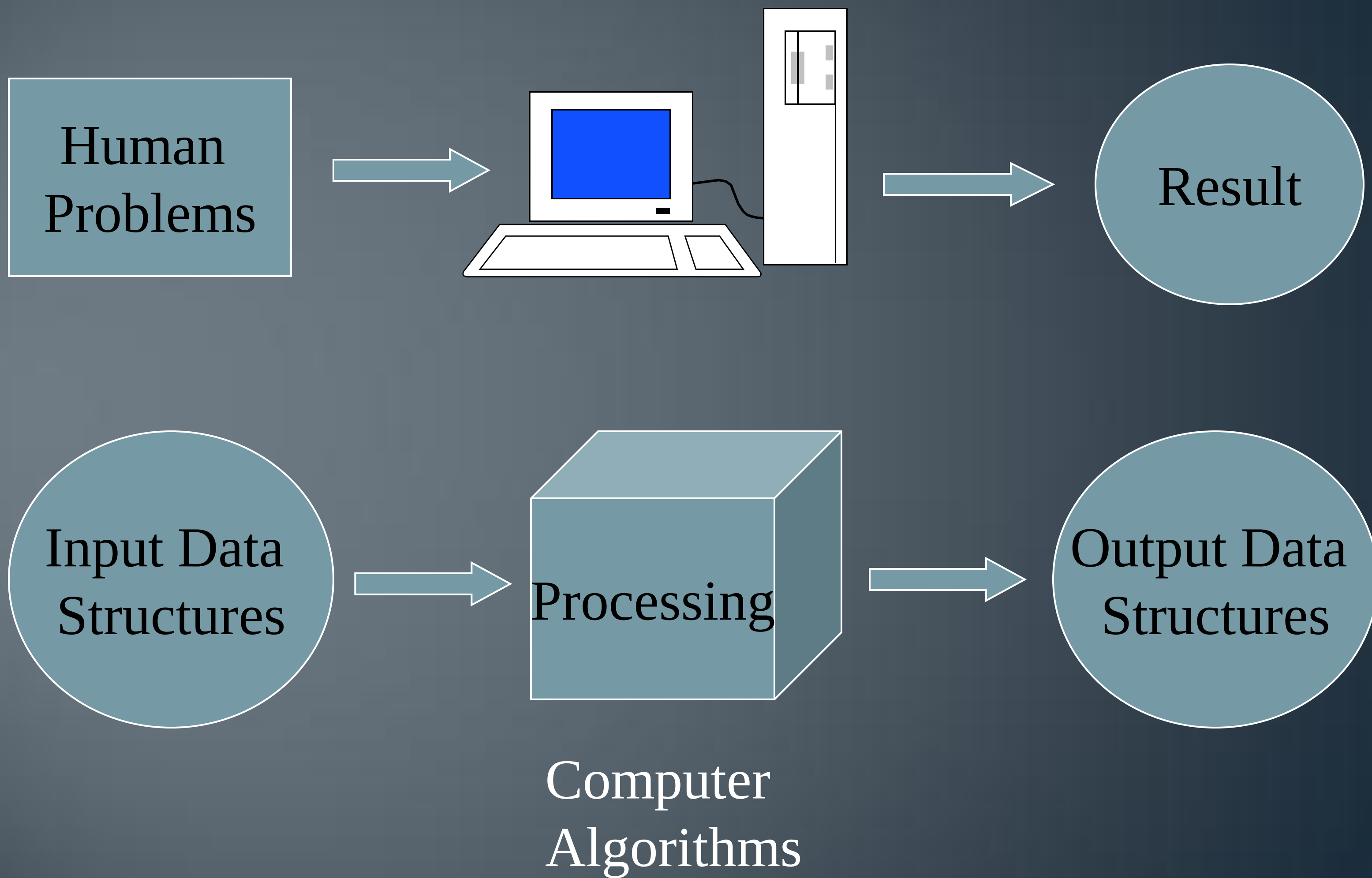
Course Name: Analysis and Design of Algorithms

Topics to be covered

- What is Backtracking
- Sum of Subsets
- Graph Coloring
- Hamiltonian Circuits
- Other Problems

Algorithm Design



Algorithm Design …

For a problem? What is an Optimal Solution?

