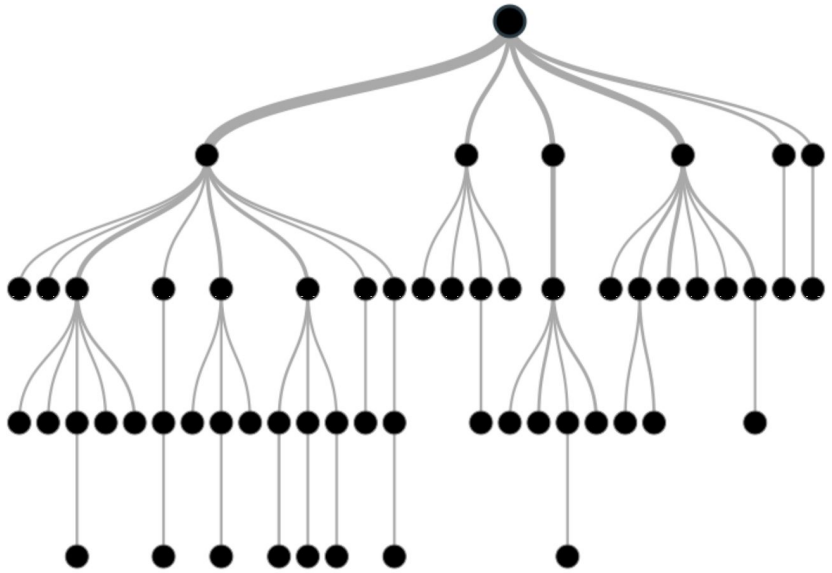


Combinatorial Problems w/ Recursion

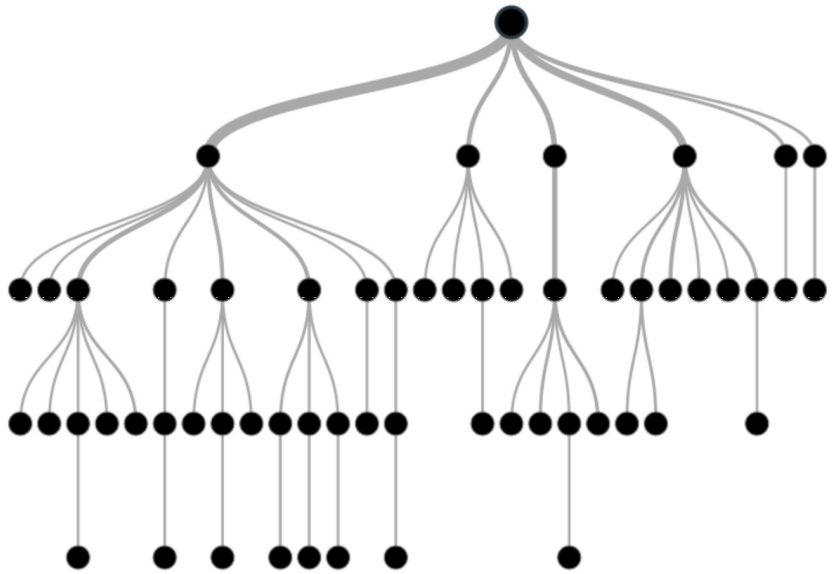
Hien Luu

The only thing which can save/help you in interviews is your thinking ability

How would you explain what recursion is to a 4-year old kid?

Agenda

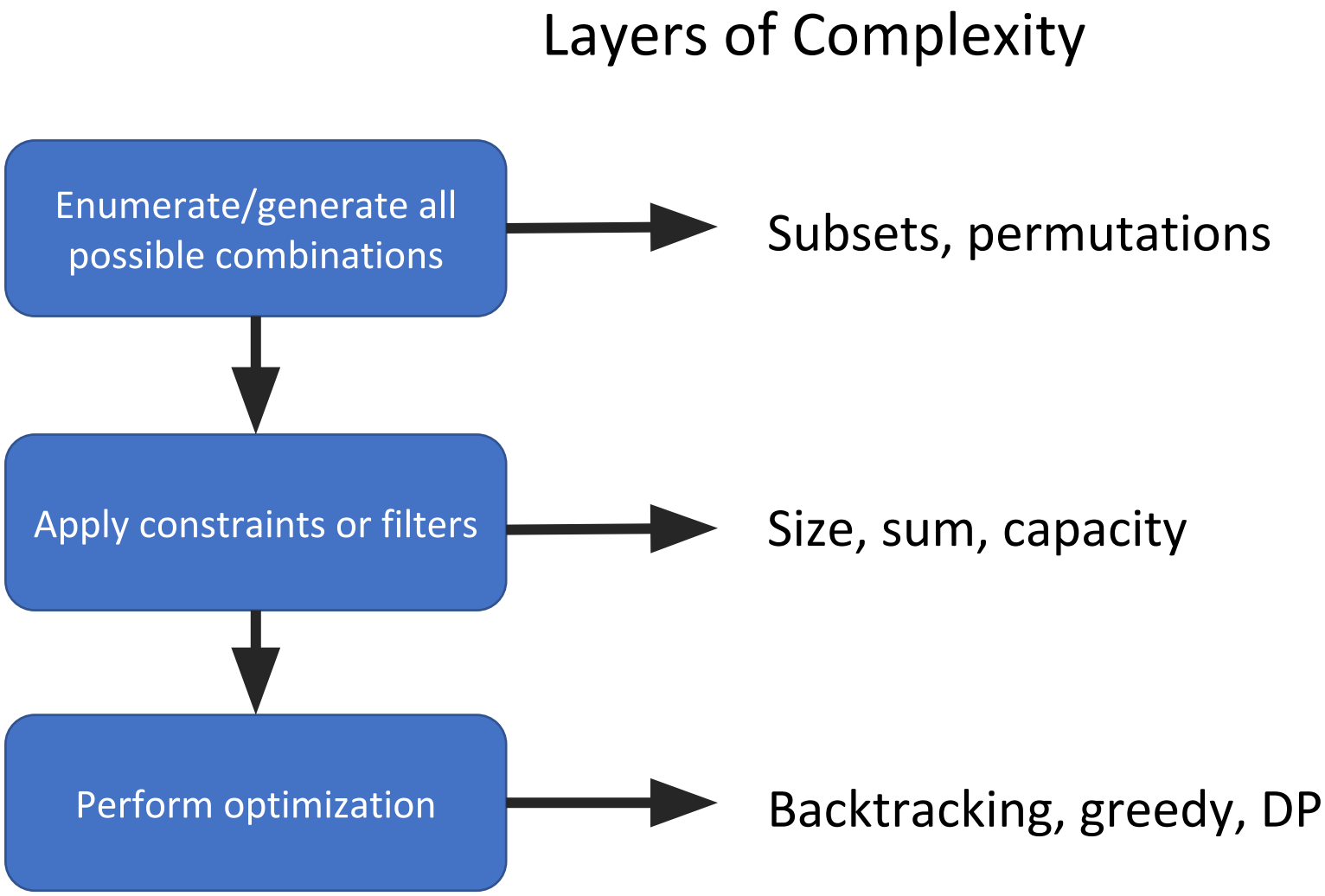
- Combinatorial Problems (~2hrs)
  - Exhaustive search, enumerate, generate
  - Classic problems – subsets, permutation
  - General solution Pattern
- Backtracking Problems (~1.5hrs)
  - General Solution Pattern



Popular interview questions, why?

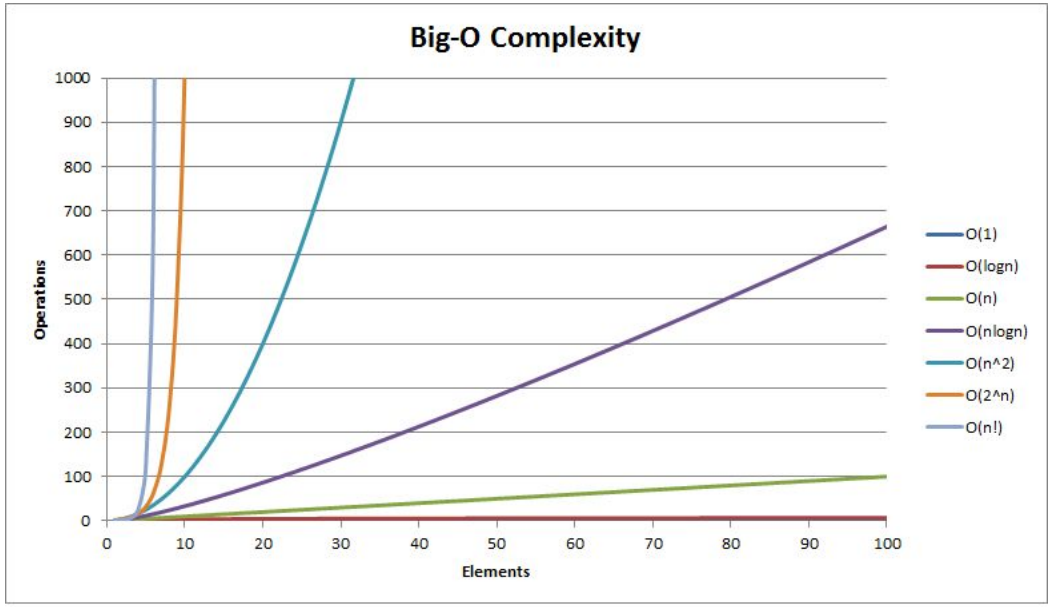
Logistics:
- break ~10:30am
- lunch at noon

Combinatorial Problems



Combinatorial Problems

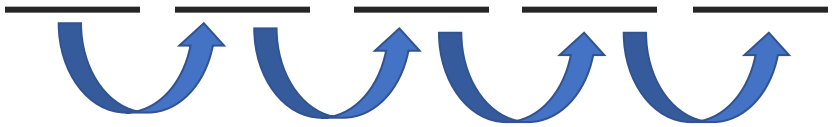
- Common question
  - Generation, enumeration, existence
- Common pattern
  - From a set of decisions (aka subproblems) and choices
  - Lend itself to recursion
- Combinatorial explosion
  - Create challenges
  - Exponential runtime



Combinatorial Problems - Mental Model

Systematic way of enumerating or generating

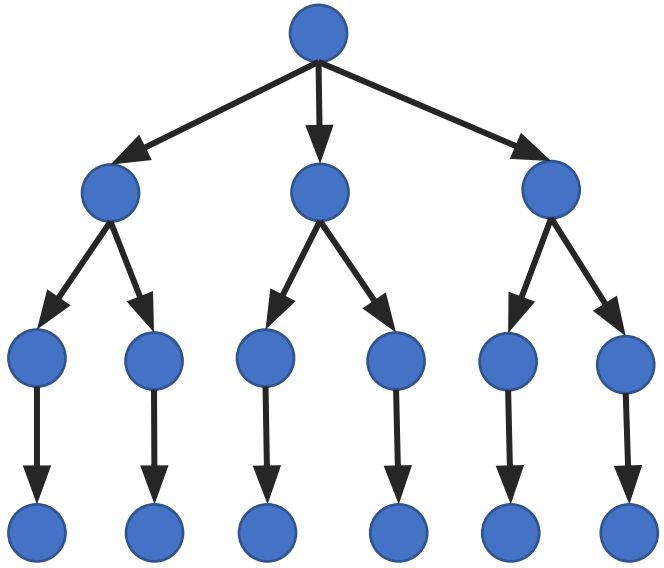
A series of blanks

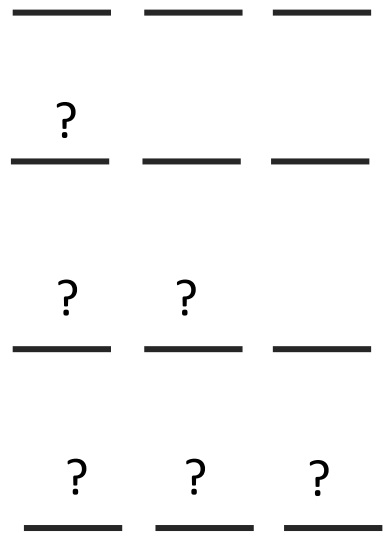


Efficient manager

Recursion Fairy

Decision Tree



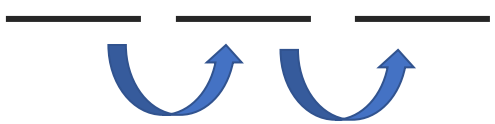


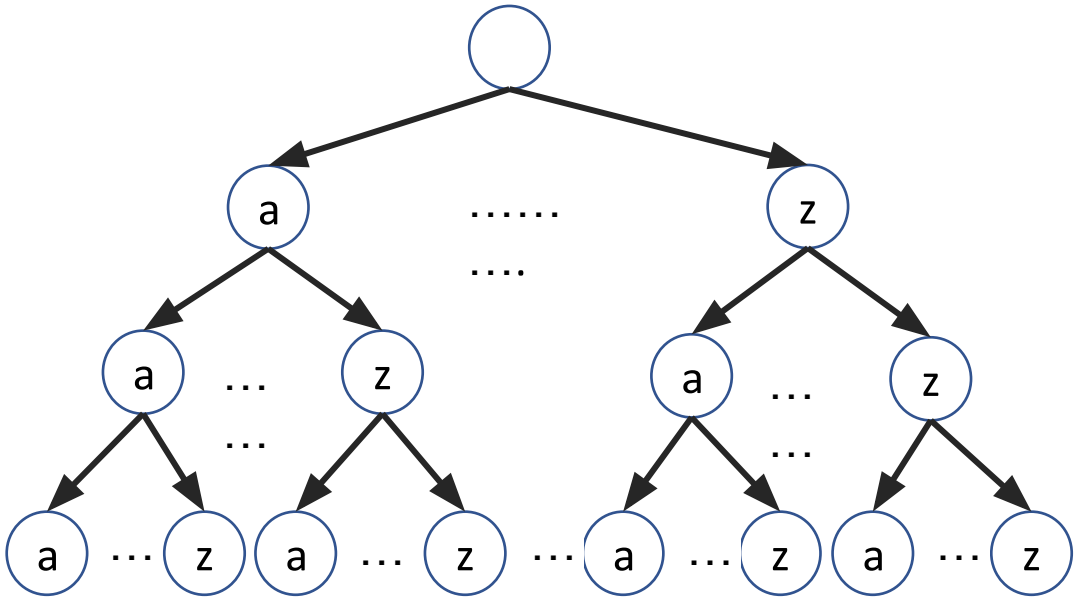
Helpful visualization tool

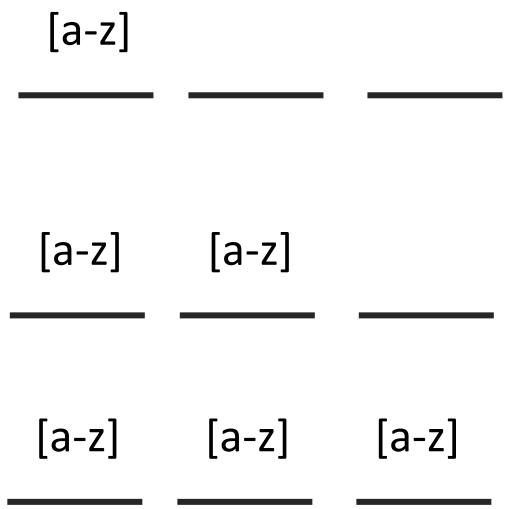
Combinatorial Problems

- Example
  - Generate all 3-letter strings (print or collect)
    - Each blank/letter is a decision
    - Each decision has 26 possible choices

