CS 106X, Lecture 10
Recursive Backtracking

reading:

Programming Abstractions in C++, Chapter 9

Personal Note about Backtracking

- Function design: 1. functionality
2. contex, precondition, assumption
3. input, output parameter

- Powerful Tool: Call/Recursive/Decision Tree
A useful way to design algorithm; See example:Sublists.
Thinking pattern: expand/penetrate single case each time to reach the base case.
Two process: down: call; up: return.

- Common mistake：misunderstanding of "each step's" choices
Thinking "steply": The function we design helps us solve problem in each step!
The parameters of the function indicate the state of each step: How many decisions (left) to make? How many available things-not necessarily choices- left ?

Common example: mix up permucation and combination question:
the former: diceRoll, 1~6 which one first?(order)
the later: subLists, include or exclude?(selection)
we need think "each step's" choices: include/exclude instead of considering all of choices we hace in this problem!!

- Other Notes:
undo: after all choices each step had been tried.

Return value function backtracking:

1. If backtrack with a function that has return value, remember use a variable to store the value while using recursion for the rest.

2. Find out the clearly meaning of the return value. Normally, the return value stores our answer to each step, so no more storage parameter is needed.

3. Relate to recursive leap of faith: I only solve one step, the rest someone can help me.

Call tree drawing: most of the parameter whose meaning should be clear should be included .

Void and Return Value Function Recuesive Thinking:

void: focus on functionality

return: focus on meaning of input args, return value.

Use of return value and parameter in a recursive function: pass and store information

parameter: give info to the next call

return value: give info to the previous call

Exhaustive search

- **exhaustive search**: Exploring every possible combination from a set of choices or values.
  - often implemented recursively

  Applications:
  - producing all permutations of a set of values
  - enumerating all possible names, passwords, etc.
  - combinatorics and logic programming

- Often the search space consists of many ***decisions***, each of which has several available ***choices***.
  - Example: When enumerating all 5-letter strings, each of the 5 letters is a *decision*, and each of those decisions has 26 possible *choices*.

Exhaustive search

A general pseudo-code algorithm for exhaustive search:

Search(decisions, storage).

Specifically: decisions:left

decisions

I decide to do sth….

function **Search** (*decisions*):

each call makes a choice

choose(try) –> explore –> …

- If there are decisions left to make:
  - // Let's handle one decision ourselves, and the rest by recursion.
  - For each available choice *C* for this decision:
    - **Choose** *C*.
    - **Search** the remaining decisions that could follow *C*.
- Otherwise, if there are no more decisions to make: Stop.

loop maybe to go over each choice you can make.

- *Observation:* The "**base case**" no longer represents a simple case of the algorithm; rather, it is the case where the algorithm is finished working and has no more choices left to make.

stop base case

Backtracking

- **backtracking**: Finding solution(s) by trying partial solutions and then abandoning them if they are not suitable.
  - a "brute force" algorithmic technique (tries all paths)
  - often implemented recursively

Applications:

- producing all permutations of a set of values
- parsing languages
- games: anagrams, crosswords, word jumbles, 8 queens
- combinatorics and logic programming
- escaping from a maze

Backtracking

A general pseudo-code algorithm for backtracking:

function **Search** (*decisions*):

- If there are decisions left to make:

  // Let's handle one decision ourselves, and the rest by recursion.

  - For each available choice *C* for this decision:
    - **Choose** *C*.
    - **Search** the remaining decisions that could follow *C*.
    - **Un-choose** *C*. *(backtrack!)*
- Otherwise, if there are no more decisions to make: Stop.

Note: **understanding what choices you have each step before starting is vitally important.**

decisions + choices/options+decision tree
choose(check valid!) –> explore –> unchoose

stop base case: validate the current solution! if true, return 1 or t; otherwise 0 or f

- Key tasks:
  - Figure out appropriate smallest unit of work (decision).
  - Figure out how to enumerate all possible choices/options for it.

Do an instance by yourself, write and divide what you've down, the smallest unit is the decision. (divide and conquer)

Backtracking in a nutshell

1. 明确每一步的decision（smallest work）和options，一次function call只干一步的事情; 同时搞清楚每一步(state)要记录的信息特征
2. 步骤：
- (选对象): 有时候在做decision选option之前，要先找到一个最选择的对象。比如decision: pick up a word; option: include or exclude
这里a word就比较抽象，就需要在已有的所有words里面选一个（参考sublists）选对象可能要做标记
- for each options
    choose->explore->unchoose
3. base case的设计：没decisions了；验证答案是否是valid candidate
验证是否得到答案（先）
4. valid candidate: have the possibility to become a solution

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?

