CS 106B, Lecture 11
Recursive Backtracking 3

reading:

Programming Abstractions in C++, Chapter 9

"Arm's length" recursion

- **Arm's length recursion**: A poor style where unnecessary tests are performed before performing recursive calls.
  - Typically, the tests try to avoid making a call into what would otherwise be a base case.

- Example: escapeMaze
  - Our code recursively tries to explore up, down, left, and right.
  - Some of those directions may lead to walls or off the board. Shouldn't we test before making calls in these directions?

Arm's Length escapeMaze

```
// This code is bad. It uses arm's length recursion.
bool escapeMaze(Maze& maze, int r, int c) {
    maze.mark(row, col);
    // recursive case: try to escape in 4 directions
    // (check each one by arm's length)
    if (maze.inBounds(r-1,c) && maze.isOpen(r-1, c)) {
        if (escapeMaze(r-1,c)) {return true; }
    }
    if (maze.inBounds(r+1,c) && maze.isOpen(r+1, c)) {
        if (escapeMaze(r+1,c)) {return true; }
    }
    if (maze.inBounds(r,c-1) && maze.isOpen(r,c-1)) {
        if (escapeMaze(r,c-1)) {return true; }
    }
    if (maze.inBounds(r,c+1) && maze.isOpen(r,c+1)) {
        if (escapeMaze(r,c+1)) {return true; }
    }
    maze.taint(row, col);
    return false;   // all 4 paths failed; taint
}
```

Escape Maze solution

```
// This code is better.
bool escapeMaze(Maze& maze, int row, int col) {
    if (!maze.inBounds(row, col)) {
        return true;            // base case 1: escaped
    } else if (!maze.isOpen(row, col)) {
        return false;           // base case 2: blocked
    } else {
        // recursive case: try to escape in 4 directions
        maze.mark(row, col);
        if (escapeMaze(maze, row - 1, col)
                || escapeMaze(maze, row + 1, col)
                || escapeMaze(maze, row, col - 1)
                || escapeMaze(maze, row, col + 1)) {
            return true;    // one of the paths worked!
        } else {
            maze.taint(row, col);
            return false;   // all 4 paths failed; taint
        }
    }
}
```

