# Recursion — Pt 3

# Finding a data item

- Suppose you have an array (or ArrayList) of N items.  How do you determine if the array contains a particular item?
  - Does the form of the array matter?
    - Unsorted
    - Sorted
  - What is the complexity of finding an item?

# Find

## works on all arrays. Sorted or Unsorted

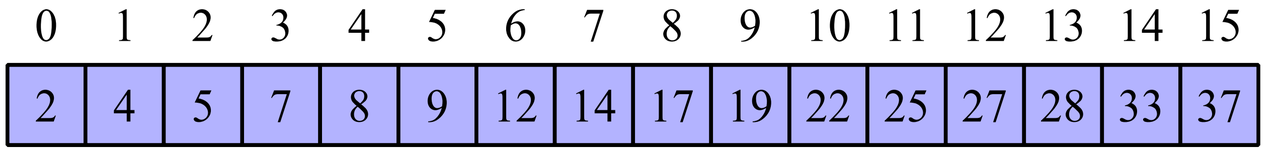
- Start at beginning
  - compare until found
  - Time Complexity??
  - Loops would work
    - but this is a unit on recursion

```
public int find(int[] arr, int num) {
        return findUtil(arr, num, 0);
   }

    /**
     * Find be looking at each item.  The array may be in any order
     * @param arr the array to be searched
     * @param num the number to be found
     * @param loc the location to consider next
     * @return the location of num in arr, or -1 if not in arr
     */
    private int findUtil(int[] arr, int num, int loc) {
        if (loc >= arr.length)
            return -1;
        if (arr[loc] == num)
            return loc;
        return findUtil(arr, num, loc + 1);
    }
```

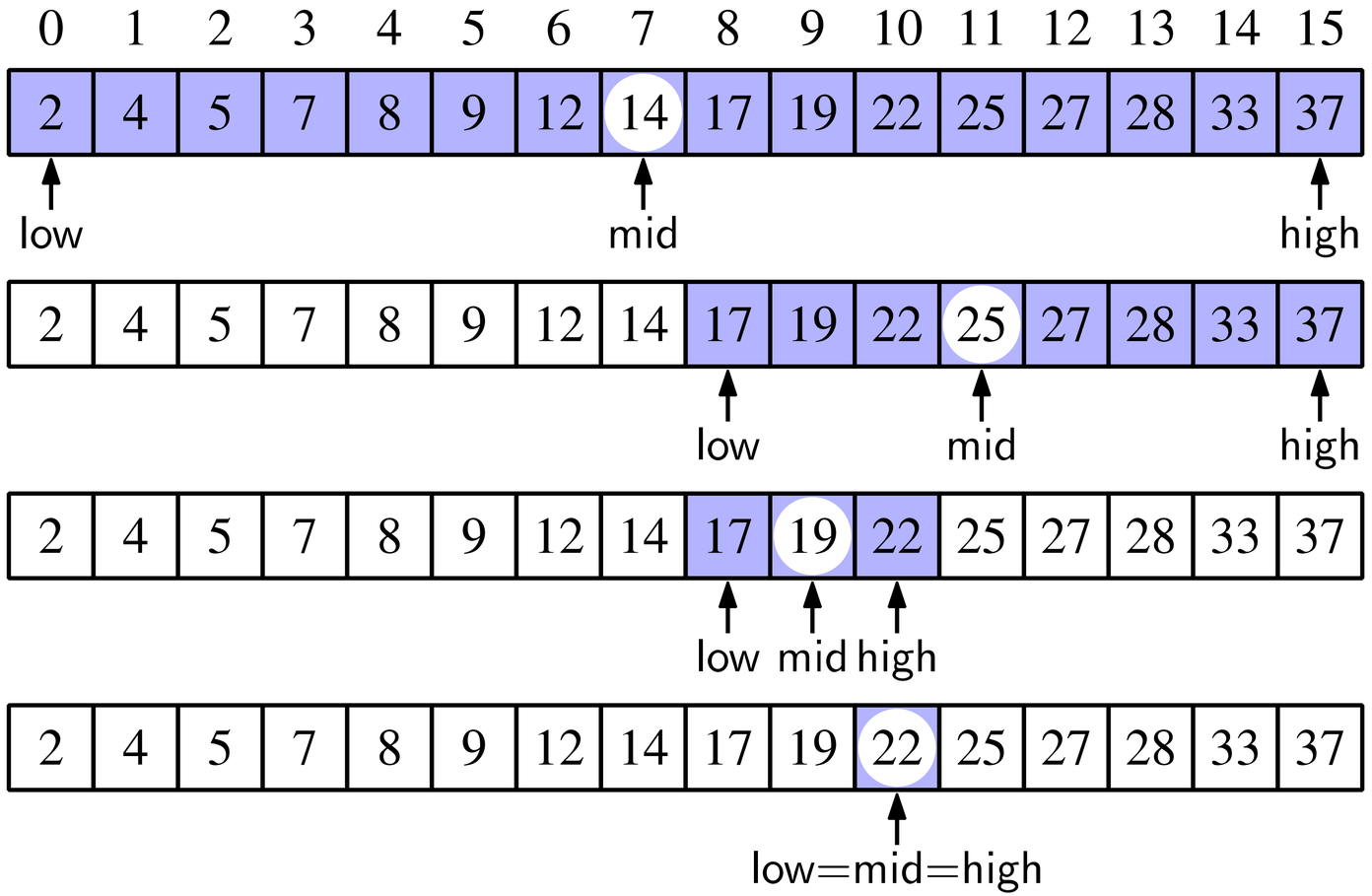
# Binary Search
# Faster find on sorted arrays