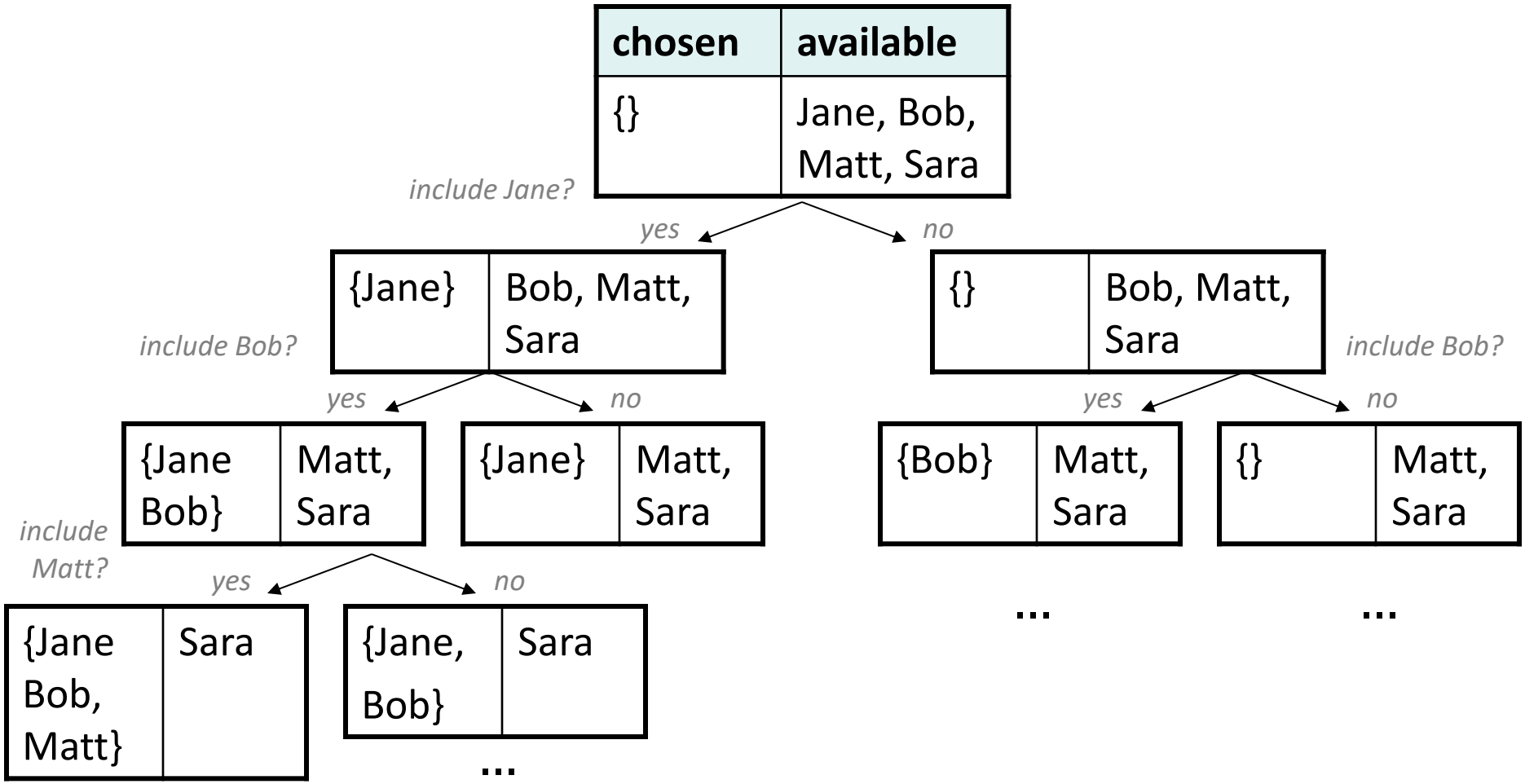
Un-choosing

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?

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?
  - *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*

The Backtracking Checklist

- ❑ **Find what choice(s) we have at each step.** What different options are there for the next step?

