CS 106B, Lecture 13
Recursive Backtracking 3

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: Permute Vector

- Write a function **`permute`** that accepts a `Vector` of strings as a parameter and outputs all possible rearrangements of the strings in that vector. The arrangements may be output in any order.
  - Example: if v contains `{"a", "b", "c", "d"}`, your function outputs these permutations:

{a, b, c, d}	{b, a, c, d}	{c, a, b, d}	{d, a, b, c}
{a, b, d, c}	{b, a, d, c}	{c, a, d, b}	{d, a, c, b}
{a, c, b, d}	{b, c, a, d}	{c, b, a, d}	{d, b, a, c}
{a, c, d, b}	{b, c, d, a}	{c, b, d, a}	{d, b, c, a}
{a, d, b, c}	{b, d, a, c}	{c, d, a, b}	{d, c, a, b}
{a, d, c, b}	{b, d, c, a}	{c, d, b, a}	{d, c, b, a}

Backtracking Model

Choosing

1. We generally iterate over **decisions**. What are we iterating over here? **The position.** What are the **choices** for each decision? **Which string to choose**. Do we need a for loop? **Yes, over strings**.

Exploring

2. How can we *represent* that choice? **Vector<string>** How should we **modify the parameters** and **store our previous choices** (avoiding *arms-length* recursion)? **Build up the result Vector, remove chosen strings from the options Vector**
   a) Do we need to use a **wrapper** due to extra parameters? **Yes!**
3. How should we **restrict** our choices to be valid? **Only choose strings we haven't used**
4. How should we use the **return value** of the recursive calls? **No return value**. Are we looking for all solutions or just one? **all solutions**

Backtracking Model

Un-choosing

5. How do we **un-modify** the parameters from step 3? **Add the chosen string back to our Vector of options, remove it from the result Vector we're building**. 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**? **Print the result Vector**
7. Is there a case for when there **aren't any valid choices left** or a "bad" state is reached (usually return false)? **Not in this case**
8. Are the base cases ordered properly? Are we avoiding **arms-length** recursion? **We should always avoid arms-length recursion!**

Permute solution

```
// Outputs all permutations of the given vector.
void permute(Vector<string>& v) {
    Vector<string> chosen;
    permuteHelper(v, chosen);
}

void permuteHelper(Vector<string>& v, Vector<string>& chosen) {
    if (v.isEmpty()) {
        cout << chosen << endl;   // base case
    } else {
        for (int i = 0; i < v.size(); i++) {
            string s = v[i];
            v.remove(i);
            chosen.add(s);                      // choose
            permuteHelper(v, chosen);           // explore
            chosen.remove(chosen.size() - 1);   // un-choose
            v.insert(i, s);
        }
    }
}
```

