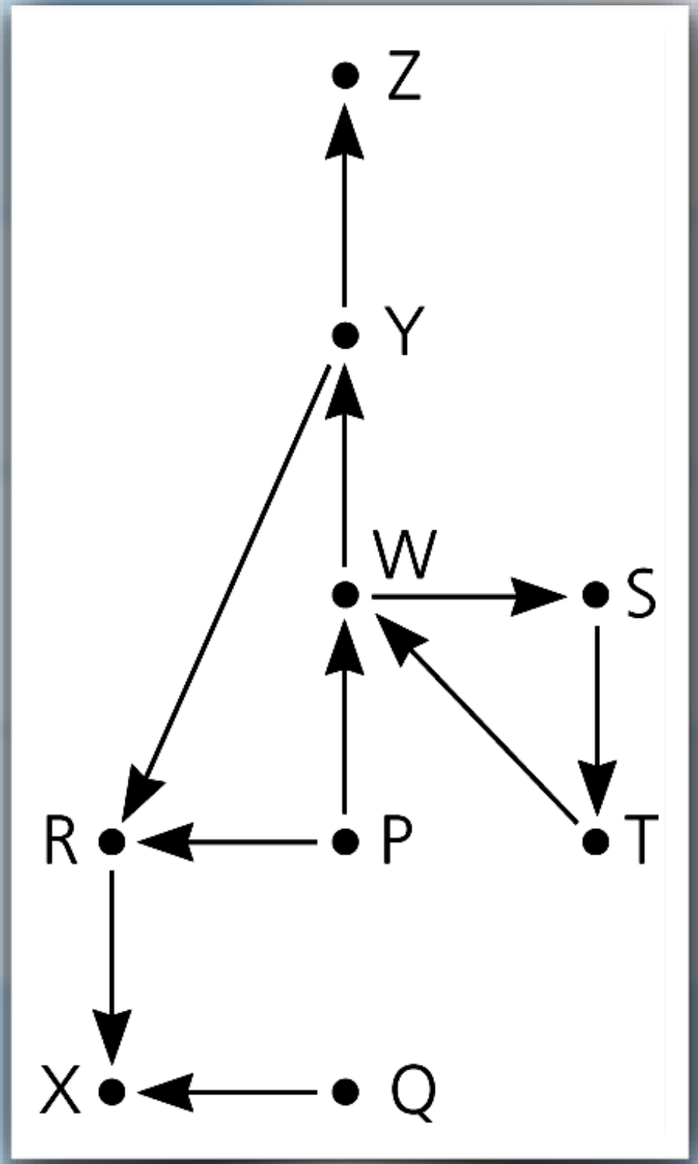
BACKTRACKING

Backtracking

- **Searching for a solution and backing up when an impasse (dead-end) is reached**
  - Depth-first Search
- **Recursion and backtracking can be combined to solve problems**
  - Searching Labyrinth for Item
  - Search for best airline route between cities
    - Beginning at the origin city, try every possible sequence of flights until either
      - A sequence that gets to the destination city is found
      - Can determine that no such sequence exists
    - Backtracking can be used to recover from choosing a wrong city