A series of blanks







Solution Pattern – (pseudo code)

```
driver(problems) {
  helper(problems, empty-partial-solution)
}

helper(sub problems, partial-solution) {
  if (no more sub problems) { // base case
    print/collect partial-solution
  } else {
    for (each possible choices) {
      make a choice  // add to partial-solution
      helper(remaining-decisions, partial-solution)
      unmake choice  // if mutable data structure is used
    }
  }
}
```

Letter Case Permutation

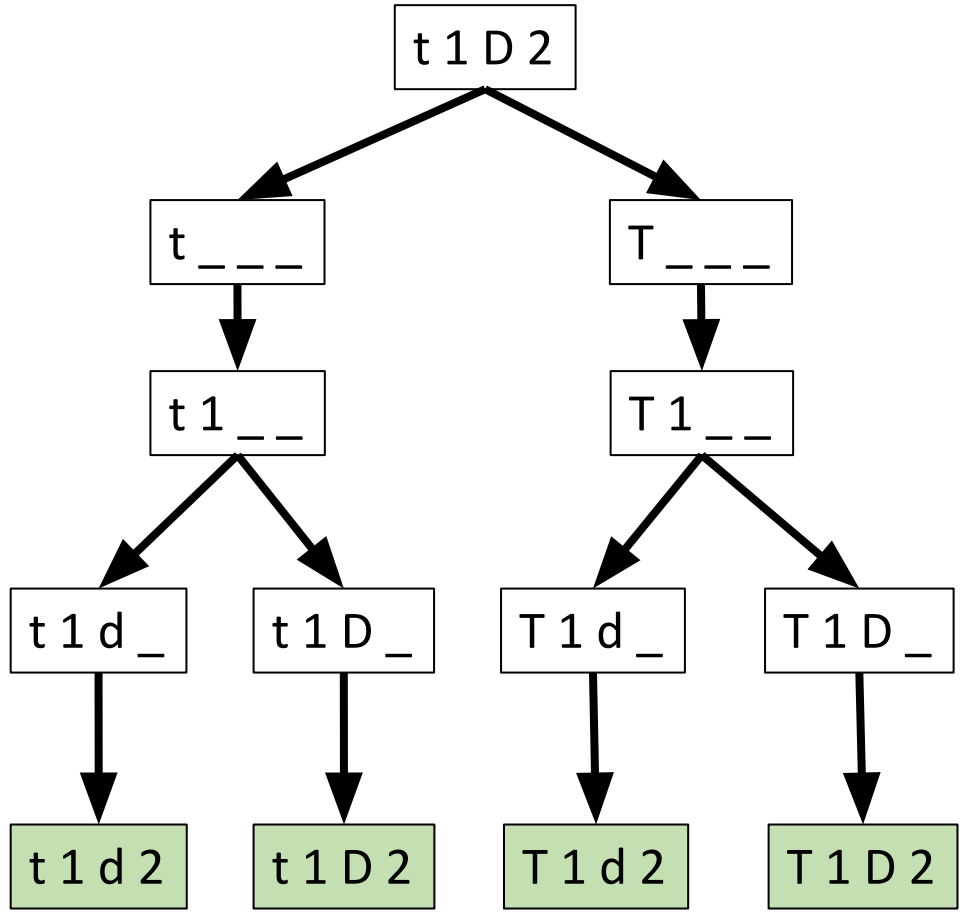
- Given a string S
  - Transform every letter to lowercase or uppercase to create another string
  - Return a list of all possible strings we could create
- Example:
  - Input: ”t1D2”
  - Output: [”t1D2", ”t1d2”, ”T1D2", ”T1d2"]

  - Input: “12345”
  - Output: ["12345"]

  - Input: “abc”
  - Output: [????]

Letter Case Permutation – Decision Tree

Input: "t1D2"



Draw the execution order

Letter Case Permutation

```
// Return a list of all possible strings

List<String> driver(String input) {
   List<String> coll = new ArrayList<>();
   helper(input, 0, "", coll);
   return coll;
}

void helper(String input, int idx, String soFar, List<String> coll) {
  // base case
  // else case

}
```

Letter Case Permutation

```
void helper(String input, int idx, String soFar, List<String> coll)
{
    // base case
    // else case
}
```

Letter Case Permutation

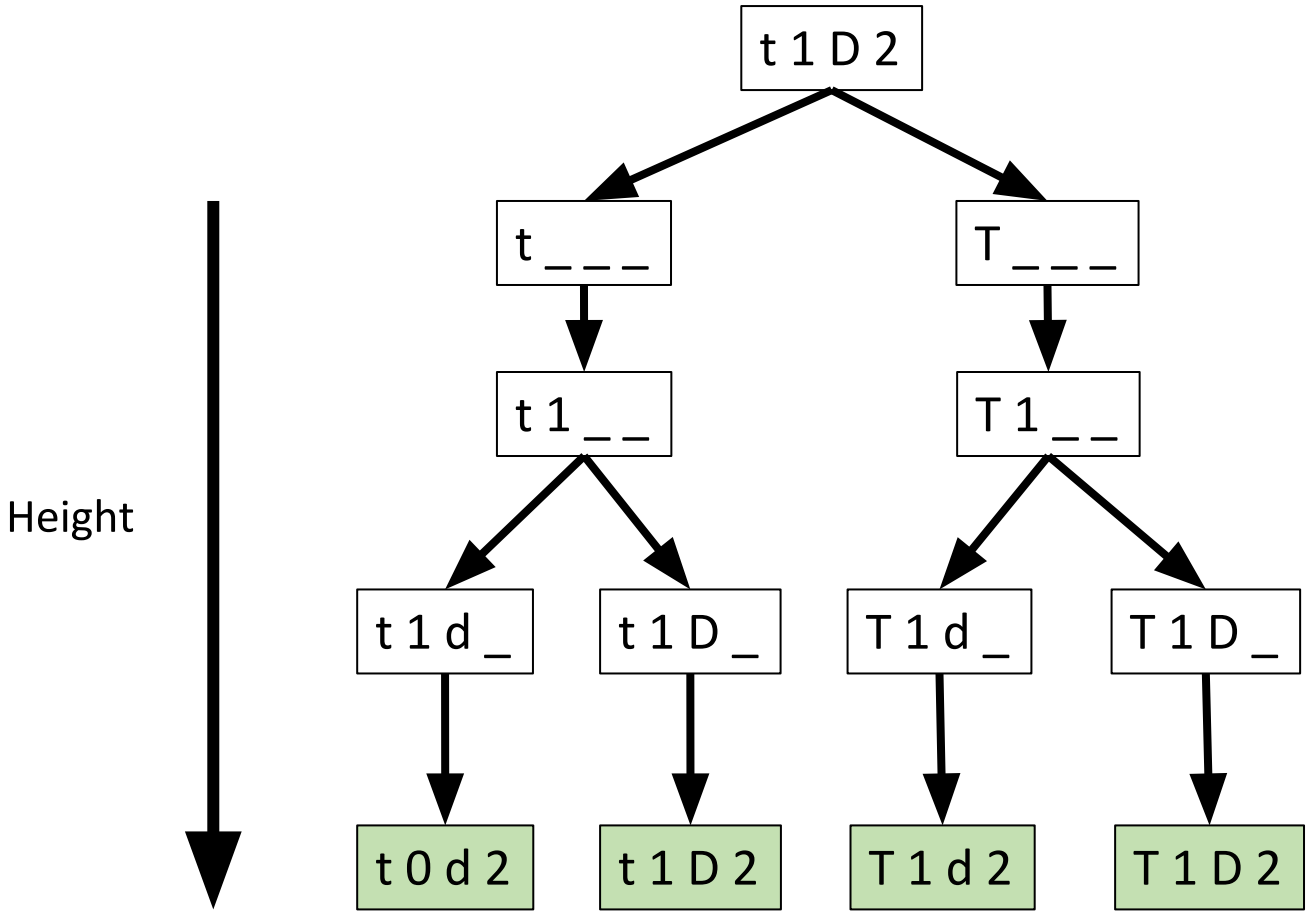
```
void helper(String input, int idx, String soFar, List<String> coll)
{
    if (idx == input.length()) {          // base case
        coll.add(soFar);
    } else {                              // else case
      char currChar = input.charAt(idx);
      if (Character.isDigit(currChar)) {
        helper(input, idx+1, soFar + currChar, coll);
     } else {
        helper(input, idx+1, soFar +
               Character.toLowerCase(currChar), coll);
        helper(input, idx+1, soFar +
               Character.toUpperCase(currChar), coll);
     }
}
```

Letter Case Permutation

- Time complexity
  - # of calls * amount of work/call
- Space complexity
  - Explicit (collector) +
  - Implicit (stack)

Time complexity:
- $O(2^{n+1} * n)$

Space complexity:
- $O(2^n * n + n^2)$



Letter Case Permutation

```
// using mutable data structure
void helper(String input, int idx, List<Character> buf, List<String> coll) {
   if (idx == input.length()) {
     coll.add(new String(buf));
   } else {
     char currChar = input.charAt(idx);
     if (Character.digit(currChar)) {
       buf.append(Character.toLowerCase(currChar));
       helper(input, idx + 1, buf, coll);
       buf.remove(buf.size()-1)
     } else {

       buf.append(Character.toLowerCase(currChar));
       helper(input, idx + 1, buf, coll);
       buf.remove(buf.size()-1)

       buf.append(Character.toUpperCase(currChar));
       helper(input, idx + 1, buf, coll);
       buf.remove(buf.size()-1)
     }
  }
```

Letter Case Permutation

```
// using mutable data structure
void helper(String input, int idx, char[] buf, List<String> coll) {
   if (idx == input.length()) {
     coll.add(new String(buf));
   } else {
     char currChar = input.charAt(idx);
     if (Character.isAlphabetic(currChar)) {
       // what should go here
     } else {
       // what should go here?
     }
  }
}
```

Letter Case Permutation

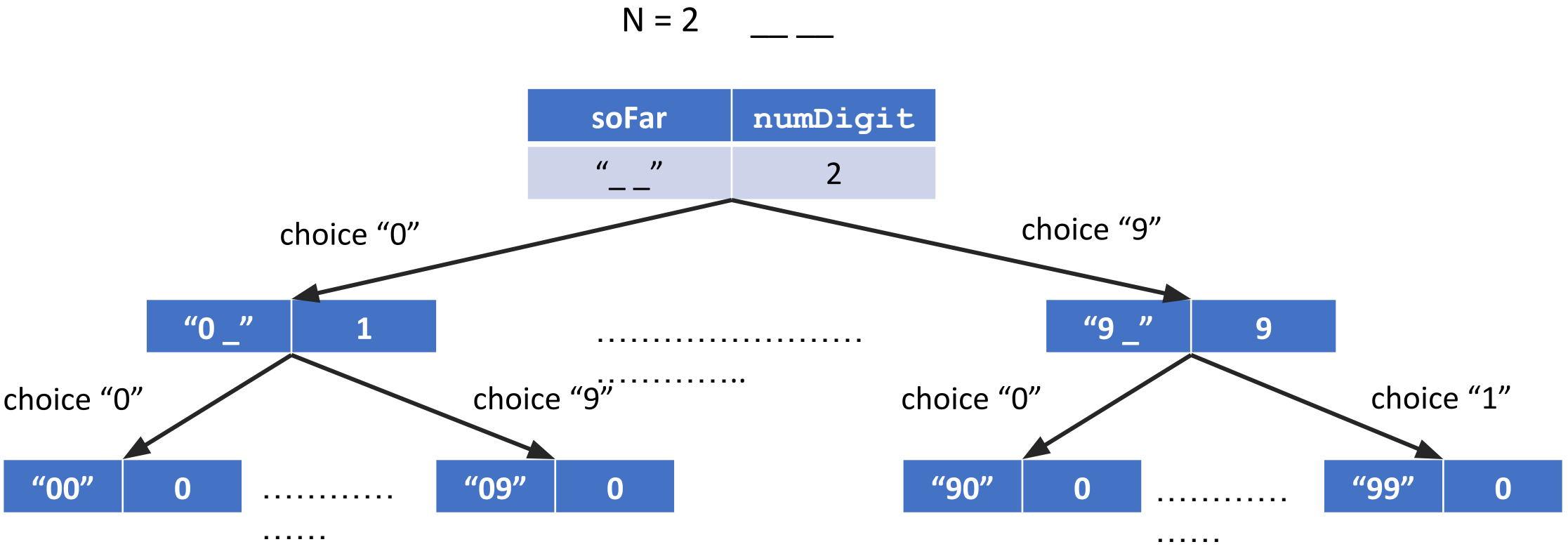
```
// using mutable data structure
void helper(String input, int idx, char[] buf, List<String> coll) {
   if (idx == input.length()) {
     coll.add(new String(buf));
   } else {
     char currChar = input.charAt(idx);
     if (Character.isDigit(currChar)) {
       buf[idx] = currChar;
       helper(input, idx + 1, buf, coll);
     } else {
       buf[idx] = Character.toLowerCase(currChar); // make a choice
       helper(input, idx + 1, buf, coll);
       buf[idx] = Character.toUpperCase(currChar); // make a choice
       helper(input, idx + 1, buf, coll);
     }
  }
}
```

Print Decimal Strings - Optional

- Print all decimal numbers with n digits in ascending order
- Example:
  - N = 2
    - Output: 00, 01, 02, 03,04,05,06,07,08,09,10,11,12,13,14,15,16,17,18,19…99
- Approach
  - What are the decisions?
  - What are the choices?
  - How to represent the partial solution?

