CS 106X, Lecture 10
Recursive Backtracking

reading:

Programming Abstractions in C++, Chapter 9

Plan For Today

- Ex. Printing Binary
- What is Recursive Backtracking?
- Ex. Dice Sums
- **Announcements**
- Ex. Subsets

Plan For Today

- Ex. Printing Binary
- What is Recursive Backtracking?
- Ex. Dice Sums
- **Announcements**
- Ex. Subsets

Exercise: printAllBinary

- Write a recursive function **`printAllBinary`** that accepts an integer number of digits and prints all binary numbers that have exactly that many digits, in ascending order, one per line.
  - `printAllBinary(2);`

```
00
01
10
11
```

`printAllBinary(3);`

```
000
001
010
011
100
101
110
111
```

❑Wrapper? ❑Base Case? ❑Recursive case? ❑All inputs?

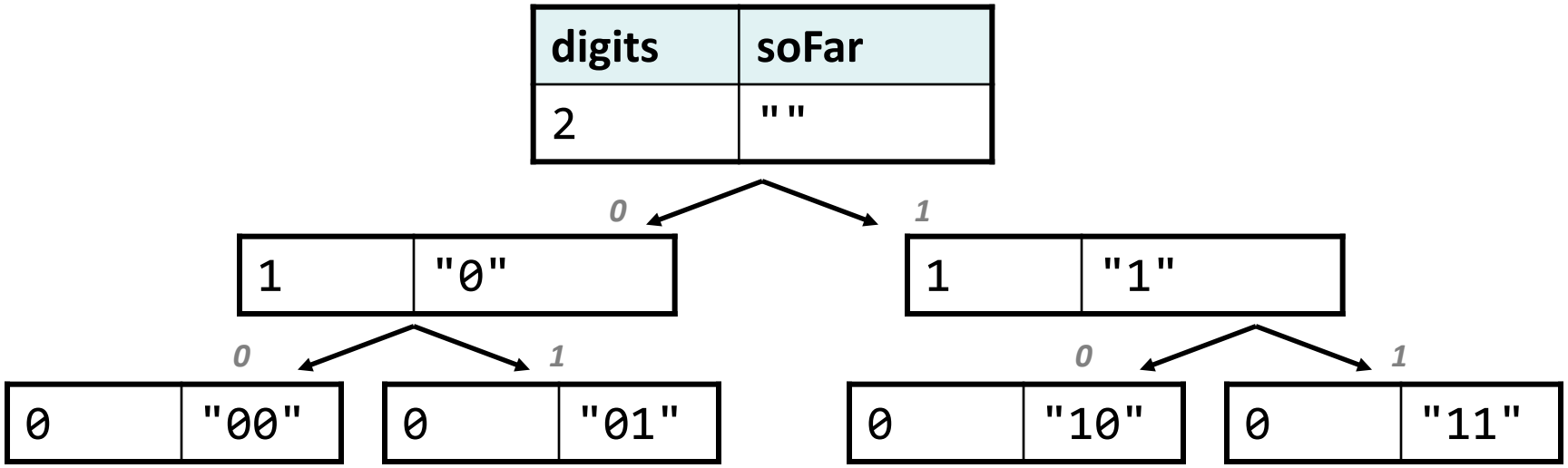
printAllBinary solution

```
void printAllBinary(int numDigits) {
    printAllBinaryHelper(numDigits, "");
}

void printAllBinaryHelper(int digits, string soFar) {
    if (digits == 0) {
        cout << soFar << endl;
    } else {
        printAllBinaryHelper(digits - 1, soFar + "0");
        printAllBinaryHelper(digits - 1, soFar + "1");
    }
}
```

A tree of calls

- `printAllBinary(2);`



  - This kind of diagram is called a *call tree* or *decision tree*.
  - Think of each call as a choice or decision made by the algorithm:
    - Should I choose 0 as the next digit?
    - Should I choose 1 as the next digit?

The base case

```
void printAllBinaryHelper(int digits, string soFar) {
    if (digits == 0) {
        cout << soFar << endl;
    } else {
        printAllBinaryHelper(digits - 1, soFar + "0");
        printAllBinaryHelper(digits - 1, soFar + "1");
    }
}
```

- The **base case** is where the code stops after doing its work.
  - `pAB(3) -> pAB(2) -> pAB(1) -> pAB(0)`
- Each call should keep track of the work it has done.