Exercise: sublists

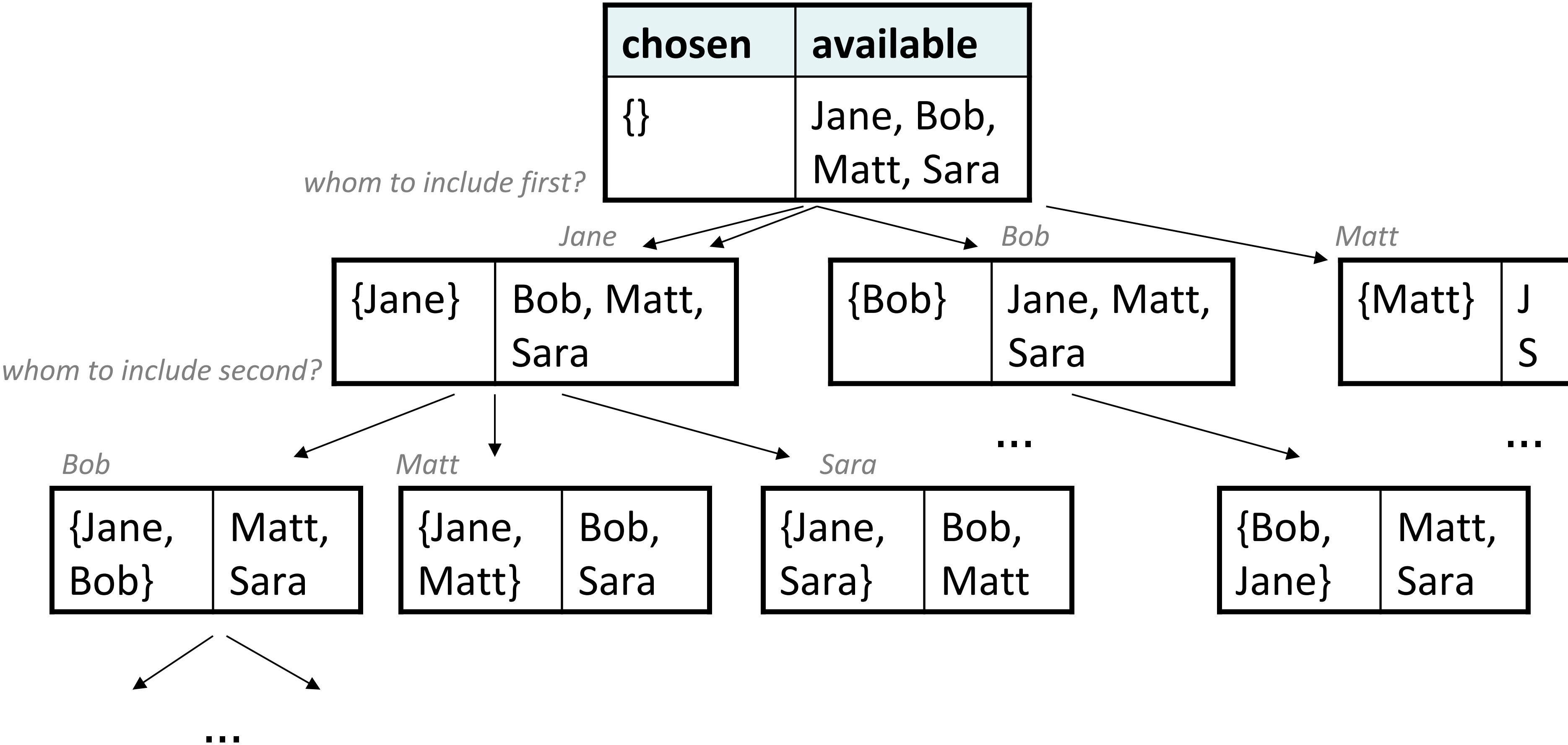
printSubVectors

- 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:

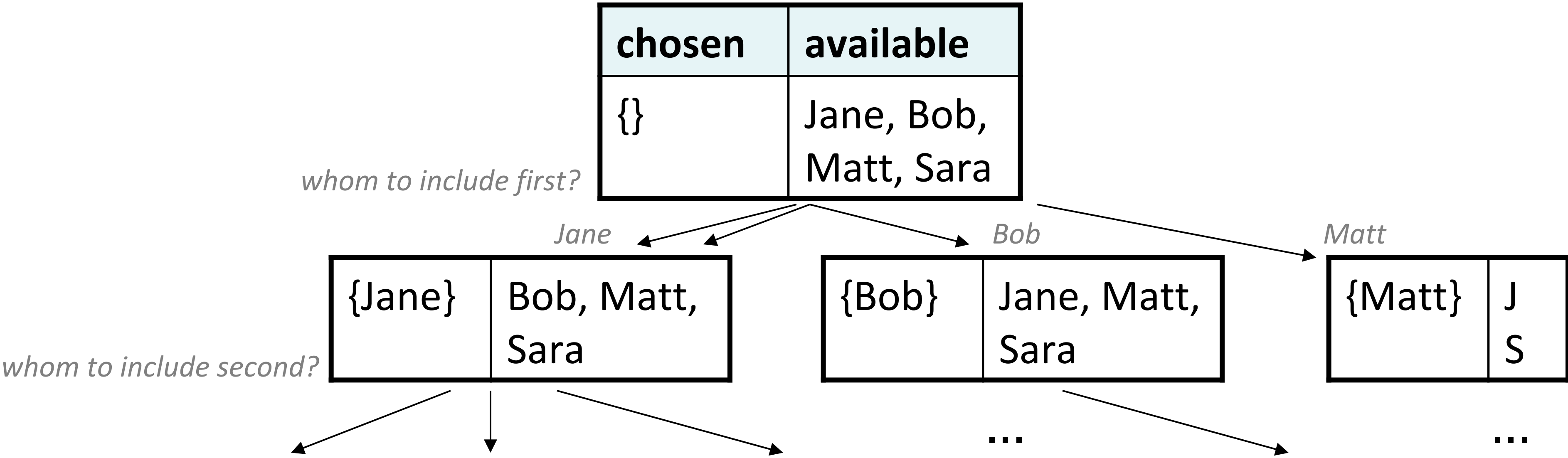
```
{Jane, Bob, Matt, Sara}
{Jane, Bob, Matt}
{Jane, Bob, Sara}
{Jane, Bob}
{Jane, Matt, Sara}
{Jane, Matt}
{Jane, Sara}
{Jane}
{Bob, Matt, Sara}
{Bob, Matt}
{Bob, Sara}
{Bob}
{Matt, Sara}
{Matt}
{Sara}
{}
```