Print Decimal Strings – Decision Tree

Print Decimal Strings



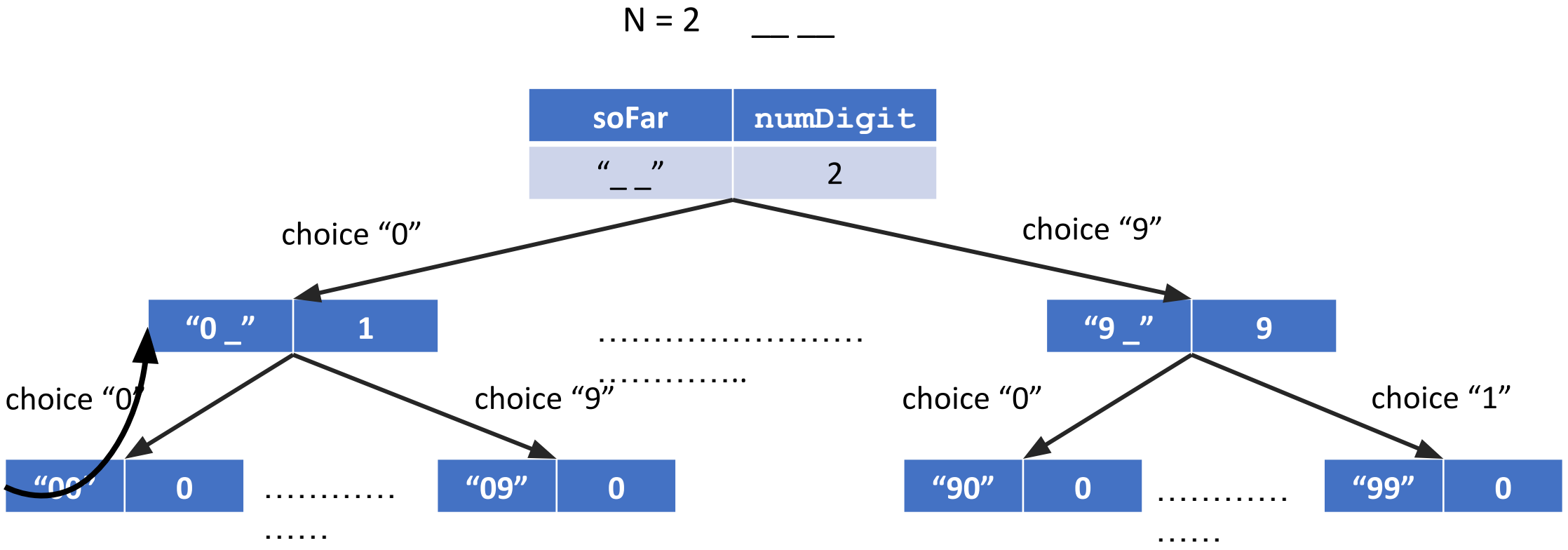
All Decimal Strings

```
driver(int numDigit) {
 helper(numDigit, new ArrayList<Integer>());
}

helper(int numDigit, List<Integer> soFar) {
  // base case
  // else case
}
```

All Decimal Strings

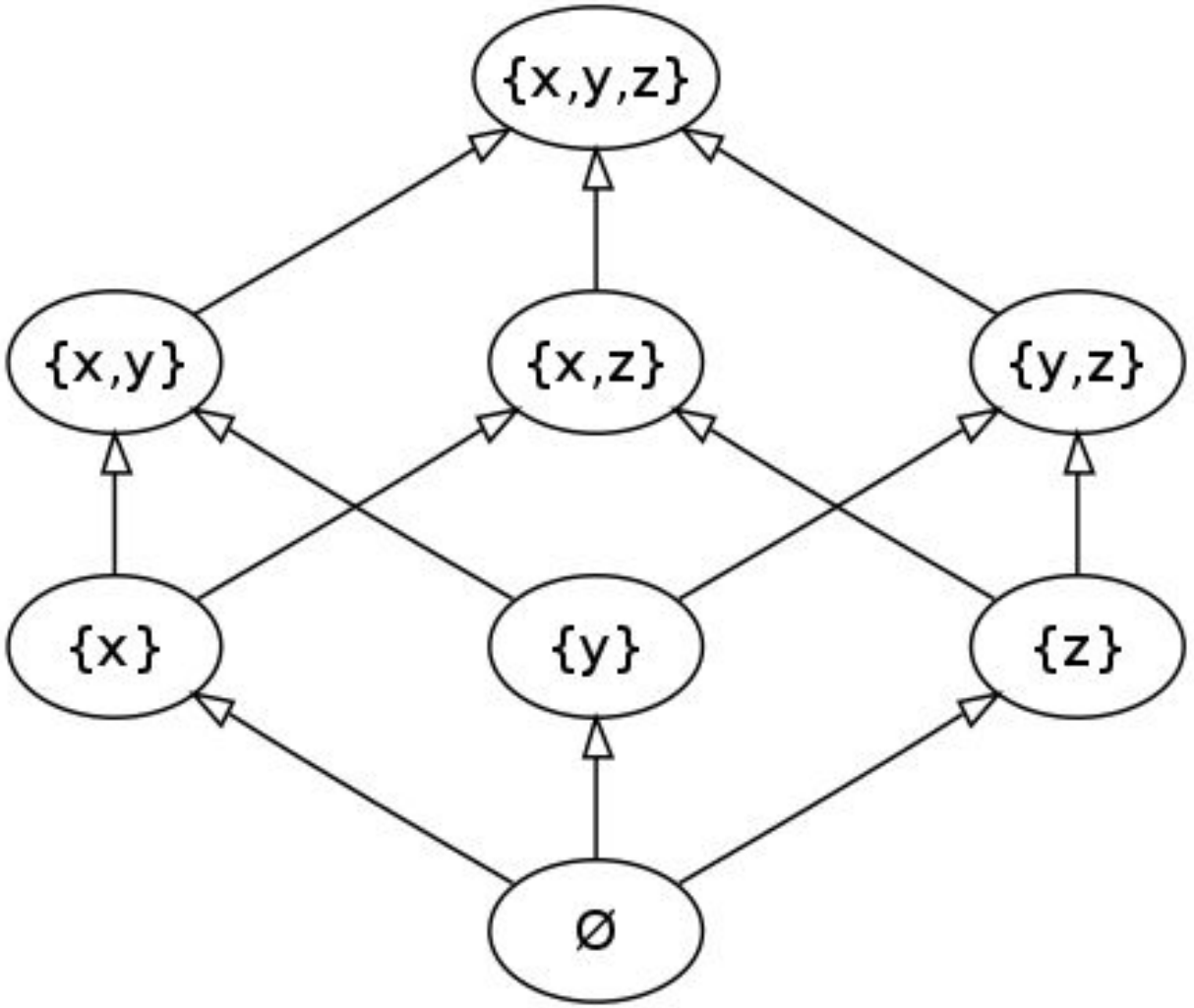
```
driver(int numDigit) {
 helper(numDigit, new ArrayList<Integer>());
}
helper(int numDigit, List<Integer> soFar) {
  if (numDigit == 0) {
    System.out.println(soFar);
  } else {
    for (int choice = 0; choice <= 9; choice++) {
       soFar.add(choice);
       helper(numDigit-1, soFar);
       soFar.remove(soFar.size()-1); // very important step
    }
  }
}
```

Print Decimal Strings



Generate Powerset

Classical Combinatorial Structure



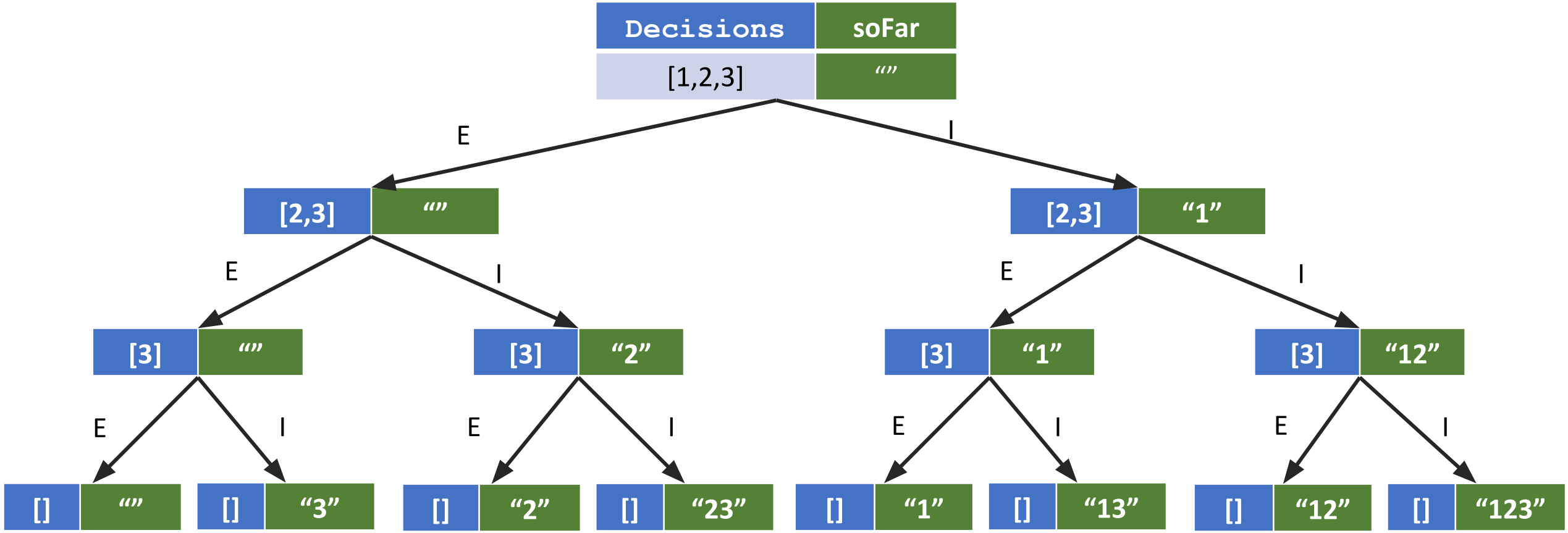
All Subsets aka Powerset

- Given a set of distinct integers, return all subsets
  - A subset can be empty or all the integers
  - An integer can be a part of or not a part of a subset
- Example:
  - Input: [1,2,3]
  - Output: {},{1},{2},{3},{1,2},{1,3},{2,3},{1,2,3}

- Approach
  - What are the decisions?
  - What are the choices?
  - How to represent the partial solution?

All subsets – printSubsets([1,2,3])

All subsets – printSubsets([1,2,3])



O(n) = O(n-1) + O(n-1) =====> 2O(n-1)

All subsets – printSubset([1,2,3])

```
List<String> driver(int[] numbers) {
 // decision, partial solution, collector
 List<String> coll = new ArrayList<>();
 helper(???);
}

// let's write code together
void helper(???) {
}
```

All subsets – printSubset([1,2,3])

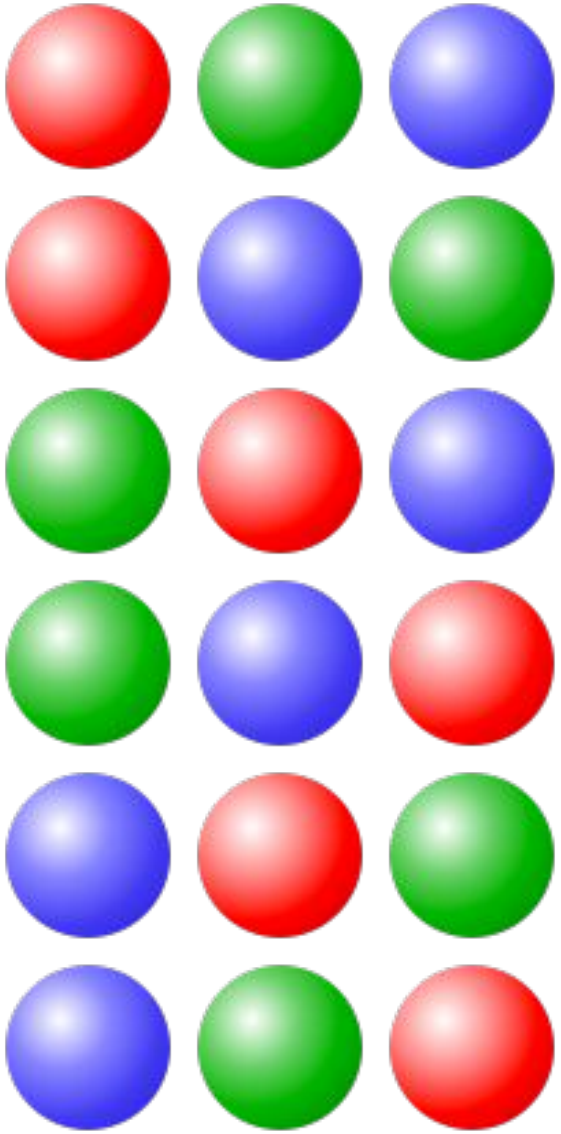
```
// using mutable data structure
helper(int[] numbers, int idx, StringBuffer soFar,
        List<String> coll) {
  if (idx == numbers.length) {
    coll.add(soFar.toString()); // a copy of soFar
  } else {
    helper(numbers, idx+1, soFar);  // exclude

    soFar.append(numbers[idx]);     // include
    helper(numbers, idx+1, soFar);
    soFar.deleteCharAt(soFar.length()-1);
  }
}
```

All subsets – printSubset([1,2,3])

- Time complexity
  - $O(2^n)$
- Space complexity when returning all subsets
  - Mutable: $O(2^n * n + n)$
- Space complexity when only printing subsets
  - ??