For each valid choice:

- ❑ **Make it and explore recursively.** Pass the information for a choice to the next recursive call(s).
- ❑ **Undo it after exploring.** Restore everything to the way it was before making this choice.

- ❑ **Find our base case(s).** What should we do when we are out of decisions?

Plan For Today

- **Recap:** what is recursive backtracking?
- **Recap:** dice rolls
- **Recap:** subsets
- Selection Problems: Subsets, Combinations and Permutations
- **Break:** Announcements
- Practicing types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*

含返回值的backtrack，非常推荐回到递归的方式去理解返回值的含义：当前这一步的返回值和之后返回值的联系和含义（recursive step + leap of faith）
explore的步骤似乎就是：当前要做decision：参加高等教育：有种choices：高考，出国，艺考。我把每种choice都explore（模拟人生？）一遍，比较不同choice的结果得出当下的结论（返回值）。
key：如果我把后面的可能都explore一遍并且假设我都看到了这些可能的结果，当前这一步我要怎么做，未来和explore的部分有什么联系？

one solution：假设我做了这个选择，如果我知道后面能找到一个true的解，说明我当前的选的这个选项是对的，我也return true；
best solution：我当下所有可能的choice都选一遍，假设知道了所有他们的结果，再比较找最优
count solution：当前选A能带来a个解，选B能带来b个解，那么当前这一步总的解的个数就是a+b个。

Types of Selection Problems

- **Subsets** are zero or more elements from a group of elements.
- **Combinations** are ways you can choose exactly K elements from a group of elements.
- **Permutations** are ways you can order a group of elements.

Backtracking: All Solutions

A general pseudo-code algorithm for backtracking problems searching for all solutions

Backtrack(*decisions*):

- if there are no more decisions to make:
  - if our current solution is valid, add it to our list of found solutions
  - else, do nothing or return
- else, let's handle one decision ourselves, and the rest by recursion. for each available **valid** choice *C* for this decision:
  - **Choose** *C* by modifying parameters.
  - **Explore** the remaining decisions that could follow *C*. Keep track of which solutions the recursive calls find.
  - **Un-choose** *C* by returning parameters to original state (if necessary).
- Return the list of solutions found by all the helper recursive calls.

Backtracking: One Solution

A general pseudo-code algorithm for backtracking problems searching for one solution

Backtrack(*decisions*):

- if there are no more decisions to make:
  - if our current solution is valid, return **true**
  - else, return **false** tell upper function stop
- else, let's handle one decision ourselves, and the rest by recursion.
  for each available **valid** choice *C* for this decision:
  - **Choose** *C* by modifying parameters. receive information from called function bool result = call function.
  - **Explore** the remaining decisions that could follow *C*. If any of them find a solution, return **true**
  - **Un-choose** *C* by returning parameters to original state (if necessary).

after going through all solutions: if nothing found the return false

- If no solutions were found, **return false**

Find a Solution

- **Base case:** validate the solution so far; return the solution if it's valid, or an empty solution otherwise.
- **Recursive step:** check all potential choices. If one returns a valid solution, return that. Otherwise, return an empty solution.
- Consider passing the solution by reference, and returning a boolean indicating whether a solution was found.

Count Solutions

- Base case: is it a valid solution? If so, return 1. Otherwise, return 0.
- Recursive step: return the sum of all the recursive calls.

参考leetcode n-queens ii

This approach is useful because sometimes we want to make sure that there is *exactly* one solution. For instance, a maze!

Find the Best Solution

- Base case: is it a valid solution? If so, return it. Otherwise, return a default/empty solution.
- Recursive step: check all potential choices, then output the "best" of all of them.