  - 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?

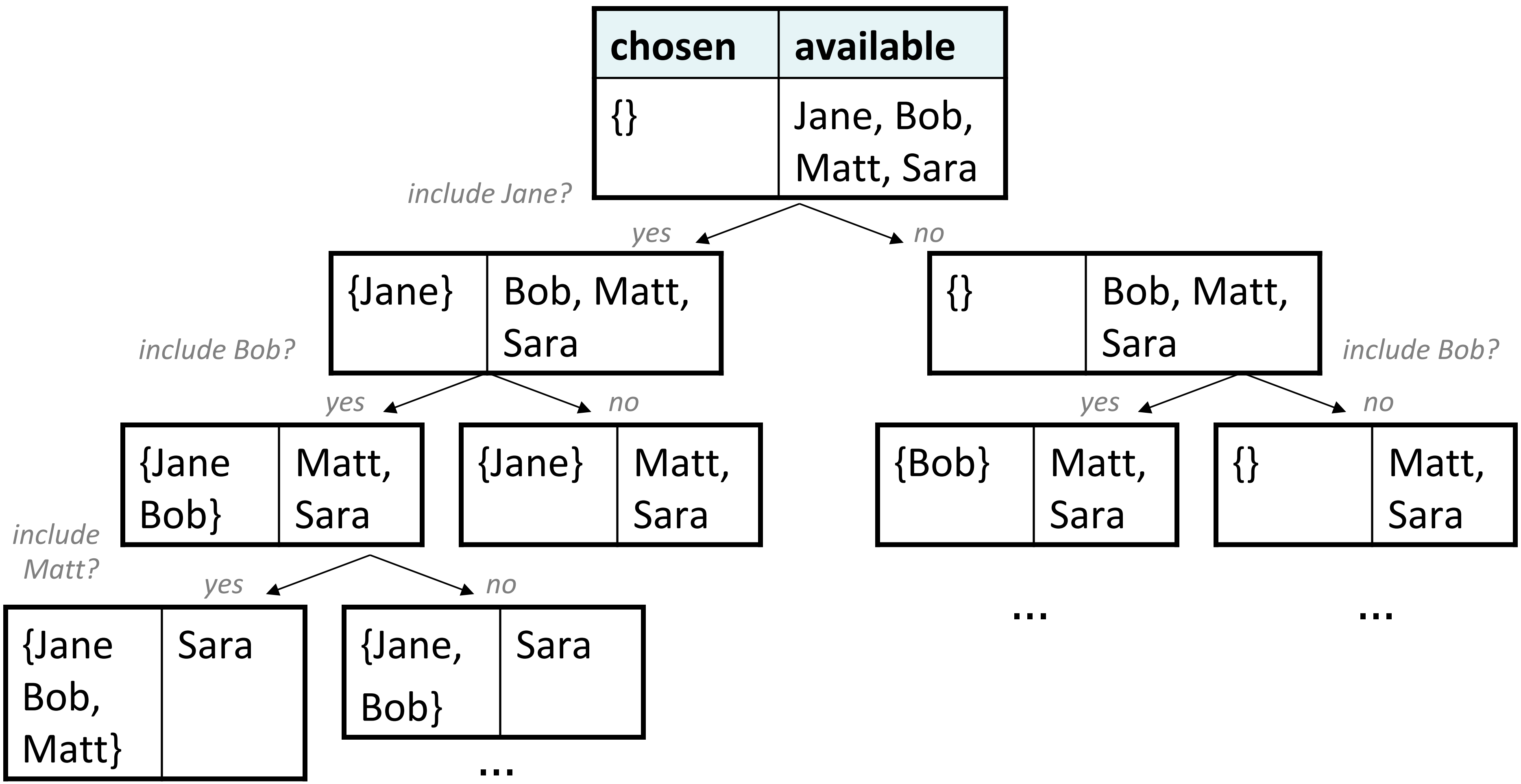
A. It does not actually end up finding every possible sublist.