- Base case should print the result of the work done by prior calls.
  - Work is kept track of in some variable(s) - in this case, `string soFar`.

Plan For Today

- Ex. Printing Binary
- What is Recursive Backtracking?
- Ex. Dice Sums
- **Announcements**
- Ex. Subsets

Recursive Backtracking

- **Recursive Backtracking:** using recursion to explore solutions to a problem and abandoning them if they are not suitable.
  - Determine whether a solution exists
  - Find a solution
  - Find the best solution
  - Count the number of solutions
  - Print/find all the solutions
- Applications:
  - Puzzle solving (Sudoku, Crosswords, etc.)
  - Game playing (Chess, Solitaire, etc.)
  - Constraint satisfaction problems (scheduling, matching, etc.)

The Recursion Checklist

- ❑ **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?
- ❑ **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?
- ❑ **Find our recursive step**. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?
- ❑ **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?

The Recursion Checklist

- ❑ **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?
- ❑ **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?
- ❑ **Find our recursive step**. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?
- ❑ **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?

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?

printAllBinary solution

```
void printAllBinary(int numDigits) {
    printAllBinaryHelper(numDigits, "");
}

void printAllBinaryHelper(int digits, string soFar) {
    if (digits == 0) {
        cout << soFar << endl;
    } else {
        printAllBinaryHelper(digits - 1, soFar + "0");
        printAllBinaryHelper(digits - 1, soFar + "1");
    }
}
```

❑Choose ❑Explore ❑Un-choose ❑Base case

Plan For Today

- Ex. Printing Binary
- What is Recursive Backtracking?
- Ex. Dice Sums
- **Announcements**
- Ex. Subsets

Exercise: Dice rolls

diceRoll

- Write a recursive function **`diceRoll`** that accepts an integer representing a number of 6-sided dice to roll, and output all possible combinations of values that could appear on the dice.

`diceRoll(2);`

```
{1, 1}  |  {3, 1}  |  {5, 1}
{1, 2}  |  {3, 2}  |  {5, 2}
{1, 3}  |  {3, 3}  |  {5, 3}
{1, 4}  |  {3, 4}  |  {5, 4}
{1, 5}  |  {3, 5}  |  {5, 5}
{1, 6}  |  {3, 6}  |  {5, 6}
{2, 1}  |  {4, 1}  |  {6, 1}
{2, 2}  |  {4, 2}  |  {6, 2}
{2, 3}  |  {4, 3}  |  {6, 3}
{2, 4}  |  {4, 4}  |  {6, 4}
{2, 5}  |  {4, 5}  |  {6, 5}
{2, 6}  |  {4, 6}  |  {6, 6}
```

`diceRoll(3);`

```
{1, 1, 1}
{1, 1, 2}
{1, 1, 3}
{1, 1, 4}
{1, 1, 5}
{1, 1, 6}
{1, 2, 1}
{1, 2, 2}
   ...
{6, 6, 4}
{6, 6, 5}
{6, 6, 6}
```

  - How is this problem recursive (self-similar)?

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?

The Backtracking Checklist

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

For each val

What die value should I choose next?

- ❑ **Make it** information for a cho
- ❑ **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?

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?

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** to the way it was be
- ❑ **Find our b** when we are out of decisions?

We need to communicate the dice chosen so far to the next recursive call.

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?

The Backtracking Checklist

- ❑ **Find what choice(s) we have at each step.** What different o…?
- For each val…
  - ❑ **Make it** …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?

We need to be able to remove the die we added to our roll so far.

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?

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 cho
- ❑ **Undo it** g to the way it was be

When we have no dice left to choose, print them out.