不同choices之间出现的结果要比较找best

Exercise: subsets

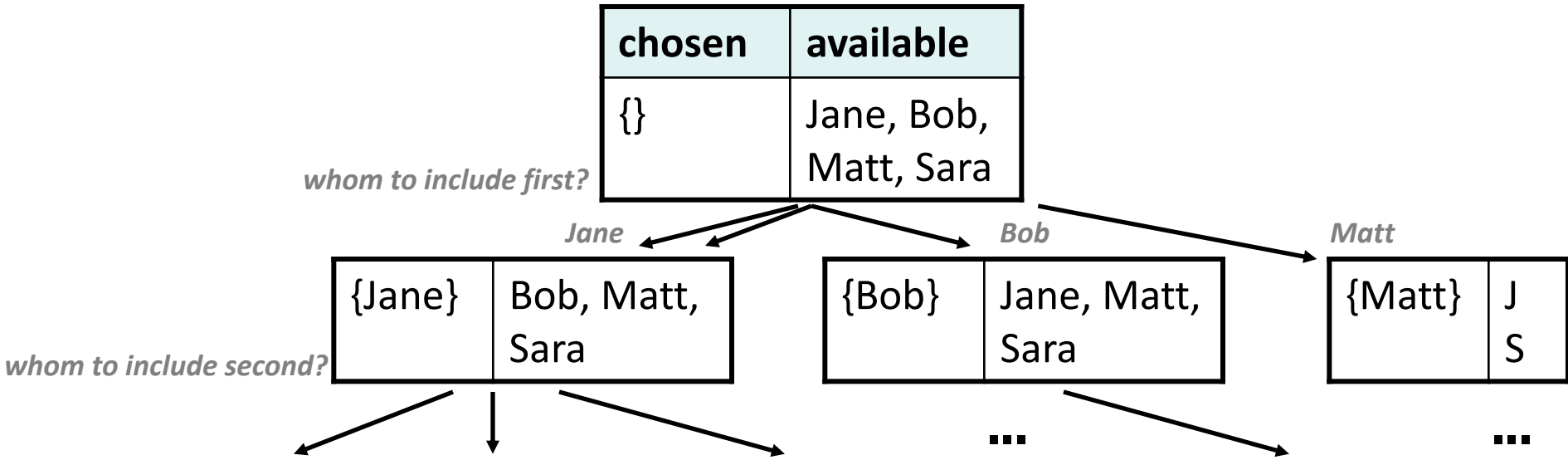
printSubVectors

- Write a function **`subsets`** 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 **`subsets`**`(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 subsets out in any order, one per line.
    - *What are the "choices" in this problem? (choose, explore)*

Wrong decision tree



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

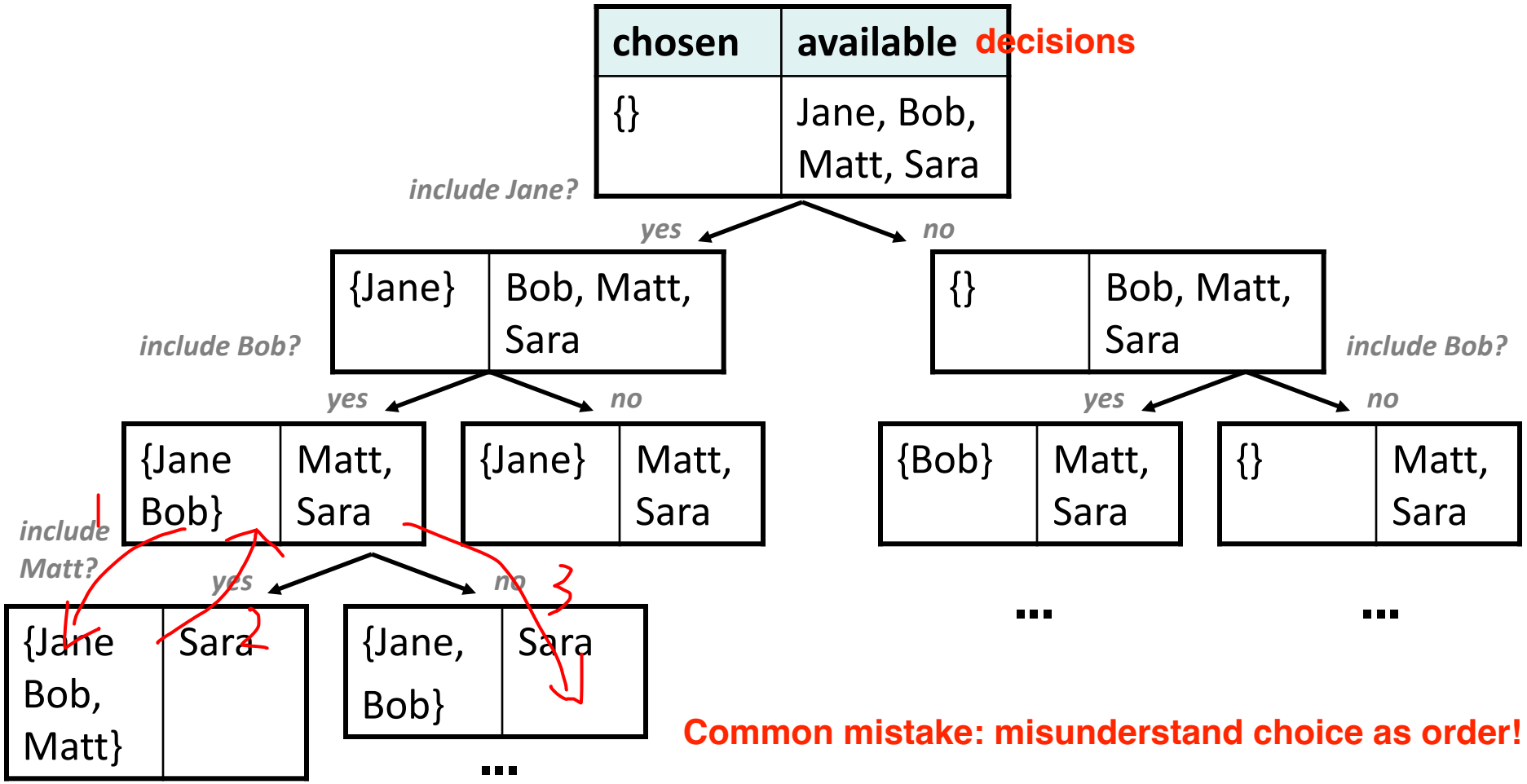
A. It does not actually end up finding every possible subset.

