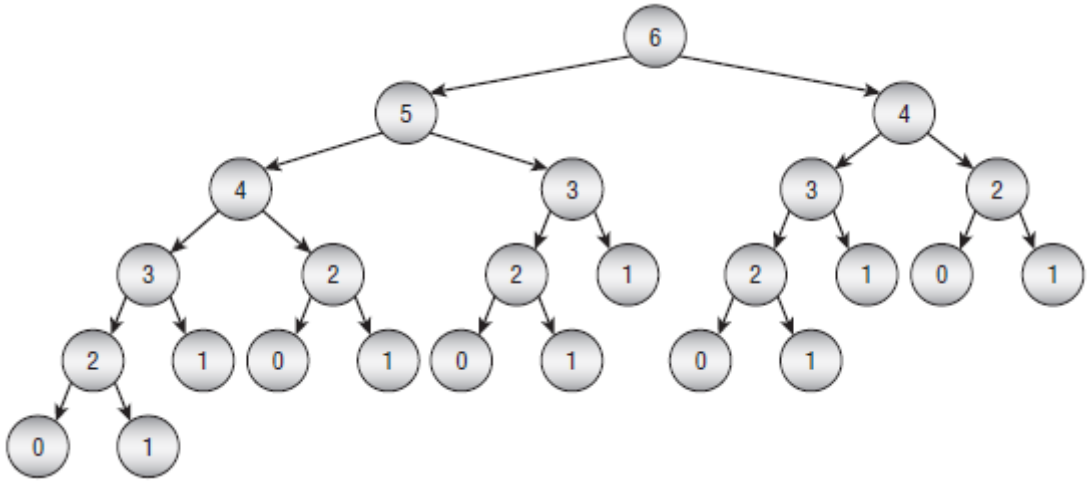
Recursion, Part 2



Agenda

- Backtracking Algorithms
- Selections and Permutations
- Recursion Removal
- Summary
- Exercises

Backtracking Algorithms

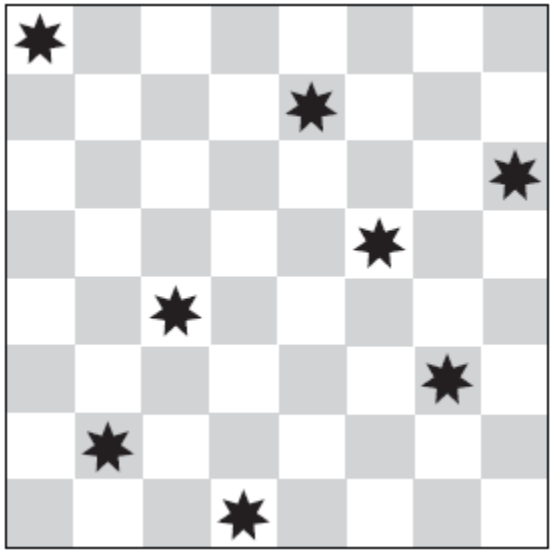
- At each step, extend the test solution
- If the test solution is a final solution, save it
- If the extension make a solution impossible, backtrack to the previous test solution

Backtracking Pseudocode

```
Boolean: LeadsToSolution(Solution: test_solution)
    // If the partial solution cannot lead to a full solution, return false.
    If <test_solution cannot solve the problem> Then Return false

    // If this is a full solution, return true.
    If <test_solution is a full solution> Then Return true

    // Extend the partial solution.
    Loop <over all possible extensions to test_solution>
        <Extend test_solution>

        // Recursively see if this leads to a solution.
        If (LeadsToSolution(test_solution)) Then Return true

        // This extension did not lead to a solution. Undo the change.
        <Undo the extension>
    End Loop

    // If we get here, this partial solution cannot
    // lead to a full solution.
    Return false
End LeadsToSolution
```

