Cairo University, Faculty of Computers and AI

CS213 – 2022 / 2023

Programming II

Lecture 16: Backtracking

By

Dr. Mohammad El-Ramly

Lecture Objectives

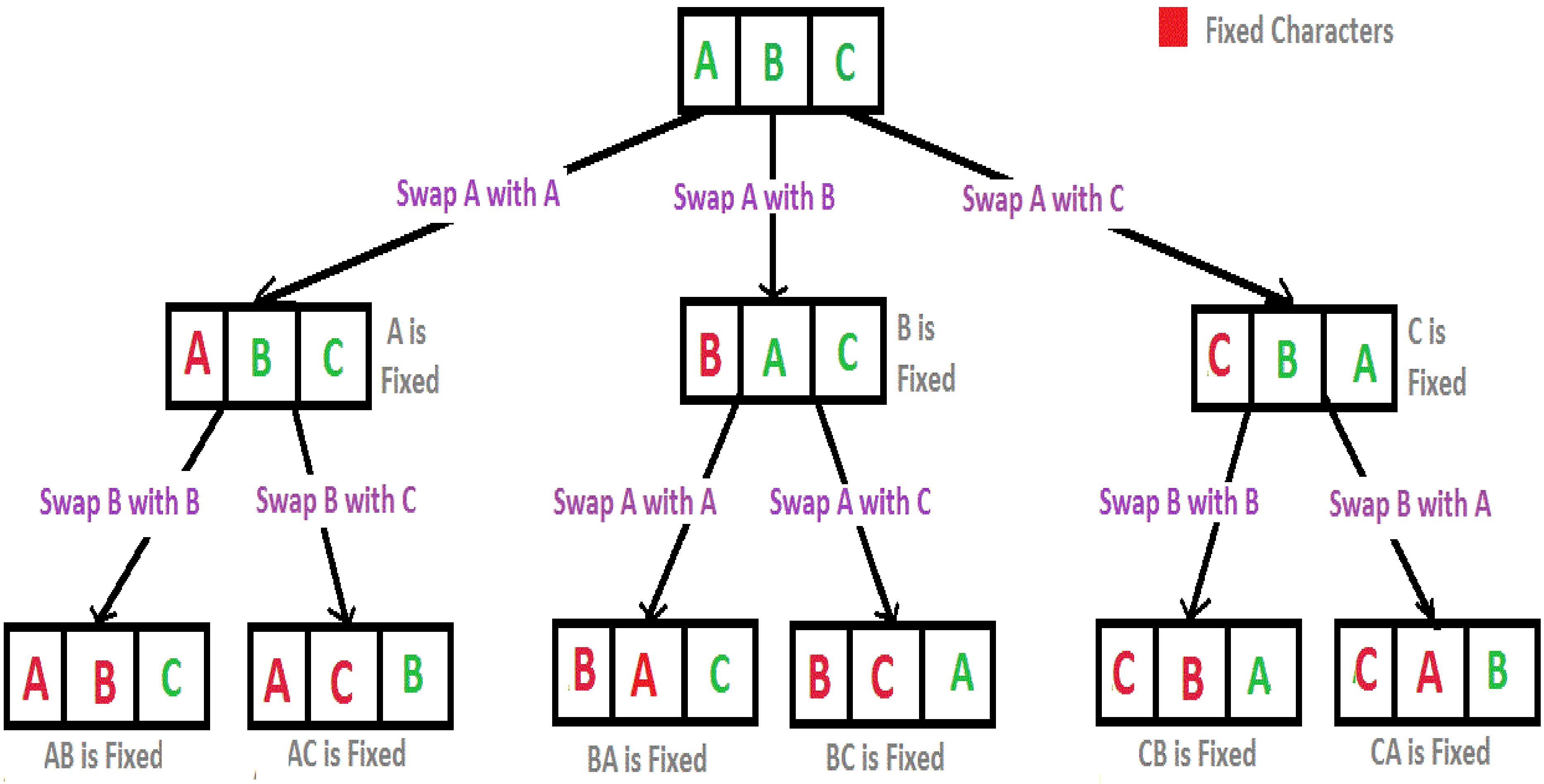
- Understanding exhaustive recursion
  - Finding all words formed from some letters
- Understand backtracking
- Apply it to some popular problems
  - Solving a maze
  - Nim

Exhaustive Recursion

- Exhaustive recursion is when we explore a big space of possibilities and we want to try ***all*** of them
- For example you may want to get all the words that can be formed from the letters of a string (permutations).

Exhaustive Recursion

- Permutations
  - Want to enumerate all rearrangements: ABCD permutes to DCBA, CABD, etc.
- Solving recursively
  - Choose a letter from input to append to output
  - Recursively permute remaining letters onto output
- How to ensure each letter is used exactly once?
- What is the base case?