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

C. It does find all subset, but it finds them multiple times.

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

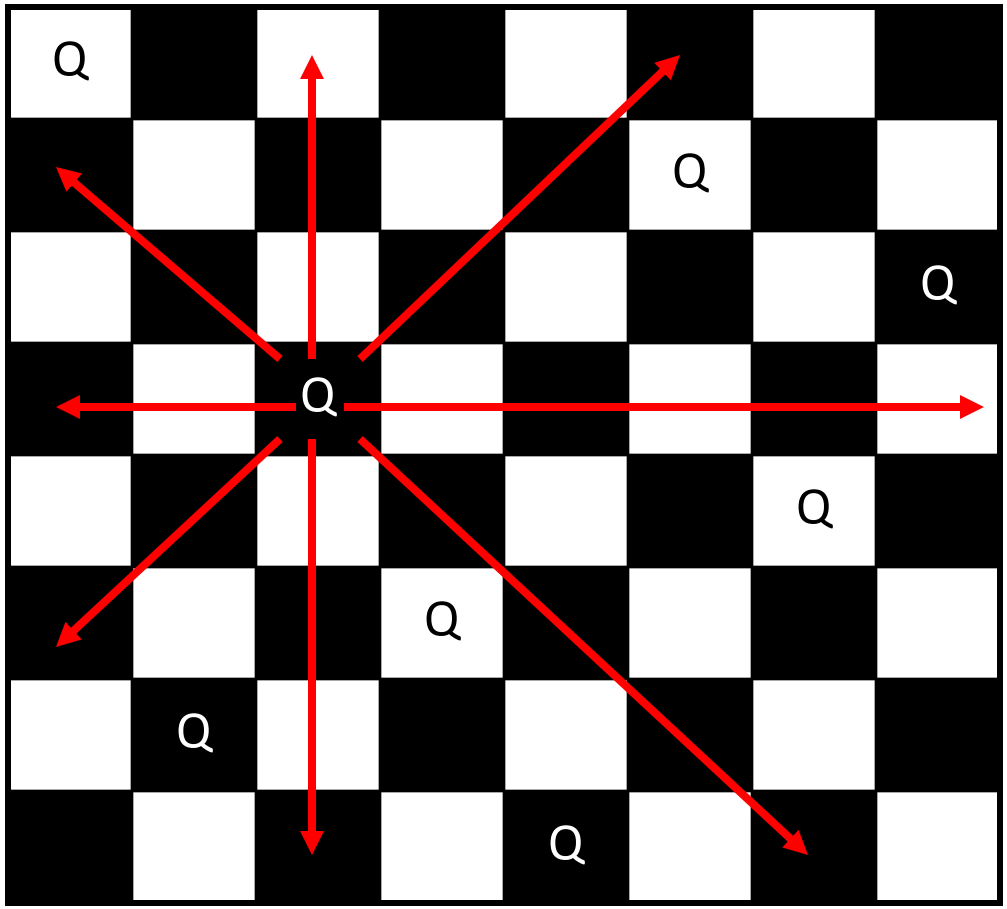
```
void subSets(const Set<string>& masterSet) {
    Set<string> chosen;
    listSubsetsRec(masterSet, chosen);
}

void listSubsetsRec(const Set<string>& masterSet, const Set<string>& used) {
    if (masterSet.isEmpty()) {
        cout << used << endl;
    } else {
        string element = masterSet.first();

        listSubsetsRec(masterSet - element, used);           // Without
        listSubsetsRec(masterSet - element, used + element); // With
    }
}
```