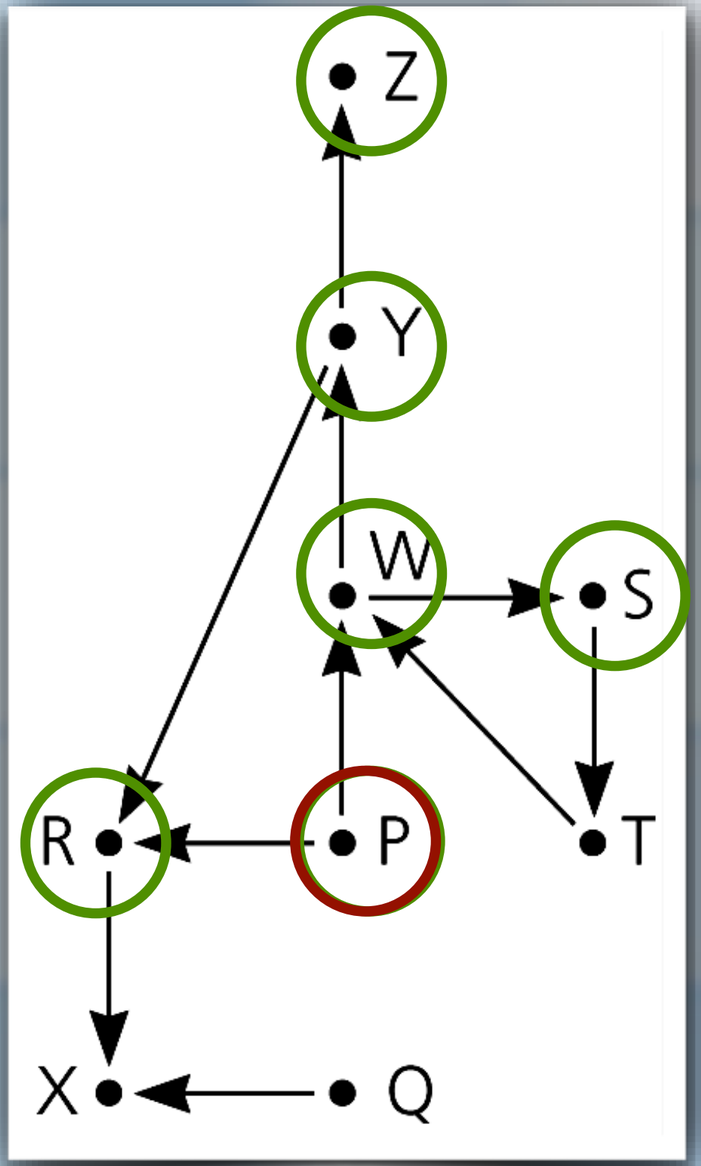
Backtracking

- **High Planes Airline Company (HPAir)**
  - Goal:
    - Indicate whether a sequence of HPAir flights exists from the origin city to the destination city
    - Can we fly from P to Z?
- **The flight map for HPAir is a graph**
  - Adjacent vertices are two vertices that are joined by an edge
  - A directed path is a sequence of directed edges



Backtracking

- **Recursive Search Problem**
  - Possible outcomes of the recursive search strategy
    - You eventually reach the destination city and can conclude that it is possible to fly from the origin to the destination
    - You reach a city, **X**, from which there are no departing flights
    - You go around in circles
      - Mark visited cities to prevent cycles



Backtracking

- **Recursive Search Strategy**

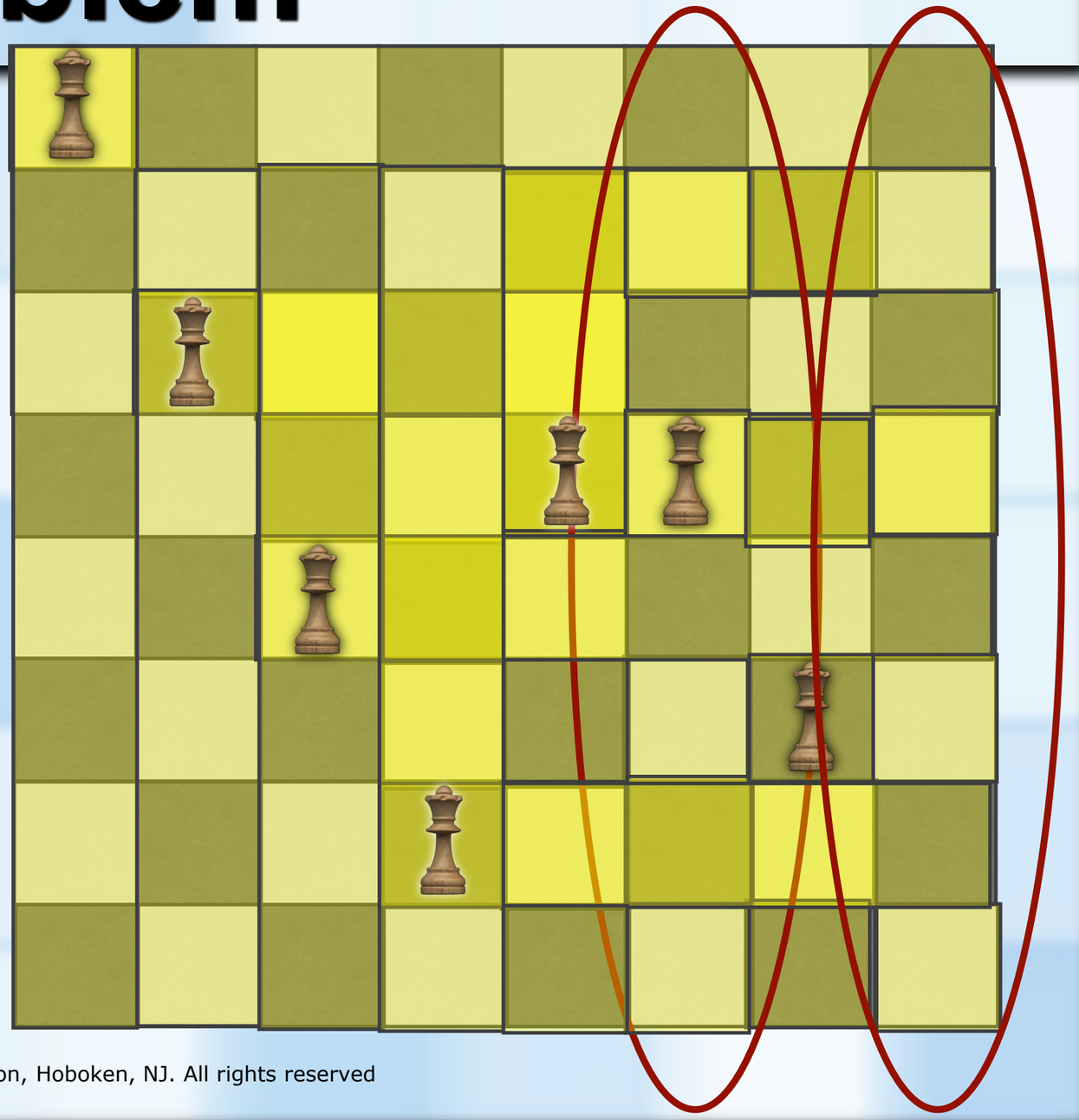
```
// Discovers whether a sequence of flights
// from origin City to destination City exists.
searchR(originCity: City, destinationCity: City): boolean
     Mark originCity as visited
     if (originCity is destinationCity)
            Terminate—the destination is reached
     else
            for (each unvisited city C adjacent to originCity)
                  searchR(C, destinationCity)
```