- Search for an integer (22) in an ordered list




- $mid = \left\lfloor \frac{low + high}{2} \right\rfloor = \left\lfloor \frac{0 + 15}{2} \right\rfloor = 7$
  - `target==data[mid]`, found
  - `target>data[mid]`, recur on second half
  - `target<data[mid]`, recur on first half

# target = 22


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low mid high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low=mid=high

# Binary Search Code

```
/**
     * An alternate version of finding number in an array
     * This version requires that the array be sorted
     * @param arr the array - it must be sorted
     * @param num the number to be found
     * @param minPos the lowest position that the item to be found could be at
     * @param maxPos the highest possible location
     * @return the location, or -1 if not found
     */
    private int findUtilB(int[] arr, int num, int minPos, int maxPos) {
        if (minPos >= maxPos)
            return -1;
        int loc = (minPos + maxPos) / 2;
        if (arr[loc] == num)
            return loc;
        if (num > arr[loc])
            return findUtilB(arr, num, loc + 1, maxPos);
        else
            return findUtilB(arr, num, minPos, loc - 1);
    }
```

Why loc+1?

Why loc-1?

What would need to be changed for this code to work on ArrayList?

# Binary Search Analysis

- Each recursive call divides the array in half
- If the array is of size n, it divides (and searches) at most $\log_2 n$ times before the current half is of size $1$
- $O(\log_2 n)$

# Practice

## list merging

- Given two ArrayLists, create a new ArrayList in which the contents of the two given lists are interweaved
  - For example: [1,3,5,7,9] [2,4] would yield [1,2,3,4,5,7,9]
  - For example: [2,33,444,33] [0,1,10,11,100,101,110, 111]  yields [2,0,33,1,444,11,33,100,101,110,111]
- Do this recursively
  - Do not change the contents of the two provided ArrayLists
  - You may assume that the ArrayLists contain Integers
- Problems:
  - How do you take from A and then from B?
  - How to you handle different length lists?
  - What is (are) the base case(s)?

# Recursion and Backtracking

- All problems considered so far progress steadily towards an answer.
- Consider a maze.  Sometimes you need to “backtrack”.
  - RECURSION makes backtracking easy!
- Idea:
  - 1. Somehow  make a copy of where you are,
  - 2. Try to go forward one step.
    - A. If success,
      - Mark your step on the copy.
      - return to step 1
    - B. If failure
      - throw out copy  (ie go backwards) -- perhaps mark what you have tried (problem dependent)
      - go some other direction using your original

# Mazes and Recursion

- English language instructions for solving a maze (written so that a person could follow without thought and solve the maze).