Generate Permutations

- Rearranging the members of a set into a sequence
- Order matters



Generating Permutations

- Given a string of characters, print all the possible permutations
- Definition: the arrangement of characters in which the order is important

- Example:
  - Input: “HAT”

HAT
HTA
AHT
ATH
THA
TAH

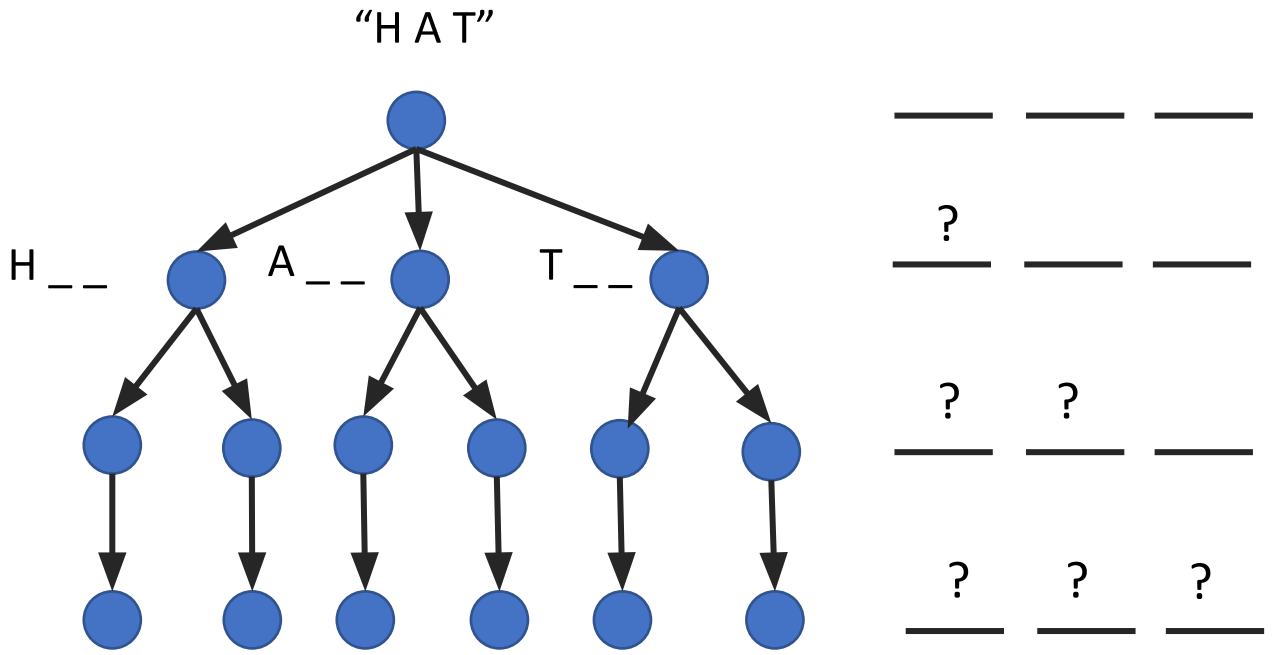
H A T

___ ___ ___

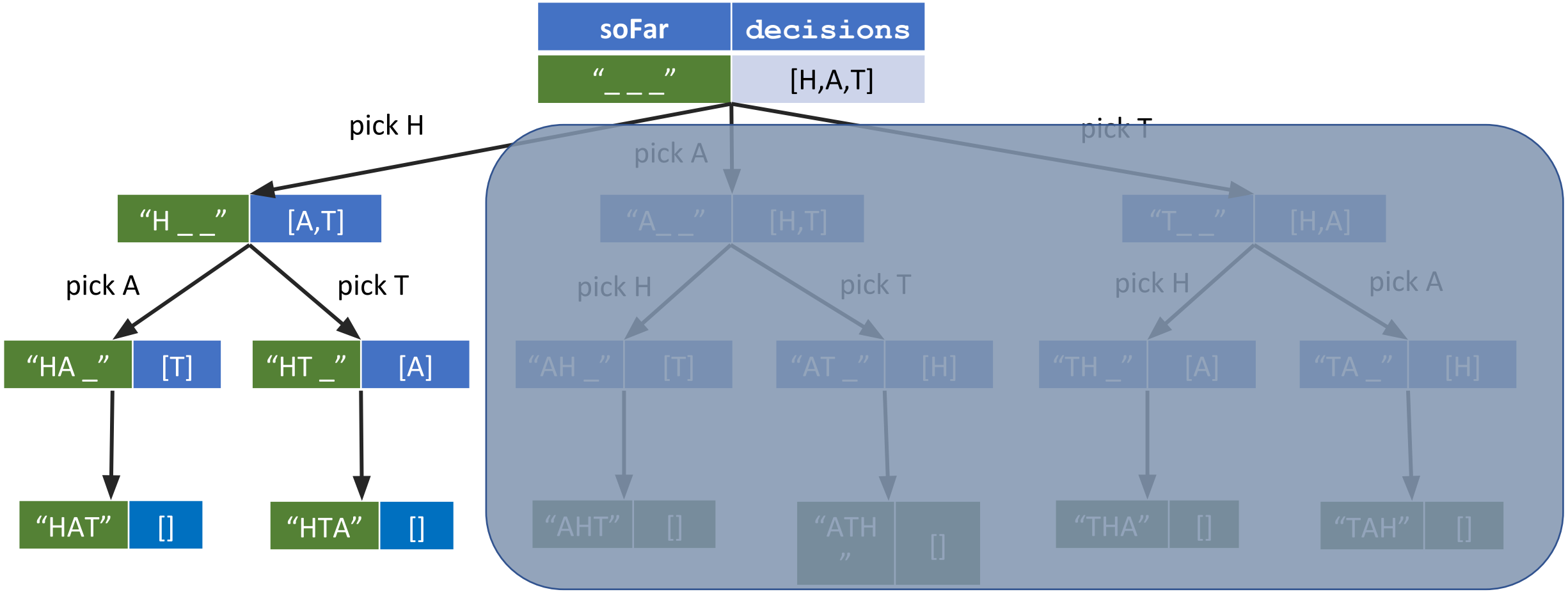
Generating Permutations

- Rearranging the members of a set
  - Print them
- Example:
  - Input: “HAT”