Eight Queens Problem

- Position 8 queens so none can attack the others



Eight Queens Problem

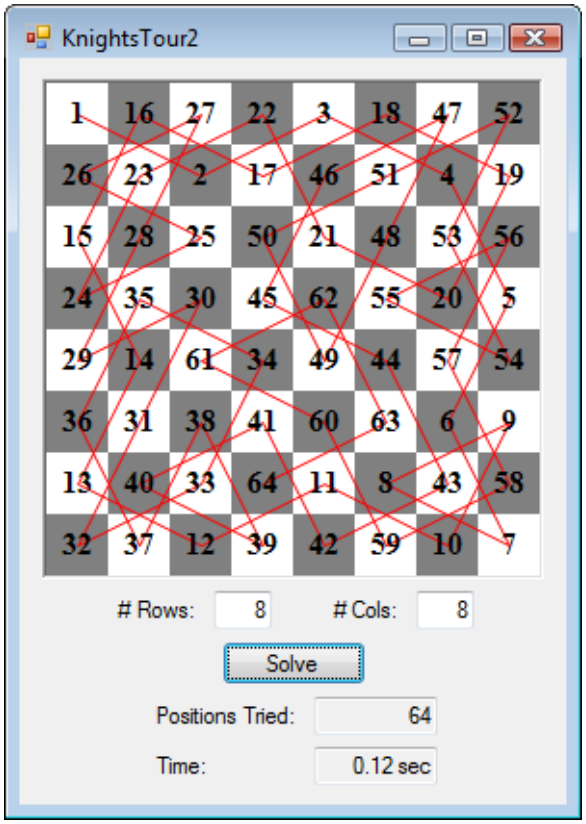
$\binom{64}{8}$ = 4,426,165,368 possible arrangements

Recall: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

- Backtracking removes almost 1.3 billion arrangements without considering them

Knight's Tour

- Make a knight visit every square on the board



- For a closed tour, return to the starting point

Selections and Permutations

- Selection (or combination) – An unordered subset

  All two-item selections of the set {A, B, C} are:

  {A, B} {A, C} {B, C}

- Permutation - An ordered subset

  All two-item permutations of the set {A, B, C} are:

  (A, B) (A, C) (B, A) (B, C) (C, A) (C, B)

With Duplicates

- Selections with duplicates

  All two-item selections with duplicates of {A, B, C} are:

  {A, A} {A, B} {A, C} {B, B}, {B, C} {C, C}

- Permutations with duplicates

  All two-item permutations with duplicates of {A, B, C} are:

  (A, A) (A, B) (A, C) (B, A) (B, B) (B, C) (C, A) (C, B) (C, C)

Selections With Loops

```
// Generate selections of 3 items allowing duplicates.
List<string>: Select3WithDuplicates(List<string> items)
    List<string>: results = New List<string>
    For i = 0 To <Maximum index in items>
        For j = i To <Maximum index in items>
            For k = j To <Maximum index in items>
                results.Add(items[i] + items[j] + items[k])
            Next k
        Next j
    Next i
    Return results
End Select3WithDuplicates
```

Selections Without Duplicates

```
// Generate selections of 3 items without allowing duplicates.
List<string>: Select3WithoutDuplicates(List<string> items)
    List<string>: results = new List<string>()
    For i = 0 To <Maximum index in items>
        For j = i + 1 To <Maximum index in items>
            For k = j + 1 To <Maximum index in items>
                results.Add(items[i] + items[j] + items[k])
            Next k
        Next j
    Next i
    Return results
End Select3WithoutDuplicates
```

Recursive Selections

- The problem with the preceding code is that it requires you to know how many items to select
- For a more general solution, use recursion
- Pass an array or list to hold results into the algorithm

Recursive Selection Code

```
// Generate combinations allowing duplicates.
SelectKofNwithDuplicates(Integer: index, Integer: selections[],
  Data: items[], List<List<Data>> results)
    // See if we have made the last assignment.
    If (index == <Length of selections>) Then
        // Add the result to the result list.
        List<Data> result = New List<Data>()
        For i = 0 To <Largest index in selections>
            result.Add(items[selections[i]])
        Next i
        results.Add(result)
    Else
        // Get the smallest value we can use for the next selection.
        Integer: start = 0 // Use this value if this is the first index.
        If (index > 0) Then start = selections[index - 1]
        // Make the next assignment.
        For i = start To <Largest index in items>
            // Add item i to the selection.
            selections[index] = i
            // Recursively make the other selections.
            SelectKofNwithDuplicates(index + 1, selections, items, results)
        Next i
    End If
End SelectKofNwithDuplicates
```