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

Examining the problem

- We want to generate all possible sequences of values.

```
for (each possible first die value):
    for (each possible second die value):
        for (each possible third die value):
            ...
                print!
```

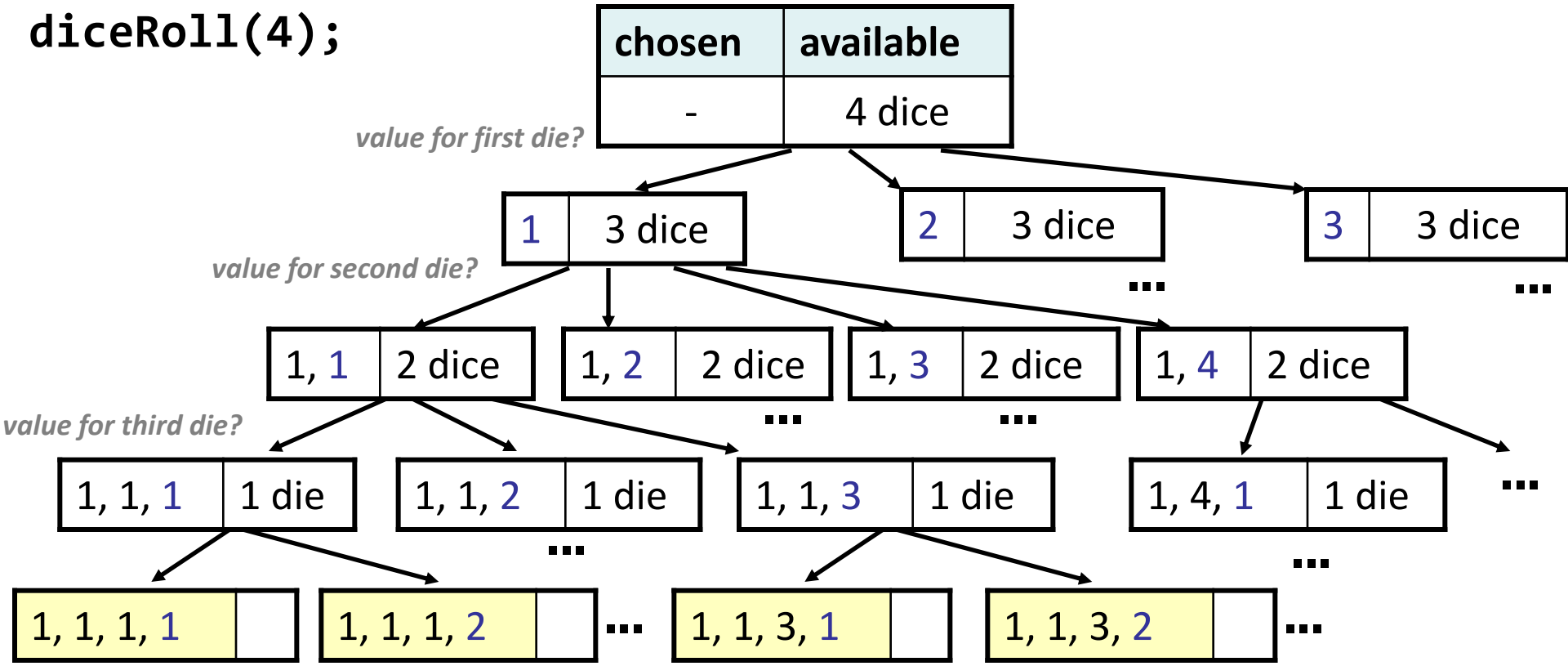
  - This is called a **depth-first search**

  - You may think that `for` loops are the way to approach this problem.
    - But how many loops are needed?

  - How can we completely explore such a large search space?

A decision tree

```
diceRoll(4);
```



diceRolls pseudocode

```
function diceRolls(dice):
    if dice == 0:
        nothing to do.
    else:
        // handle all roll values for a single die; let recursion do the rest.
        for each die value i in range [1..6]:
            choose that the current die will have value i.
            diceRolls(dice-1).          // explore the remaining dice.
            un-choose the value i.
```

- How do we keep track of our choices?

diceRolls solution

```
// Prints all possible outcomes of rolling the given
// number of six-sided dice in {#, #, #} format.
void diceRolls(int dice) {
    Vector<int> chosen;
    diceRollHelper(dice, chosen);
}

// private recursive helper to implement diceRolls logic
void diceRollHelper(int dice, Vector<int>& chosen) {
    if (dice == 0) {
        cout << chosen << endl;                 // base case
    } else {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);                      // choose
            diceRollHelper(dice - 1, chosen);   // explore
            chosen.remove(chosen.size() - 1);   // un-choose
        }
    }
}
```