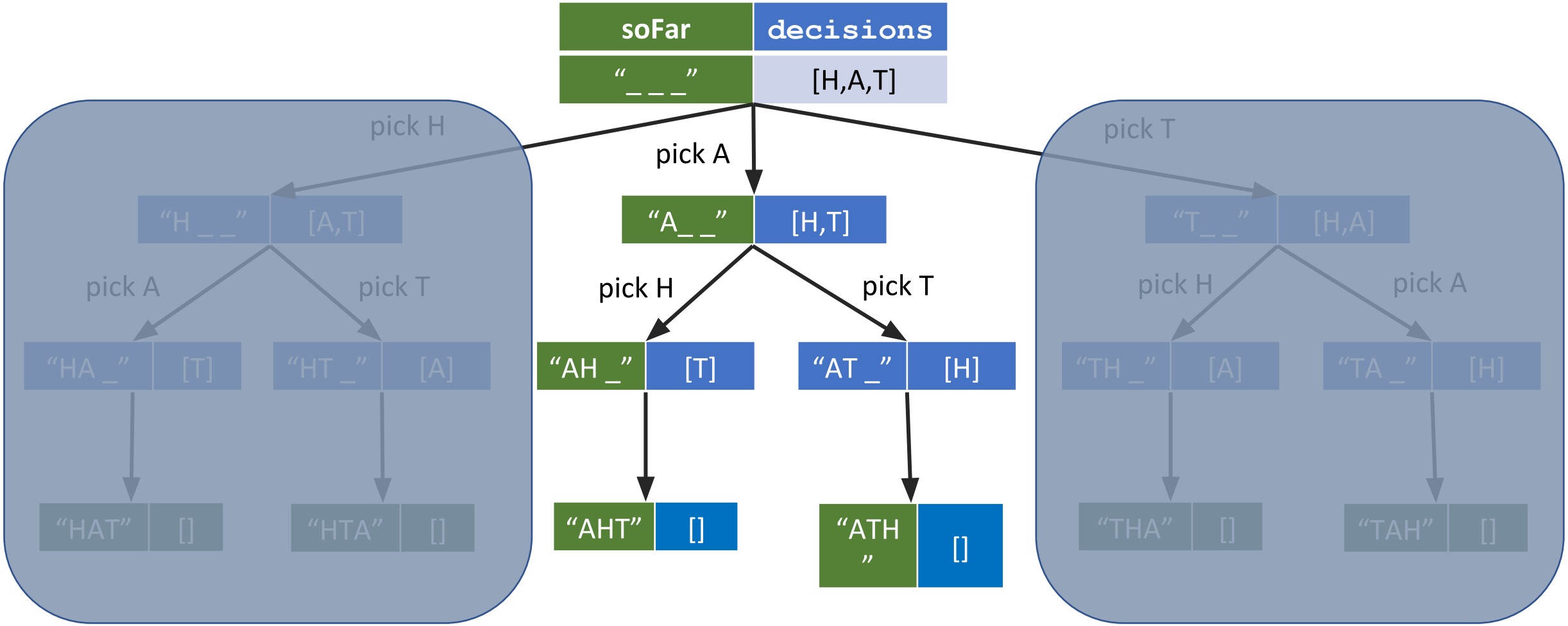
HAT
HTA
AHT
ATH
THA
TAH



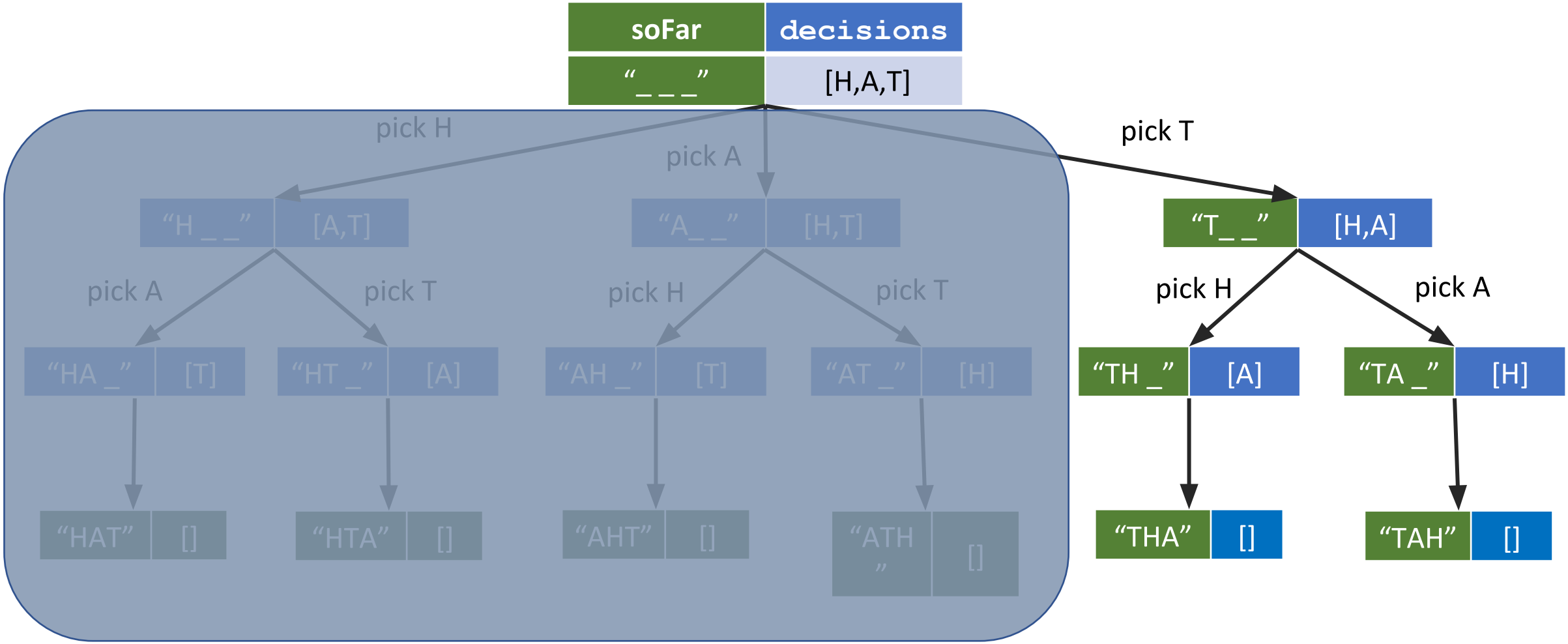
Generating Permutations



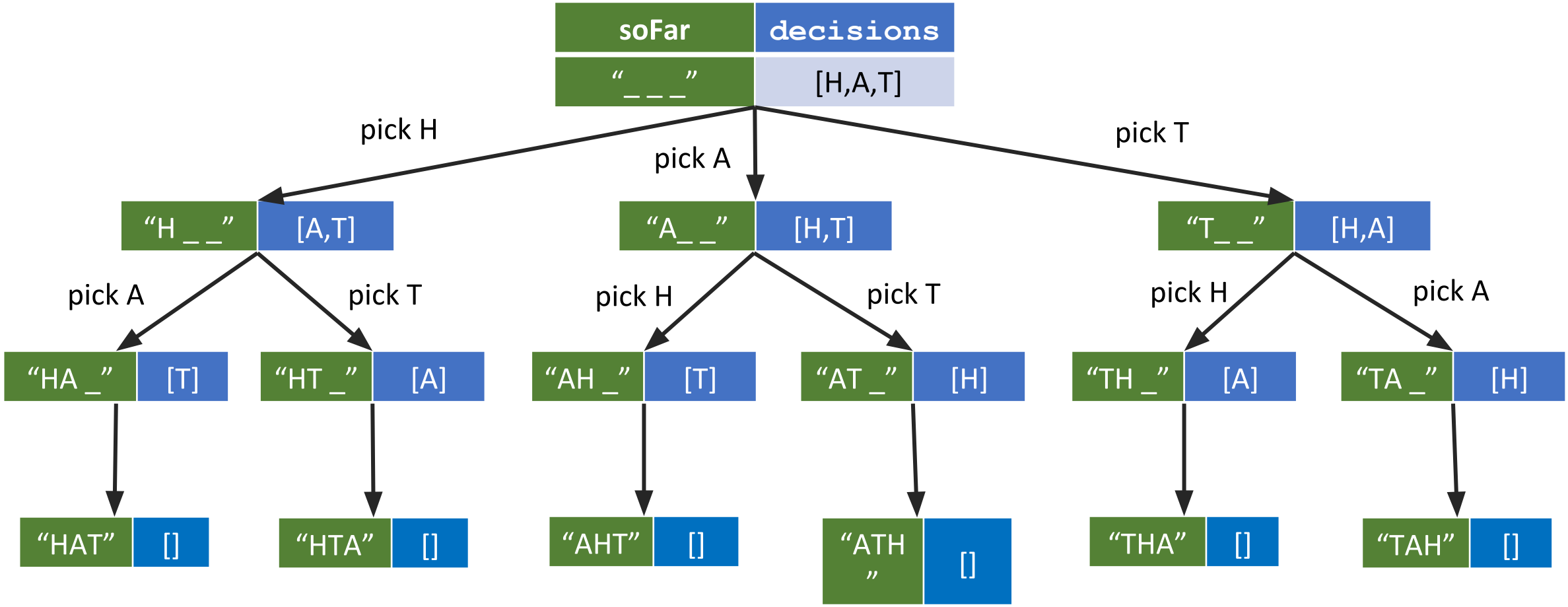
Generating Permutations



Generating Permutations

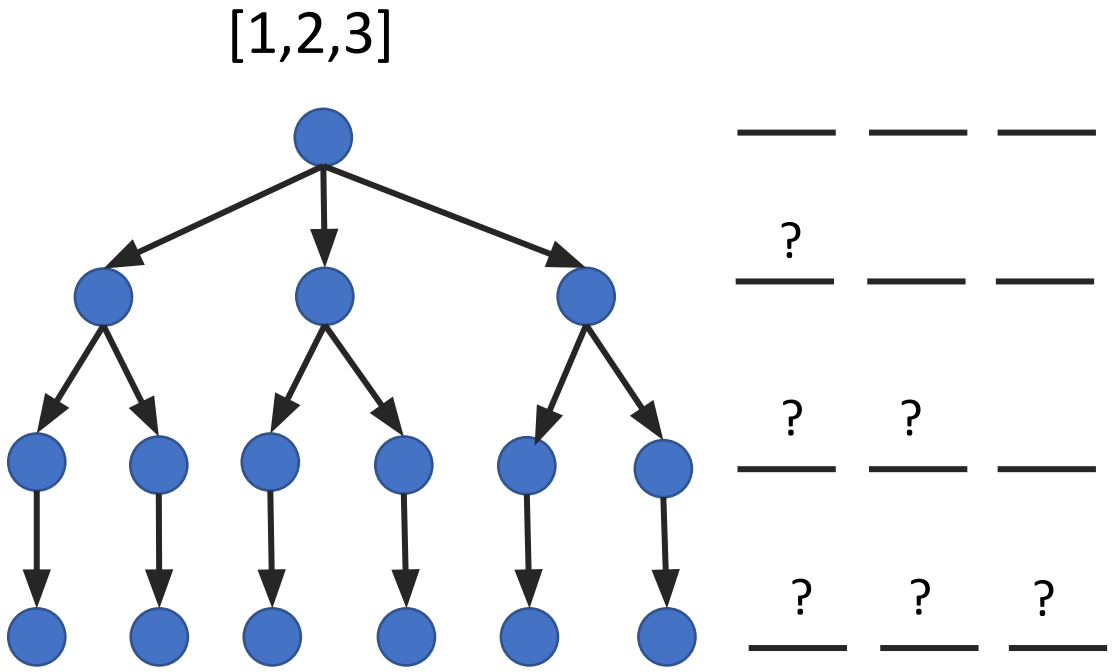


Generating Permutations



Generating Permutations

- Rearranging the members of a set
  - Print them
- Approach:
  - What are the decisions?
  - What are the choices?
  - How to represent the subproblems?
    - Remove the chosen choice from the decision list
    - Move the choice to the front and use an index to tell subordinate where to start from



Generating Permutations

```
void driver (String input) {
  helper(input, “”);
}

void helper(String remaining, String soFar) {
}
```

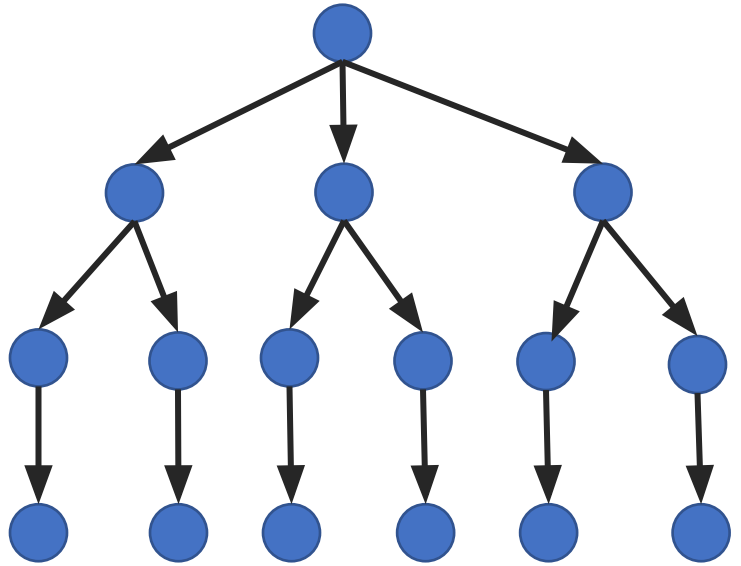
Generating Permutations

```
void driver (String input) {
  helper(input, “”);
}

void helper(String remaining, String soFar) {
  if (remaining.isEmpty()) {
    System.out.println(soFar);
  } else {
    for (int i = 0; i < remaining.length(); i++) { // choices
      String prefix = remaining.subString(0,i);
      String suffix = remaining.substring(i+1);
      helper(prefix+suffix, soFar + remaining.chatAt(i));
    }
  }
}
```

Generating Permutations

Runtime & Space Complexity Analysis of Permutation

```
void helper(String remaining, String soFar) {
  if (remaining.isEmpty()) {
    System.out.println(soFar);
  } else {
    for (int i = 0; i < remaining.length(); i++) {
      String prefix = remaining.subString(0,i);
      String suffix = remaining.substring(i+1);
      helper(prefix+suffix, soFar + remaining.chatAt(i));
    }
  }
}
```



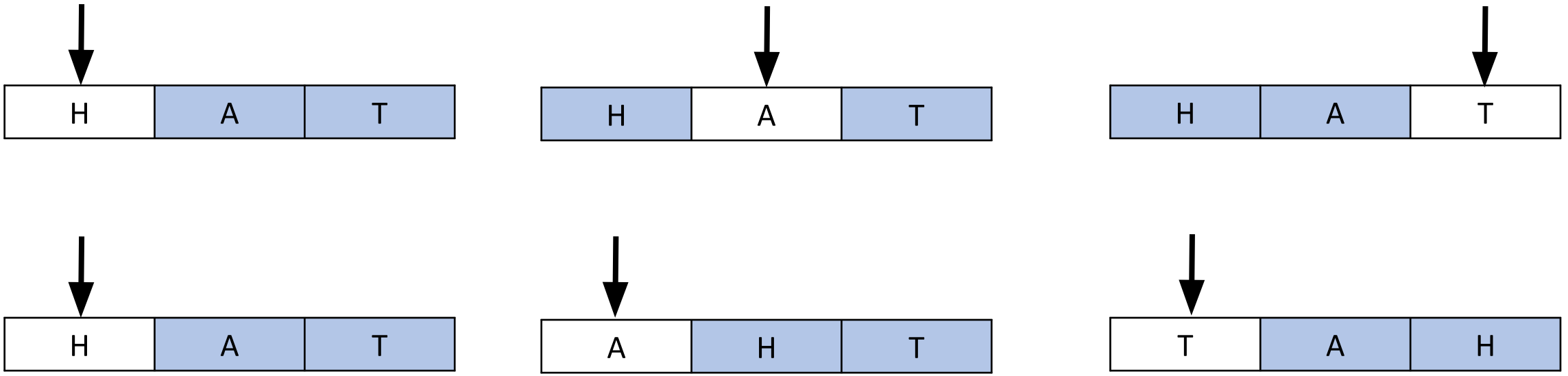
- Time complexity
  - # of calls * amount of work/call
- Space complexity
  - Explicit (collector) + Implicit (stack)

Generating Permutations

Using Mutable Data Structure for Subproblems

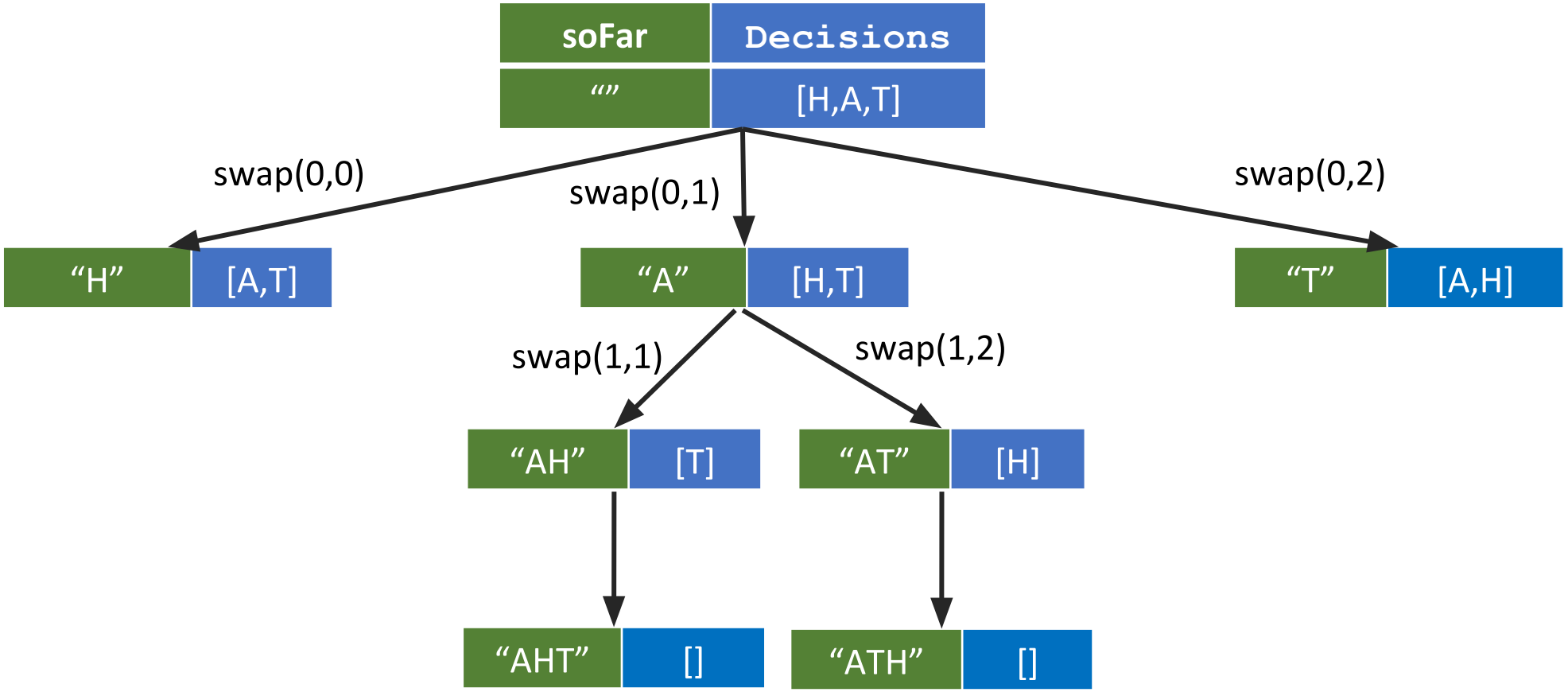
?

Input: [H,A,T], idx=0

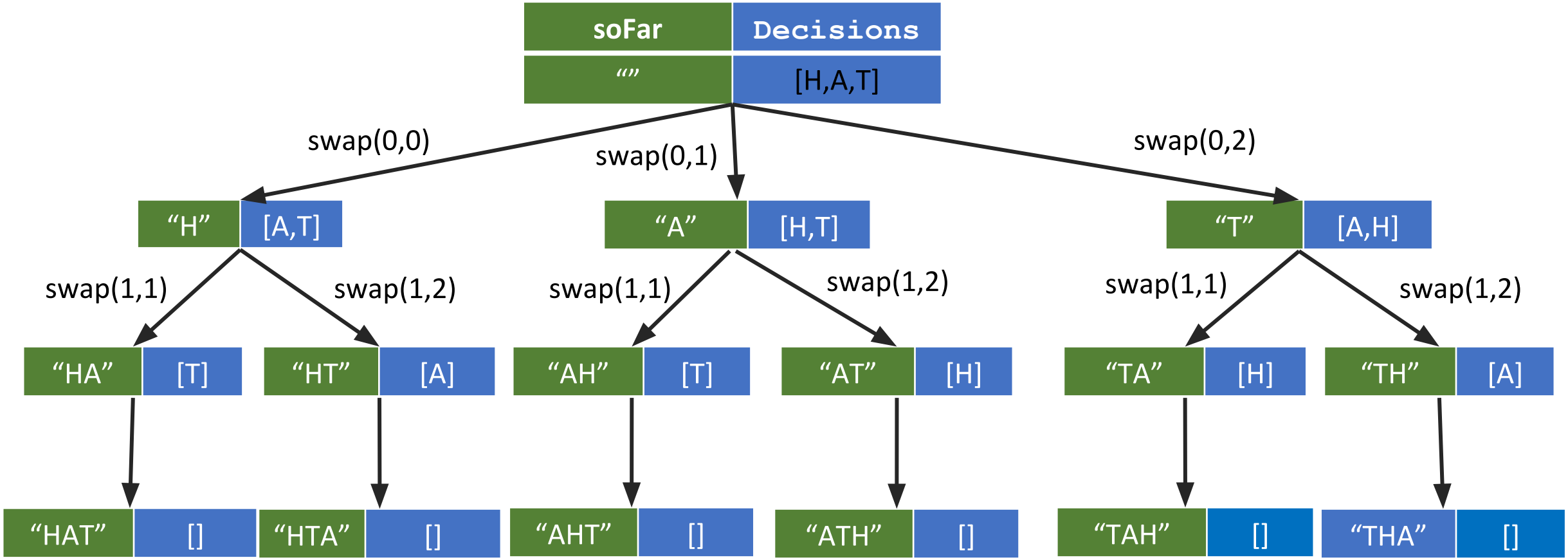


How to pass down the remaining work to subordinates?

