# CS 106B, Lecture 13
# Recursive Backtracking

# Plan for Today

- More backtracking!
  - Make sure to practice, in section, on CodeStepByStep, with the book
- Some notes on the midterm

# "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
- Can lead to **functionality bugs** as well as **less readable code**
- Applies to all recursive code but **especially backtracking**

# Backtracking Model

**Choosing**

1. We generally iterate over **decisions**. What are we iterating over here? What are the **choices** for each decision? Do we need a for loop?

**Exploring**

2. How can we *represent* that choice? How should we **modify the parameters** and **store our previous choices** (avoiding *arms-length* recursion)?
   a) Do we need to use a **wrapper** due to extra parameters?
3. How should we **restrict** our choices to be valid?
4. How should we use the **return value** of the recursive calls? Are we looking for all solutions or just one?

**Un-choosing**

5. How do we **un-modify** the parameters from step 3? Do we need to explicitly un-modify, or are they copied? Are they un-modified at the same level as they were modified?

**Base Case**

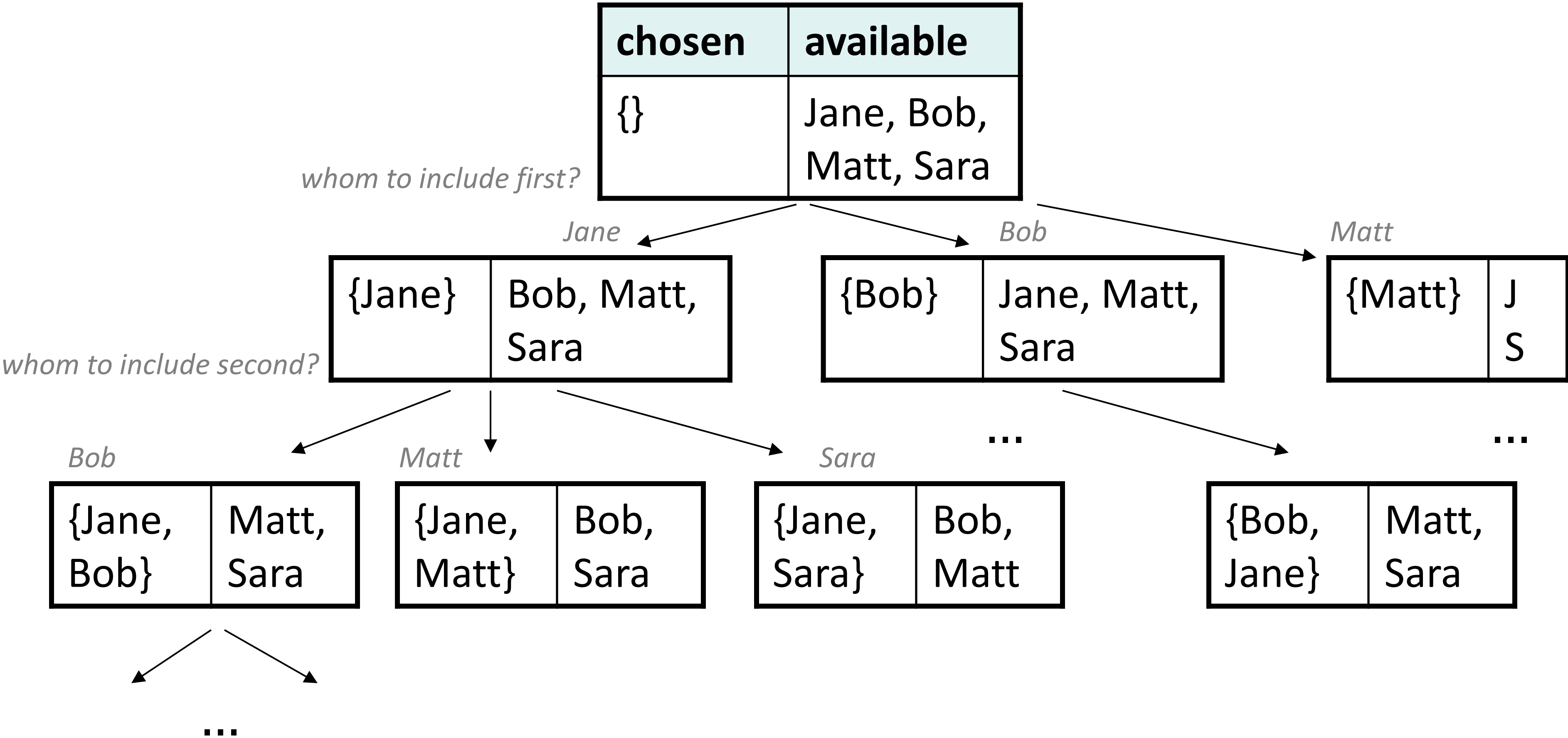
6. What should we do in the base case when we're **out of decisions** (usually return true)?
7. Is there a case for when there **aren't any valid choices left** or a "bad" state is reached (usually return false)?
8. Are the base cases ordered properly? Are we avoiding **arms-length** recursion?