- **Minimum CPU time**
- **Minimum memory**

Example: Given 4 numbers, sort it to nonincreasing order.

Method 1: Sequential comparison

1. Find the largest (3 comparisons)
2. Find the second largest (2 comparisons)
3. Find the third largest (1 comparisons)
4. Find the fourth largest

A total of 6 comparisons

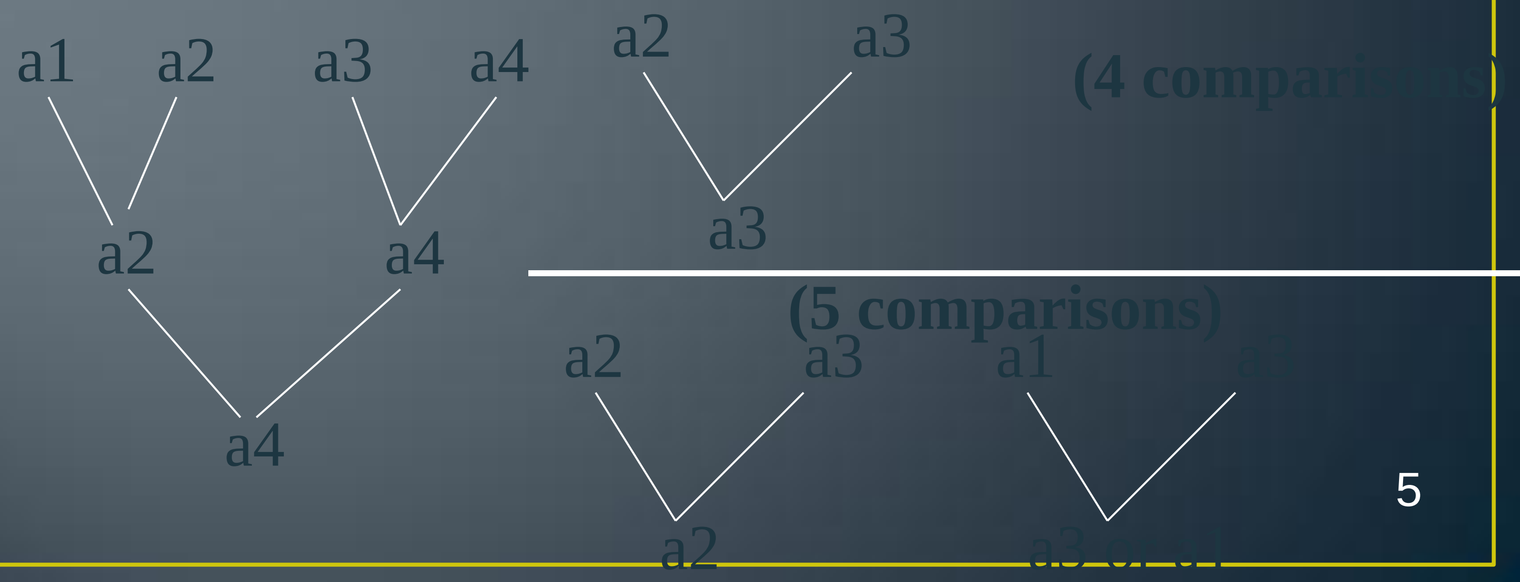
Algorithm Design …

For a problem? What is an Optimal Solution?

- **Minimum CPU time**
- **Minimum memory**

Example: Given 4 numbers, sort it to nonincreasing order.