Backtracking

- **Eight-Queens Problem**
  - Place eight queens on the chessboard so that no queen can attack any other queen
  - One strategy: Guess at a solution
    - There are **4,426,165,368** ways to arrange 8 queens on a chessboard of 64 squares
  - An observation to eliminate many arrangements
    - No queen can reside in a row or a column that contains another queen
    - Only **40,320** or **8!** arrangements of queens to be checked for attacks along diagonals

Eight Queens Problem

- **Searching strategy**
  - Place queens one column at a time
  - If you reach an impasse, backtrack to the previous column
- **Base case**
  - If there are no more columns to consider
    - You're finished (do nothing - implicit base case)
- **Recursive step**
  - If you successfully place a queen in the current column
    - Consider the next column
  - If you cannot place a queen in the current column
    - You need to **backtrack**



PROCESSING EXPRESSIONS

Recognizing Palindromes

- **Palindrome**
  - A string that reads the same from left to right as it does from right to left
- **Language**
  - Palindromes = {***w*** : ***w*** reads the same left to right as right to left}
- **Grammar**
  - **<char>** rule is the base case for success

```
<palindrome> = empty string | <char> | a <palindrome> a |
                b  <palindrome> b | …| Z <palindrome> Z

<char> = a | b | … | z | A | B | … | Z
```

Recognizing Palindromes

- **Recognition algorithm**

```
isPal(in w:string):boolean
   if (w is the empty string or w is of length 1)
        return true

   else if (w's first and last characters are the same letter )
        return isPal(w minus its first and last characters)

   else
        return false
```

```
<palindrome> = empty string | <char> | a <palindrome> a |
                b  <palindrome> b | …| Z <palindrome> Z

<char> = a | b | … | z | A | B | … | Z
```

Algebraic Expressions

- **Infix expressions**
  - An operator appears between its operands
    - `a + b`
- **Prefix expressions**
  - An operator appears before its operands
    - `+ a b`
- **Postfix expressions**
  - An operator appears after its operands
    - `a b +`
- **Advantages of prefix and postfix expressions**
  - No precedence rules
  - No association rules
  - No parentheses
  - Simple grammars
  - Straightforward recognition and evaluation algorithms

Algebraic Expressions

- **To convert a fully parenthesized infix expression to a prefix form**
  - Move each operator to the position marked by its corresponding open parenthesis
  - Remove the parentheses
    - Infix expression: `( ( a + b) * c )`
    - Prefix expression: `* + a b c`
- **To convert a fully parenthesized infix expression to a postfix form**
  - Move each operator to the position marked by its corresponding closing parenthesis
  - Remove the parentheses
    - Infix expression: `( ( a + b) * c )`
    - Postfix expression: `a b + c *`