Generating Permutations



Generating Permutations



Show the swapping and Unswapping

Generating Permutations

```
void driver(int[] input) {
  helper(input, 0, “”);
}

void helper(int[] input, int idx, String soFar) {
  if (idx == input.length) {
    System.out.println(soFar);
  } else {
    ??
  }
}
```

Generating Permutations

```
void driver(int[] input) {
  helper(input, 0, “”);
}

void helper(int[] input, int idx, String soFar) {
  if (idx == input.length) {
    System.out.println(soFar);
  } else {
    for (int i = idx; i < input.size); i++) {
      swap(input, idx, i);
      helper(input, idx+1, soFar + input[i]);
      swap(input, idx, i); //restore state
    }
  }
}
```

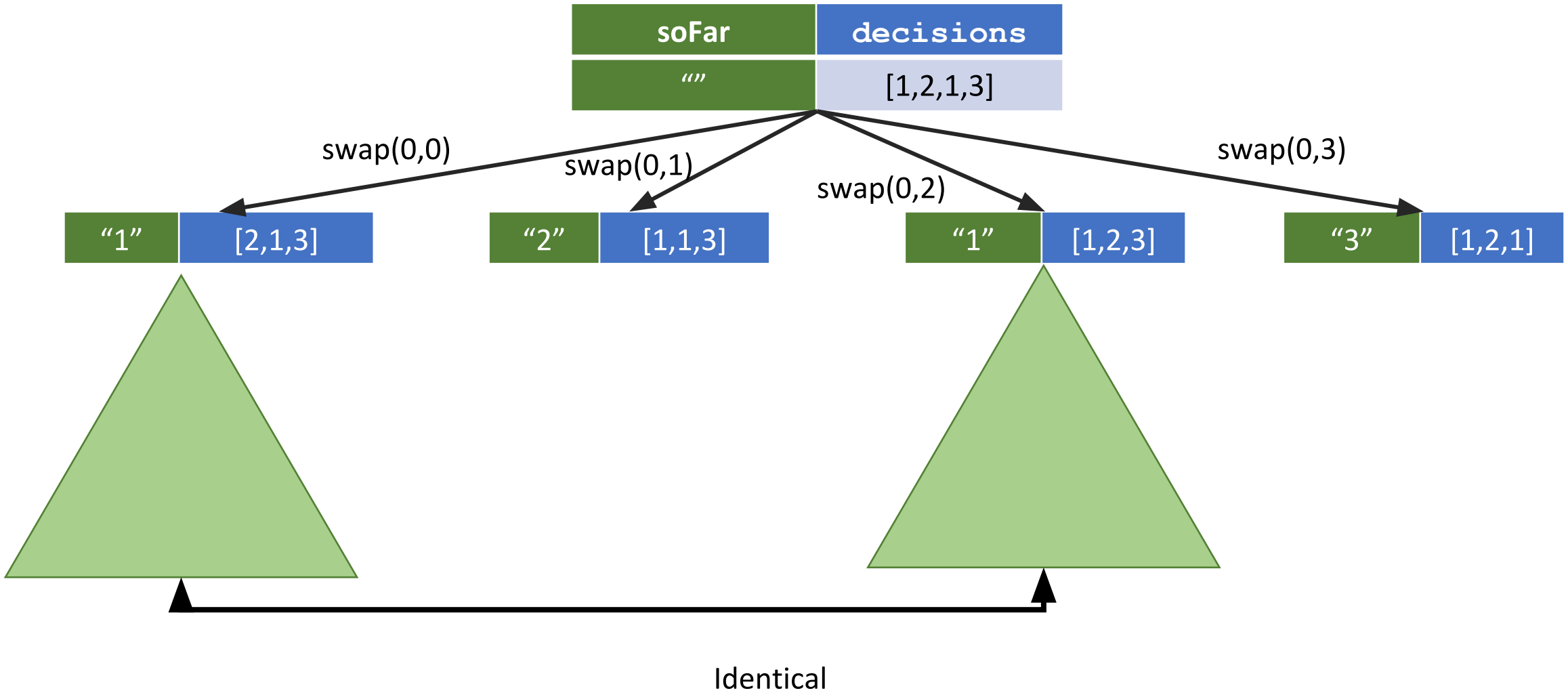
Generating Permutations

```
void helper(int[] input, int idx, StringBuffer soFar) {
  if (idx == input.length) {
    System.out.println(soFar.toString());
  } else {
    for (int i = idx; i < input.size); i++) {
      swap(input, idx, i);
      soFar.append(input[i]);

      helper(input, idx+1, soFar);
      soFar.deleteCharAt(soFar.length()-1);

      swap(input, idx, i); //restore state
    }
  }
}
```

Generating Permutations w/ Duplicates

- What if input has duplicates
  - If don’t handle them properly then there will be duplicates in the output
- Example:
  - Input: [1,2,3,1,2]
- Approach
  - ??

Generating Permutations w/ Duplicates



Generating Permutations w/ Duplicates

- What if input has duplicates
  - If don’t handle them properly then there will be duplicates in the output
- Example:
  - Input: [1,2,3,1,2]
- Approach
  - Keep track of the choices
  - If seen it before, then skip it
  - Don’t make the same choice twice

Generating Permutations w/ Duplicates

```
void helper(int[] input, int idx, String soFar) {
  // base case
  // other case
}
```

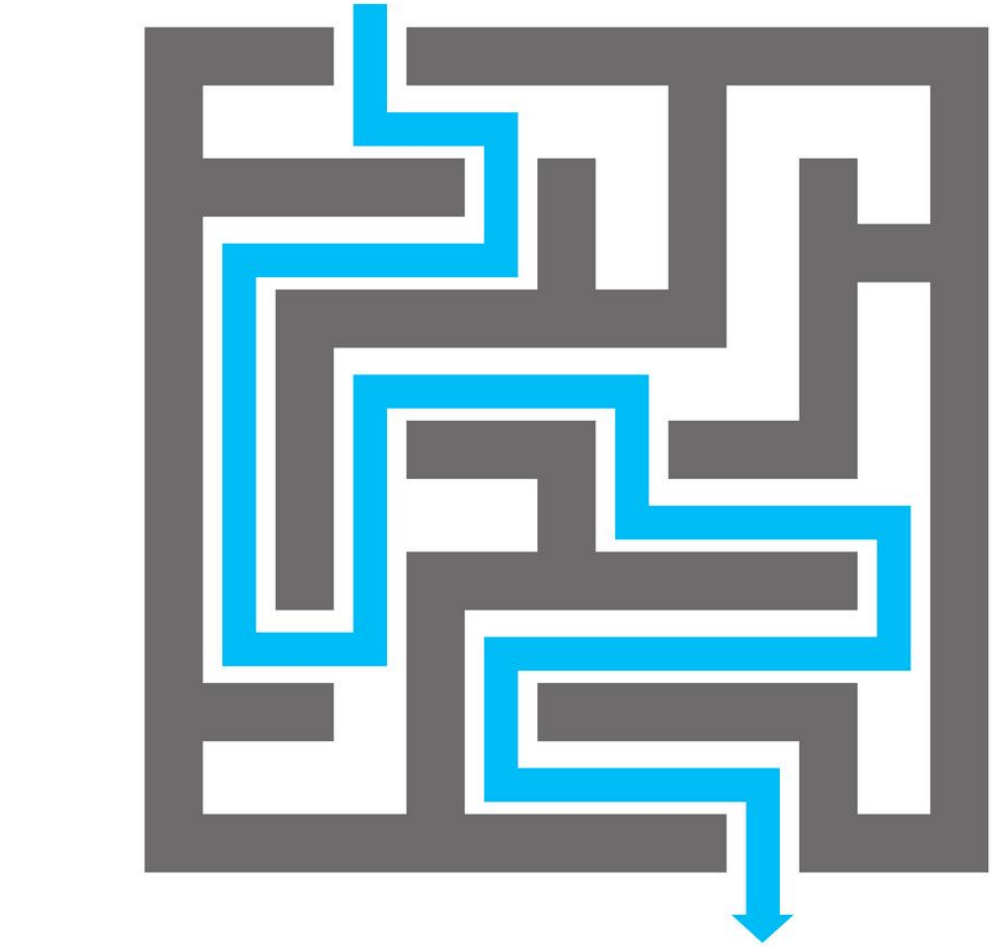
Generating Permutations w/ Duplicates

```
void helper(int[] input, int idx, StringBuffer soFar) {
  if (idx == input.length) {
    System.out.println(soFar.toString());
  } else {
    Set<Integer> cache = new HashSet<Integer>();
    for (int i = idx; i < input.length; i++) {
      if (cache.contains(input[i]) continue;
      cache.add(input[i]);

      swap(input, idx, i);
      soFar.append(input[i]);
      helper(input, idx+1, soFar);
      soFar.deleteCharAt(soFar.length()-1);
      swap(input, idx, i); //restore state
    }
  }
}
```

Exhaustive Search w/ Constraints

Backtracking

- Find one or all possible solutions that meet a certain set of constraints
- Incrementally build candidates to solutions
- Abandon a candidate solution as soon as it can't satisfy the given constraints



Backtracking

- Find solution(s) by trying all possible paths with constraints
  - Determine all possible solutions
  - Determine feasibility (one solution)
  - Determine the number of possible solutions
  - Determine the best (maximum, minimum) solution
- Applications
  - Parsing languages
  - Games: n-queens, Sudoku, Minesweeper

Backtracking - general approach

- Incremental approach
  - Perform exhaustive search w/o constraints
  - Apply the constraints
    - Add the backtracking steps at appropriate places

- Runtime & space complexity
  - Still exponential in worst-case scenario

Backtracking – all possible solutions

```
helper(sub problems, partial-solution) {
  if (no more sub problems) { // base case
    print/collect partial-solution
  } else {
    for (each possible choices) {
      make a choice  // add to partial-solution
      helper(remaining-decisions, partial-solution)
      unmake choice  // if mutable data structure is used
    }
  }
}
```

Backtracking – feasibility

```
helper(sub problems, partial-solution) {
  if (violate constraint) {  // backtracking case
    return;
  }

  if (no more sub problems) { // base case
    print/collect partial-solution
  } else {
    for (each possible choices) {
      make a choice  // add to partial-solution
      helper(remaining-decisions, partial-solution)
      unmake choice  // if mutable data structure is used
    }
  }
}
```

Backtrack Problem – Subset Size

Medium 1236 62 Add to List Share

Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*.

Example:

```
Input: n = 4, k = 2
Output:
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

Backtrack Problem – Subset Size

```
// w/o enforcing the constraint
helper(int[] numbers, int idx, StringBuffer soFar,
       List<String> coll, int k) {
  if (idx == numbers.length) {
    coll.add(soFar.toString()); // a copy of soFar
  } else {
    helper(numbers, idx+1, soFar);  // exclude

    soFar.append(numbers[idx]);     // include
    helper(numbers, idx+1, soFar);
    soFar.deleteCharAt(soFar.length()-1);
  }
}
```

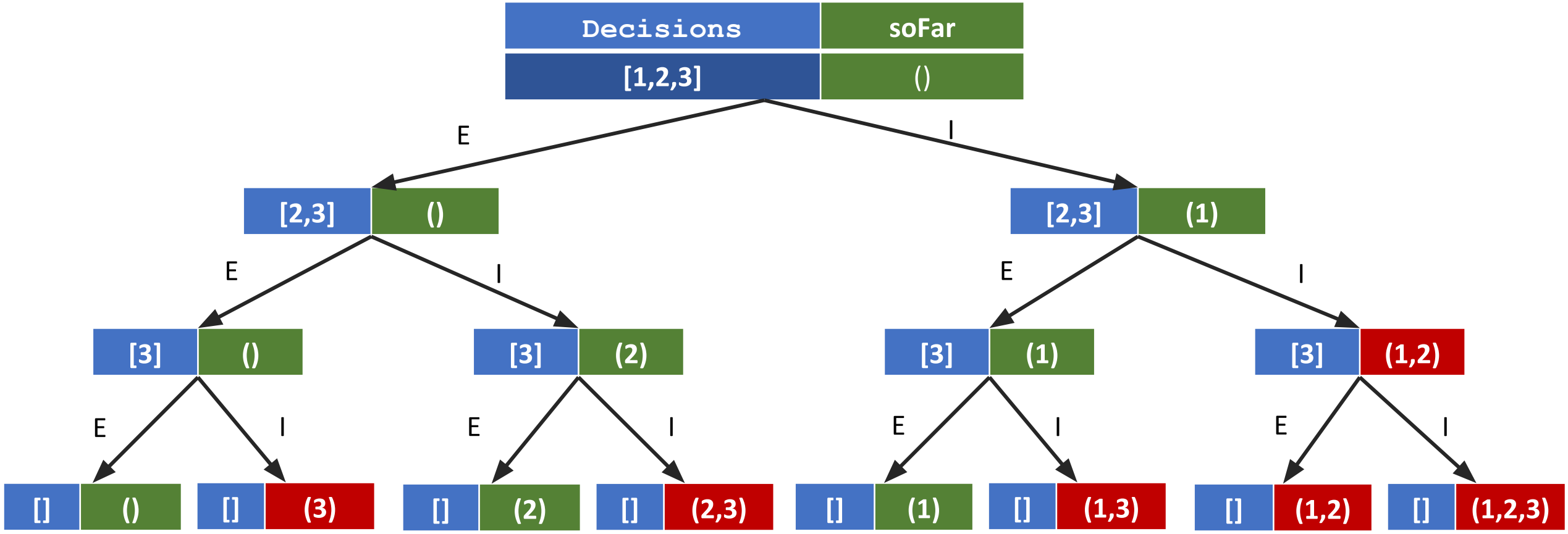
Backtrack Problem – Subset Size

```
// with enforcing the constraint
helper(int[] numbers, int idx,StringBuffer soFar,
       List<String> coll, int k) {
  if (soFar.length() == k) {  // backtracking case
    coll.add(soFar.toString());
  } if (idx == numbers.length) {  // base case
    return;
  } else {
    helper(numbers, idx+1, soFar);  // exclude

    soFar.append(numbers[idx]);     // include
    helper(numbers, idx+1, soFar);
    soFar.deleteCharAt(soFar.length()-1);
  }
}
```

Backtrack Problem – Subset Sum

- Print all subsets of a set of distinct integers – [1,2,3]
  - With sum equals to K = 2
- What question(s) should we ask about the integers?
  - All positive integers
  - All positive integers with 0
  - Positive, 0 and negative
- Approach
  - When to compute the subset sum?
    - At the leaf level
    - On the fly

All subsets – Subset Sum([1,2,3], 2)



violate the constraint

Backtrack Problem – Subset Sum

```
// printing subsets w/o constraint
void helper(int[] input, int idx, List<Integer> path, int target)
{
  if (idx == input.length) {
     System.out.println(path);
  } else {
    helper(input, idx+1, path, target); // exclude

    path.add(input[idx]);
    helper(input, idx+1, path, target); // include
    path.remove(path.size()-1);
  }
}
```

Backtrack Problem – Subset Sum w/ pruning

```
void helper(int[] input, int idx, List<Integer> path,
            int sumSoFar, int target) {

  if (sumSoFar == target) System.out.println(path); return;
  if (idx == input.length || sumSoFar > target) {
     return;
  } else {
    helper(input, idx+1, path, sumSoFar, target); // exclude

