CS 106B, Lecture 10
Recursive Backtracking 2

reading:

Programming Abstractions in C++, Chapters 8.2 - 8.3; 9

Backtracking

A general pseudo-code algorithm for backtracking:

function **Explore** (*decisions*):

- If there are decisions left to make:

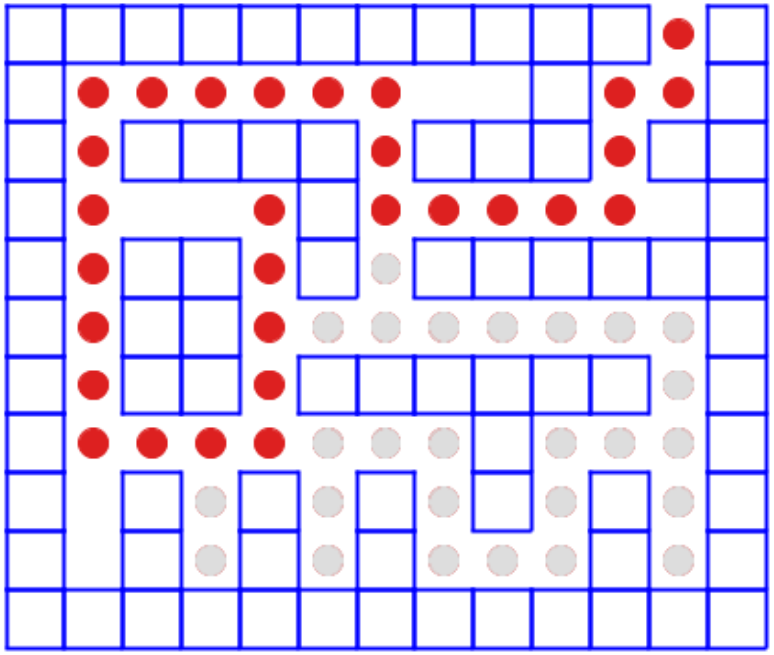
  // Let's handle one decision ourselves, and the rest by recursion.

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

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

Escape Maze exercise

- Write a function **`escapeMaze`**`(maze, row, col)` that searches for a path out of a given 2-dimensional maze.
  - Return `true` if able to escape, or `false` if not.
  - "Escaping" means exiting the maze boundaries.
  - You can move 1 square at a time in any of the 4 directions.
  - "Mark" your path along the way.
  - "Taint" bad paths that do not work.
  - Do not explore the same path twice.



Maze class

- `#include "Maze.h"`

Member name	Description
`m.inBounds(row, col)`	true if within maze boundaries
`m.isMarked(row, col)`	true if given cell is marked
`m.isOpen(row, col)`	true if given cell is empty (no wall or mark)
`m.isTainted(row, col)`	true if given cell has been tainted
`m.isWall(row, col)`	true if given cell contains a wall
`m.mark(row, col);`	sets given cell to be marked
`m.numRows(), m.numCols()`	returns dimensions of maze
`m.taint(row, col);`	sets given cell to be tainted
`m.unmark(row, col);`	sets given cell to be not marked if marked
`m.untaint(row, col);`	sets given cell to be not tainted if tainted