Prefix Expressions

- **A recursive recognition algorithm**
  - **Base case:** Single lowercase letter `<identifier>`
  - **Recursive:** `<operator> <prefix> <prefix>`

```
<prefix> = <identifier> | <operator> <prefix> <prefix>
<operator> = + | - | * | /
<identifier> = a | b | … | z
```

Prefix Expressions

```
evaluatePrefix(in strExp:string):float
   ch  = first character of expression strExp
   Remove first character from strExp
   if (ch is an identifier)
      return value of the identifier
   else if (ch is an operator named op)
   {
      operand1 = evaluatePrefix(strExp)
      operand2 = evaluatePrefix(strExp)
      return operand1 op operand2
   }
```

Recursively Processing Arrays

RECURSIVE LINEAR SEARCH


```
int linearSearch(string target, string[] theStrings,
                 int first, int last)
{
  int result = -1;

  if (first > last)
    result = -1;

  else if (target
    result = firs

  else
    result = line                              rst + 1, last);

  return result;
} // end linearSearch

int linearSearch(string target, string[] theStrings, int size)
{
  return linearSearch (target, theStrings, 0, size-1);
} // end linearSearch
```

We want to return the index of where the item is found.

Pearson

RECURSIVE BINARY SEARCH

Mark
?

first 0

mid 3

last 6

Chris 0

Derek 1

Josh 2

Kirk 3

Mark 4

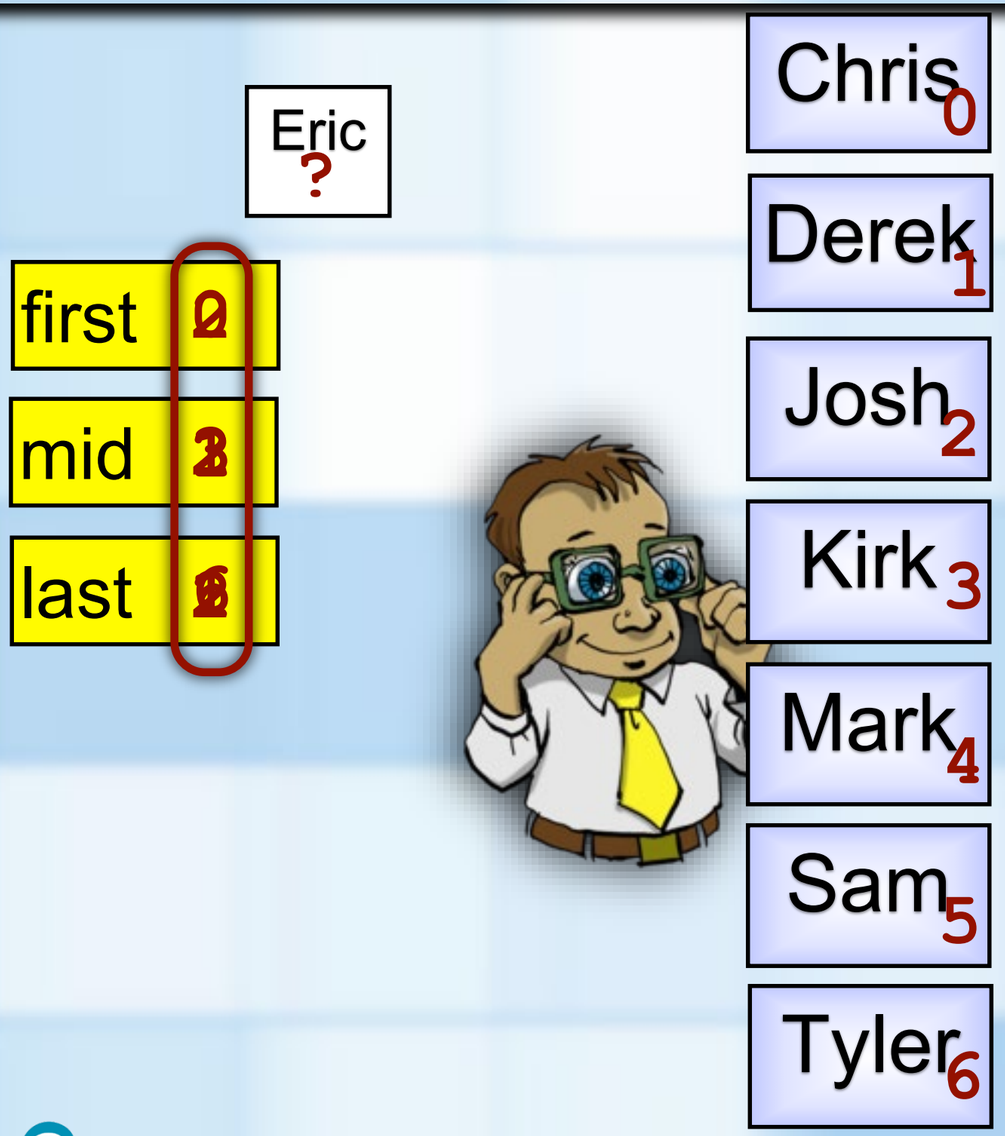
Sam 5

Tyler 6

```
int binarySearch(string target, String[] strings, int size)