    // include
    path.add(input[idx]);
    helper(input, idx+1, path, sumSoFar + input[idx] ,target);
    path.remove(path.size()-1);
  }
}
```

Backtrack Problem – Subset Sum

- Print all subsets of a set of distinct integers
  - With sum equals to K
- What question(s) should we ask about the integers?
  - All positive integers
  - All positive integers with 0
  - Positive, 0 and negative

Backtrack Problem – Dice Roll Sum

- Print all combinations of dice values that add up to given sum
  - Each dice has 6 values: {1,2,3,4,5,6}
- For example:
  - Dice=2, Sum=7
    - Output: {1,6}, {2,5}, {3,4}, {4,3},{5,2}, {6,1}
  - Dice=3, Sum=7
    - Output: {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}

Backtrack Problem – Dice Roll Sum

- Print all combinations of dice values that add up to given sum
  - Each dice has 6 values: {1,2,3,4,5,6}
  - diceSum(numDice, target sum)
  - diceSum(3, 7)
- Approach
  - What are the decisions to iterate over?
  - What are the choices at each decision?
  - Base case?
  - What does the recursion tree look like?

Backtrack Problem – Dice Roll Sum

```
void driver(int numDice, int targetSum) {
  helper(??);
}

void helper(??) {
}
```

Backtrack Problem – Dice Roll Sum

```
void helper(int numDice, int targetSum, List<Integer> path) {

   if (numDice == 0) {
     if (computeSum(path) == targetSum) {
       System.out.println(path);
     }
   } else {
     for (int value = 1; value <= 6; value++) {
       path.add(value);
       helper(numDice-1, targetSum, path);
       path.remove(path.size()-1);
     }
   }
}
// what are the opportunities for pruning?
```

Backtrack Problem – Dice Roll Sum

- Analyzing **`helper`**
  - Exhaustive search even for paths that not lead to success
  - Current sum is already too high
- What are the opportunities for pruning
  - We must roll every remaining dice
  - How can we take advantage of that info.?
  - sumSoFar + 1*numDice > targetSum
  - sumSoFar + 6*numdice < targetSum

Backtrack Problem – Dice Roll Sum

```
void helper(int numDice, int targetSum, int sumSoFar
                    List<Integer> path) {
    if (numDice == 0) {
        if (computeSum(path) == targetSum)
            System.out.println(path);
    } else if (sumSoFar + 1*numDice > targetSum ||
               sumSoFar + 6*numDice < targetSum) {
      return;
    } else {
      for (int value = 1; value <= 6; value++) {
        path.add(value);
        helper(numDice-1, targetSum, sumSoFar + value, path);
        path.remove(path.size()-1);
      }
    }
}
```

Backtrack Problem – Generate Parentheses

- Given *n* pairs of parentheses
  - Generate all combinations of well-formed parentheses

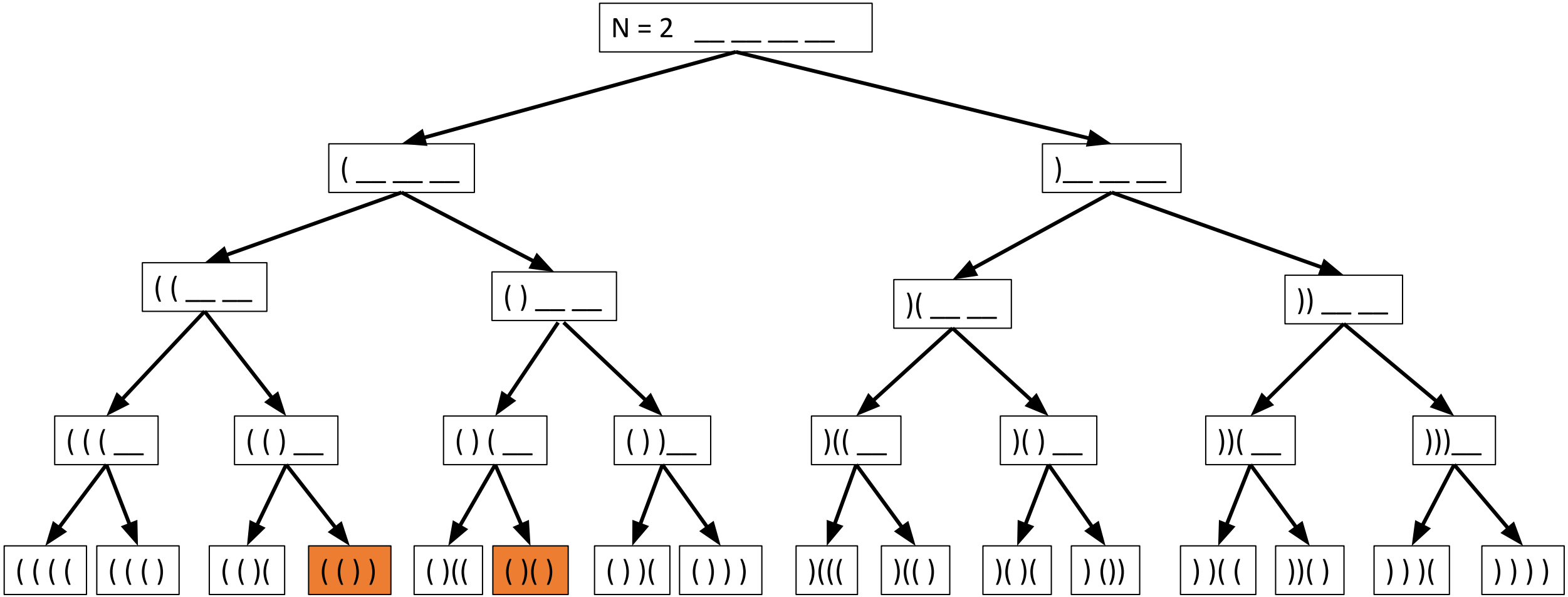
- For example
  - N=1 - [“( )”]
  - N=2 - [“(( ))”, “( ) ( )”]
  - N = 3 - ["(( ) ( ))”, "((( )))”, "(( )) ( )”, "( ) ( ) ( )”, "( ) (( ))”]

How comfortable do you feel you can solve this problem at this point?

Backtrack Problem – Generate Parentheses

- Given *n* pairs of parentheses
  - Print all combinations of well-formed parentheses
- For example
  - N=1 - [“( )”]
  - N=2 - [“(( ))”, “( ) ( )”]
  - N = 3 - ["(( ) ( ))”, "((( )))”, "(( )) ( )”, "( ) ( ) ( )”, "( ) (( ))”]
- Approach
  - What are the decisions?
  - What are the choices?
  - What are the constraints? When to apply them?

Backtrack Problem – Generate Parentheses



Backtrack Problem – Generate Parentheses

```
// without constraints - print
void driver(int n) {
    helper(2*n, “”);
}

// what pattern can we leverage? subproblems? partial solution?
void helper(int remaining, String soFar) {
  ??
}
```

Backtrack Problem – Generate Parentheses

```
// without constraints - print
void driver(int n) {
    helper(2*n, “”);
}

// what pattern can we leverage? subproblems? partial solution?
void helper(int remaining, String soFar) {
  if (remaining == 0) {
    if (isWellFormed(soFar)) print(soFar);
  } else {
    helper(remaining-1, soFar + “(”);
    helper(remaining-1, soFar + “)”);
  }
}
```

Backtrack Problem – Generate Parentheses

```
//"()("

boolean isWellFormed(String input) {
   ??
}
```

Backtrack Problem – Generate Parentheses

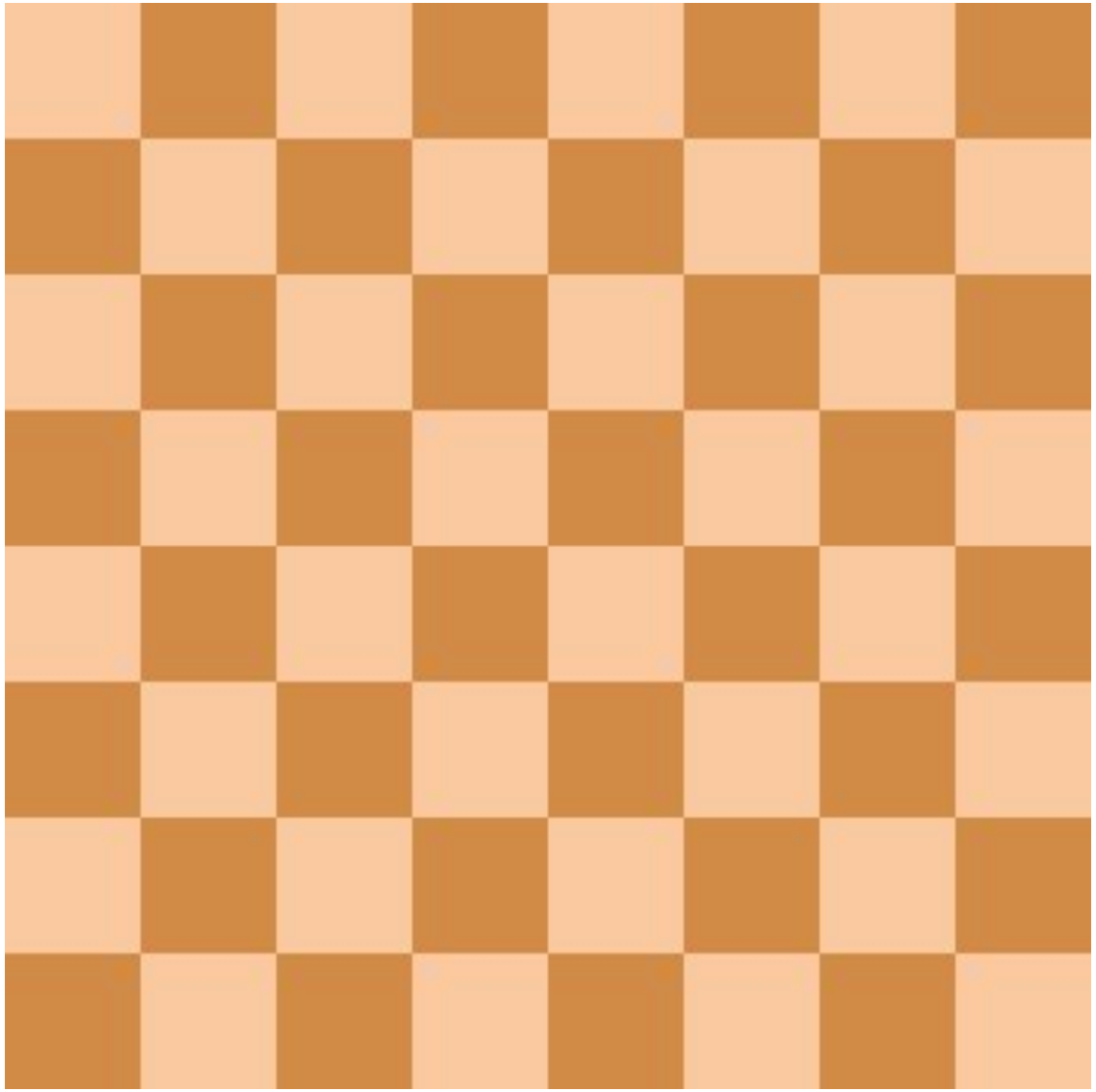
- Given *n* pairs of parentheses
  - Generate all combinations of well-formed parentheses
- For example
  - N=1 - [“( )”]
  - N=2 - [“(( ))”, “( ) ( )”]
  - N=3 - ["(( ) ( ))”, "((( )))”, "(( )) ( )”, "( ) ( ) ( )”, "( ) (( ))”]
- Formalize the constraints
  - The number of remaining “(“ must be <= to the remaining “)” at any point in time

Backtrack Problem – Generate Parentheses

```
void driver(int n) {
    helper(n, n, new StringBuffer());
}
void helper(int lParanCnt, int rParanCnt, StringBuffer soFar) {
  if (lParanCnt > rParanCnt) return;
  if (lParanCnt < 0 || rParanCnt < 0)  return;

  if (lParanCnt == 0 && rParanCnt == 0) {
    System.out.print(soFar.toString());
  } else {
    soFar.append("(");
    helper(lParanCnt - 1, rParanCnt, soFar);
    soFar.removeCharAt(soFar.length()-1);

