin data$

				t			

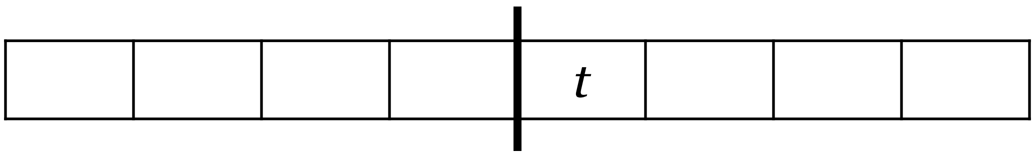
- Brute force
  - Check all values for *t*
  - Return when found
  - Worst-case: O(*n*)
- *Optimization:* stop searching if *data[i]* > *t*
  - Doesn't improve worst case

Input: *data*: sorted array to search
Input: n: length of *data*
Input: t: target value
Output: Index i such that $data[i] = t$, or -1 if $t \notin data$

```
Algorithm: ExhaustiveSearch
for i = 1 to n do
    if data[i] = t then
        return i
    end
end
return −1
```

Naïve binary search

- **Input**
  - *data*: sorted array of length *n*
  - *t*: target value
- **Output:** index *i* such that *data[i]* = *t*, or -1 if $t \notin data$



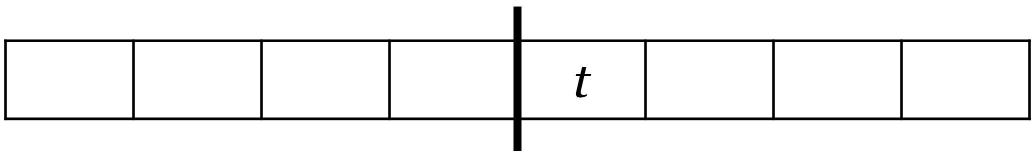
- Split *data* in half
- Search each half
- Return *t* if found
- Base case: array size 1

- $T(n) = 2T(n/2) + \Theta(1)$
  - $T(n) = \Theta(n)$ by Master Theorem
  - No better than linear search!

```
Input: data: sorted array to search
Input: n: length of data
Input: t: target value
Output: Index i such that data[i] = t, or
        −1 if t ∉ data
Algorithm: NaïveBinSearch
if n = 1 then
    if data[1] = t then
        return 1
    else
        return −1
    end
end
mid = ⌊n/2⌋
lhs = NaïveBinSearch(data[1..mid])
rhs = NaïveBinSearch(data[mid + 1..n])
if lhs ≠ −1 then
    return lhs
else if rhs ≠ −1 then
    return mid + rhs
else
    return −1
end
```

Better binary search

- **Input**
  - *data*: sorted array of length *n*
  - *t*: target value
- **Output:** index *i* such that *data[i]* = *t*, or -1 if $t \notin$ *data*



- *Observation*
  - In sorted array, only one half has *t*

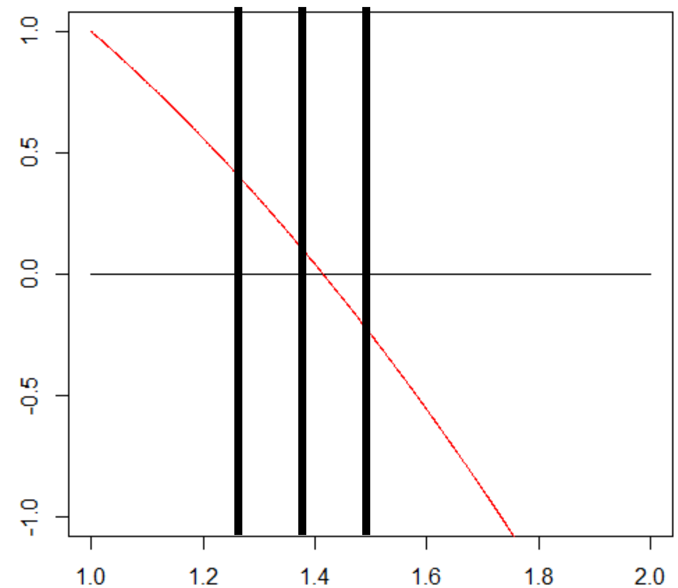
- Reduces number of recursive calls to 1
  - $T(n) = T(n/2) + \Theta(1)$
  - $T(n) = \Theta(\lg n)$ by MT