{
  return binarySearch(target, strings, 0, size - 1);
} // end binarySearch

int binarySearch(string target, string[] strings,
                                        int first, int last)
{
  int result;
  int mid = first + (last - first) / 2;

  if(first > last)
    result = -1; // target not in array                    BASE CASE

  else if (target == strings[mid])
    result = mid; // target found at a[mid]                BASE CASE

  else if(target < strings[mid])
    result = binarySearch(target, strings, first, mid - 1);

  else
    result = binarySearch(target, strings, mid + 1, last);

  return result;

} // end binarySearch
} // end SortedStringSearcher
```

Pearson

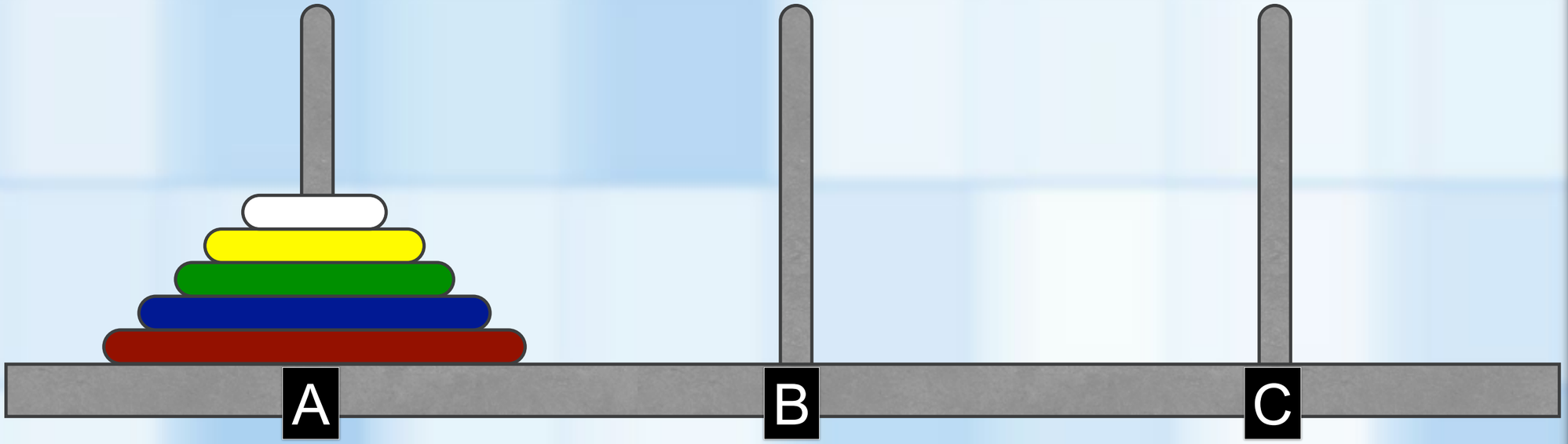
Recursive Binary Search




```
int binarySearch(string target, String[] strings)
{
  return binarySearch(target, strings, 0, strings.length - 1);
} // end binarySearch

int binarySearch(string target, string[] strings,
                                          int first, int last)
{
  int result;
  int mid = first + (last - first) / 2;

  if(first > last)
    result = -1; // target not in array

  else if (target == strings[mid])
    result = mid; // target found at a[mid]

  else if(target < strings[mid])
    result = binarySearch(target, strings, first, mid - 1);

  else
    result = binarySearch(target, strings, mid + 1, last);

  return result;

 } // end binarySearch
} // end SortedStringSearcher
```