Exercise: Dice roll sum

- Write a function **`diceSum`** similar to `diceRoll`, but it also accepts a desired sum and prints only combinations that add up to exactly that sum.

```
diceSum(2, 7);
```

```
{1, 6}
{2, 5}
{3, 4}
{4, 3}
{5, 2}
{6, 1}
```

```
diceSum(3, 7);
```

```
{1, 1, 5}
{1, 2, 4}
{1, 3, 3}
{1, 4, 2}
{1, 5, 1}
{2, 1, 4}
{2, 2, 3}
{2, 3, 2}
{2, 4, 1}
{3, 1, 3}
{3, 2, 2}
{3, 3, 1}
{4, 1, 2}
{4, 2, 1}
{5, 1, 1}
```

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?

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 cho
- ❑ **Undo it** g to the way it was be

When we have no dice left to choose, print them out ***only if they equal our sum***.

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

Minimal modification

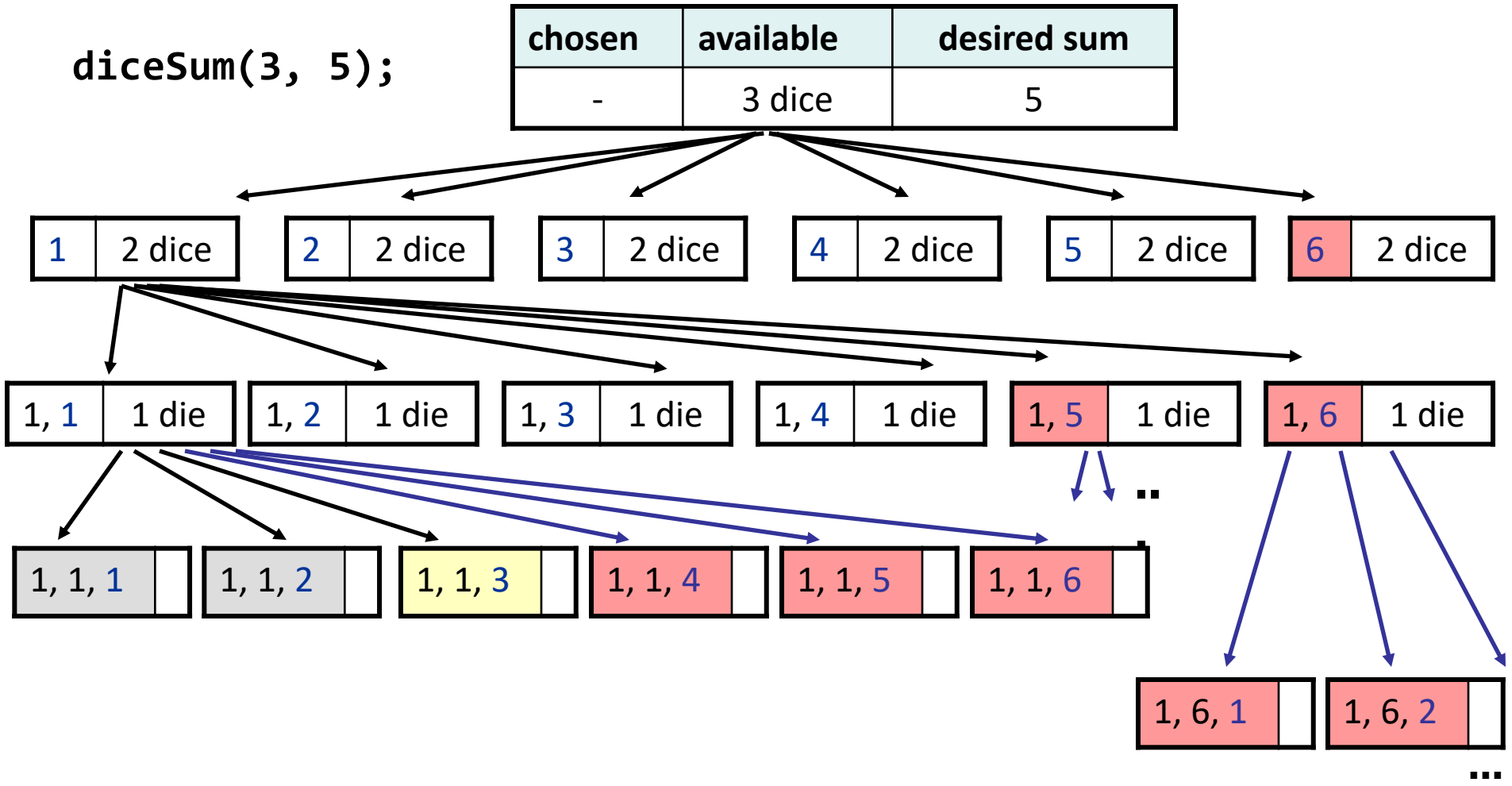
```
void diceSum(int dice, int desiredSum) {
    Vector<int> chosen;
    diceSumHelper(dice, desiredSum, chosen);
}

void diceSumHelper(int dice, int desiredSum, Vector<int>& chosen) {
    if (dice == 0) {
        if (sumAll(chosen) == desiredSum) {
            cout << chosen << endl;                          // base case
        }
    } else {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);                                   // choose
            diceSumHelper(dice - 1, desiredSum, chosen);     // explore
            chosen.remove(chosen.size() - 1);                // un-choose
        }
    }
}

int sumAll(const Vector<int>& v) {
    int sum = 0;
    for (int k : v) { sum += k; }
    return sum;
}
```