# Exercise: sublists

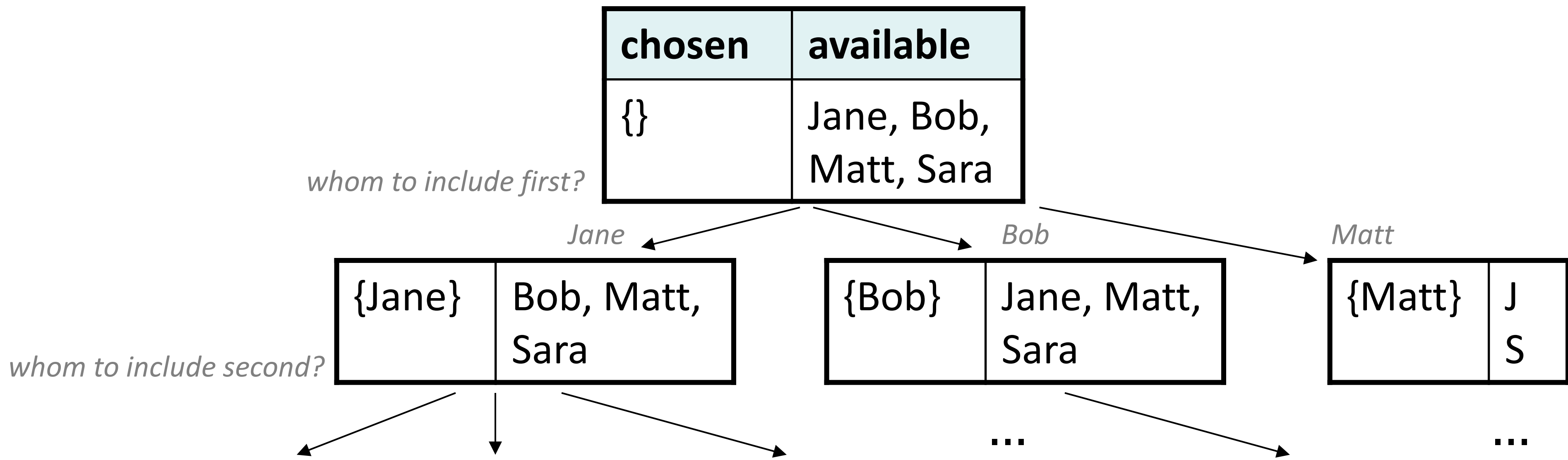
- Write a function **`sublists`** that finds every possible sub-list of a given vector. A sub-list of a vector *V* contains ≥ 0 of *V*'s elements.
  - Example: if *V* is `{Jane, Bob, Matt, Sara}`, then the call of **`sublists`**`(V);` prints:

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

# Decision tree?



# Wrong decision tree




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

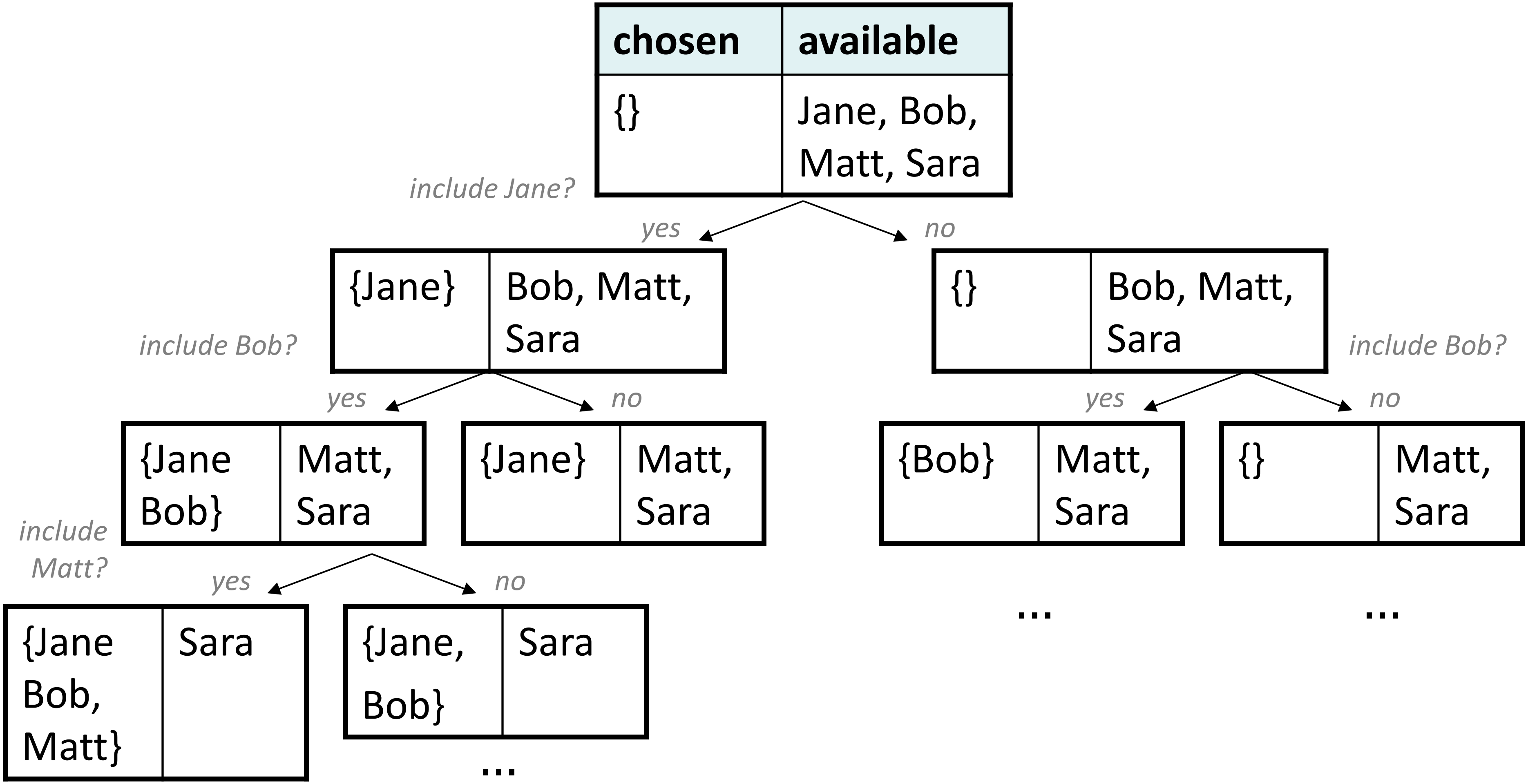
**A.** It does not actually end up finding every possible sublist.

**B.** It does find all sublists, but it finds them in the wrong order.

**C.** It does find all sublists, 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**.

# Mental Model

- **Choose**: What decisions do we have to make? What are our choices?
- **Explore**: How should we modify our parameters after making a choice?
- **Un-Choose**: How do we revert our choice?
- **Base Case**: What should we do when we are out of decisions to make?

# Mental Model

- **Choose**: What decisions do we have to make? What are our choices?
  - *Whether to include a person or not*
- **Explore**: How should we modify our parameters after making a choice?
  - *Build up a vector containing people chosen so far*
- **Un-Choose**: How do we revert our choice?
  - *Remove the person previously inserted into the vector*
- **Base Case**: What should we do when we are out of decisions to make?
  - *Print the result vector*