Method 2: Somewhat clever method



Backtracking Problems

- Find your way through the well-known maze of hedges by Hampton Court Palace in England? Until you reached a dead end.
- 0-1 Knapsack problem – exponential time complexity.
- N-Queens problem.

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"

Introduction

- Backtracking is used to solve problems in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion.
- Backtracking is a modified depth-first search of a tree.
- Backtracking involves only a tree search.
- Backtracking is the procedure whereby, after determining that a node can lead to nothing but dead nodes, we go back ("backtrack") to the node's parent and proceed with the search on the next child.

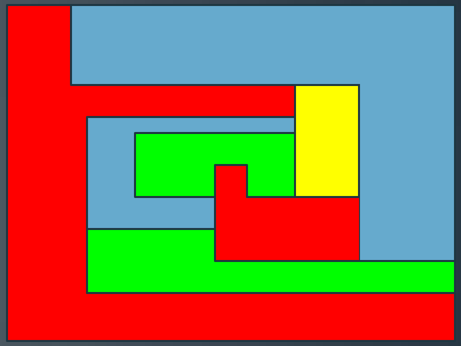
Introduction …

- We call a node nonpromising if when visiting the node we determine that it cannot possibly lead to a solution. Otherwise, we call it promising.
- In summary, backtracking consists of
  - Doing a depth-first search of a state space tree,
  - Checking whether each node is promising, and, if it is nonpromising, backtracking to the node's parent.
- This is called pruning the state space tree, and the subtree consisting of the visited nodes is called the pruned state space tree.

Solving a maze

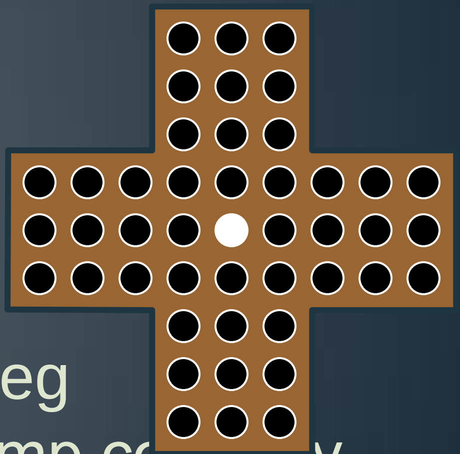
- Given a maze, find a path from start to finish
- At each intersection, you have to decide between three or fewer choices:
  - Go straight
  - Go left
  - Go right
- You don’t have enough information to choose correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many types of maze problem can be solved with backtracking