Towers of Hanoi

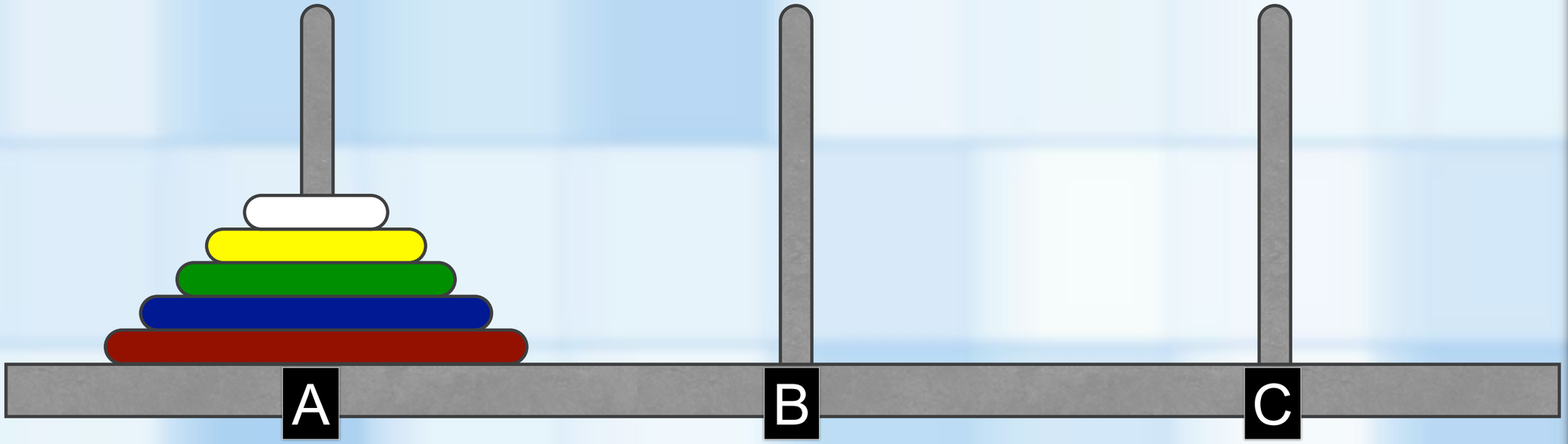
- Move the disks from post A to post C
- Can only move one disk at a time
- Cannot place a disk on one smaller than itself



Pearson

Towers of Hanoi

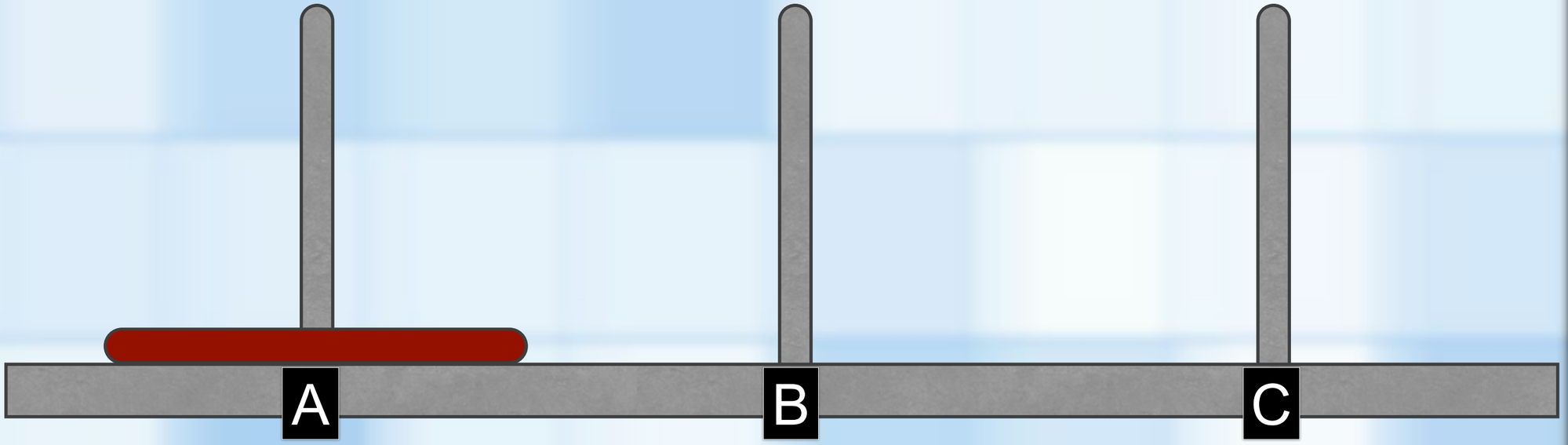
- **Move the disks from post A to post C**
  - **post A** - Source (`src`)
  - **post B** - Spare (`spare`)
  - **post C** - Destination (`dst`)
- `solveTowers(count, src, dst, spare)` `// signature`
- `solveTowers(5, A, C, B)` `// example`