    soFar.append(")");
    helper(lParanCnt, rParanCnt - 1, soFar);
    soFar.removeCharAt(soFar.length()-1);
  }
}
```

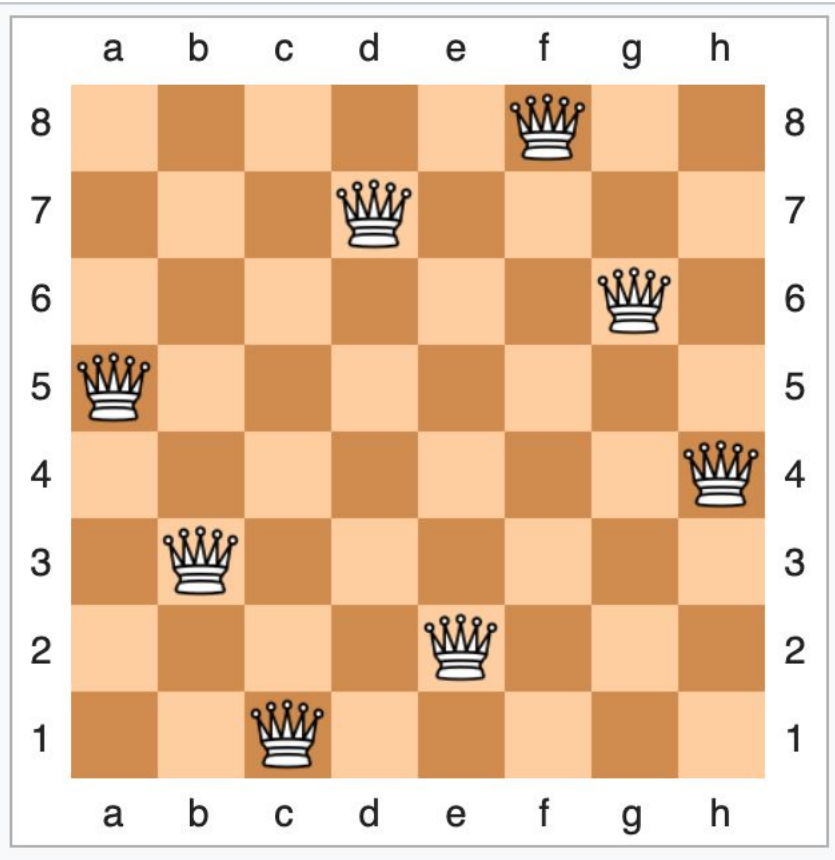
Backtrack - Venerable N-Queens Problem

- Given an NxN board, place N queens such that they can’t attack each other
  - A queen can attack another queen if they are on same row or column or diagonal line
- Return all possible boards



Backtrack Problem - N Queens

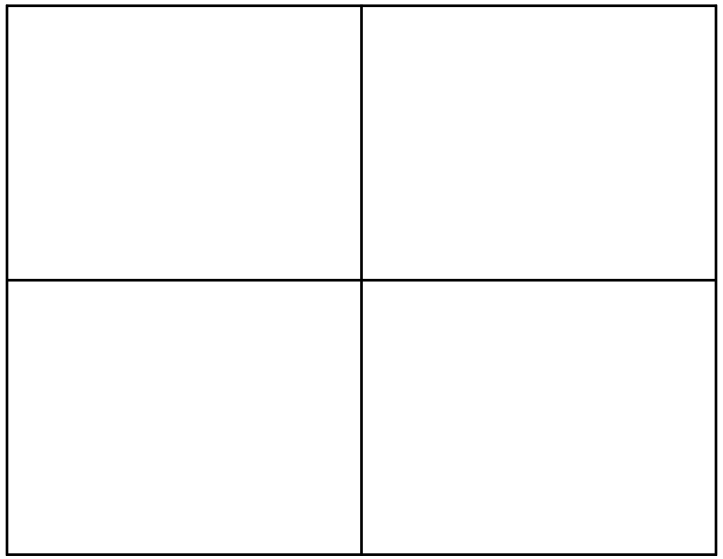
- Published in 1848 by chess composer Max Bezzel
- First solution was published 1850
- For an 8x8 board
  - 4,426,165,368 possible arrangements
  - Only 92 are valid



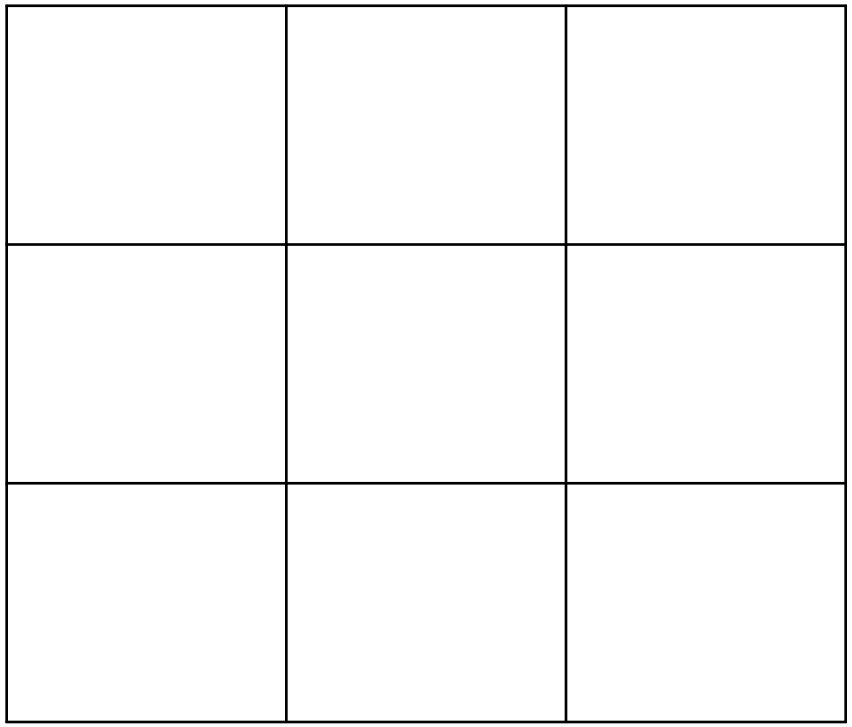
Backtrack Problem - N Queens

A queen can attack another queen if they are on same row or column or diagonal line

2 x 2

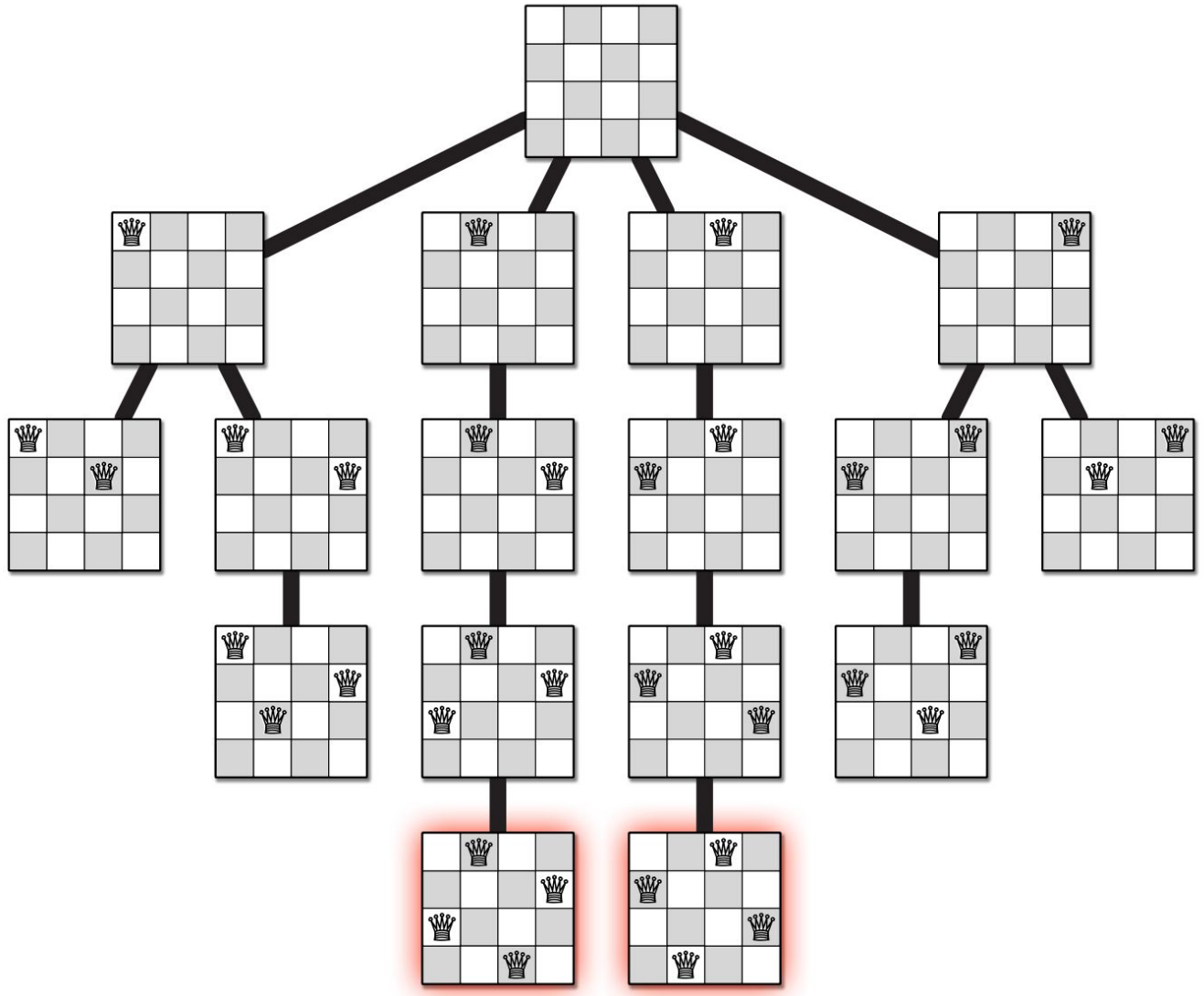


3 x 3



Backtrack Problem - N Queens

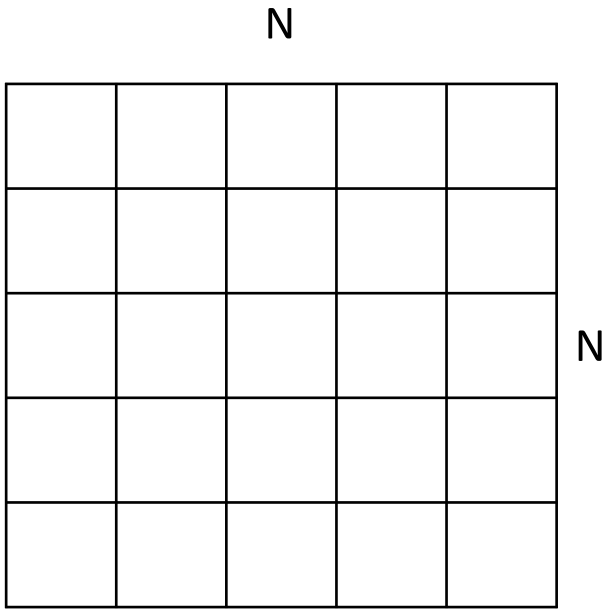
4 Queens



Backtrack Problem - N Queens

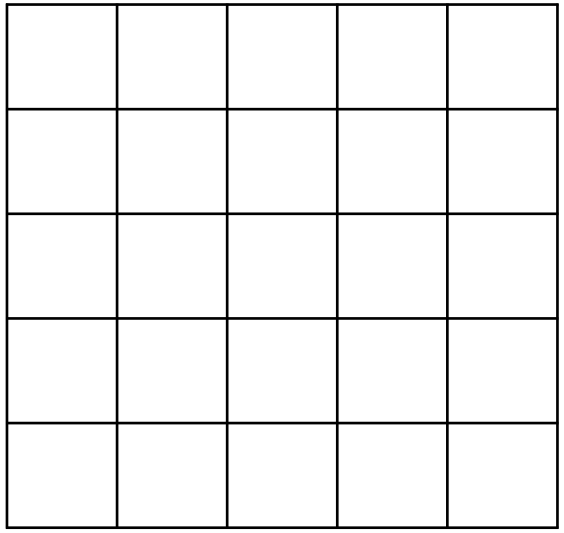
- Start with naïve approach - without the constraints
  - Place each queen in all possible cells
  - How many possible choices for 1st queen?
  - How many possible choices for 2nd queen?
  - How many possible choices for 3rd queen?

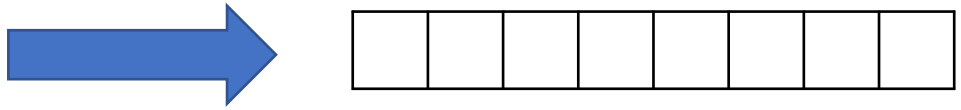
  - $N^2 \times N^2 - 1 \times N^2 - 2$, etc…



Backtrack Problem - N Queens

- Start with a slightly better approach - with row constraint
  - At the minimum - each queen must be on a separate **row**
  - Place a queen in all possible columns in each row
    - Check if each combination meets the constraints?
  - How many possible choices for 1^{st} queen, 2^{nd} queen, 3^{rd} queen?
    - NxNxN => N^N





Each cell represents a row
Each cell value represents the column

Backtrack Problem - N Queens

- A better approach with row & column constraint
  - Place one queen per row and explore all possible different columns
    - Check if each combination meets the constraints?
  - How many possible choices for 1st queen, 2nd queen, 3rd queen?
    - N, N-1, N-2 => N!

1-D array

Index -> row number
Value -> column number

Backtrack Problem - N Queens

- Translate to code
  - Which template from earlier problems we can leverage?
    - How many decisions?
    - How many choices at each decision?
  - Need a way to keep track which queens have placed on the board?
  - The board can be represented by 1-d array
    - since each queen must not be on the same row
    - Index represents the row number
    - The value in each cell represents the column
  - How to represent the subproblems?
  - How to represent the partial solution?

Backtrack Problem - N Queens

Decision tree/general strategy

Backtrack Problem - N Queens

```
List<List<Integer>> driver(int boardSize) {
  List<List<Integer>> boards = new ArrayList<>();
  List<Integer> board = new ArrayList<>();
  helper(??)
  return boards;
}

void helper(??) {
  // what are the decision?
  // what are the choices at each decision?
}
```

Backtrack Problem - N Queens

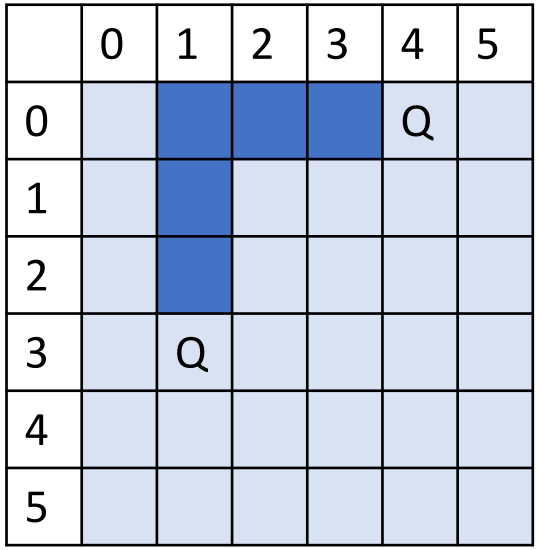
```
// write code here
void helper(??) {
  // what are the decision?
  // what are the choices at each decision?
}
```

Backtrack Problem - N Queens

```
void helper(int boardSize, int row, List<Integer> board,
            List<List<Integer>> coll) {

  if (row == boardSize) {
     coll.add(new ArrayList<>(board));
  } else {
    // explore possible columns for a particular row
    for (int col = 0; col < boardSize; col++) {
     if (isSafeToPlaceQueenAt(board, row, col) {
        board.append(col);
        helper(boardSize, row+1, board, coll);
        board.remove(board.size()-1);
     }
  }
}
```

Backtrack Problem - N Queens

```
boolean isSafeToPlaceQueenAt(List<Integer> board, int row, int col) {
  // check for column conflict
  // check for diagonal conflict
}
```



Q1: (0,4), Q2: (3,1)

ABS(0-3) == ABS(4-1)

Backtrack Problem - N Queens

```
boolean isSafeToPlaceQueenAt(List<Integer> board, int row2,
                                    int col2) {
   for (int col1 : board) {
       if (col1 == col2) return false;
   }
   // diagonal
   for (int row1 = 0; row1 < board.size(); row1++) {
      int xDist = Math.abs(row1 - row2);
      int yDist = Math.abs(board.get(row1) - col2);

      if (xDist == yDist) return false;
   }
   return true;
}
```

Summary

- Identify the problem type
  - Permutation
  - Combination
- Visualize by drawing recursion tree
- Identify the decision points and choices
  - How to represent the subproblems
  - How to represent the partial solution (mutable vs immutable data structure)
- Apply the solution template
  - Base case and recursive case
  - Apply the constraints if needed
    - Abstract out the utility/helper function

The only thing which can save/help you in interviews is your thinking ability

Thank You

“Tell me and I will forget. Show me and I will remember. Involve me and I will understand.”

Quick survey at the end of the class