Coloring a map



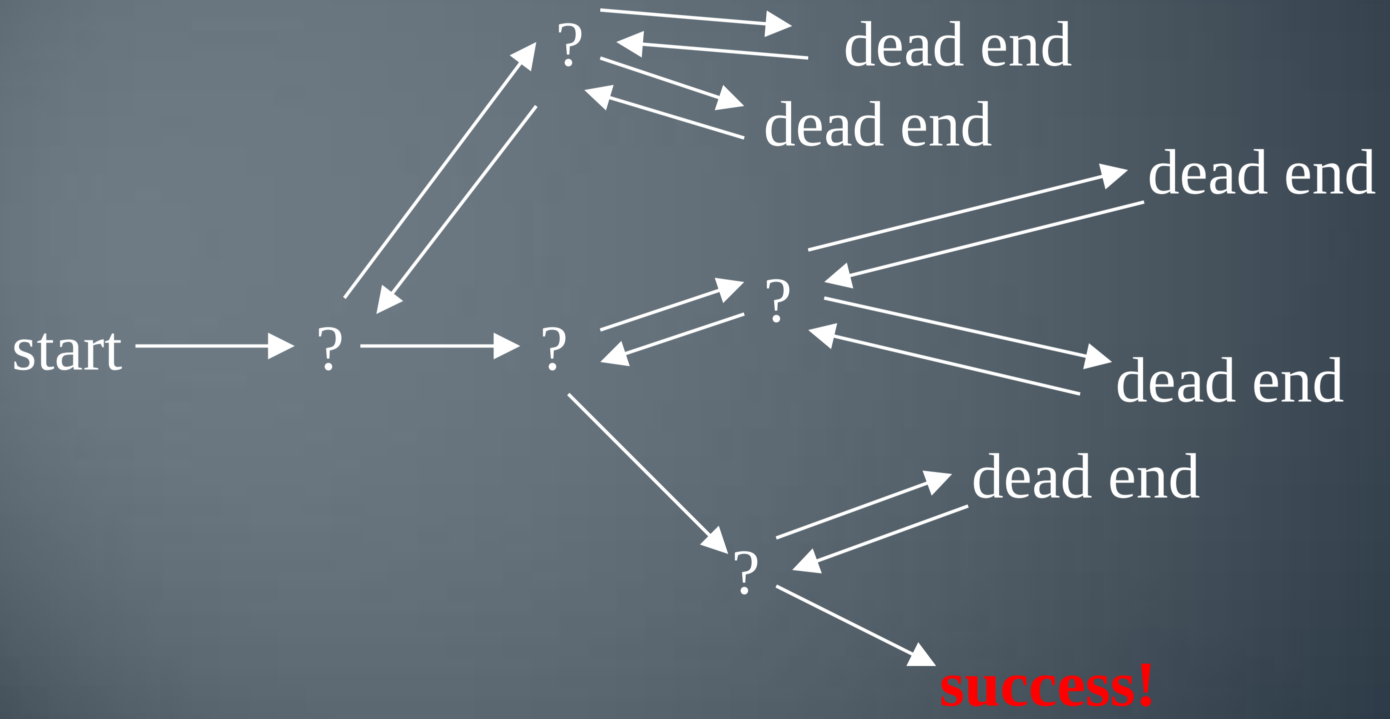
- You wish to color a map with not more than four colors
  - red, yellow, green, blue
- Adjacent countries must be in different colors
- You don't have enough information to choose colors
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many coloring problems can be solved with backtracking

Solving a puzzle



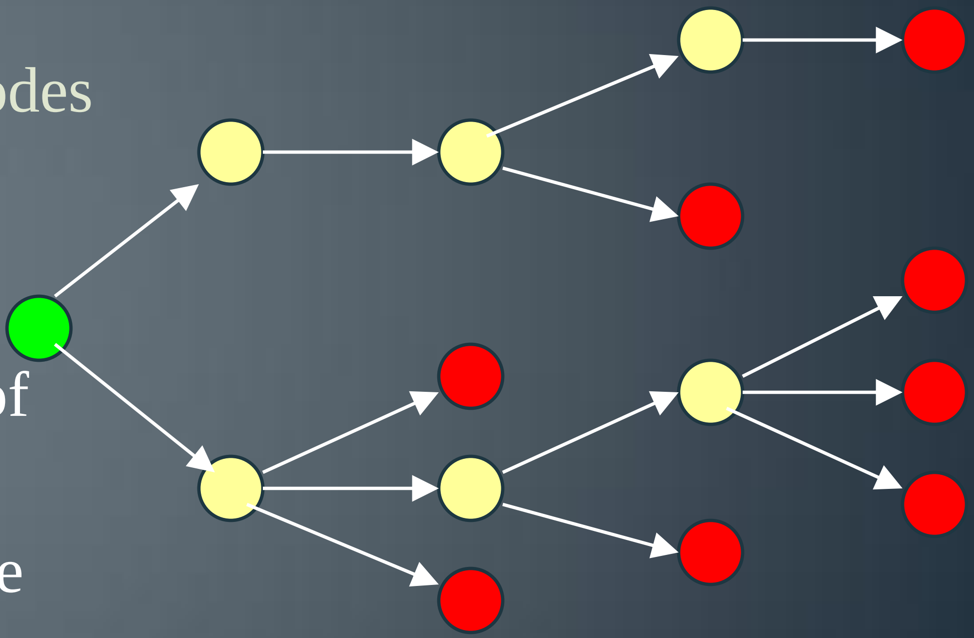
- In this puzzle, all holes but one are filled with white pegs
- You can jump over one peg with another
- Jumped pegs are removed
- The object is to remove all but the last peg
- You don't have enough information to jump correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many kinds of puzzle can be solved with backtracking