Wasteful decision tree

```
diceSum(3, 5);
```



Optimizations

- We need not visit every branch of the decision tree.
  - Some branches are clearly not going to lead to success.
  - We can preemptively stop, or **prune**, these branches.

- Inefficiencies in our dice sum algorithm:
  - Sometimes the current sum is already **too high**.
    - (Even rolling 1 for all remaining dice would exceed the desired sum.)
  - Sometimes the current sum is already **too low**.
    - (Even rolling 6 for all remaining dice would fall short of the desired sum.)
  - The code must **re-compute** the sum many times.
    - (1+1+1 = ..., 1+1+2 = ..., 1+1+3 = ..., 1+1+4 = ..., ...)

diceSum solution

```
void diceSum(int dice, int desiredSum) {
    Vector<int> chosen;
    diceSumHelper(dice, 0, desiredSum, chosen);
}

void diceSumHelper(int dice, int sum, int desiredSum, Vector<int>& chosen) {
    if (dice == 0) {
        if (sum == desiredSum) {
            cout << chosen << endl;                          // base case
        }
    } else if (sum + 1*dice <= desiredSum
            && sum + 6*dice >= desiredSum) {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);                                   // choose
            diceSumHelper(dice - 1, sum + i, desiredSum, chosen); // explore
            chosen.remove(chosen.size() - 1);                // un-choose
        }
    }
}
```

For you to think about...

- How would you modify **`diceSum`** so that it prints only unique combinations of dice, ignoring order?
  - (e.g. don't print both `{1, 1, 5}` and `{1, 5, 1}`)

```
diceSum2(2, 7);
```

```
{1, 6}
{2, 5}
{3, 4}
{4, 3}
{5, 2}
{6, 1}
```

```
diceSum2(3, 7);
```

```
{1, 1, 5}
{1, 2, 4}
{1, 3, 3}
{1, 4, 2}
{1, 5, 1}
{2, 1, 4}
{2, 2, 3}
{2, 3, 2}
{2, 4, 1}
{3, 1, 3}
{3, 2, 2}
{3, 3, 1}
{4, 1, 2}
{4, 2, 1}
{5, 1, 1}
```

diceSum solution

```
void diceSum(int dice, int desiredSum) {
    Vector<int> chosen;
    diceSumHelper(dice, 0, desiredSum, chosen, 1);
}

void diceSumHelper(int dice, int sum, int desiredSum, Vector<int>& chosen,
    int minDieValue) {
    if (dice == 0) {
        if (sum == desiredSum) {
            cout << chosen << endl;                          // base case
        }
    } else if (sum + 1*dice <= desiredSum
            && sum + 6*dice >= desiredSum) {
        for (int i = minDieValue; i <= 6; i++) {
            chosen.add(i);                                   // choose
            diceSumHelper(dice - 1, sum + i, desiredSum, chosen, i);//explore
            chosen.remove(chosen.size() - 1);                // un-choose
        }
    }
}
```