Pearson

Towers of Hanoi

- **Move the disks from post A to post C**
- **Base Case**
  - ***One Disk*** - we can move it and not violate the rules
- `if (count == 1)`
  - `moveDisk(src, dst)`



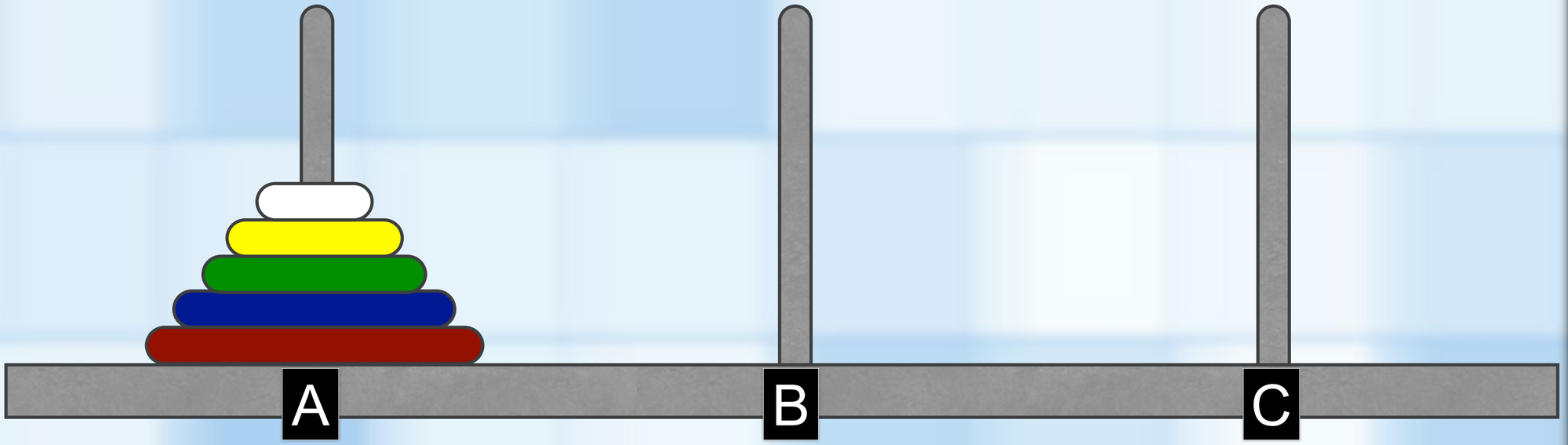
Pearson

Towers of Hanoi

- **Move the disks from post A to post C**
- **Smaller problems**
  - move others disk out of the way so we can move red disk
  - `solveTowers(count, src, dst, spare)`

```
solveTowers( 5, A, C, B )
```

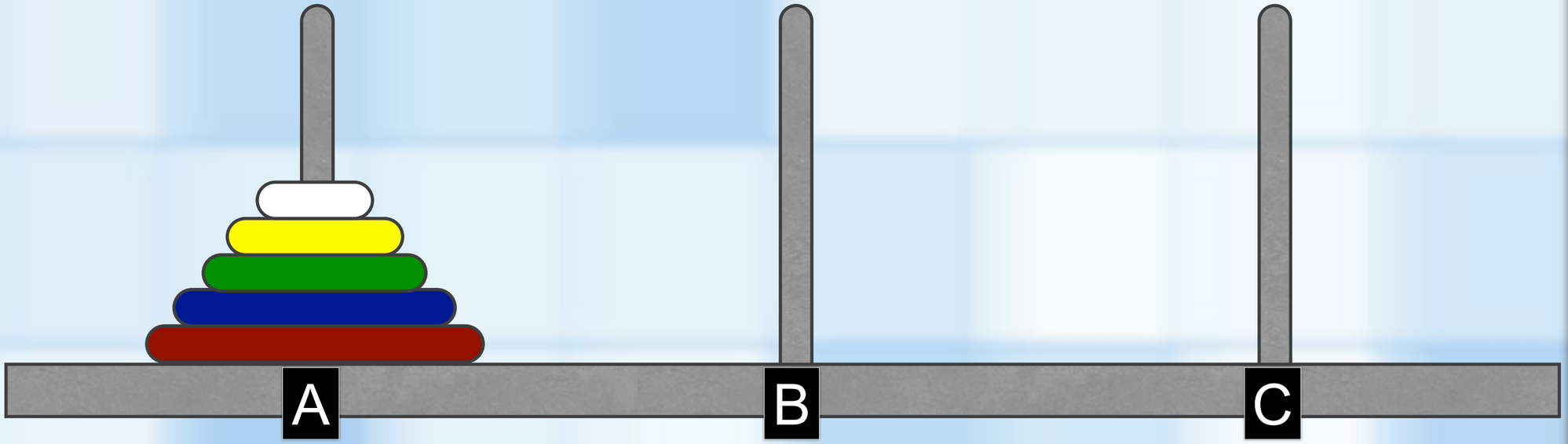
```
solveTowers( 4, A, B, C )
moveDisk( A, C )
solveTowers( 4, B, C, A )
```