Input: *data*: sorted array to search
Input: n: length of *data*
Input: t: target value
Output: Index i such that $data[i] = t$, or -1 if $t \notin data$

```
Algorithm: BinSearch
if n = 1 then
    if data[1] = t then
        return 1
    else
        return −1
    end
end
mid = ⌊n/2⌋
if data[mid] = t then
    return mid
else if data[mid] > t then
    return BinSearch(data[1..mid − 1])
else
    return mid + BinSearch(data[mid + 1..n])
end
```

One-sided binary search

- Binary search can be applied to many different types of problems
  - E.g., estimating a root of $2-n^2$ between 1 and 2
    - Guess 1.5 (too high)
    - Guess 1.25 (too low)
    - Guess 1.375 (too low)
    - …
    - Repeat until sufficiently close
- Needs upper and lower bounds



$f(1.25) = 0.4375$ $\quad f(1.5) = -0.25$

- Alternative: one-sided search
  - Start with min value
  - Double until too large
  - Binary search between last success and first failure

One-sided binary search example

- Compute $\lfloor \lg n \rfloor$ for positive n
  - Lower bound?
  - Upper bound?

- Exhaustive solution
  - Try every power of 2 until you exceed *n*
  - $\Theta(\lg n)$ time

- One-sided binary search
  - *Double* power of 2 until too high
  - Binary search between last success and first failure
  - $\Theta(\lg \lg n)$ time

```
Input: n: positive integer
Output: ⌊lg n⌋
Algorithm: LinearScanLog
f = 1
x = 0
while f < n do
    f = 2f
    x = x + 1
end
return x
```

```
Input: n: positive integer
Output: ⌊lg n⌋
Algorithm: OneSidedLog
lo = 0
hi = 1
while 2^hi < n do
    lo = hi
    hi = 2hi
end
while hi − lo > 1 do
    mid = ⌊(hi − lo)/2⌋
    if 2^mid < n then
        hi = mid
    else
        lo = mid
    end
end
return lo
```

Divide-and-conquer analysis

- **Pros**
  - Reduces complexity for several problems
  - Easy to prove correctness via strong induction
  - Easy to analyze with Master Theorem
  - Works very well with parallel computing

- **Cons**
  - Problem must exhibit *optimal substructure*
    - Solution must be related to subproblem solutions
  - Sometimes has poor complexity even with optimal substructure
    - *Dynamic programming* solves this problem

Divide-and-conquer exercise

- Develop an efficient algorithm for *natural number exponentiation*
- **Input**
  - a: base of exponent (real number)
  - n: exponent (nonnegative integer)
- **Output:** a^n

- Divide problem into equal parts
- Solve subproblems recursively
- Combine solutions
- Solve small instances directly

- *Hint:* $a^n = \underbrace{a \cdot a \cdot a \cdots a}_{n \text{ times}}$
  - Focus on even n first
  - Think about how to handle odd n

Divide-and-conquer solution

- Develop an efficient algorithm for *natural number exponentiation*
- **Input**
  - a: base of exponent (real number)
  - n: exponent (nonnegative integer)
- **Output:** a^n

Input: a: base of exponent (real number)
Input: n: exponent (nonnegative integer)
Output: a^n
Algorithm: QuickPow
if $n = 0$ **then**
    return 1
end
$t = \text{QuickPow}(a, \lfloor n/2 \rfloor)$
if n is even **then**
    return t^2
else
    return at^2
end

Divide-and-conquer application

- Consider matrix addition and multiplication:
  - Given n-by-n matrices A and B, compute the sum $C = A$ + B
  - Given an n-by-n matrices A and B, compute the product $C = AB$
- Addition
  - Compute $c_{ij} = a_{ij} + b_{ij}$, for every cij
  - Complexity: $\Theta(1)\Theta(n^2) = \Theta(n^2)$
- Multiplication
  - Compute c_{ij} = sum-product of row i of A and column j of B
    - $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$
  - Complexity: $\Theta(n)\Theta(n^2) = \Theta(n^3)$
  - Fastest known algorithm until 1969