Plan For Today

- Ex. Printing Binary
- What is Recursive Backtracking?
- Ex. Dice Sums
- **Announcements**
- Ex. Subsets

Announcements

- Pair Programming on HW4
  - Pair programming means that two people work *together* on an assignment, completely.
  - Pair programmers must never be working on the assignment independently, and should both be looking at the same screen, with one of the students typing (they should take turns).
  - Students may ask conceptual questions in the LaIR and on Piazza independently, but if you are in a pair you must get help on the code together.
  - If one student has taken the course before, there can be no overlapping code from that student's prior work unless they worked on it with the exact same people then as now.

Plan For Today

- Ex. Printing Binary
- What is Recursive Backtracking?
- Ex. Dice Sums
- **Announcements**
- Ex. Subsets

Exercise: subsets

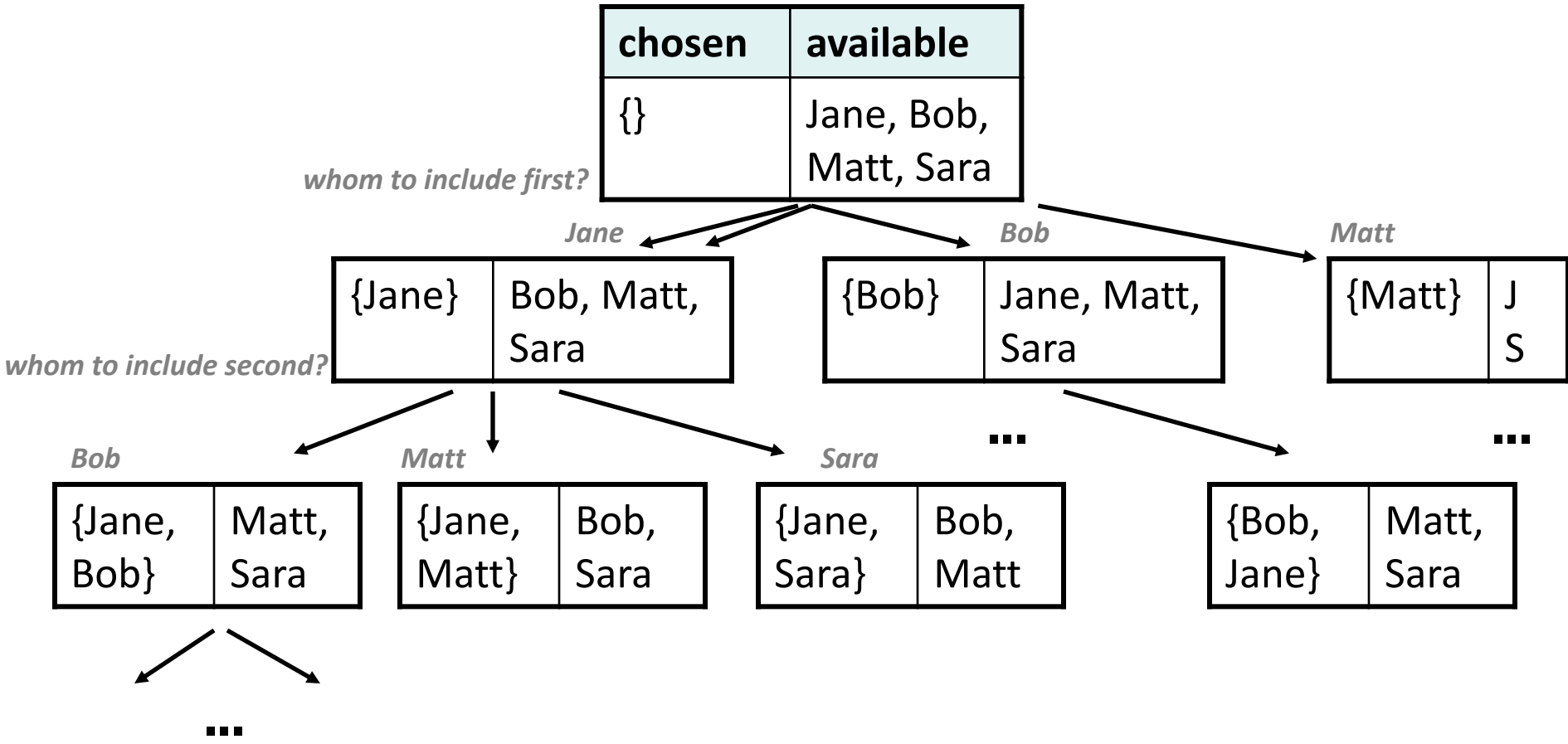
printSubVectors

- 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:

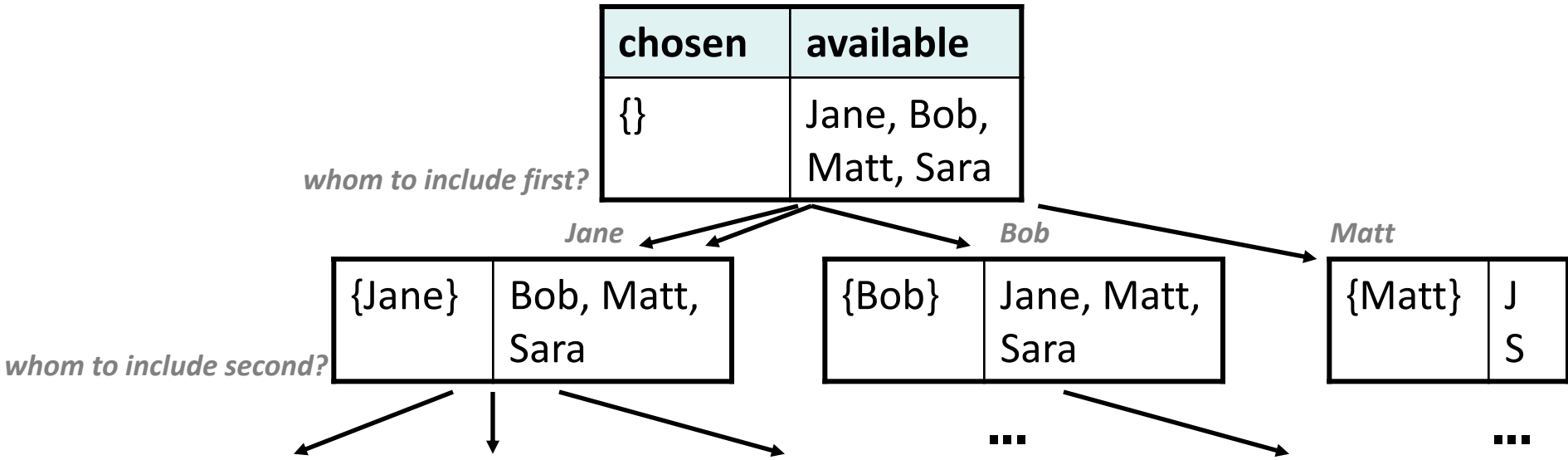
```
{Jane, Bob, Matt, Sara}
{Jane, Bob, Matt}
{Jane, Bob, Sara}
{Jane, Bob}
{Jane, Matt, Sara}
{Jane, Matt}
{Jane, Sara}
{Jane}
{Bob, Matt, Sara}
{Bob, Matt}
{Bob, Sara}
{Bob}
{Matt, Sara}
{Matt}
{Sara}
{}
```

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

Decision tree?