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**.

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?

Backtracking Model

Choosing

1. We generally iterate over **decisions**. What are we iterating over here?

    Each element.

    What are the **choices** for each decision?

    Whether to include that element in the sublist.

    Do we need a for loop?

    No – only two options.

Exploring

2. How can we *represent* that choice?

    Vector<string>

    How should we **modify the parameters** and **store our previous choices** (avoiding *arms-length* recursion)?

    Build up the result Vector, keep track of which *index* to include

3. Are we looking for all solutions or just one?

    All solutions

Backtracking Model

Un-choosing

5. How do we **un-modify** the parameters from step 2?

   Remove the element from the Vector, if it was added.

Base Case

6. What should we do in the base case when we're **out of decisions**?

   Print the result Vector

7. Is there a case for when there **aren't any valid choices left** or a "bad" state is reached (usually return false)?

   Not in this case

sublists solution

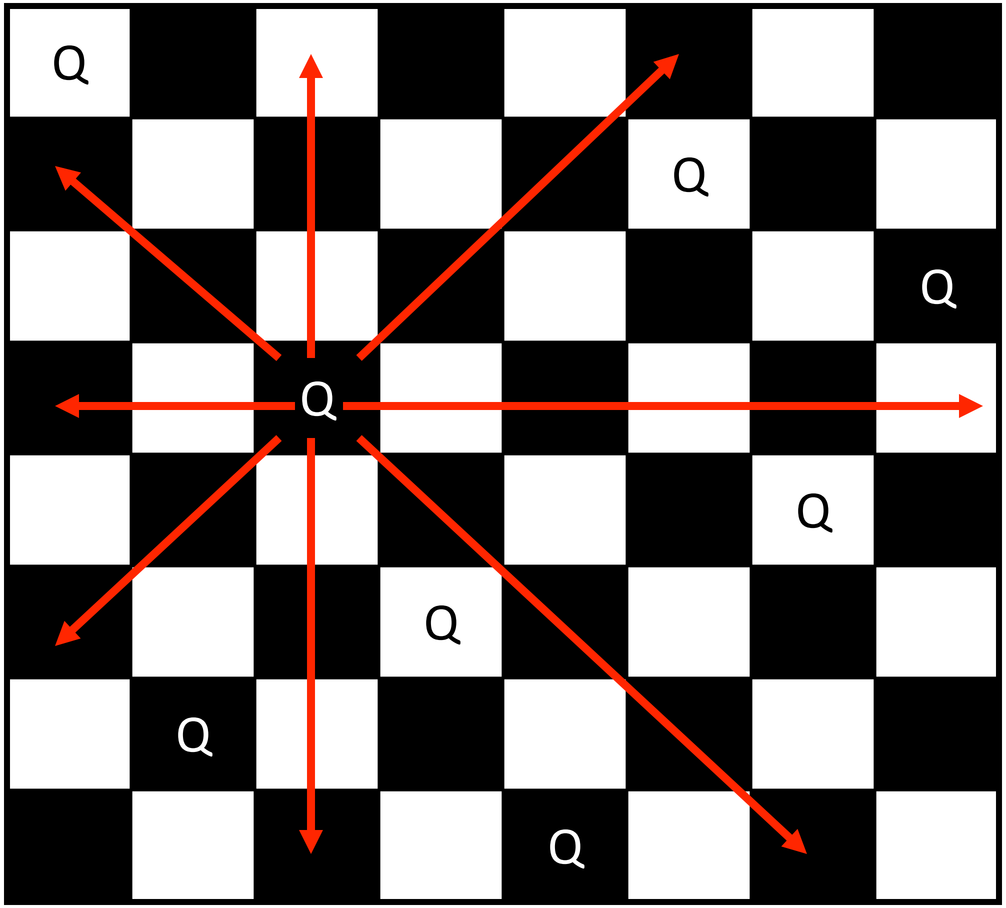
```