# Recursion and Backtracking

- All problems considered so far progress steadily towards an answer.
- Consider a maze.  Sometimes you need to “backtrack”.
  - RECURSION makes backtracking easy!
- Idea:
  - 1. Somehow  make a copy of where you are
  - 2. Identify all of the possible moves you can make
  - 3. Try to go forward one step.
    - A. If you can go forward … ,
      - If needed, mark your step on the copy.
      - return to step 1
    - B. If failure --
      - try a different forward step
  - 4. If you run out of forward steps, backtrack

- Twiddle
  - especially with mazes mark places you have been so you do not retry failed paths

# N Queens problem

- Place N queens on an NxN chessboard such that no queen can take another
- Strategy:
  - on row N
    - move across the columns trying a spot for OK
      - if found a spot, then recur with row N+1
    - if have checked everything in a row and there is no place that is OK
      - backtrack
        - undo placement of queen in row N-1 and continue across that row

# N Queens

## setup

- board just a 2d array of chars
- will do recursion with a private utility function

```
public class NQueens {
    private char[][] board;
    private int size = 0;

    public NQueens(int siz) {
        size = siz;
        board = new char[siz][siz];
        for (int i = 0; i < siz; i++) {
            for (int j = 0; j < siz; j++) {
                board[i][j] = '.';
            }
        }
    }

    private void showBoard() {
        for (int r = 0; r < size; r++) {
            for (int c = 0; c < size; c++) {
                System.out.print(board[r][c]);
            }
            System.out.print("\n");
        }
    }


    public void doQueens() {
        doQueensUtil(0);
    }
```

# N Queens

## recursion

- base case:
  - the row being asked to consider is off board
    - return true;
- in the row
  - go across every column
    - put queen in a column
      - check if that is OK
        - if it is, go to recur to next row
        - if found solution return true;
      - if NOT OK, remove queen from column
  - if cannot find a place to put a queen, return false

```
private boolean doQueensUtil(int roww) {
     if (roww >= size)
         return true;
     if (rowOccupied(roww))
         return doQueensUtil(roww + 1);
     for (int col = 0; col < size; col++) {
         board[roww][col] = 'Q';
         if (OKBoard()) {
             boolean v = doQueensUtil(roww + 1);
             if (v)
                 return true;
         } else {
             System.out.println("NOT OK");
             showBoard();
             System.out.println("NOT OK" + roww + " " + col)
         }
         board[roww][col] = '-';
     }
     return false;
}
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | | 4 | | 9 | | |
| | | 3 | 6 | | 8 | 5 | 7 | |
| 1 | | | 3 | 5 | 9 | | 4 | |
| 2 | 6 | | | | 4 | | | 7 |
| | 1 | 4 | 8 | 6 | | 3 | | |
| | 8 | | 7 | | | 2 | | 4 |
| 4 | | 7 | | | 3 | | 1 | |
| | | | | 9 | 6 | | | 3 |
| | | | 5 | | | | 9 | 8 |

# Sudoku

- Solvable using really stupid recursion

```
Puzzle solve(Puzzle p, int xloc, int yloc)
    if isSolved(p)
        return p
    if not isSolvable(p)
        return null
    if (yloc>9)
        xloc++
        yloc=0
    if (xloc>9)
        return null
    if (p(xloc, yloc) != 0)
        return solve(p, xloc, yloc+1)
    else
        legalmoves = legalmovesat(p, xloc, yloc)
        foreach legalmove : legalmoves
            set p(xloc, yloc) to legalmove
            np = solve(copy(p), xloc, yloc+1)
            if (np!=null)
                return np
        return null
```

# Sudoku

## 2

```
Puzzle
    at its simplest this could be just a 2d array (specifically 9x9) of int


boolean isSolved(Puzzle p)
    return true if the puzzle is completely solved false otherwise


boolean isSolvable(Puzzle p)
    return true if the puzzle still might be solvalble, false otherwise


List legalmovesat(Puzzle p, int xloc, int yloc)
    return the list of numbers that can be legally put into the position given
the current board


Puzzle copy(Puzzle p)
    return a new instance of puzzle that is an exact copy of the provided
puzzle.  Importantly, making a change in the copy should have no effect on the
original.
```