Recursion Tree for Permutations of String "ABC"

https://www.geeksforgeeks.org/c-programs-gq/cc-backtracking-programs-gq/

Permute Strategy

- Result is empty starting input is "abcd"
- Choose a letter to be first say "**a**"
- Result so far is "a“ remaining input is "**bcd**"
- Recursively permute to get all "**bcd**" combos
- After finishing permutations with "a" in front need to go again with "**b**" in front and then "c" and so on

Permute Code

```
void RecPermute(string soFar, string rest)
{
  if (rest == "") {
     cout << soFar << endl;
  } else {
     for (int i = 0; i < rest.length(); i++) {
          string next = soFar + rest[i];
          string remaining = rest.substr(0, i)
                               + rest.substr(i+1);
          RecPermute(next, remaining);
     }
  }
}
// "wrapper" function
void ListPermutations(string s) {
  RecPermute("", s);
}
```

- Now: soFar rest : c   at
- Now: soFar rest : ca   t
- Now: soFar rest : cat
- **cat**

- Now: soFar rest : ct   a
- Now: soFar rest : cta
- **cta**

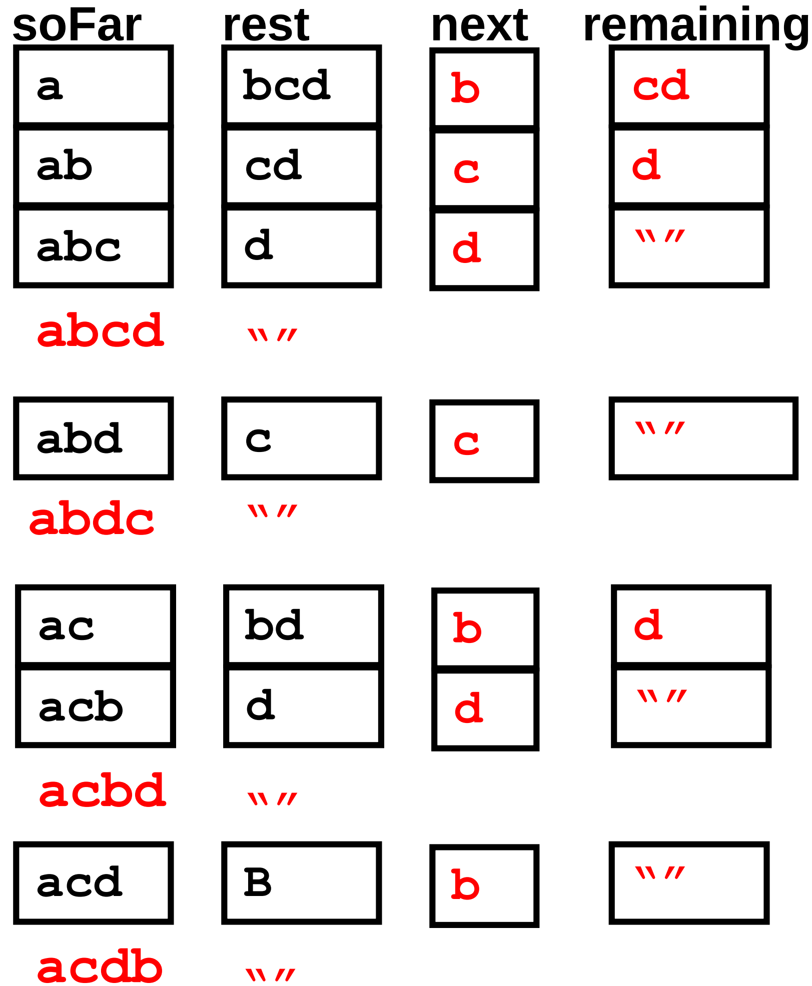
- Now: soFar rest : a   ct
- Now: soFar rest : ac   t
- Now: soFar rest : act
- **act**