# 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
    }
}
```

# Announcements

- Assignment 4 goes out tonight. You should receive Assn2 feedback by end of the day
  - A small part of Assn. 4 uses structs which are covered tomorrow.
- Exam logistics
  - Midterm review session in class on 7/23
  - Midterm is on Wednesday, July 24, from 7:00-9:00PM
  - Midterm info (list of topics covered and study tips) online: https://web.stanford.edu/class/cs106b/exams/midterm.html
  - **Highly Recommended:** Complete assignment 4 (or parts of it) before the midterm – backtracking will be tested. Assignment 4 will not be due until July 25th though
  - Lectures 14 and 15 are NOT included on the midterm

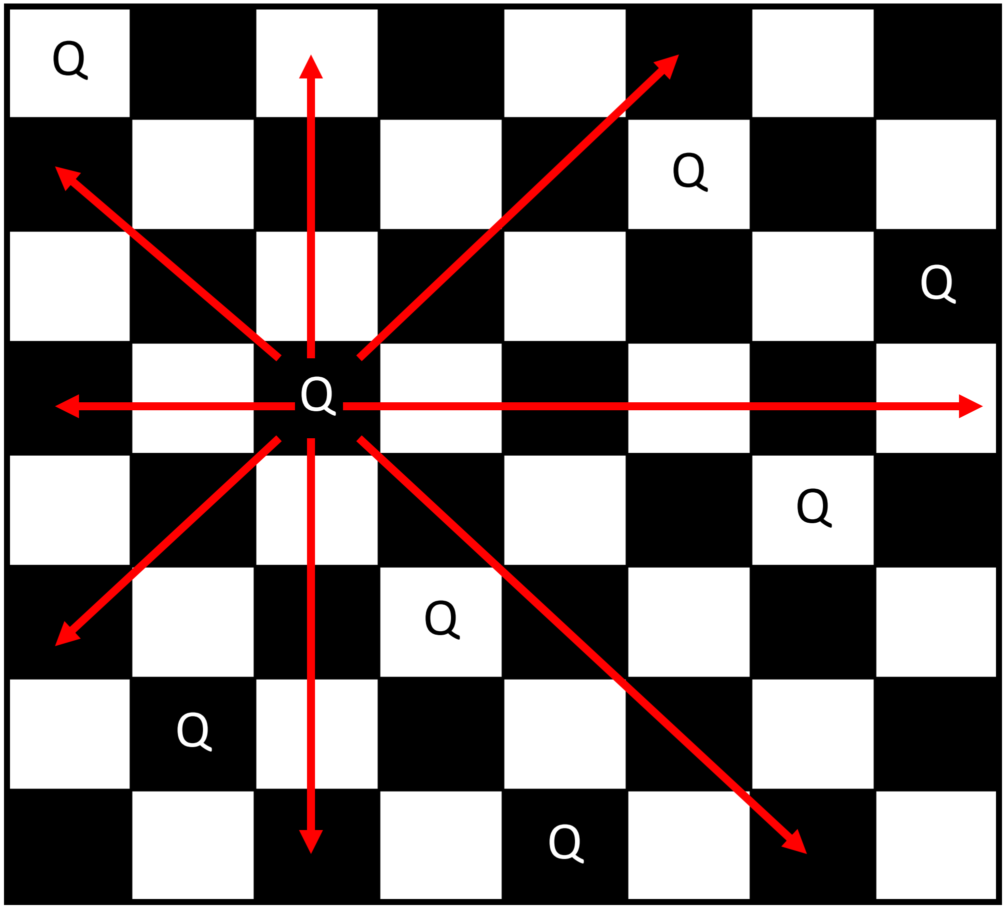
# Announcements

- Practice midterm is released. You need BlueBook to use it.
  - Download the file as a .json file
  - Open BlueBook, follow instructions

- Save your answers in a separate document to compare to the practice midterm answers.
  - Practice midterm answers will be released in a few days

# 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.



# 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 |

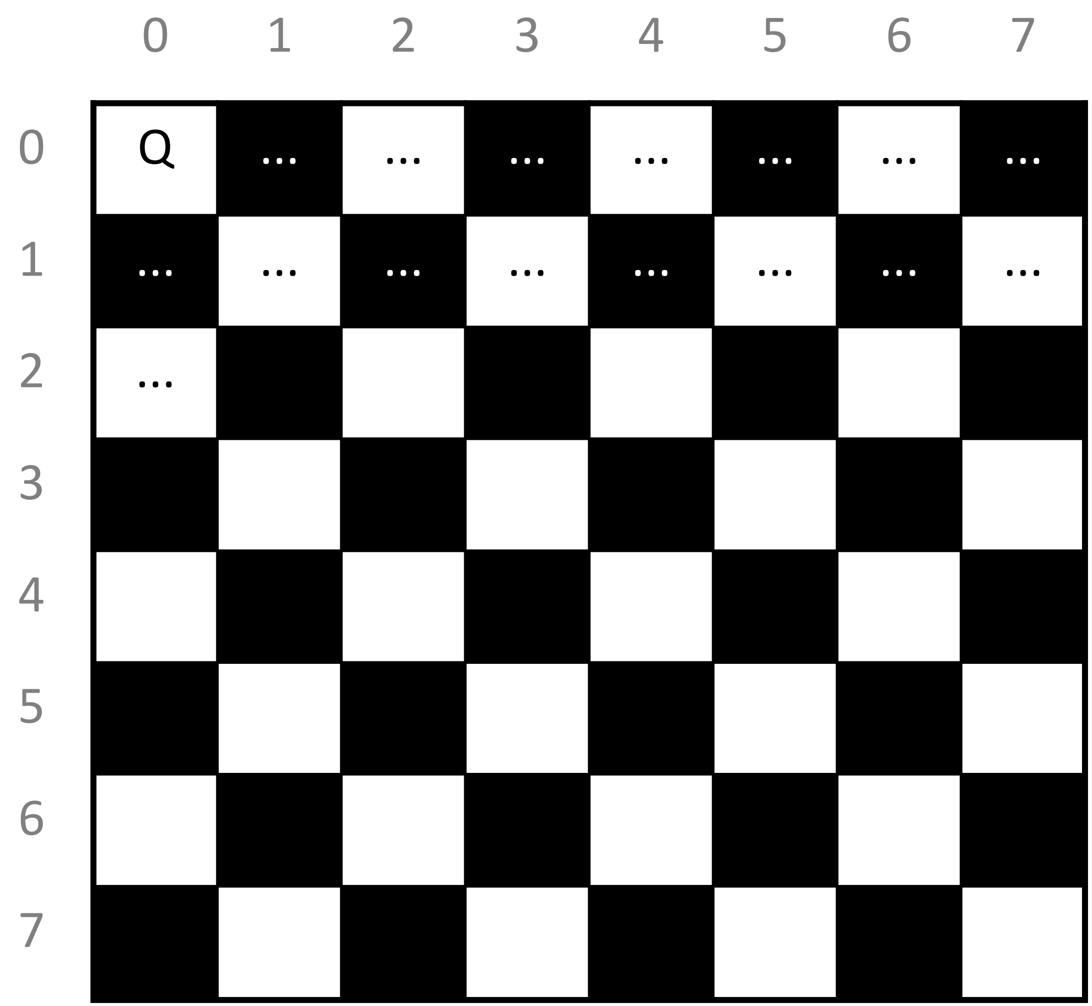