Recursive Permutation Code

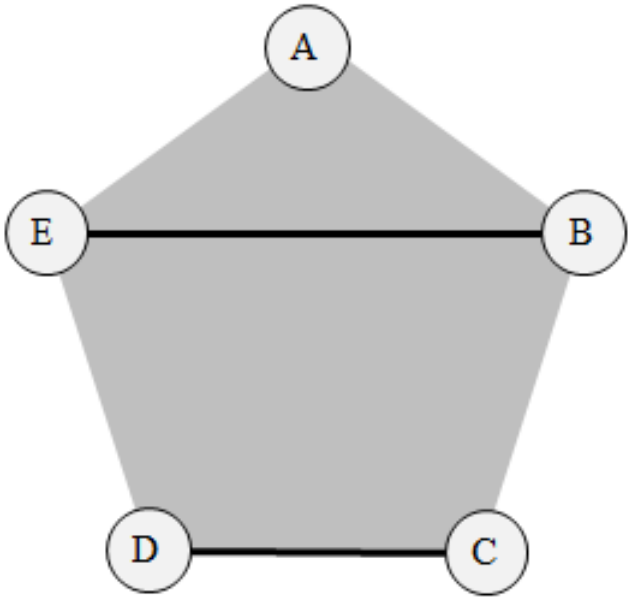
```
// Generate permutations allowing duplicates.
PermuteKofNwithDuplicates(Integer: index, Integer: selections[],
  Data: items[], List<List<Data>> results)
    // See if we have made the last assignment.
    If (index == <Length of selections>) Then
        // Add the result to the result list.
        List<Data> result = New List<Data>()
        For i = 0 To <Largest index in selections>
            Result.Add(items[selections[i]])
        Next i
        results.Add(result)
    Else
        // Make the next assignment.
        For i = 0 To <Largest index in items>
            // Add item i to the selection.
            selections[index] = I

            // Recursively make the other assignments.
            PermuteKofNwithDuplicates(index + 1, selections, items, results)
        Next i
    End If
End PermuteKofNwithDuplicates
```

Counting Selections and Permutations

Picking k of n items:

- Selections with duplicates: $\binom{n+k-1}{k}$
- Selections without duplicates: $\binom{n}{k}$
- Permutations with duplicates: $n \times n \times \ldots \times n = n^k$
- Permutations without duplicates: $n \times (n-1) \times (n-2) \times \ldots \times (n-k+1)$

Round Robin Scheduling



Recursion Removal

- Recursion may exhaust the stack space

Tail Recursion

- Occurs when recursion is the last thing the algorithm does

```
Integer: Factorial(Integer: n)
    If (n == 0) Then Return 1
    Integer: result = n * Factorial(n - 1)
    Return result
End Factorial
```

Tail Recursion Removal

- Replace recursion with a loop that prepares for “recursion”

```
Integer: Factorial(Integer: n)
    // Make a variable to keep track of the returned value.
    // Initialize it to 1 so we can multiply it by returned results.
    // (The result is 1 if we do not enter the loop at all.)
    Integer: result = 1
    // Start a loop controlled by the recursion stopping condition.
    While (n != 0)
        // Save the result from this “recursive” call.
        result = result * n
        // Prepare for “recursion.”
        n = n - 1
    Loop
    // Return the accumulated result.
    Return result
End Factorial
```

Dynamic Programming
(Storing Intermediate Values)

- The Fibonacci algorithm contains many repeated values

```
// Calculated values.
Integer: FibonacciValues[100]

// The maximum value calculatued so far.
Integer: MaxN

// Set Fibonacci[0] and Fibonacci[1].
InitializeFibonacci()
    FibonacciValues[0] = 0
    FibonacciValues[1] = 1
    MaxN = 1
End InitializeFibonacci
```

Using Intermediate Values

```
// Return the nth Fibonacci number.
Integer: Fibonacci(Integer: n)
    // If we have not yet calculated this value, calculate it.
    If (MaxN < n) Then
        FibonacciValues[n] = Fibonacci(n - 1) + Fibonacci(n - 2)
        MaxN = n
    End If

    // Return the calculated value.
    Return FibonacciValues[n]
End Fibonacci
```

Bottom-Up Programming
(Intermediate Values on the Fly)

```
// Return the nth Fibonacci number.
Integer: Fibonacci(Integer: n)
    If (n > MaxN) Then
        // Calculate values between Fibonacci(MaxN) and Fibonacci(n).
        For i = MaxN + 1 To n
            FibonacciValues[i] = Fibonacci(i - 1) + Fibonacci(i - 2)
        Next i
        // Update MaxN.
        MaxN = n
    End If

    // Return the calculated value.
    Return FibonacciValues[n]
End Fibonacci
```

General Recursion Removal

- Mimic the behavior of the computer during recursion
- To recurse:
  - Push values into a stack
  - Prepare variables for recursion
  - Loop to the beginning of the recursive method
- To end recursion:
  - Pop values off of the stack

Summary

- Backtracking Algorithms
  - Eight Queens Problem
  - Knight's Tour
- Selections and Permutations
  - Selections
  - Permutations
  - Counting Selections and Permutations
  - Round Robin Scheduling

Summary (continued)

- Recursion Removal
  - Tail Recursion
  - Dynamic Programming
  - Bottom-Up Programming
  - General Recursion Removal

Exercises

- Chapter 9 Exercises 17 – 23.
- Do either:
  - Chapter 9 Exercises 12 – 14, or
  - Chapter 9 Exercises 15 – 16.
- Bonus: Chapter 9 Exercise 24.
- Read *Essential Algorithms, 2e* Chapter 10 pages 285 – 316. (Stop before the section "Threaded Trees.")