Exercise: sublists

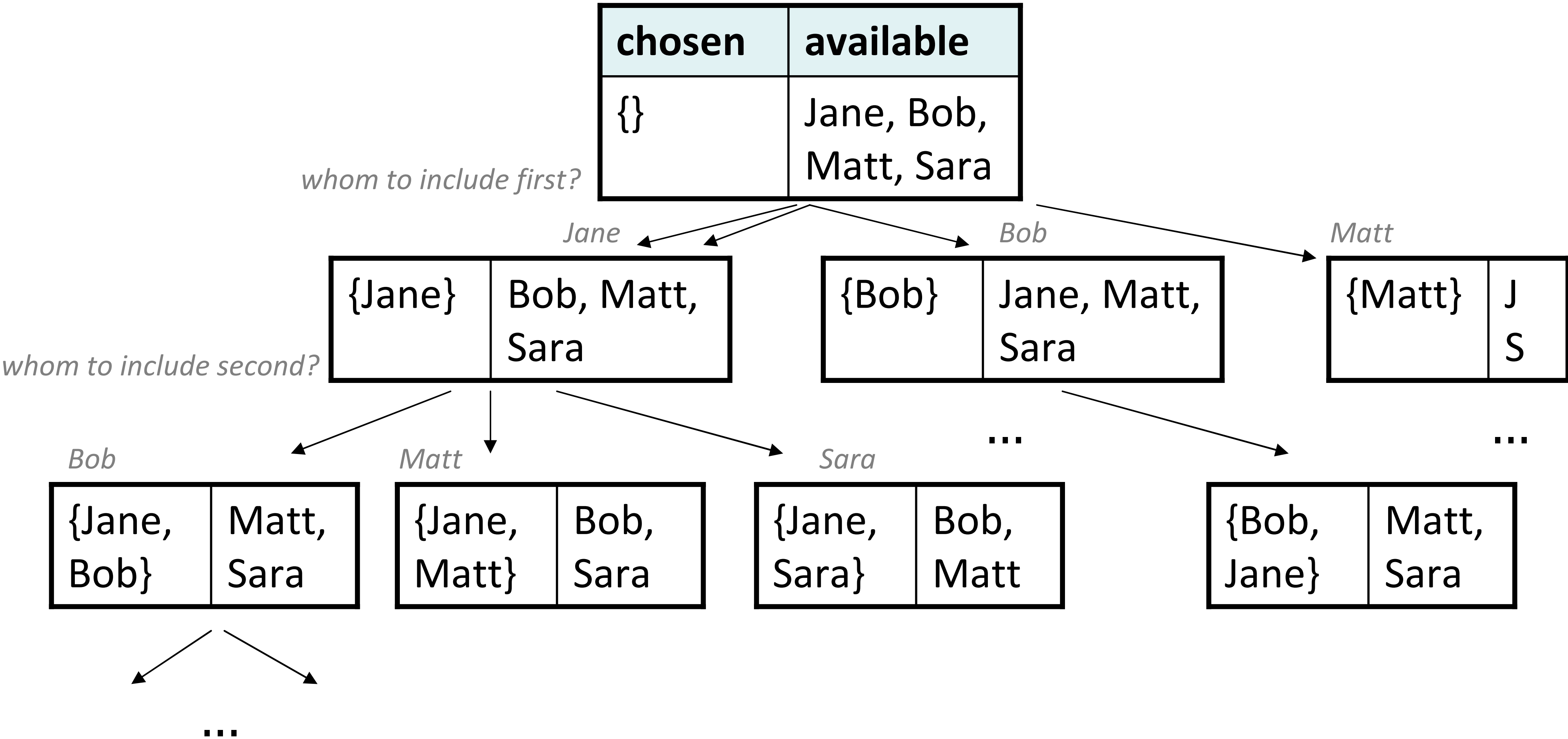
printSubVectors

- Write a function **`sublists`** that finds every possible sub-list of a given vector. A sub-list of a vector *V* contains ≥ 0 of *V*'s elements.
  - Example: if *V* is `{Jane, Bob, Matt, Sara}`, then the call of **`sublists`**`(V);` prints:

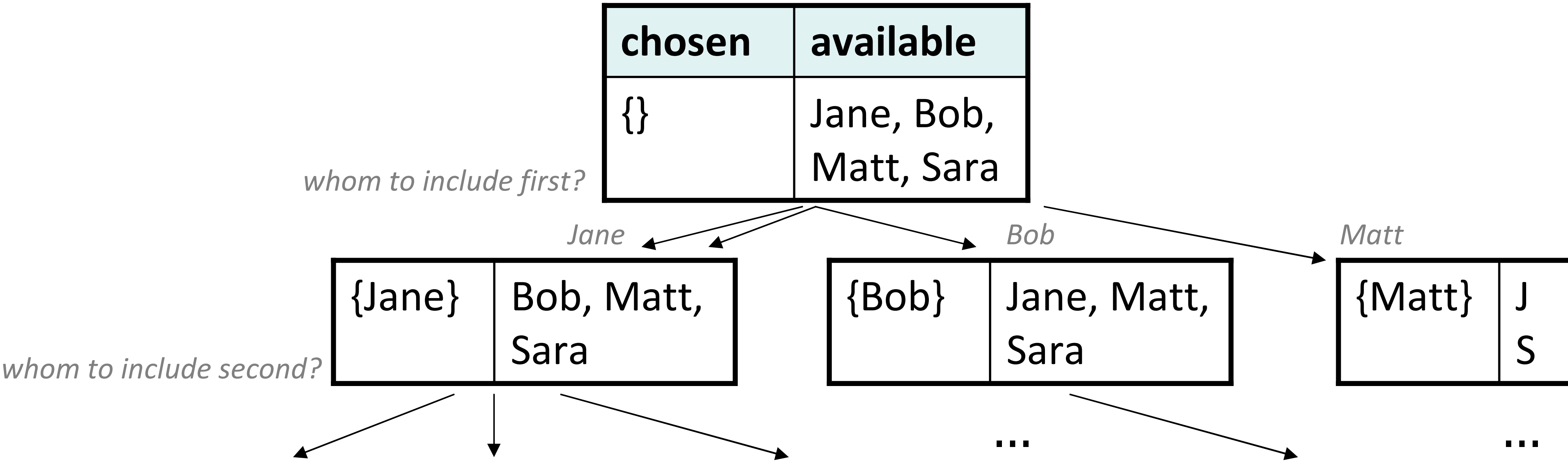
```
{Jane, Bob, Matt, Sara}
{Jane, Bob, Matt}
{Jane, Bob, Sara}
{Jane, Bob}
{Jane, Matt, Sara}
{Jane, Matt}
{Jane, Sara}
{Jane}
{Bob, Matt, Sara}
{Bob, Matt}
{Bob, Sara}
{Bob}
{Matt, Sara}
{Matt}
{Sara}
{}
```