- Write a function **`solveQueens`** that accepts a `Board` as a parameter and tries to place 8 queens on it safely.
  - Your method should return a board with the queens placed if it's possible.

# Mental Model

- **Choose**: What decisions do we have to make? What are our choices?
- **Explore**: How should we modify our parameters after making a choice?
- **Un-Choose**: How do we revert our choice?
- **Base Case**: What should we do when we are out of decisions to make?

# Naive algorithm

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




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

**A.** 64 choices

**B.** 64 * 8

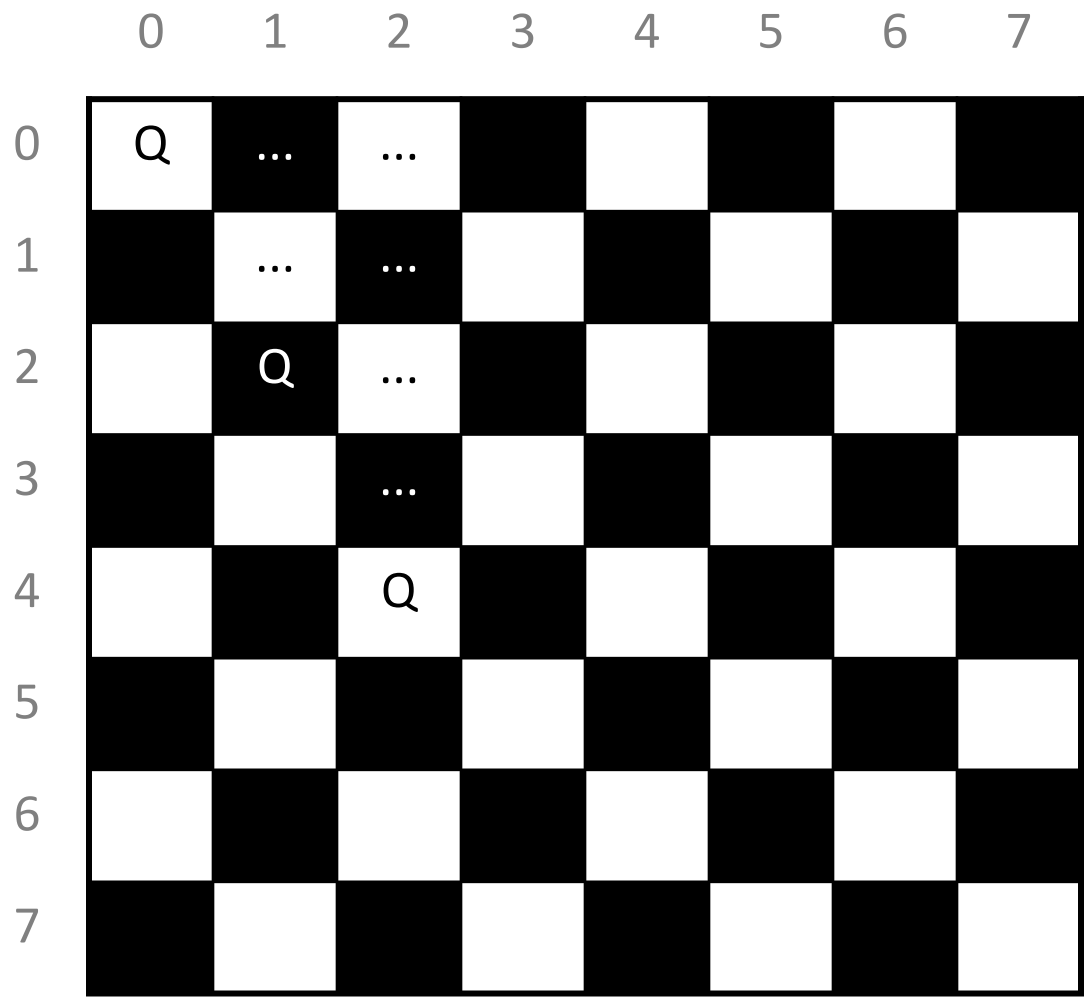
**C.** $64^{8}$

**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 * ...



# Mental Model

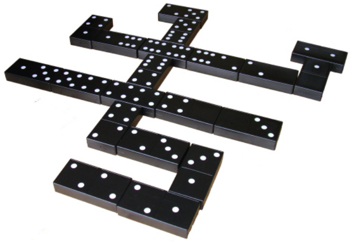
- **Choose**: What decisions do we have to make? What are our choices?
  - *Where in a column to place a queen*
- **Explore**: How should we modify our parameters after making a choice?
  - *Place the queen on the board, move on to the next column*
- **Un-Choose**: How do we revert our choice?
  - *Remove the queen that we placed previously*