Strassen's Algorithm

- Described by Volker Strassen (1969)
- Used divide-and-conquer to reduce complexity of matrix multiplication
- Divide matrices into 4 quadrants:

$$\begin{array}{|c|c|}\hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} \times \begin{array}{|c|c|}\hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array} = \begin{array}{|c|c|}\hline C_{11} & C_{12} \\ \hline C_{21} & C_{22} \\ \hline \end{array}$$

- Compute the following:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

- Then:

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Strassen's complexity

- Strassen's equations:

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

  - Total complexity: $T(n) = 7T(n/2) + \Theta(n^2)$
    - 18 additions/subtractions $\Theta(n^2)$
    - 7 multiplications $7T(n/2)$
- c = $\log_2(7)$ ≈ 2.81
- $f(n)$ vs. $n^{2.81} \Rightarrow f(n) = O(n^{\lg 7 - 0.8})$
- $T(n) = \Theta(n^c) \approx \Theta(n^{2.81})$

Dynamic programming

- Algorithm design strategy
- Similar to divide-and-conquer
- Useful when divide-and-conquer would be inefficient

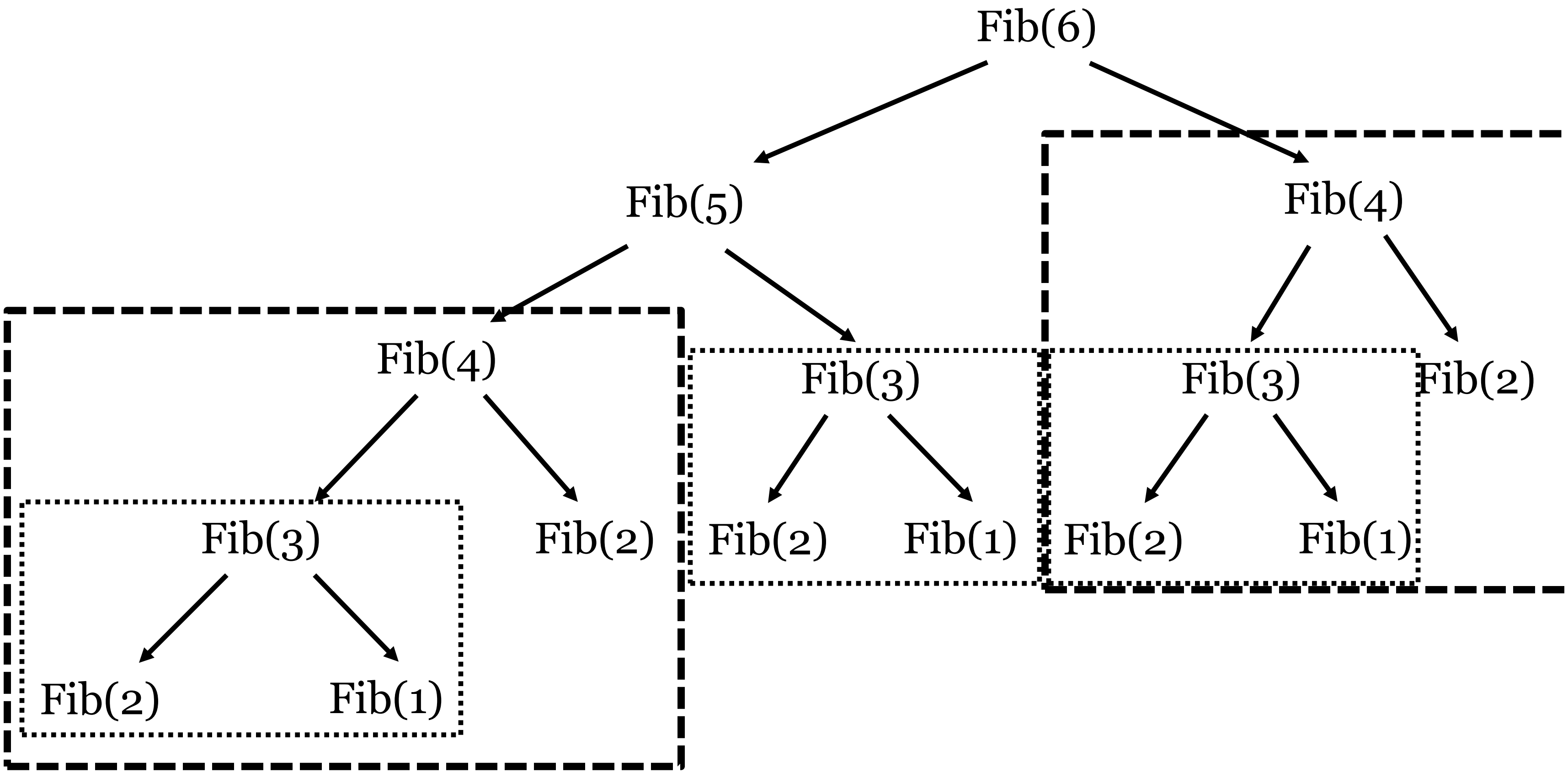
Dynamic programming motivation

$$F(n) = \begin{cases} 1, & \text{if } n = 1, 2 \\ F(n-1) + F(n-2), & \text{if } n > 2 \end{cases}$$

```
Input: n: index of Fibonacci number to compute
Output: F_n
Algorithm: Fibonacci
if n = 1 or 2 then
  | return 1
else
  | return Fibonacci(n − 1) + Fibonacci(n − 2)
end
```

- You've probably been told that this function is inefficient
- **Review:** recursive complexity
- $T(n) = T(n\text{-}1) + T(n\text{-}2) + \Theta(1)$
- Complexity: $\Theta(\tau^n)$, where $\tau = \frac{1+\sqrt{5}}{2} \approx 1.61$
  - Calculating F_{1000} takes forever!

Why does this take so long?



- We’re computing the same sub-results over and over!
- **Intuition:** save sub-results in a map