Towers of Hanoi

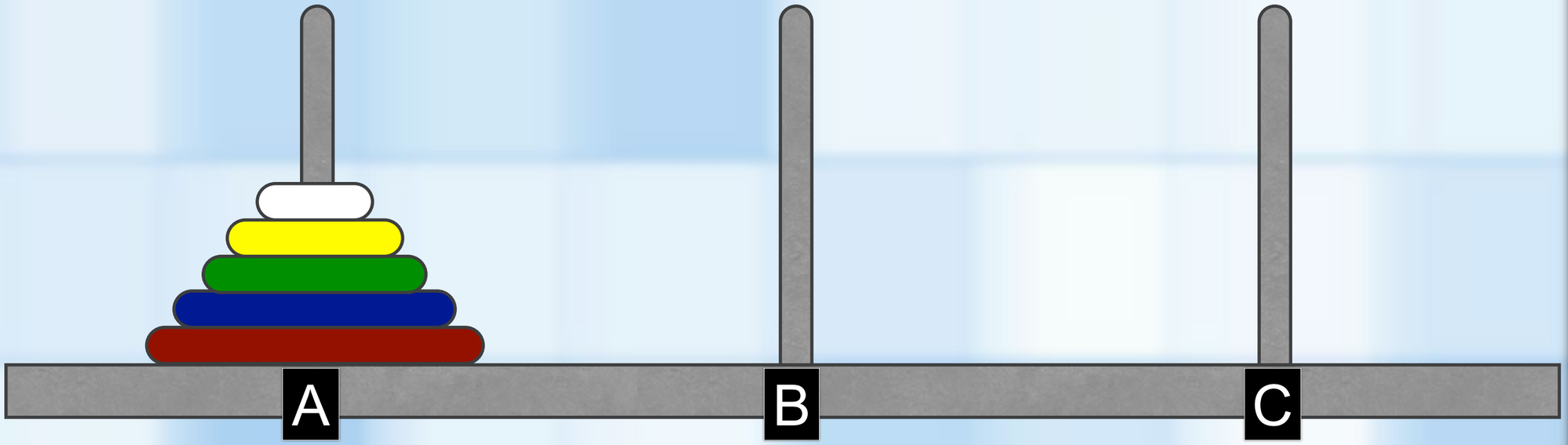
- **Move the disks from post A to post C**
- **Smaller problems**
  - move others disk out of the way so we can move red disk
  - `solveTowers(count, src, dst, spare)`

```
// generic example
solveTowers( count-1, src, spare, dst )
moveDisk( src, dst )
solveTowers( count-1, spare, dst, src )
```



Towers of Hanoi

```
void solveTowers(count, src, dst, spare) {
   if (count == 1)
      moveDisk(src, dst)
   else {
      solveTowers( count-1, src, spare, dst )
      moveDisk( src, dst )
      solveTowers( count-1, spare, dst, src )
}
```



Pearson

Towers of Hanoi

```
void solveTowers(count, src, dst, spare) {
   if (count == 1)
      moveDisk(src, dst)
   else {
      solveTowers( count-1, src, spare, dst )
      moveDisk( src, dst )
      solveTowers( count-1, spare, dst, src )
}
```

```
solveTowers(5, A, C, B)
   solveTowers(4, A, B, C)
      solveTowers(3, A, C, B)
         solveTowers(2, A, B, C)
            solveTowers(1, A, C, B)
               diskMove(A, C) // base case
            diskMove(A, B)
            solveTowers(1, C, B, A)
               diskMove(C, B) // base case
         diskMove(A, C)
         solveTowers(2, B, C, A)
```