- **Base Case**: What should we do when we are out of decisions to make?
  - *Return true*

# 8 Queens solution

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

# Exercise: Dominoes

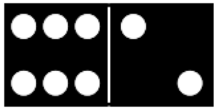
- Dominoes uses black tiles, each having 2 numbers of dots from 0-6. Players line up tiles to match dots.



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

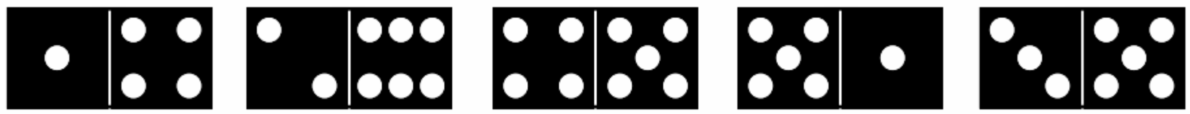
```
int first()                 // first dots value from 0-6
int second()                // second dots value from 0-6
void flip()                 // inverts 1st/2nd
bool contains(int n)        // true if 1st and/or 2nd == n
string toString()           // e.g. "(3|5)"
```

- 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.

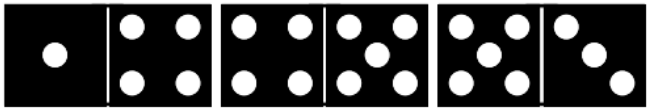


# Domino chains

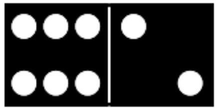
- Suppose we have the following dominoes:



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

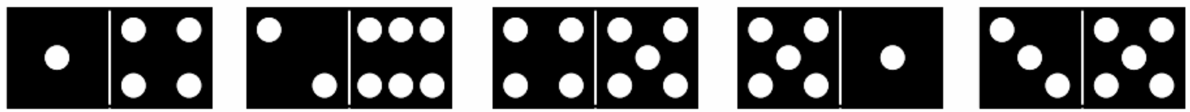


- We can "link" one domino into a "chain" from 6 to 2 as follows:



# 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:
    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;
    }
}
```