A better solution

-1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7

Input: n: index of Fibonacci number to compute
Output: F_n

Algorithm: FastFib
$fib = \text{Array}(n)$
Initialize fib to -1
return DynamicFib(n)

Algorithm: DynamicFib(n)
if $fib[n] = -1$ **then**
    if $n = 1$ or 2 **then**
        $fib[n] = 1$
    else
        $fib[n] = \text{DynamicFib}(n-1) + \text{DynamicFib}(n-2)$
    end
end
return $fib[n]$

A better solution

1	1	2	3	5	8	13
1	2	3	4	5	6	7

Input: n: index of Fibonacci number to compute
Output: F_n

```
Algorithm: FastFib
fib = Array(n)
Initialize fib to −1
return DynamicFib(n)
```

```
Algorithm: DynamicFib(n)
if fib[n] = −1 then
    if n = 1 or 2 then
        fib[n] = 1
    else
        fib[n] = DynamicFib(n − 1) +
          DynamicFib(n − 2)
    end
end
return fib[n]
```

- Each Fibonacci number computed *once*
- Total complexity: $\Theta(n)$

Dynamic programming

- General algorithm design strategy
- Recursion augmented with data structure to save sub-results
  - Not always a 1D array…
    - Depends on number of parameters
- **Useful when:**
  - Problem has recursive structure (*like divide-and-conquer*)
  - More than one recursive call
  - Problems overlap
    - Typically, 1-2 levels deep in recursion tree
- **Related concept:** memoization
  - Function that stores sub-results
  - Returns immediately when calling with previously-seen value
  - *Note:* possible to memoize non-recursive functions
    - Dynamic programming can be implemented iteratively

A more complex example

- Binomial coefficients ("*n* choose *k*")
  - **Formula:** $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
  - **Example:** $$\begin{aligned}\binom{20}{3} &= \frac{20!}{3!(17!)} \\ &= \frac{2432902008176640000}{6(355687428096000)} \\ &= 1140\end{aligned}$$