- Now: soFar rest : at   c
- Now: soFar rest : atc
- **atc**

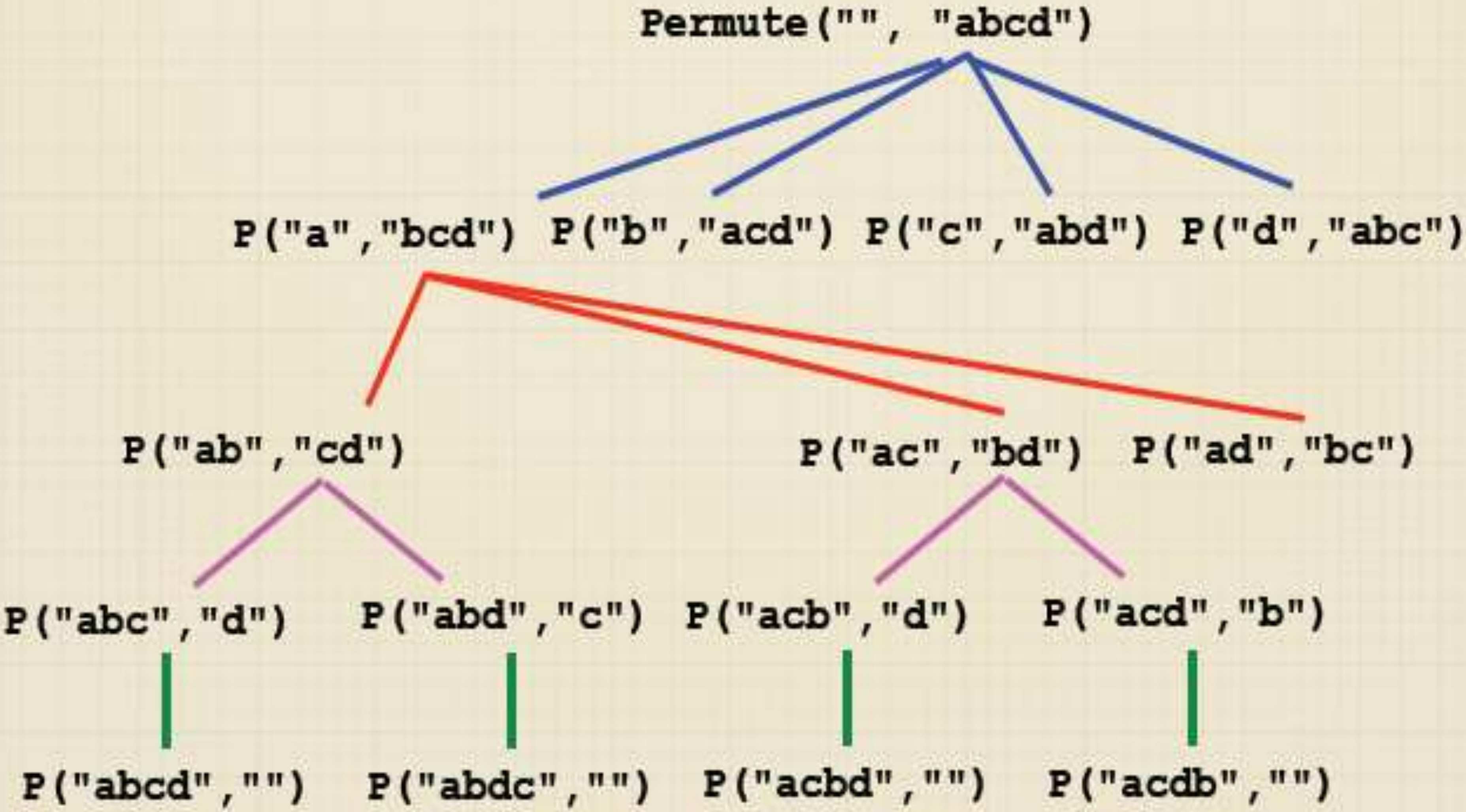
- Now: soFar rest : t   ca
- Now: soFar rest : tc   a
- Now: soFar rest : tca
- **tca**

- Now: soFar rest : ta   c
- Now: soFar rest : tac
- **tac**

- Now: soFar rest : a bcd
- Now: soFar rest : ab cd
- Now: soFar rest : abc d
- Now: soFar rest : abcd
- **abcd**
- Now: soFar rest : abd c
- Now: soFar rest : abdc
- **abdc**
- Now: soFar rest : ac bd
- Now: soFar rest : acb d
- Now: soFar rest : acbd
- **acbd**
- Now: soFar rest : acd b
- Now: soFar rest : acdb
- **acdb**
- Now: soFar rest : ad bc
- Now: soFar rest : adb c
- Now: soFar rest : adbc
- **adbc**
- Now: soFar rest : adc b
- Now: soFar rest : adcb
- **adcb**



Tree of recursive calls



Exhaustive Recursion

- Permutations/subsets have deep/wide tree of recursive calls
- Depth represents total number of decisions made
- Width of branching represents number of available options per decision
- Exhaustive recursion *explores* ***every possible option at every decision point***

Exhaustive Recursion

- Typically very expensive
- **N!** permutations (720 strings for “abcdef”)
- Recursion isn't the problem there just is a huge space to explore
- Consider partial exploration of exhaustive space
- Similar exhaustive structure but stop at first "satisfactory" outcome