Wrong decision tree



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

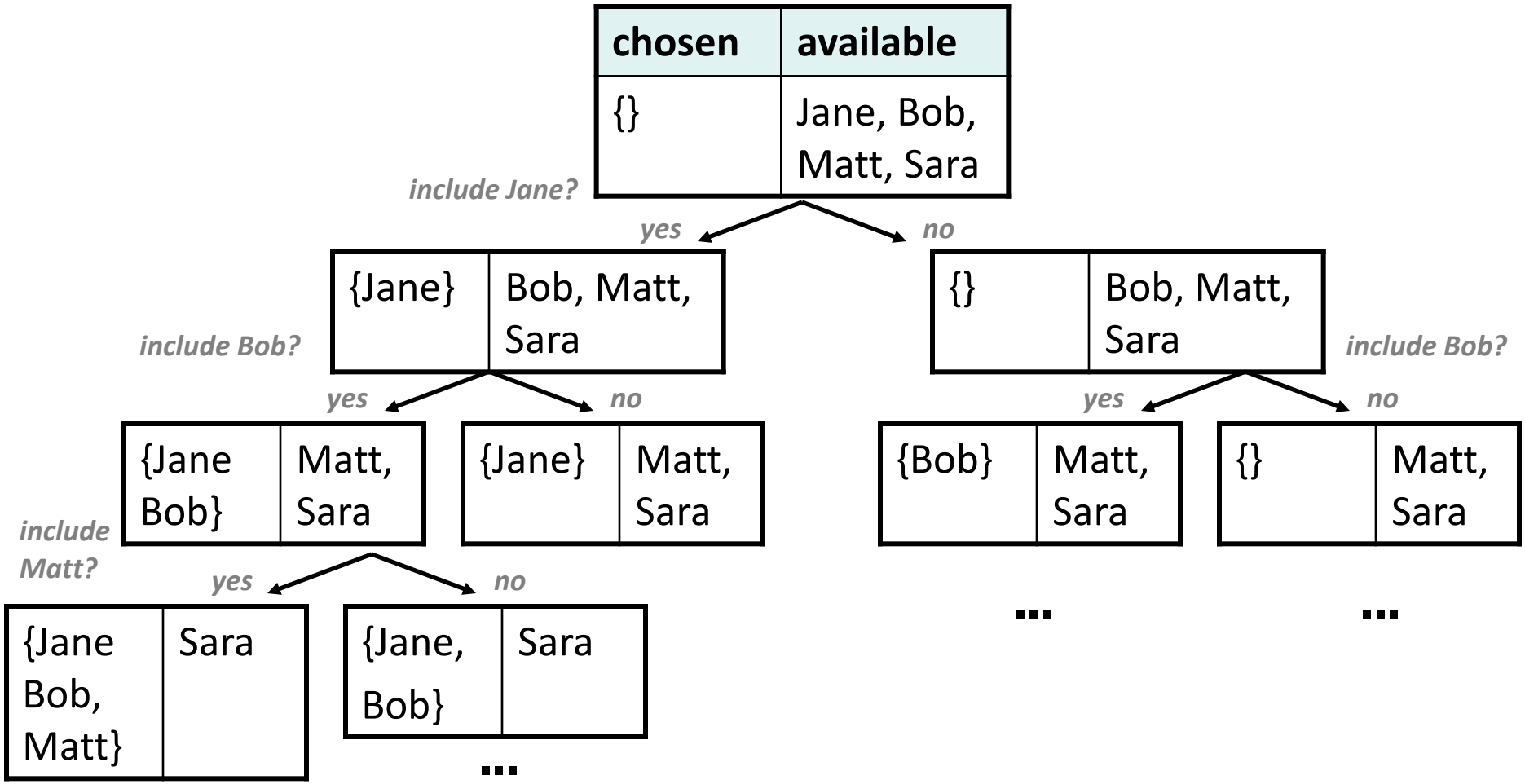
A. It does not actually end up finding every possible subset.

B. It does find all 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**.

The Backtracking Checklist

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

For each val

- ❑ **Make it** information for a cho
- ❑ **Undo it after exploring.** Restore everything to the way it was before making this choice.

Should I include the next person in the subset?

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

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?

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** to the way it was be

Find all subsets with this choice made.

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

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?

The Backtracking Checklist

- ❑ **Find what choice(s) we have at each step.** What different o…?
- For each val…
  - ❑ **Make it** …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?

Make sure the chosen set is the same as it was before.

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?

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 cho
- ❑ **Undo it** g to the way it was be

When we have no choices left, print out the subset.

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

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);
    }
}
```

Recap

- Ex. Printing Binary
- What is Recursive Backtracking?
- Ex. Dice Sums
- **Announcements**
- Ex. Subsets

Next time: more backtracking

Overflow

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}

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);
        }
    }
}
```