  - You can print the sub-lists out in any order, one per line.
    - *What are the "choices" in this problem? (choose, explore)*

Decision tree?



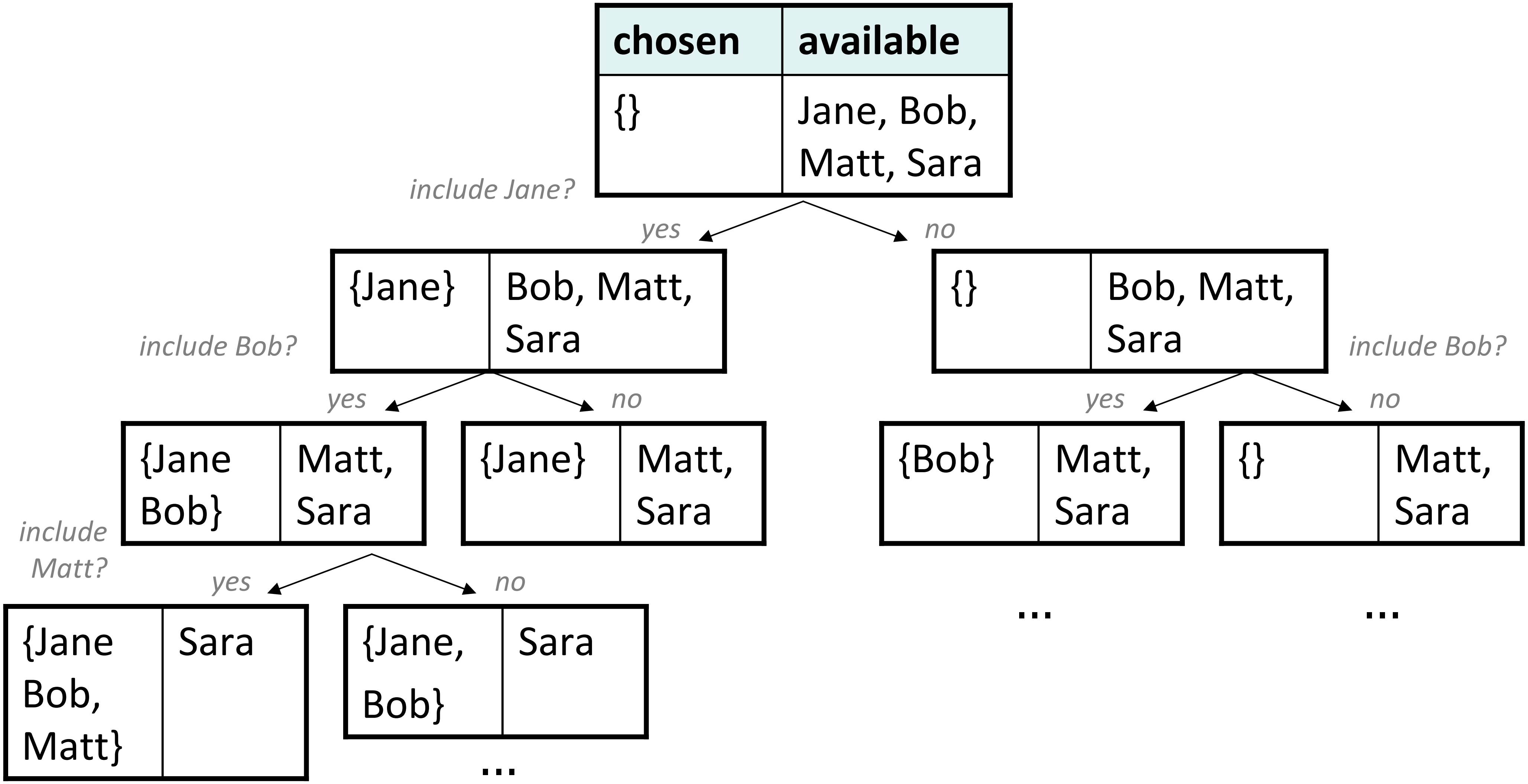
Wrong decision tree



Q: Why isn't this the right decision tree for this problem?

- **A.** It does not actually end up finding every possible subset.