Recursive Backtracking

- What if we want just any valid English words consisting of these letters?
- We need partial exploration of exhaustive space and we test each word if it is in dictionary or not.
- And we stop at the first *satisfactory* outcome.
- This is called ***backtracking***

Recursive Backtracking

- Suppose you have to make a series of *decisions* among various *choices* where
  - You don't have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem
- Backtracking is a methodical way of trying out various sequences of decisions until you find one that "***works***"

Backtracking Approach

- Design recursion function to **return success / failure**

1. At each call **choose one option** and go with it
2. Recursively proceed and see what happens
   A. If it works out great otherwise unmake choice and try again
   B. If no option worked return fail result which triggers backtracking (i.e. un-making earlier decisions)

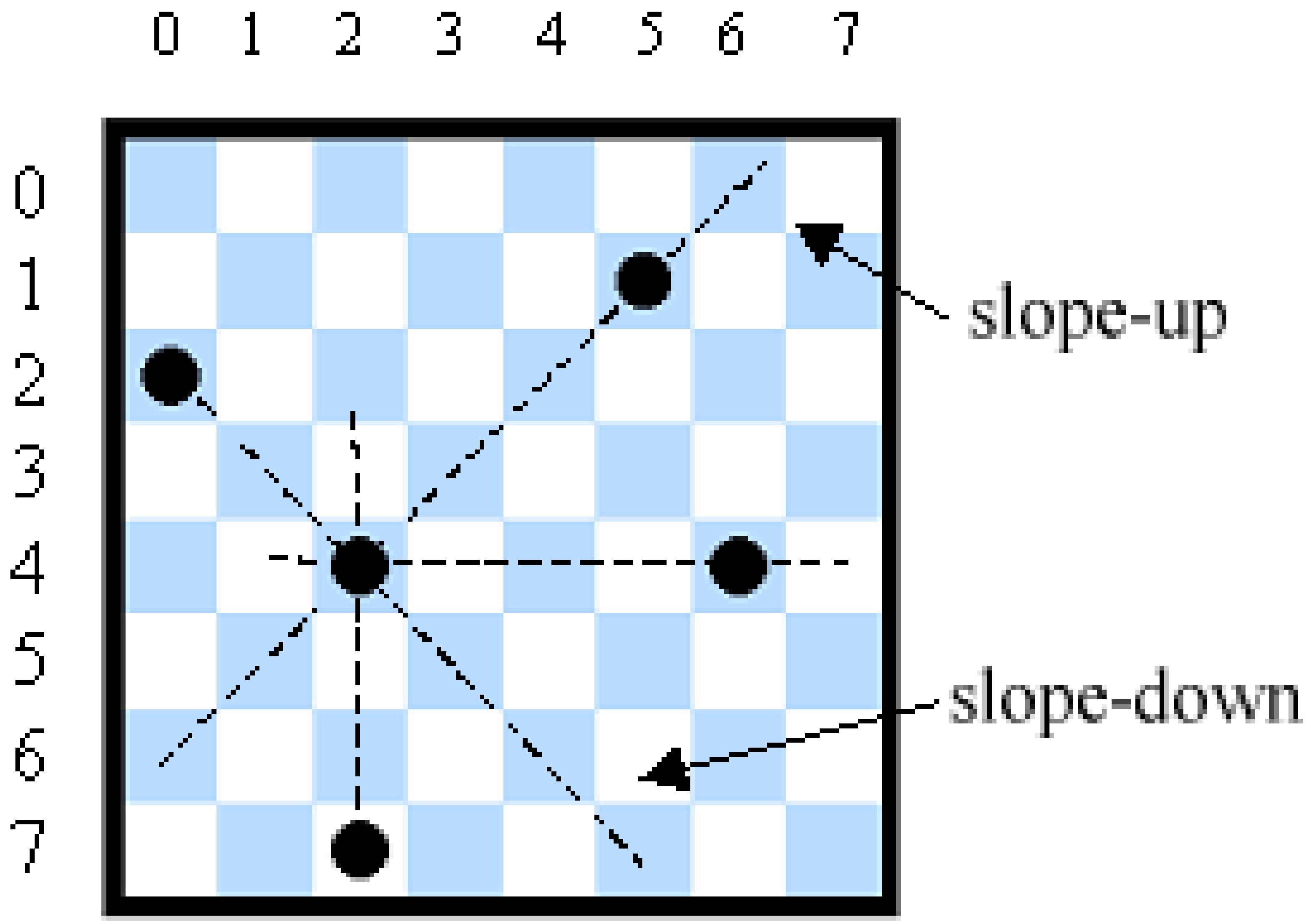
Backtracking Problems

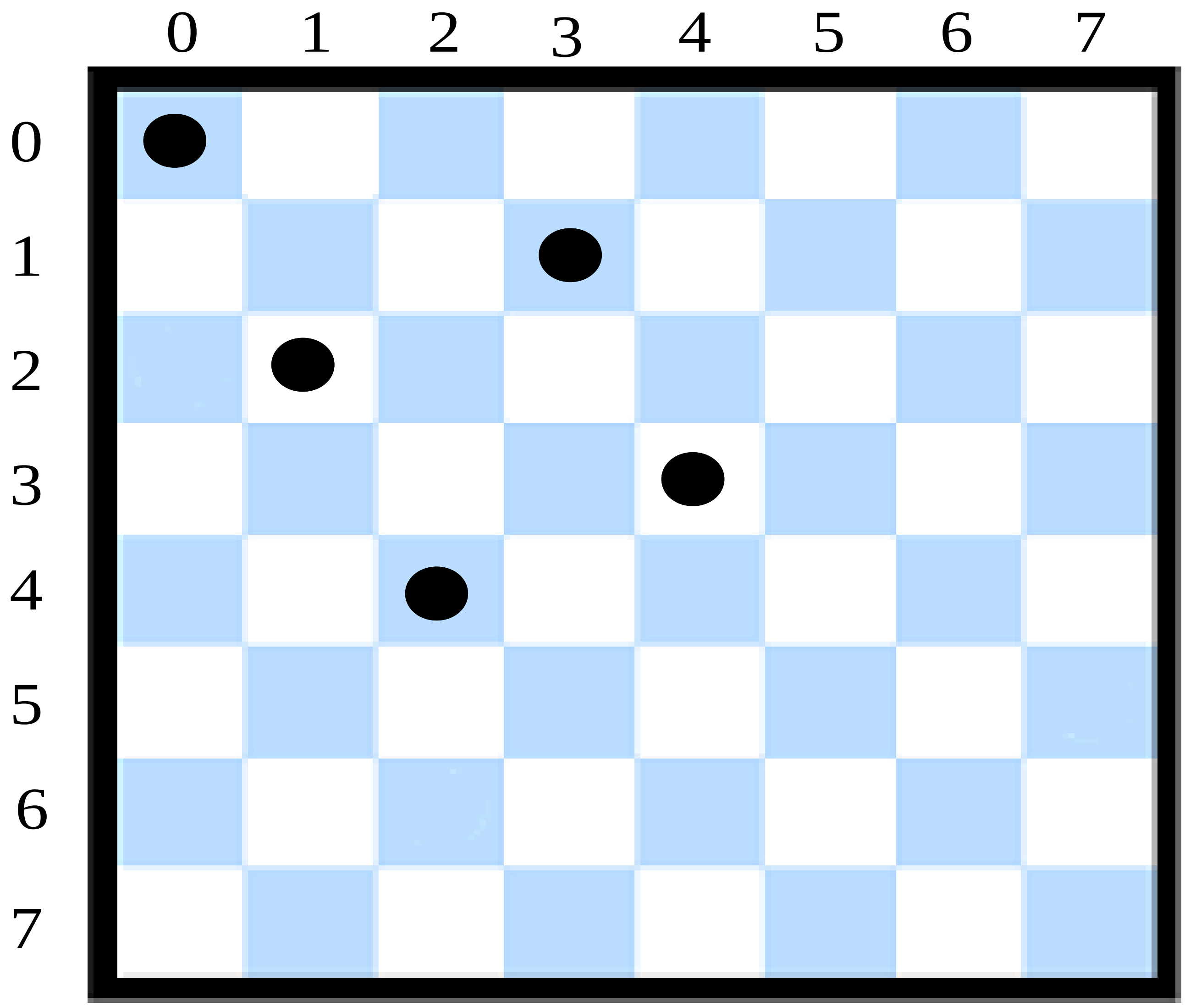
- Many problems can be solved by backtracking.
- Finding your way out of a maze.
- Map coloring
- N-Queens problem.
- https://www.brainmetrix.com/8-queens/
- https://www.cs.usfca.edu/~galles/visualization/RecQueens.html
- http://javaddlib.sourceforge.net/jdd/demos/queens.html
- http://spaz.ca/aaron/SCS/queens/