Escape Maze solution

```
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: 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}

Examining the problem

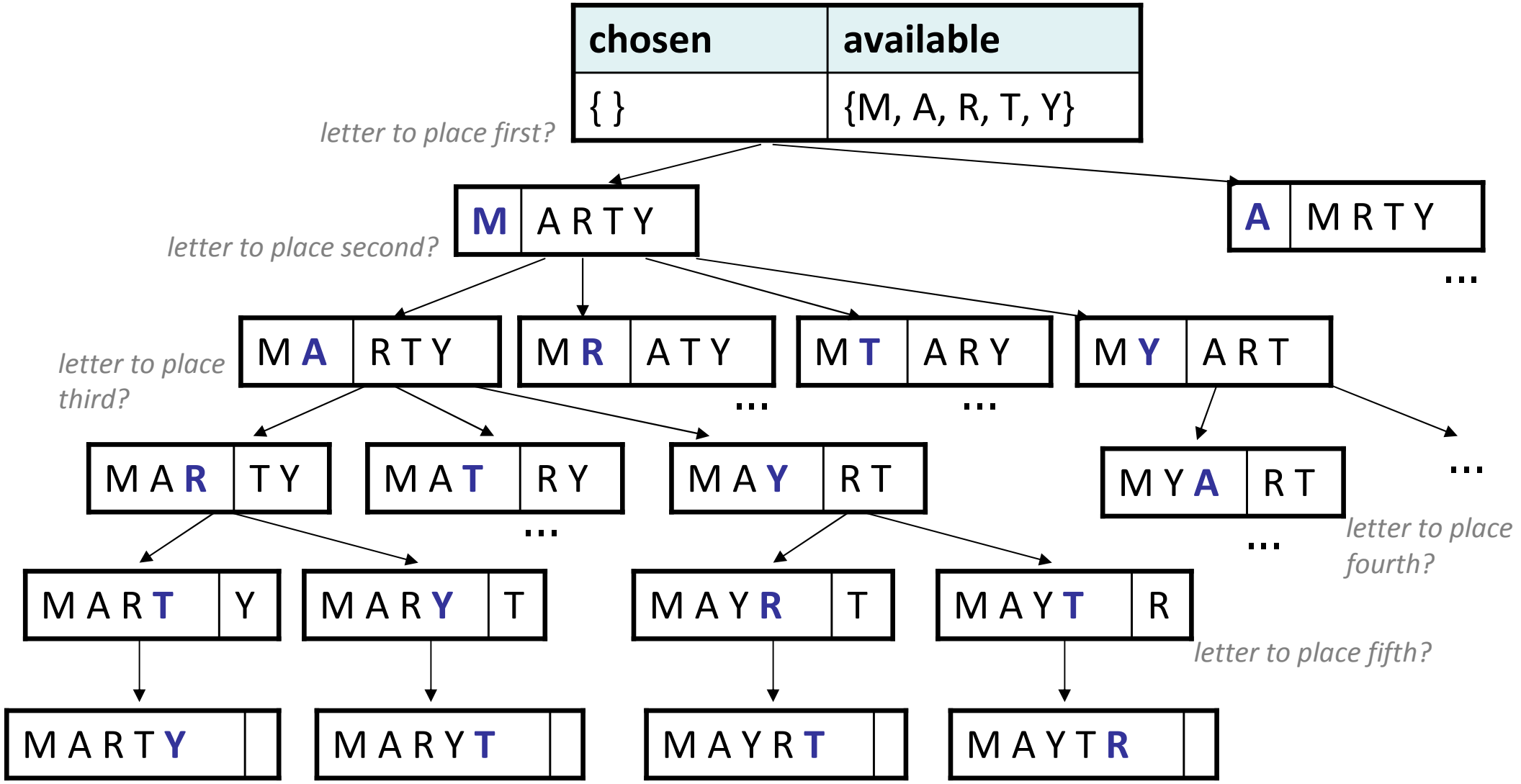
- Think of each permutation as a set of choices or **decisions**:
  - Which character do I want to place first?
  - Which character do I want to place second?
  - …
  - **solution space**: set of all possible sets of decisions to explore

- We want to generate all possible sequences of decisions.

```
for (each possible first letter):
    for (each possible second letter):
        for (each possible third letter):
            …
                print!
```

  - This is called a **depth-first search**

Decision tree



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);
        }
    }
}
```

Permute a string

```
// Outputs all permutations of the given string.
void permute(string s, ) {

}

void permuteHelper(string s, string chosen = "") {
    if (s == "") {
        cout << chosen << endl;   // base case: no choices left
    } else {
        // recursive case: choose each possible next letter
        for (int i = 0; i < s.length(); i++) {
            string rest = s.substr(0, i) + s.substr(i + 1);
            permuteHelper(rest, chosen + s[i]);   // choose/explore
        }
    }
}
```