Backtracking (animation)



Terminology I

A tree is composed of nodes



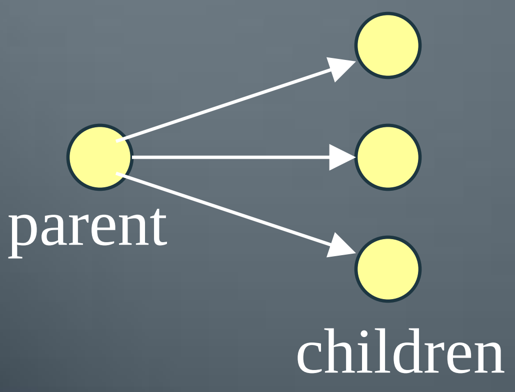
There are three kinds of nodes:

- The (one) root node
- Internal nodes
- Leaf nodes

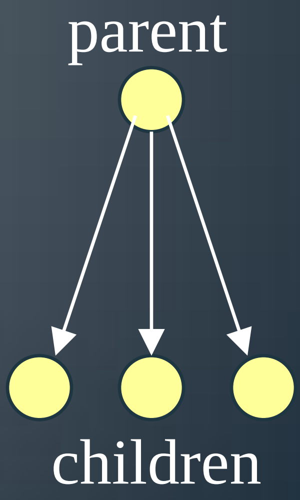
Backtracking can be thought of as searching a tree for a particular “goal” leaf node

Terminology II

- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent



Usually, however, we draw our trees *downward*, with the root at the top



Real and virtual trees

- There is a type of data structure called a tree
  - But we are **not** using it here
- If we diagram the sequence of choices we make, the diagram looks like a tree
  - In fact, we did just this a couple of slides ago
  - Our backtracking algorithm “sweeps out a tree” in “problem space”

The backtracking algorithm

- Backtracking is really quite simple--we "explore" each node, as follows:
- To "explore" node N:
  1. If N is a goal node, return "success"
  2. If N is a leaf node, return "failure"
  3. For each child C of N,
     - 3.1. Explore C
       - 3.1.1. If C was successful, return "success"
  4. Return "failure"

Sum-of-Subsets problem

- Recall the thief and the 0-1 Knapsack problem.
- The goal is to maximize the total value of the stolen items while not making the total weight exceed W.
- If we sort the weights in nondecreasing order before doing the search, there is an obvious sign telling us that a node is nonpromising.

Sum-of-Subsets problem …

- Let *total* be the total weight of the remaining weights, a node at the ith level is nonpromising if

  weight + *total* > W

Example

- Say that our weight values are 5, 3, 2, 4, 1
- W is 8
- We could have
  - 5 + 3
  - 5 + 2 + 1
  - 4 + 3 + 1
- We want to find a sequence of values that satisfies the criteria of adding up to W

Tree Space

- Visualize a tree in which the children of the root indicate whether or not value has been picked (left is picked, right is not picked)
- Sort the values in non-decreasing order so the lightest value left is next on list
- Weight is the sum of the weights that have been included at level i
- Let *weight* be the sum of the weights that have been included up to a node at level i. Then, a node at the ith level is nonpromising if

  $weight + w_{i+1} > W$

Sum-of-Subsets problem …

- Example: Show the pruned state space tree when backtracking is used with $n = 4$, $W = 13$, and $w_1 = 3$, $w_2 = 4$, $w_3 = 5$, and $w_4 = 6$. Identify those nonpromising nodes.