N-Queens Problem

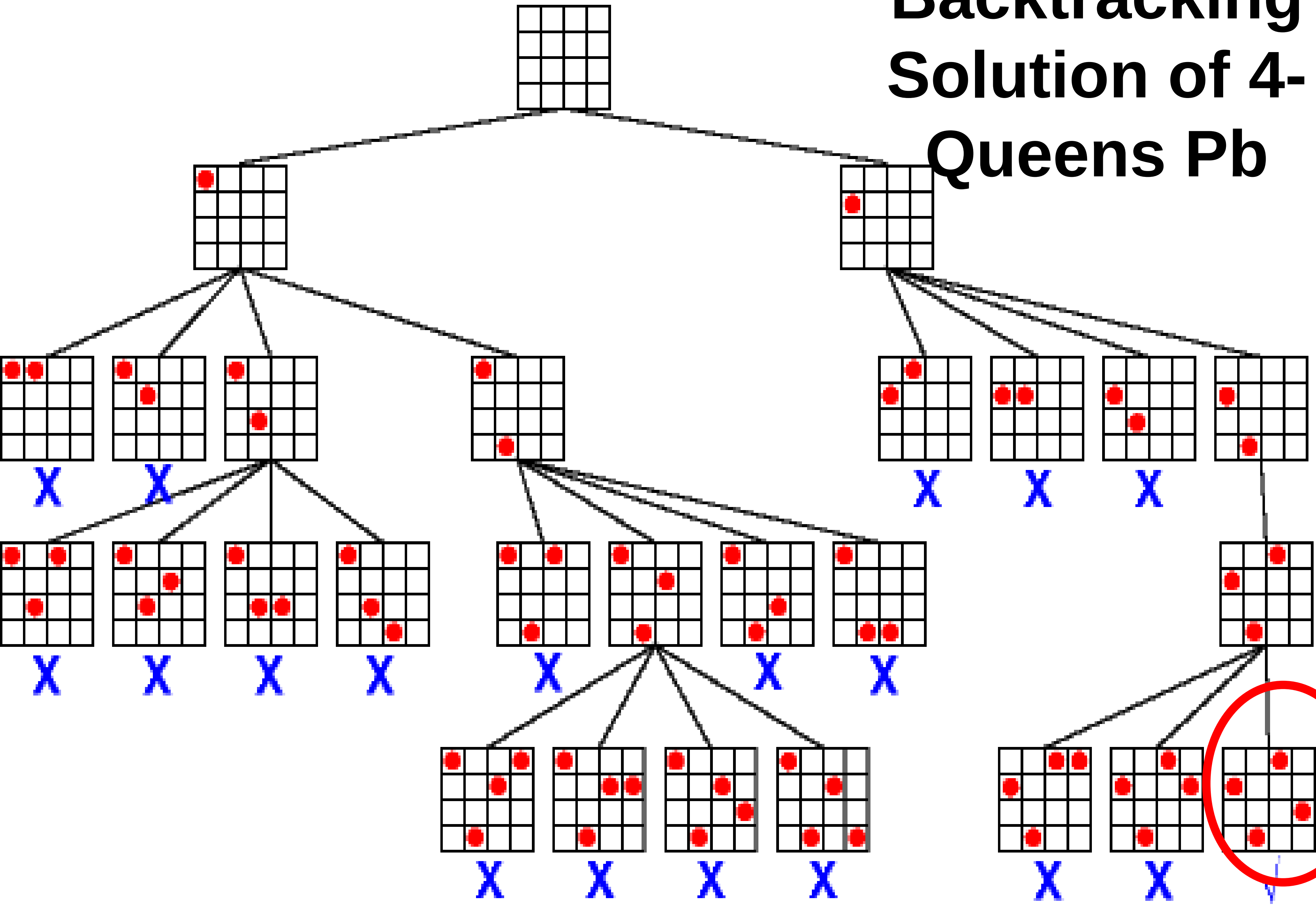
- Try to place N queens on an N * N board such that none of the queens can attack another queen.
- Queens are jealous from each other. A queen does not want to see another queen.
- Remember that queens can see / move horizontally, vertically or diagonally any distance.
- Let’s consider the 8 queen example…

The 8-Queens Example




0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7

Backtracking Solution of 4-Queens Pb



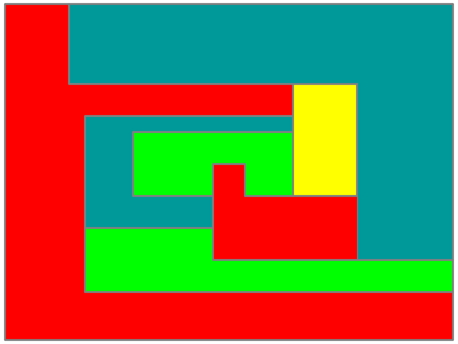
Solving N-Queens Pb

1. Start in the leftmost column
2. If all queens are placed return true
3. Try all rows in the current column. For each row
   A. If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
   B. If placing the queen in [row, column] leads to a solution return true.
   C. If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
4. If all rows have been tried and nothing worked return false to trigger backtracking.