- Formula might cause overflow
- Alternative: Pascal's triangle

$k \longrightarrow$

$n \downarrow$

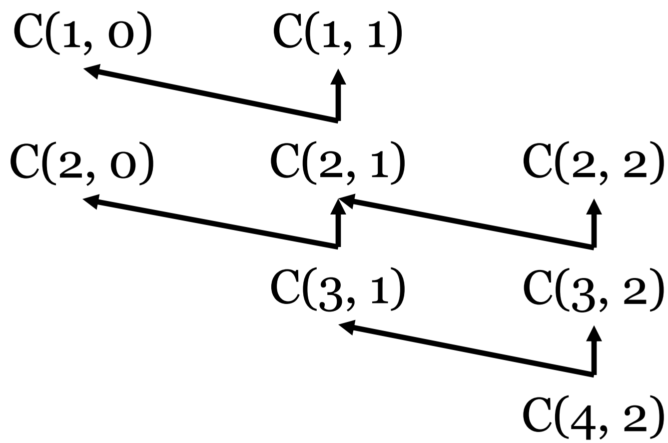
$$\begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 1 & & & \\ & & 1 & & 2 & & 1 & & \\ & 1 & & 3 & & 3 & & 1 & \\ 1 & & 4 & & 6 & & 4 & & 1 \end{array}$$

$$\binom{n}{k} = \begin{cases} 1, \text{ if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} \end{cases}$$

Binomial coefficients example

- Recursive function
- **Example:** C(4, 2)

$$\binom{n}{k} = \begin{cases} 1, \text{ if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} \end{cases}$$



- Overlapping subproblems
  - Dynamic programming!

Binomial coefficients example

- Recursive function
- **Example:** C(4, 2)

$$\binom{n}{k} = \begin{cases} 1, \text{ if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} \end{cases}$$

	0	1	2
0			
1	C(1, 0)	C(1, 1)	
2	C(2, 0)	C(2, 1)	C(2, 2)
3		C(3, 1)	C(3, 2)
4			C(4, 2)

- Overlapping subproblems
  - Dynamic programming!
- Data structure?
  - 2D array (map)

Binomial coefficients example

Recurrence: $$\binom{n}{k} = \begin{cases} 1, \text{ if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} \end{cases}$$

Algorithm: BasicBinom
if $k = 0$ or $k = n$ **then**
  return 1
else
  return BasicBinom$(n-1, k-1)$ + BasicBinom$(n-1, k)$
end

- Using 2D array:

Algorithm: AlmostMemo
if we've already calculated $binom[n, k]$ **then**
  return $binom[n, k]$
else if $k = 0$ or $k = n$ **then**
  $binom[n, k] = 1$
else
  $binom[n, k] =$ AlmostMemo$(n-1, k-1)$ + AlmostMemo$(n-1, k)$
end
return $binom[n, k]$

Algorithm: MemoBinom
if $binom[n, k] > 0$ **then**
  return $binom[n, k]$
else if $k = 0$ or $k = n$ **then**
  $binom[n, k] = 1$
else
  $binom[n, k] =$ MemoBinom$(n-1, k-1)$ + MemoBinom$(n-1, k)$
end
return $binom[n, k]$

- Final answer

Input: n, k: binomial coefficient to compute
Output: $\binom{n}{k}$
Algorithm: DPBinom
$binom = \text{Array}(n, k)$
Initialize $binom$ to 0
return MemoBinom(n, k)

Memoization summary

- Five step procedure

1. **Start with naïve recursive algorithm**
   - Based on recurrence
2. **Decide data structure and sentinel value**
3. **Add "memoization check" to the beginning of the algorithm**
   - If solution is in data structure, return it
4. **Store solution before returning**
5. **Write wrapper function to initialize data structure**

$$\binom{n}{k} = \begin{cases} 1, \text{ if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} \end{cases}$$

Input: n, k: binomial coefficient to compute
Output: $\binom{n}{k}$
Algorithm: DPBinom
$binom = \text{Array}(n, k)$
Initialize $binom$ to 0
return MemoBinom(n, k)

Algorithm: MemoBinom
if $binom[n, k] > 0$ **then**
  return $binom[n, k]$
else if $k = 0$ or $k = n$ **then**
  $binom[n, k] = 1$
else
  $binom[n, k] = \text{MemoBinom}(n-1, k-1) + \text{MemoBinom}(n-1, k)$
end
return $binom[n, k]$

Coming up

- Dynamic programming
- Greedy algorithms

- **Recommended readings:** Sections 11.3, 11.2, 12.1, 12.2, 12.5
  - *Practice problems:* R-11.3, C-11.3 (complexity only), C-11.4, A-11.2, R-12.9, C-12.6, C-12.7, C-12.9, C-12.10, A-12.1, A-12.2, A-12.5