- **B.** It does find all subsets, but it finds them in the wrong order.
- **C.** It does find all subsets, but it is inefficient.
- **D.** None of the above

Better decision tree



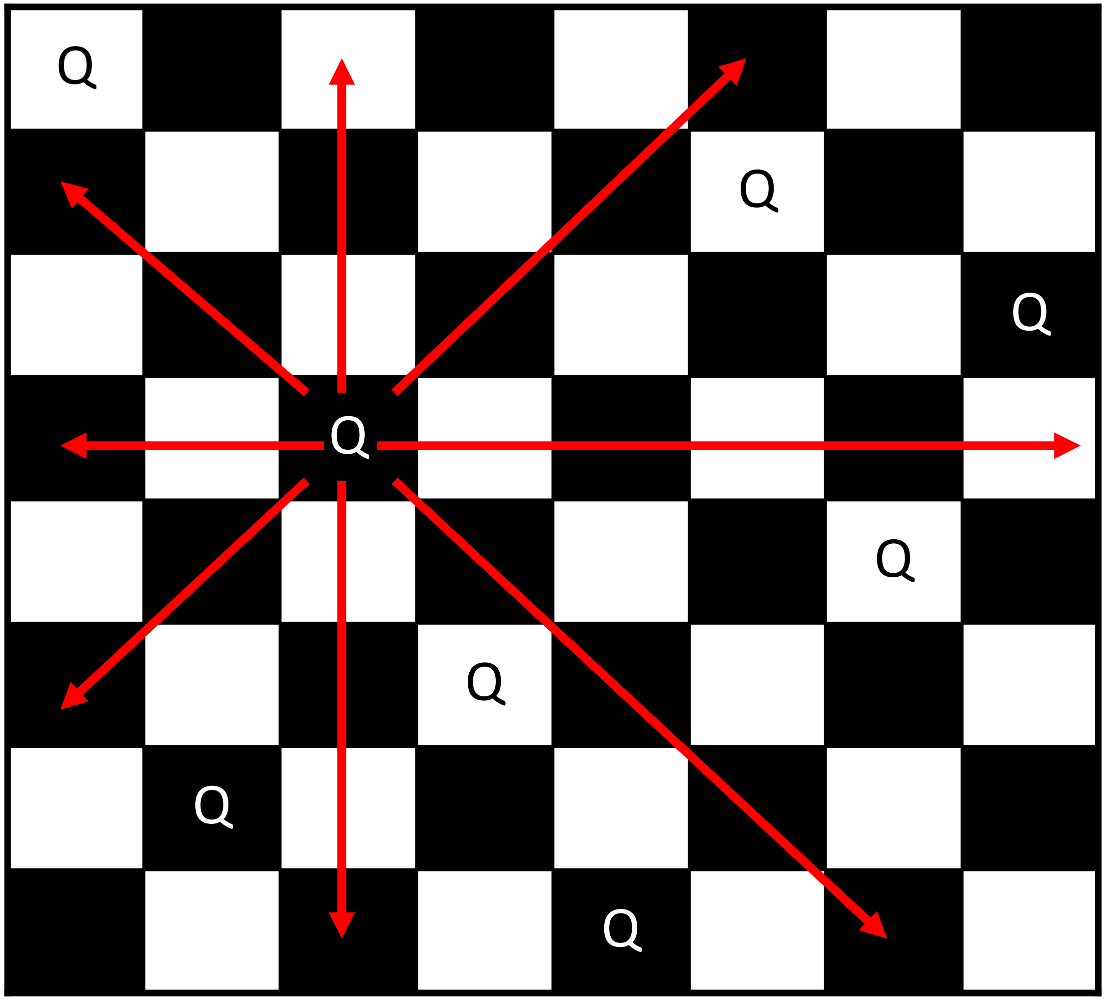
– Each decision is: "Include Jane or not?" ... "Include Bob or not?" ...
  - The **order** of people chosen does not matter; only the **membership**.