Exercise: sublists

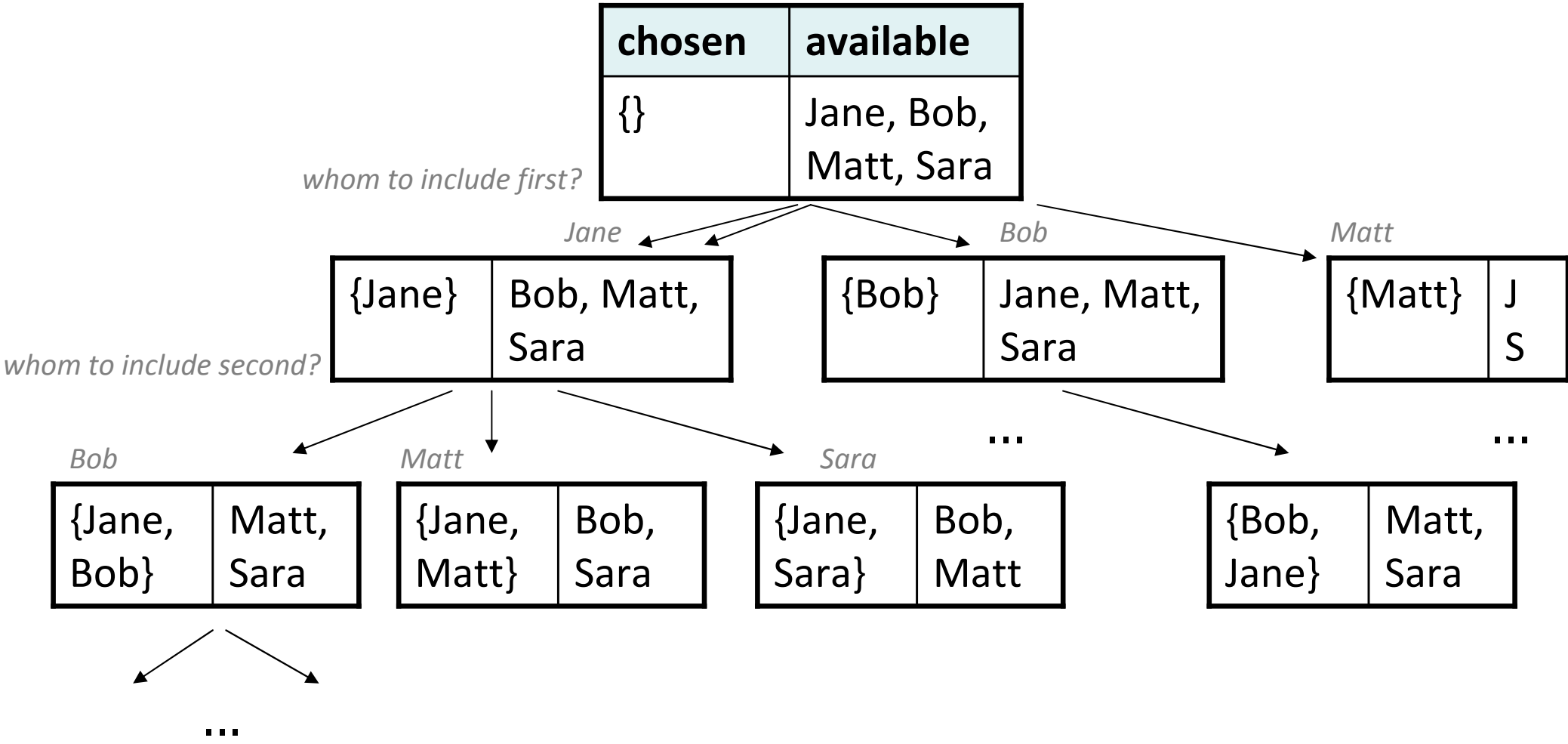
printSubVectors

- 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:

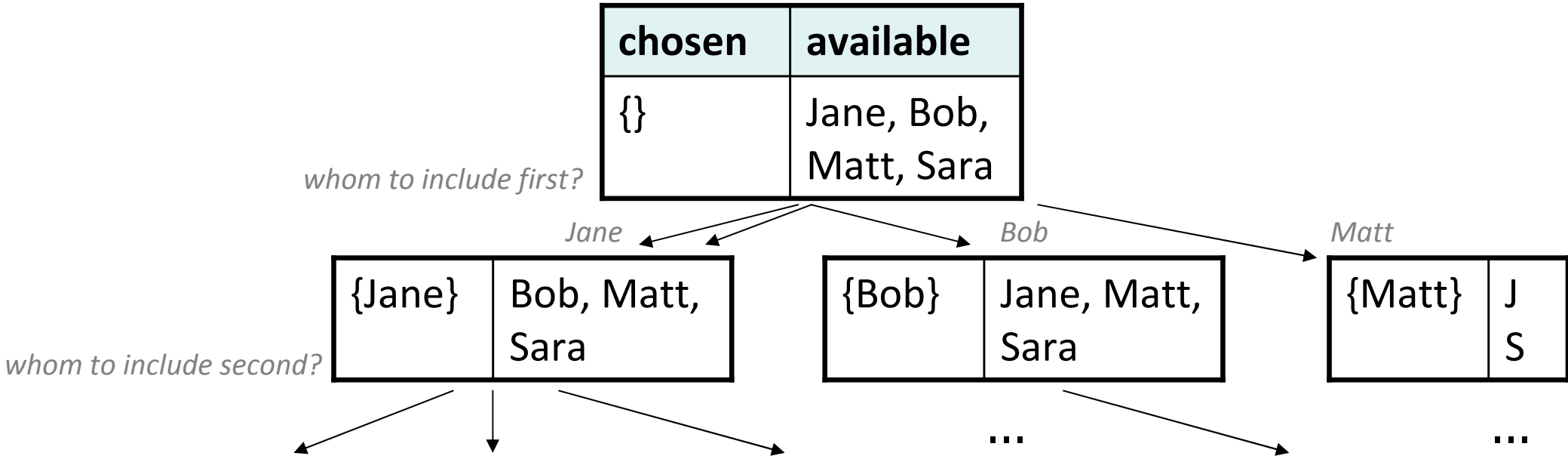
```
{Jane, Bob, Matt, Sara}
{Jane, Bob, Matt}
{Jane, Bob, Sara}
{Jane, Bob}
{Jane, Matt, Sara}
{Jane, Matt}
{Jane, Sara}
{Jane}
{Bob, Matt, Sara}
{Bob, Matt}
{Bob, Sara}
{Bob}
{Matt, Sara}
{Matt}
{Sara}
{}
```

  - 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?