- https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/

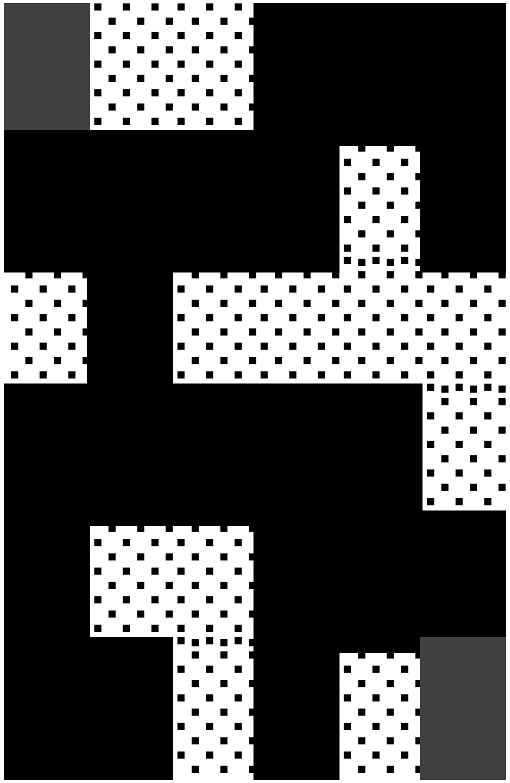
Backtracking Problem
Map Coloring

- You wish to color a map with not more than four colors
  - Red, yellow, green, blue
- Adjacent countries must be in different colors



- You don't have enough information to choose colors
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many coloring problems can be solved with backtracking

Solving a Maze



If destination is reached print the solution matrix
Else

A. Mark current cell in solution matrix as 1.
B. Move forward in the horizontal direction and recursively check if this move leads to a solution.
C. If the move chosen in the above step doesn't lead to a solution then move down and check if this move leads to a solution.
D. If none of the above solutions works then unmark this cell as 0 (BACKTRACK) and return false.

https://www.geeksforgeeks.org/rat-in-a-maze-backtracking-2/

Backtracking and Games

- Backtracking is used in 2-players strategy games.
- The first player has several choices for an initial move.
- Depending on which move is chosen the second player then has a particular set of responses.
- This process continues until the end of the game.
- The different possible positions at each turn in the game form a branching structure in which each option opens up more and more possibilities.

Backtracking and Games

- If you the computer to play the game then:
  - Get the computer follow all the branches in the list of possibilities.
  - Before making a move the computer would try every possible choice.
  - For each of these choices it would then try to determine what its opponent’s response would be by trying every possibility.
  - For each branch, it will calculate a utility function.
  - Too **expensive** to implement in practice.
  - It is actually exhaustive recursion.

Game of Nim

- The game starts with a pile of 13 coins on the table. (or any number)
- The 1st player removes 1, 2 or 3 coins.
- The 2nd player does the same.
- The player who has to remove the last coin loses.