sublists solution

```
void sublists(Vector<string>& v) {
    Vector<string> chosen;
    sublistsHelper(v, 0, chosen);
}

void sublistsHelper(Vector<string>& v, int i,
                    Vector<string>& chosen) {
    if (i >= v.size()) {
        cout << chosen << endl;   // base case; nothing to choose
    } else {
        // there are two choices to explore:
        // the subset without i'th element, and the one with it

        sublistsHelper(v, i+1, chosen);  // choose/explore (without)

        chosen.add(v[i]);
        sublistsHelper(v, i+1, chosen);  // choose/explore (with)

        chosen.remove(chosen.size() - 1);   // "undo" our choice
    }
}
```

The "8 Queens" problem

- Consider the problem of trying to place 8 queens on a chess board such that no queen can attack another queen.
  - What are the "choices"?
  - How do we "make" or "un-make" a choice?
  - How do we know when to stop?



Naive algorithm

- for (each board square):
  - Place a queen there.
  - Try to place the rest of the queens.
  - Un-place the queen.

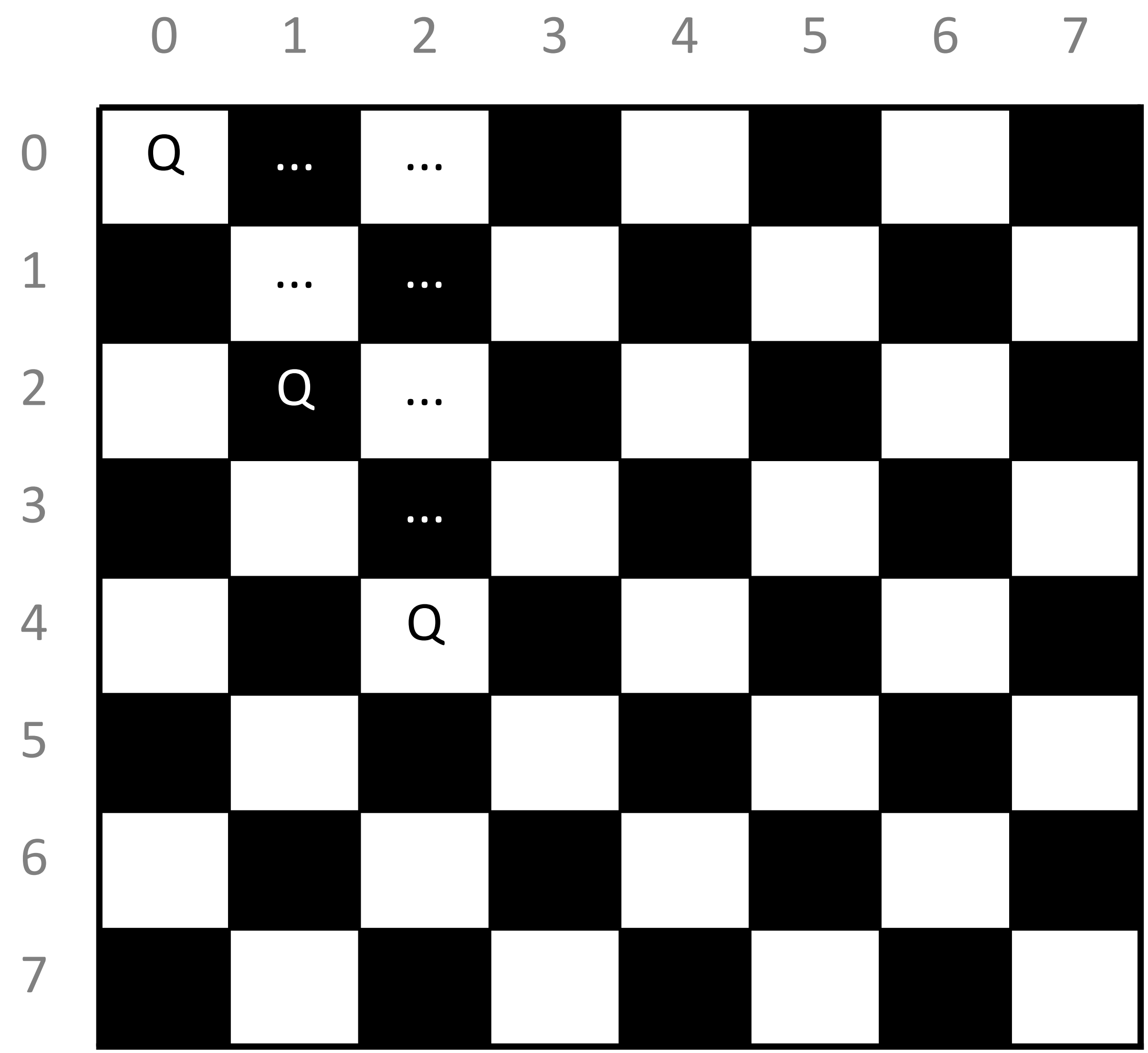
	0	1	2	3	4	5	6	7
0	Q	…	…	…	…	…	…	…
1	…	…	…	…	…	…	…	…
2	…							
3								
4								
5								
6								
7								