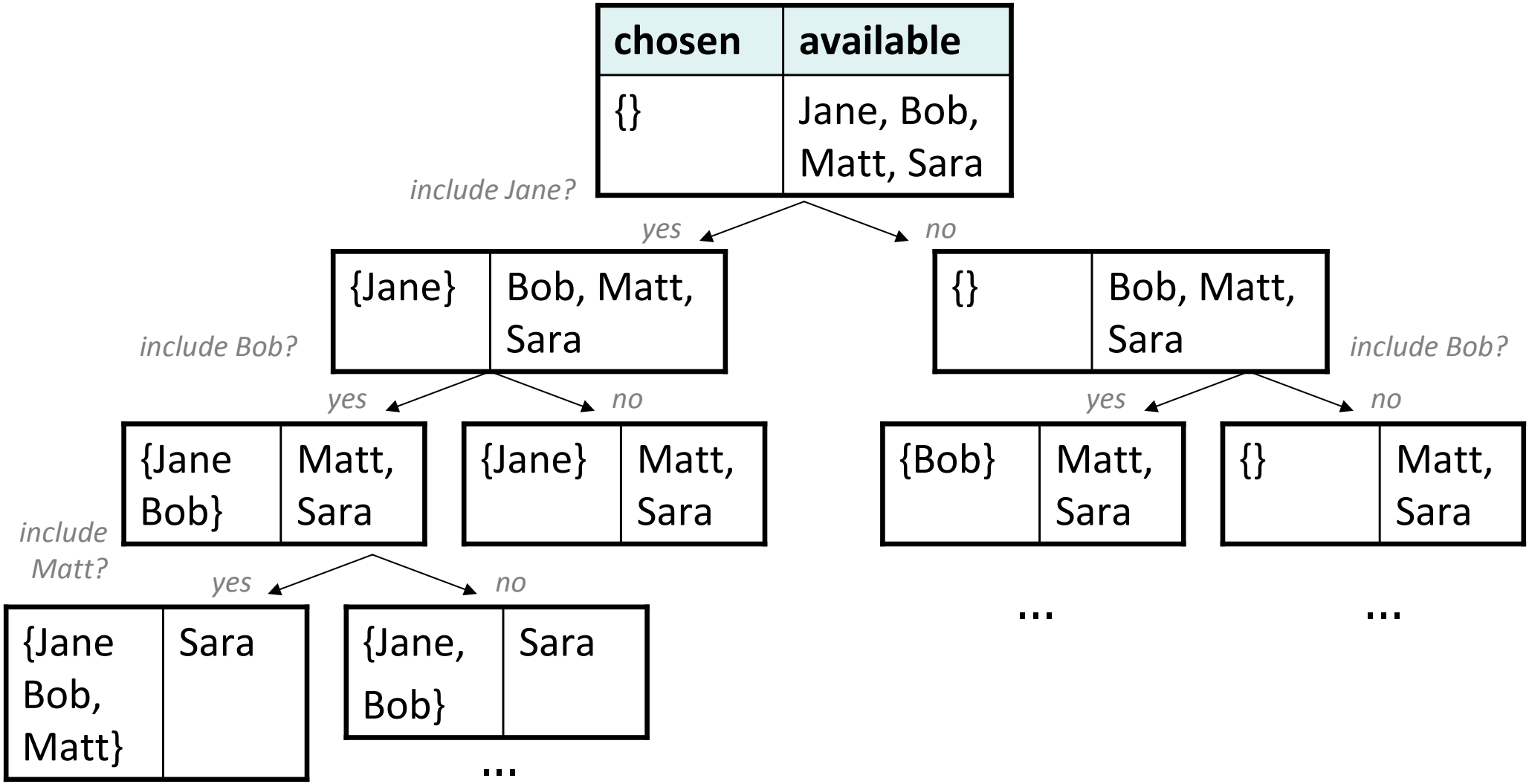
A. It does not actually end up finding every possible subset.