sublists solution

```
void subSets(Set<string>& masterSet) {
    Set<string> chosen;
    listSubsetsRec(masterSet, chosen);
}

void listSubsetsRec(Set<string>& masterSet, Set<string>& used) {
    if (masterSet.isEmpty()) {
        cout << used << endl;
    } else {
        string element = masterSet.first();

        masterSet.remove(element);
        listSubsetsRec(masterSet, used);           // Without

        used.add(element);
        listSubsetsRec(masterSet, used); // With
        masterSet.add(element);
        used.remove(element);
    }
}
```

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.

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?

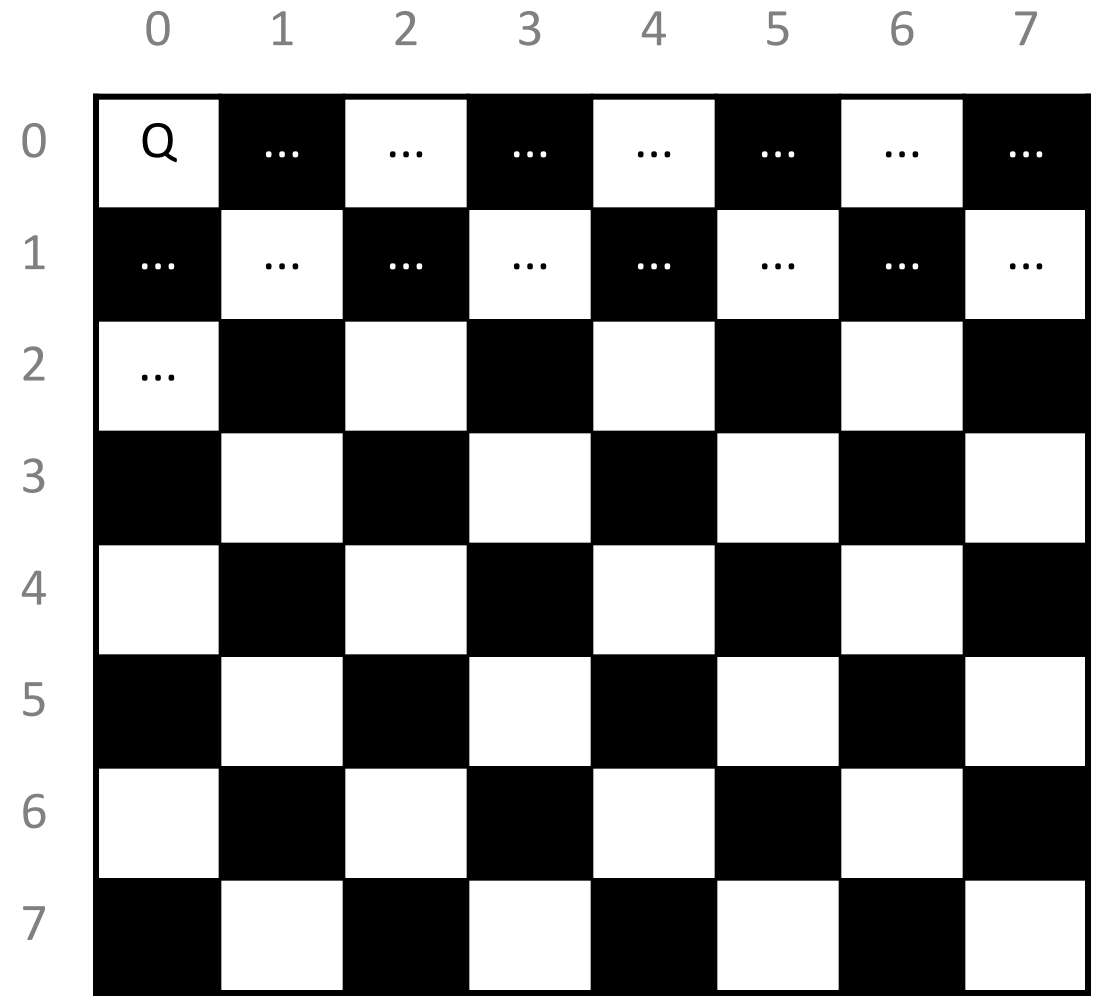
A. 64 choices

B. 64 * 8

C. 64^8

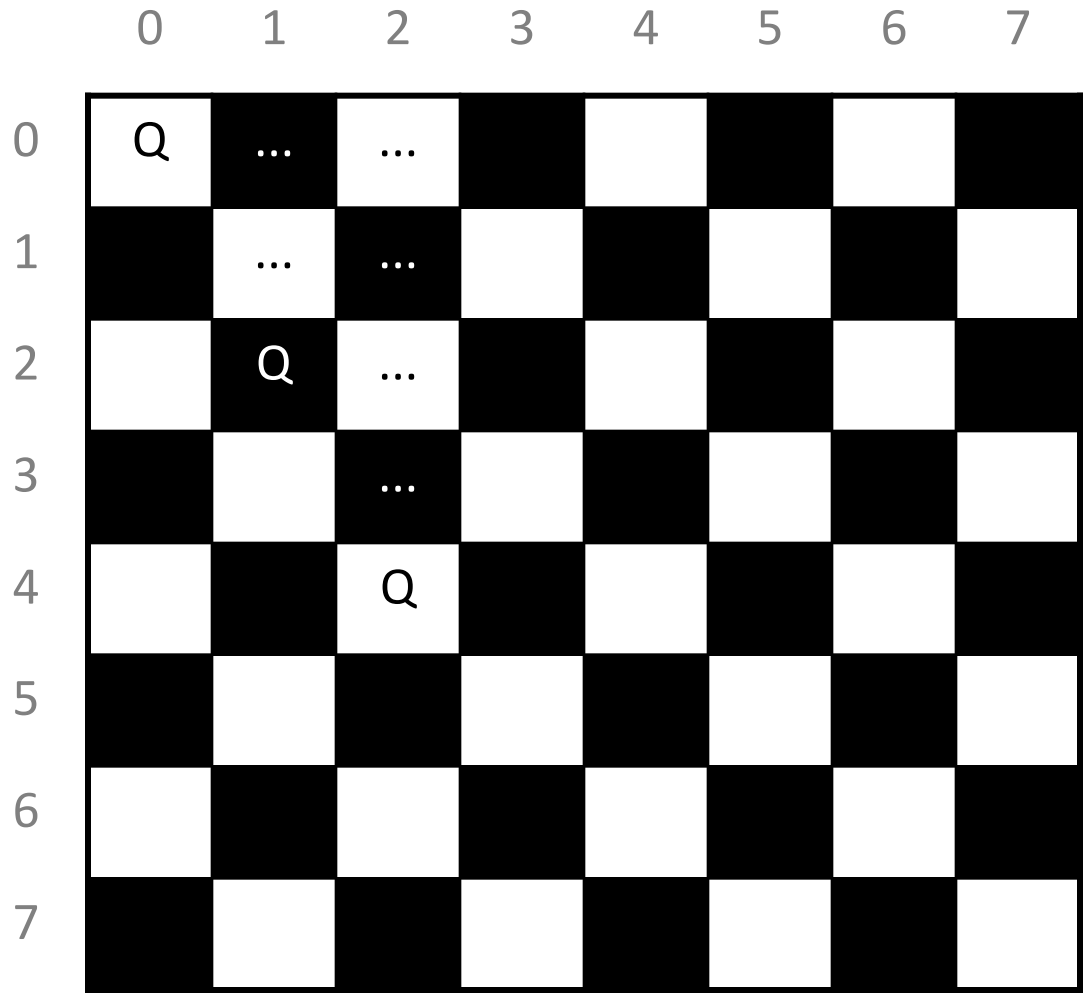
D. 64*63*62*61*60*59*58*57

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 * …



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);
}
```