void sublists(Vector<string>& v) {
    Vector<string> chosen;
    sublistsHelper(v, 0, chosen);
}

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

- Thank you to Shreya for doing a great job covering lecture!
- Grades for assignment 2 will come out early tomorrow at the latest
- Exam logistics
  - Midterm review session in one week, from 7:00-9:00PM, in Gates B01, led by SL Peter
  - Midterm is on Wednesday, July 25, from 7:00-9:00PM
  - Midterm info (list of topics covered and study tips) online: https://web.stanford.edu/class/cs106b/exams/midterm.html
  - Practice exam will be posted by end of the day tomorrow
  - General tips: practice handwriting answers, use CodeStepByStep and section handouts for further practice
  - The exam will have code trace or reading questions in addition to code writing questions
  - Complete assignment 4 before the midterm – backtracking will be tested

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.

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

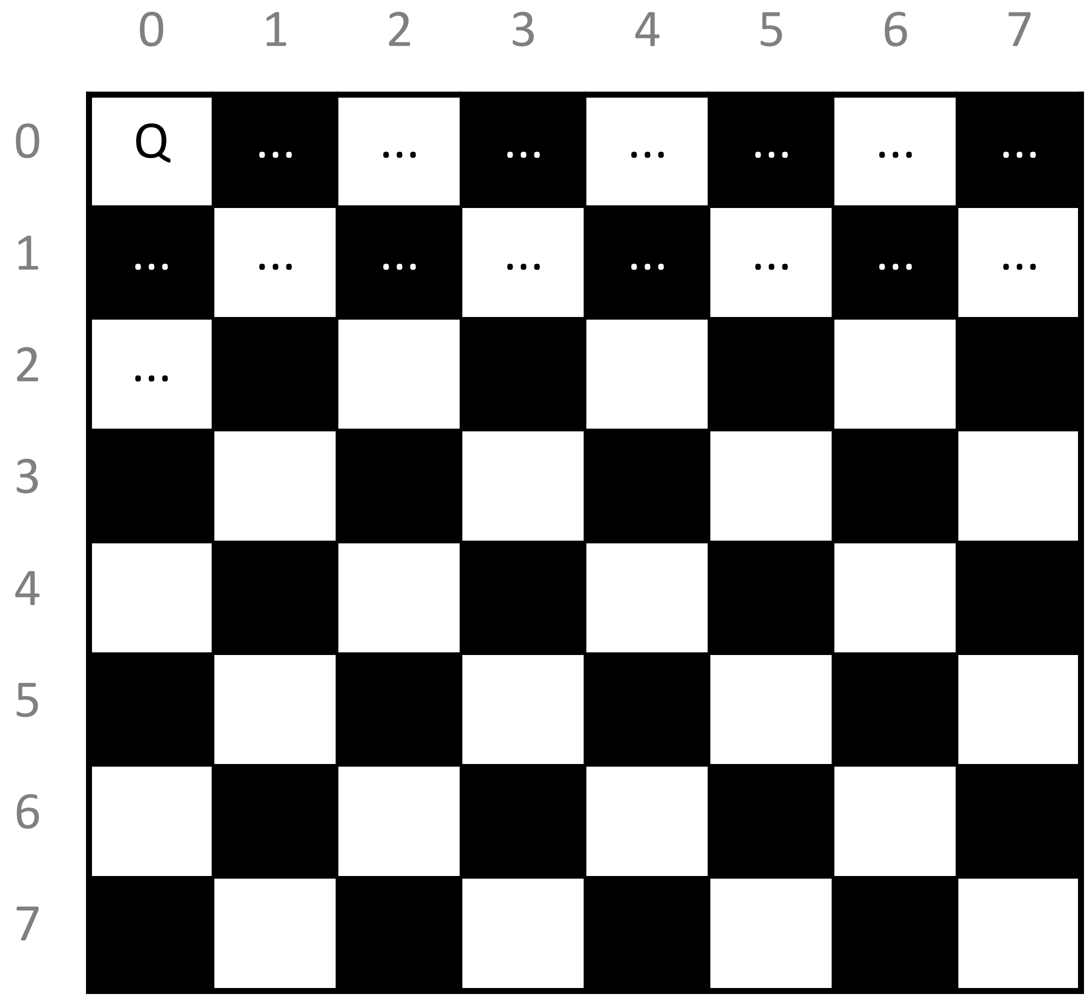
5. How do we **un-modify** the parameters from step 3? Do we need to explicitly un-modify, or are they copied? Are they un-modified at the same level as they were modified?