Q: How large is the solution space for this algorithm?

- **A.** 64 choices
- **B.** 64 * 8
- **C.** 64^{8}
- **D.** 64*63*62*61*60*59*58*57
- **E.** none of the above

Better algorithm idea

- Observation: In a working solution, exactly 1 queen must appear in each row and in each column.
  - Redefine a "choice" to be valid placement of a queen in a particular column.
  - How large is the solution space now?
    - 8 * 8 * 8 * ...



Exercise

- Suppose we have a Board class with the following methods:

Member	Description
`Board b(size);`	construct empty board
`b.isSafe(row, column)`	`true` if a queen could be safely placed here (0-based)
`b.isValid()`	`true` if all current queens are safe
`b.place(row, column);`	place queen here
`b.remove(row, column);`	remove queen from here
`cout << b << endl;` *or* `b.toString()`	print/return a text display of the board state

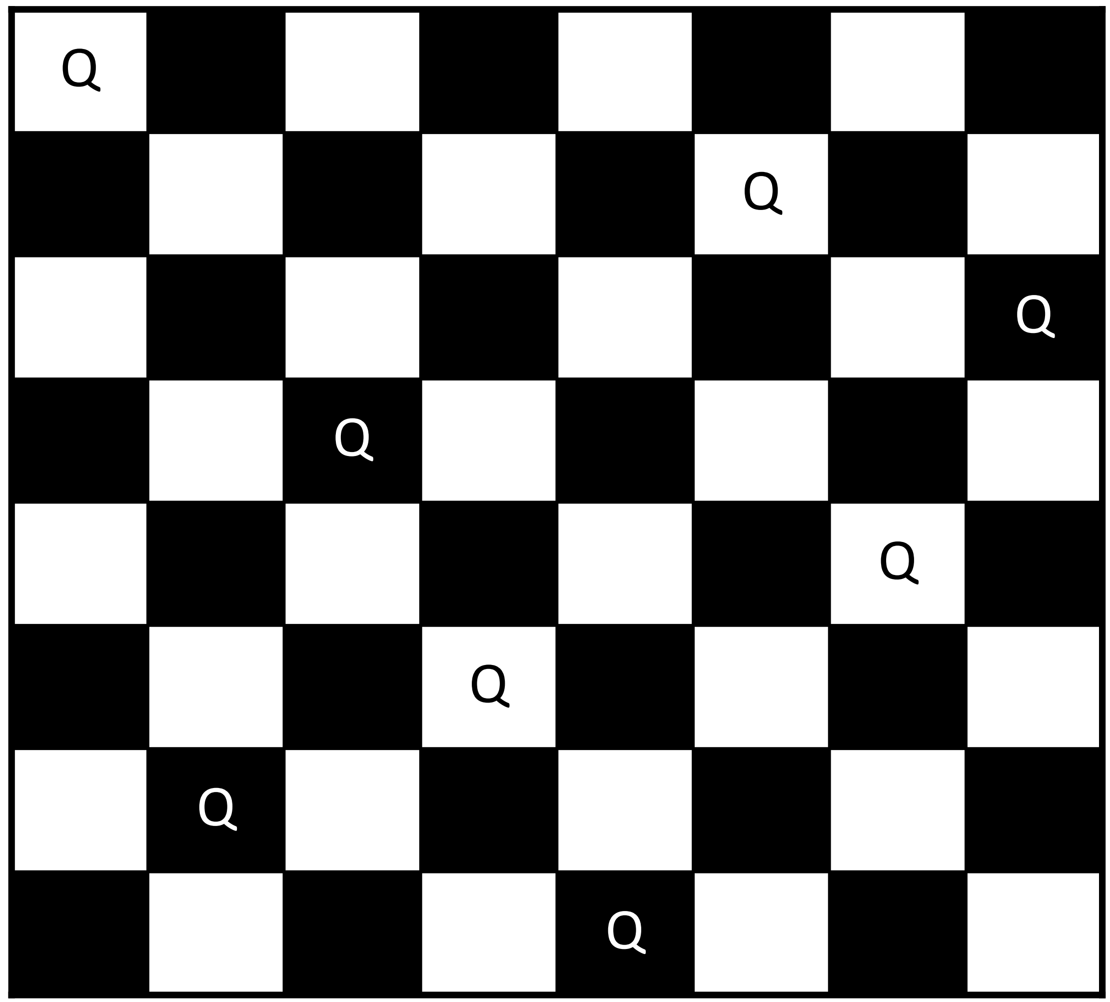
- Write a function **`solveQueens`** that accepts a Board as a parameter and tries to place 8 queens on it safely.
  - Your method should print all possible solutions.

Exercise solution

```
// Recursively searches for all solutions to N queens
// on this board, starting with the given column.
// PRE: queens have been safely placed in columns 0 to (col-1)
void solveHelper(Board& board, int col) {
    if (col >= board.size()) {
        cout << board << endl;    // base case: all columns placed
    } else {
        // recursive case: try to place a queen in this column
        for (int row = 0; row < board.size(); row++) {
            if (board.isSafe(row, col)) {
                board.place(row, col);          // choose
                solveHelper(board, col + 1);    // explore
                board.remove(row, col);         // un-choose