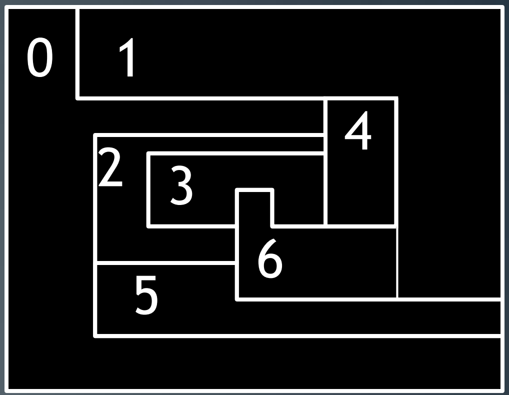
Full example: Map coloring

- The Four Color Theorem states that any map on a plane can be colored with no more than four colors, so that no two countries with a common border are the same color
- For most maps, finding a legal coloring is easy
- For some maps, it can be fairly difficult to find a legal coloring
- We will develop a complete Java program to solve this problem

Data structures

- We need a data structure that is easy to work with, and supports:
  - Setting a color for each country
  - For each country, finding all adjacent countries
- We can do this with two arrays
  - An array of “colors”, where `countryColor[i]` is the color of the i^{th} country
  - A ragged array of adjacent countries, where `map[i][j]` is the j^{th} country adjacent to country i
    - Example: `map[5][3]==8` means the 3^{th} country adjacent to country 5 is country 8

Creating the map




```
int map[][];

void createMap() {
    map = new int[7][];
    map[0] = new int[] { 1, 4, 2, 5 };
    map[1] = new int[] { 0, 4, 6, 5 };
    map[2] = new int[] { 0, 4, 3, 6, 5 };
    map[3] = new int[] { 2, 4, 6 };
    map[4] = new int[] { 0, 1, 6, 3, 2 };
    map[5] = new int[] { 2, 6, 1, 0 };
    map[6] = new int[] { 2, 3, 4, 1, 5 };
}
```

Setting the initial colors

```
static final int NONE = 0;
static final int RED = 1;
static final int YELLOW = 2;
static final int GREEN = 3;
static final int BLUE = 4;

int mapColors[] = { NONE, NONE, NONE, NONE,
                    NONE, NONE, NONE };
```

The main program

(The name of the enclosing class is ColoredMap)

```
public static void main(String args[]) {
  ColoredMap m = new ColoredMap();
  m.createMap();
  boolean result = m.explore(0, RED);
  System.out.println(result);
  m.printMap();
}
```

The backtracking method

```
boolean explore(int country, int color) {
   if (country >= map.length) return true;
   if (okToColor(country, color)) {
      mapColors[country] = color;
      for (int i = RED; i <= BLUE; i++) {
         if (explore(country + 1, i)) return true;
      }
   }
   return false;
}
```

Checking if a color can be used

```
boolean okToColor(int country, int color) {
   for (int i = 0; i < map[country].length; i++) {
      int ithAdjCountry = map[country][i];
      if (mapColors[ithAdjCountry] == color) {
         return false;
      }
   }
   return true;
}
```

Printing the results

```
void printMap() {
    for (int i = 0; i < mapColors.length; i++) {
        System.out.print("map[" + i + "] is ");
        switch (mapColors[i]) {
            case NONE:    System.out.println("none");   break;
            case RED:     System.out.println("red");    break;
            case YELLOW: System.out.println("yellow"); break;
            case GREEN:   System.out.println("green");  break;
            case BLUE:    System.out.println("blue");    break;
        }
    }
}
```

Recap

- We went through all the countries recursively, starting with country zero
- At each country we had to decide a color
  - It had to be different from all adjacent countries
  - If we could not find a legal color, we reported failure
  - If we could find a color, we used it and recurred with the next country
  - If we ran out of countries (colored them all), we reported success
- When we returned from the topmost call, we were done