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?

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

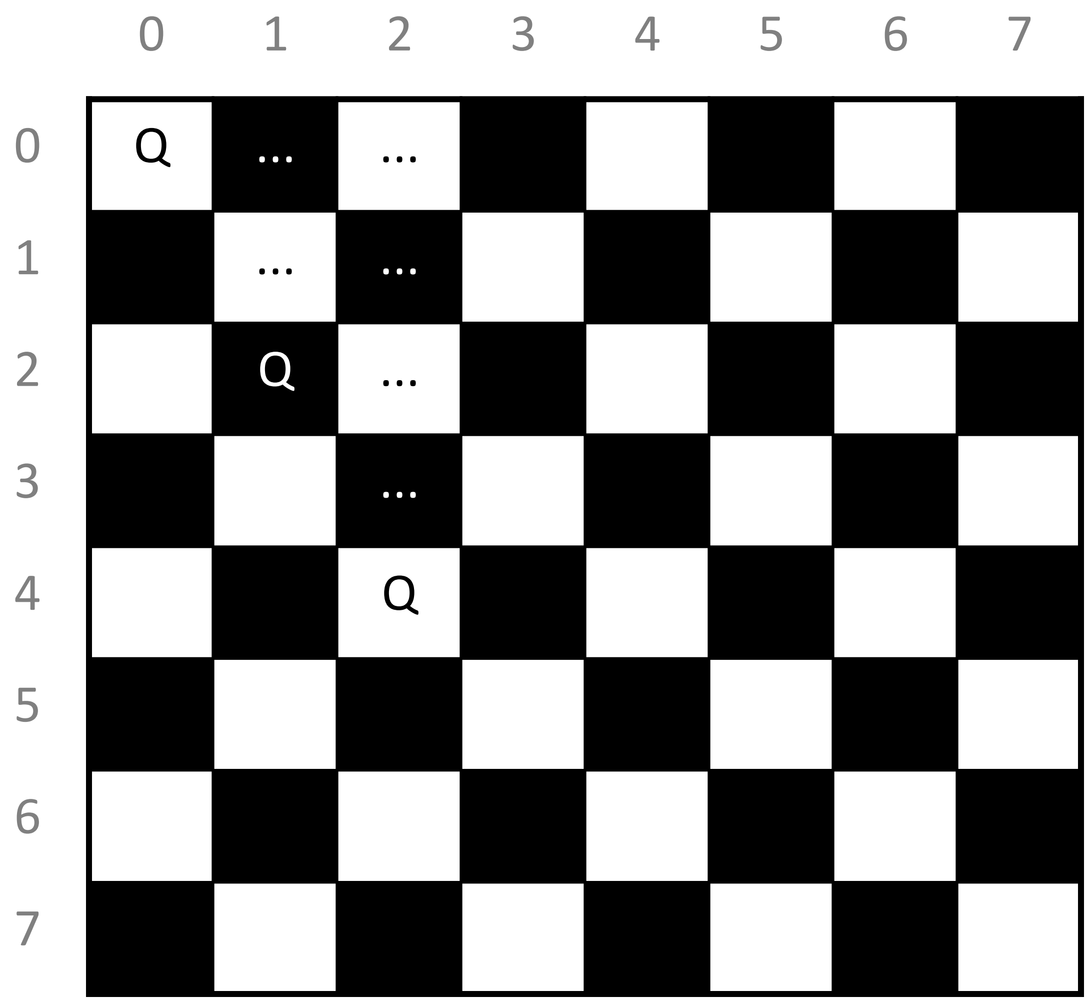
C. 64^8

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 * ...



Backtracking Model

Choosing

1. We generally iterate over **decisions**. What are we iterating over here?

   Each queen to place.

   What are the **choices** for each decision?

   Where in a column to place the queen.

   Do we need a for loop?

   Yes – 8 options.

Exploring

2. How can we *represent* that choice?

   Modify the board to place the queen

   How should we **modify the parameters** and **store our previous choices** (avoiding *arms-length* recursion)?

   Keep track of which column we should place next

3. How should we **restrict** our choices to be valid?

   Only place queens in their own column

3. Are we looking for all solutions or just one?

   Just one; we should return the board as an out parameter, and return a boolean

Backtracking Model

Un-choosing

5. How do we **un-modify** the parameters from step 2?

   Unplace the queen

Base Case

6. What should we do in the base case when we're **out of decisions**?