            }
        }
    }
}

void solveQueens(Board& board) {
    solveHelper(board, 0);
}
```

Stop after 1 solution

- Modify **solveQueens** to print just one board solution and stop.
  - How do we stop the recursion after it finds a solution?



Exercise solution

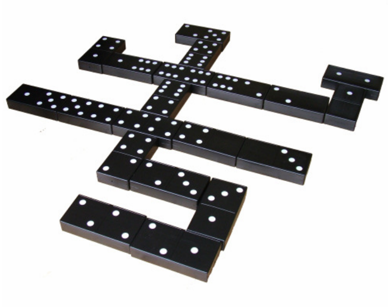
```
// Searches for a solution to the 8 queens problem
// with this board, reporting the first result found.
void solveQueens(Board& board) {
    if (solveHelper(board, 0)) {
        cout << "One solution is as follows:" << endl;
        cout << board << endl;
    } else {
        cout << "No solution found." << endl;
    }
}

...
```

Exercise solution, cont'd.

```
// Recursively searches for a solution to 8 queens on this
// board, starting with the given column, returning true if a
// solution is found and storing that solution in the board.
// PRE: queens have been safely placed in columns 0 to (col-1)
bool solveHelper(Board& board, int col) {
    if (col >= board.size()) {
        return true;   // base case: all columns are placed
    } else {
        // recursive case: place a queen in this column
        for (int row = 0; row < board.size(); row++) {
            if (board.isSafe(row, col)) {
                board.place(row, col);              // choose
                if (solveHelper(board, col + 1)) {  // explore
                    return true;   // solution found
                }
                board.remove(row, col);             // un-choose
            }
        }
        return false;   // no solution found
    }
}
```

Exercise: Dominoes

chainExists



- Dominoes uses black tiles, each having 2 numbers of dots from 0-6. Players line up tiles to match dots.