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
    }
}
```

Overflow (extra) slides

Exercise: Combinations

- Write a function **`combinations`** that accepts a string *s* and an integer *k* as parameters and outputs all possible *k* -letter strings that can be formed from unique letters in that string. The arrangements may be output in any order.
  - Example:
    `combinations("GOOGLE", 3)`
    outputs the sequence of
    lines at right.

EGL	LEG
EGO	LEO
ELG	LGE
ELO	LGO
EOG	LOE
EOL	LOG
GEL	OEG
GEO	OEL
GLE	OGE
GLO	OGL
GOE	OLE
GOL	OLG

  - To simplify the problem, you may assume that the string *s* contains at least *k* unique characters.

Exercise solution

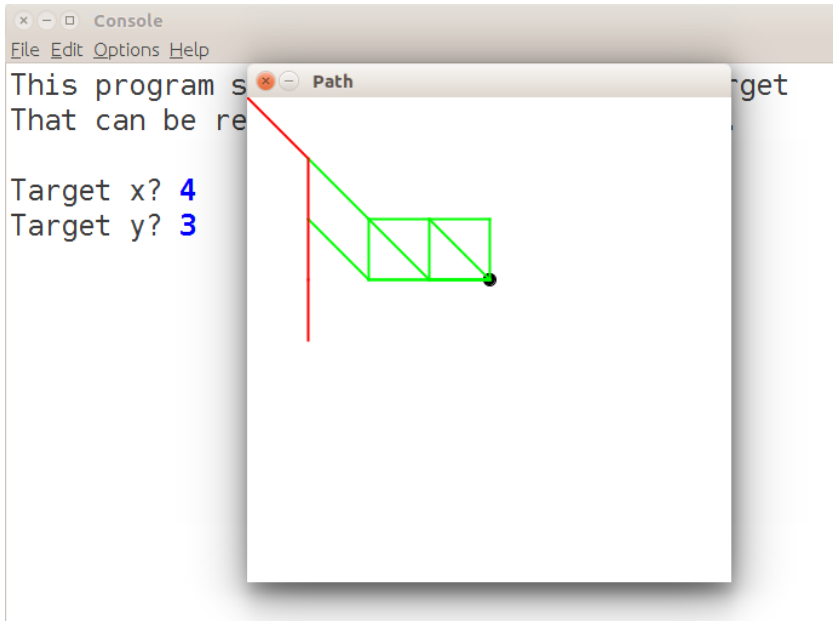
```
// Outputs all unique k-letter combinations of the given string.
void combinations(string s, int length) {
    Set<string> found;
    combinHelper(s, length, "", found);
}

void combinHelper(string s, int length, string chosen,
                  Set<string>& found) {
    if (length == 0 && !found.contains(chosen)) {
        cout << chosen << endl;   // base case: no choices left
        found.add(chosen);
    } else {
        for (int i = 0; i < s.length(); i++) {
            // try this letter, if not already used
            if (chosen.find(s[i]) == string::npos) {
                combinHelper(s.substr(0, i) + s.substr(i + 1),
                             length - 1, chosen + s[i], found);
            }
        }
    }
}
```

Exercise: Travel

- Write a function **`travel`** that accepts an x/y position (`GPoint`) parameter and prints or displays all ways to walk from (0, 0) to that point by taking single steps North, East, or Northeast.
  - Example: `travel(2, 1);` might print:

```
E  E  N
E  N  E
E  NE
N  E  E
NE  E
```

  - For extra fun, try drawing the paths on a graphical window.



Travel solution 1

```
// basic version (no drawing)
void travel(const GPoint& target, const GPoint& curr,
            const string& choices = "") {
    if (target == curr) {
        // base case: found a path
        cout << choices << endl;
    } else if (curr.getX() <= target.getX()
            && curr.getY() <= target.getY()) {
        // try each of 3 paths
        GPoint n(curr.getX(), curr.getY() + 1);
        GPoint e(curr.getX() + 1, curr.getY());
        GPoint ne(curr.getX() + 1, curr.getY() + 1);

        travel(window, target, n, choices + "N ");
        travel(window, target, e, choices + "E ");
        travel(window, target, ne, choices + "NE ");
    }
}
```

Travel solution 2

```
const int XY_SCALE = 50;

bool travel(GWindow& window, const GPoint& dst, GPoint prev, GPoint cur) {
    GLine line(prev * XY_SCALE, cur * XY_SCALE);
    line.setColor("red");
    window.add(line);
    if (dst == cur) {   // base case: found a path
        line.setColor("green");
        return true;
    } else if (cur.getX() <= dst.getX() && cur.getY() <= dst.getY()) {
        GPoint north(cur.getX(), cur.getY() + 1);   // try each of 3 paths
        GPoint east(cur.getX() + 1, cur.getY());
        GPoint northeast(cur.getX() + 1, cur.getY() + 1);
        bool result1 = travel(window, dst, cur, northeast);
        bool result2 = travel(window, dst, cur, north);
        bool result3 = travel(window, dst, cur, east);
        if (result1 || result2 || result3) {
            line.setColor("green");
            return true;
        }
    }

    window.remove(line);   // didn't find a path
    return false;
}
```