   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)?

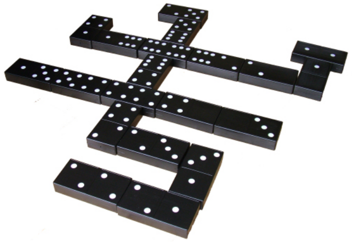
   Yes, the board could be invalid – that should be a base case.

   At the end of the function, we should return false

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



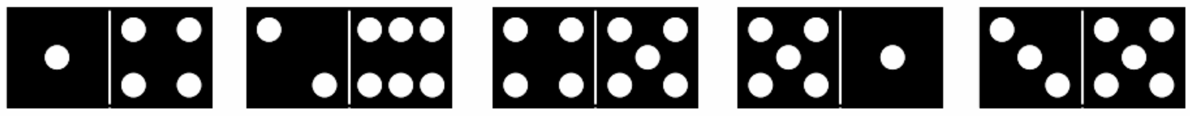
- Dominoes uses black tiles, each having 2 numbers of dots from 0-6. Players line up tiles to match dots.
- Given a class `Domino` with the following members:

```
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)"
```

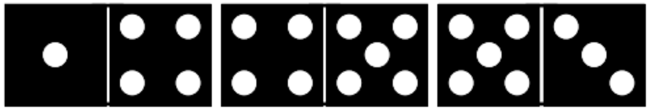
- Write a function **`chainExists`** that takes a `Vector` of dominoes and a starting/ending dot value, and returns whether the dominoes can be made into a chain that starts/ends with those values.

Domino chains

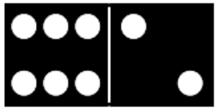
- Suppose we have the following dominoes:



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

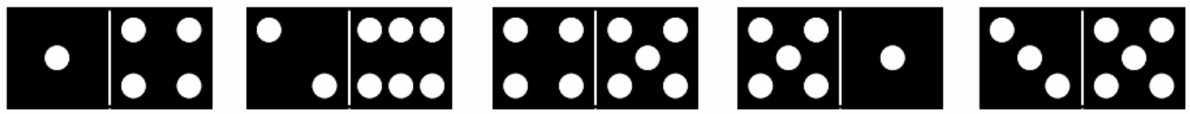


- We can "link" one domino into a "chain" from 6 to 2 as follows:



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.

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?

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;
    }
}
```