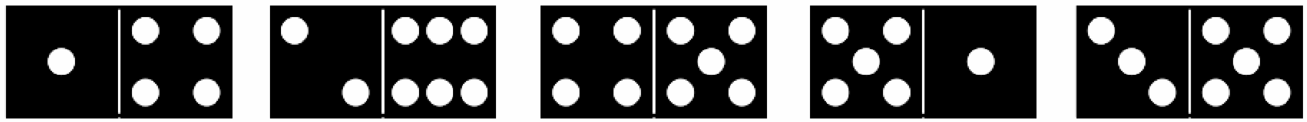
- Given a class `Domino` with the following members:

```
int first()                  // first dots value from 0-6
int second()                 // second dots value from 0-6
void flip()                  // inverts 1st/2nd
bool contains(int n)         // true if 1st and/or 2nd == n
string toString()            // e.g. "(3|5)"
```

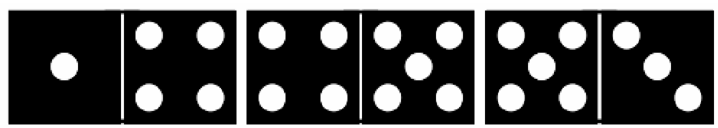
- Write a function **`chainExists`** that takes a `Vector` of dominoes and a starting/ending dot value, and returns whether the dominoes can be made into a chain that starts/ends with those values.

Domino chains

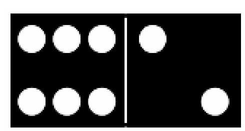
- Suppose we have the following dominoes:



- We can link them into a chain from 1 to 3 as follows:
  - Notice that the 3|5 domino had to be flipped.

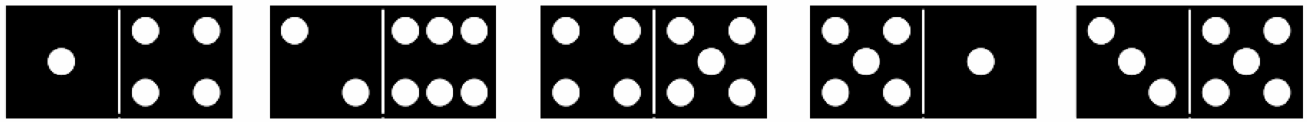


- We can "link" one domino into a "chain" from 6 to 2 as follows:



Enumerating choices

- If we have these dominoes, and we want a chain from 1 to 3:



Q: What are the "choices" your code should explore?

A. The numbers 0-6 that can appear on a domino.

B. The set of all of the dominoes above.

C. The set of dominoes above whose first number is 1.

D. The set of dominoes above whose second number is 3.

E. The set of dominoes above whose first or second number is 1.

hasChain pseudocode

```
function chainExists(dominoes, start, end):
    if dominoes is empty: nothing to do.
    if start == end:
        if any domino in dominoes contains start, return true.
    else:
        // handle all choices for a single letter; let recursion do the rest.
        for each domino d in dominoes:
            if d contains start:
                choose d.
                if chainExists(dominoes):    // explore remaining dominoes.
                    return true.
                un-choose d.

        return false.   // no chain found
```

hasChain solution

```
bool chainExists(Vector<Domino>& dominoes, int start, int end) {
    if (start == end) {                         // base case
        for (Domino d : dominoes) {
            if (d.contains(start)) { return true; }
        }
        return false;
    } else {
        for (int i = 0; i < dominoes.size(); i++) {
            Domino d = dominoes[i];
            if (d.second() == start) {
                d.flip();
            }
            if (d.first() == start) {
                dominoes.remove(i);             // choose
                if (d.second() == end ||        // explore
                        chainExists(dominoes, d.second(), end)) {
                    dominoes.insert(i, d);
                    return true;
                }
                dominoes.insert(i, d);          // un-choose
            }
        }
        return false;
    }
}
```

Exercise: Print chain

- Write a variation of your **`chainExists`** function that also prints the chain of dominoes that it finds, if any.

```
hasChain(dominoes, 1, 3);

[(1|4), (4|5), (5|3)]
```