Game of Nim

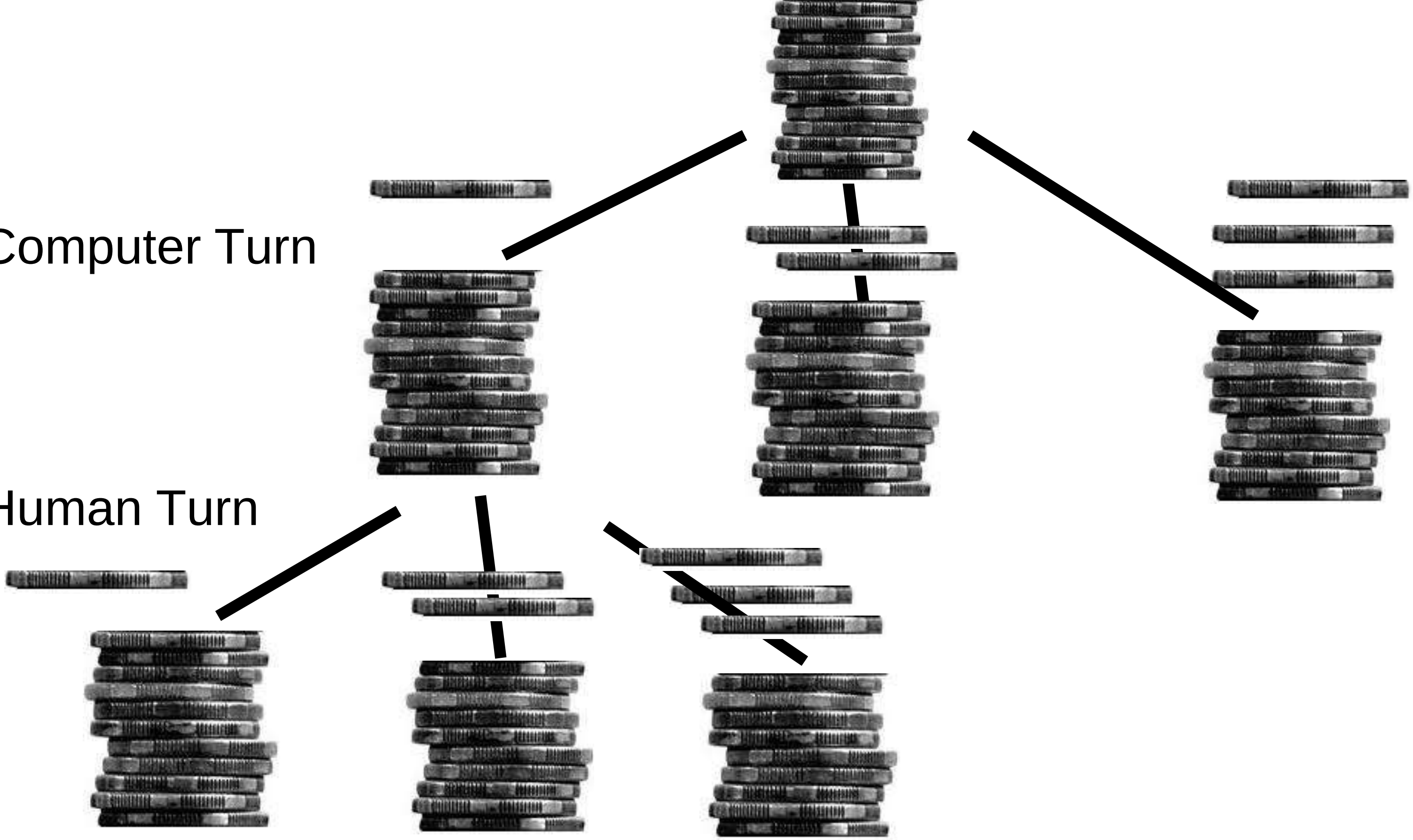
- Working from the end:
  - If you have 1 coin left you are doomed (bad position)
  - If you have 2, 3 or 4 coins you can win.
  - If you have 5 coins you may easily lose (bad position)
  - If you have 6 coins, move one and you can win (good move)
  - ……………
- A move is good
  - If it puts your opponent in bad position
- A position is bad
  - If there are no good moves (mutual recursion)

Game of Nim

```
int FindGoodMove(int nCoins) {
    for (each possible move) {
        Evaluate the position that results
        from making that move. If the
        resulting position is bad
        return that move.
    }
    Return a sentinel value indicating
    that no good move exists.
}
```

Computer Turn

Human Turn



Game of Nim

```
int FindGoodMove(int nCoins) {
  for (int n = 1; n <= MAX_MOVE; n++)
     if (IsBadPosition(nCoins - n))
          return n;
  return NO_GOOD_MOVE;
}

bool IsBadPosition(int nCoins) {
     if (nCoins == 1) return true;
     return FindGoodMove(nCoins) ==
                                   NO_GOOD_MOVE;
}
```

Design of Nim

- Welcome message
- While (nCoins > 1)
  - Display state
  - If *human's turn*
    - Get user move
  - If *computer's turn*
    - Get best move
  - Update state and decide next turn
- Declare winner

Design of Nim

```
int main() {
    int nCoins = N_COINS;
    playerT whoseTurn = Human;
    GiveInstructions();

    while (nCoins > 1) {
        DisplayState(nCoins, whoseTurn);
        switch (whoseTurn) {
            case Human: n = GetUserMove(nCoins);
                break;
            case Computer: n =
                ChooseComputerMove(nCoins);
                cout << "I take " << n << "." << ;
        }
        UpdateState (nCoinsn, whoseTurn);
    }
    AnnounceWinner(nCoins, whoseTurn);
}
```

Common Concepts

- Game State defines the current the status of the game (board, turne, tc.) at a specific point in time.
- A move is a possible action on the game.

- Generally, you may capture these in types **`stateT`** and **`moveT`**.

`main` Program for *a Game*

```
int main() {
    GiveInstructions();
    stateT state = NewGame();
    moveT move;
    while (!GameIsOver(state)) {
        DisplayGame(state);
        switch (WhoseTurn(state)) {
          case Human:
            move = GetUserMove(state);
            break;
          case Computer:
            move = ChooseComputerMove(state);
            DisplayMove(move);
            break;
        }
        MakeMove(state, move);
    }
    AnnounceResult(state);
    return 0;
}
```