CS 106X, Lecture 12
Recursive Backtracking 3

reading:

Programming Abstractions in C++, Chapter 9

Plan For Today

- **Recap:** Selection Problems
- **Recap:** Types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*
- Arms-length recursion
- Unit Testing

Plan For Today

- **Recap:** Selection Problems
- **Recap:** Types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*
- Arms-length recursion
- Unit Testing

Grab the Popcorn 

- Write a function **`printMovieOrders`** that accepts a `Vector` of movie names (Strings) as a parameter and outputs all possible orders in which we could watch those movies. The arrangements may be output in any order.
  - Example: if v contains `{"Up", "Argo", "Black Panther", "Inside Out"}`, your function outputs permutations like:

{“Up”, “Argo”, “Black Panther”, “Inside Out”}	“Inside Out”, “Argo”, “Black Panther”, “Up”}
{“Up”, “Argo”, “Inside Out”, “Black Panther”}	“Inside Out”, “Argo”, “Up”, “Black Panther”}
{“Up”, “Black Panther”, “Argo”, “Inside Out”}	“Inside Out”, “Black Panther”, “Up”, “Argo”}
{“Up”, “Black Panther”, “Inside Out”, “Argo”}	“Inside Out”, “Black Panther”, “Argo”, “Up”}
...	...

Movie Permutations

```
void printMovieOrders(Vector<string>& allMovies,
                      Vector<string>& chosen) {
    if (allMovies.isEmpty()) {
        cout << chosen << endl;   // base case
    } else {
        for (int i = 0; i < allMovies.size(); i++) {
            string currMovie = allMovies[i];
            allMovies.remove(i);
            chosen.add(currMovie);                  // choose
            printMovieOrders(allMovies, chosen);    // explore
            chosen.remove(chosen.size() - 1);     // un-choose
            allMovies.insert(i, currMovie);
        }
    }
}
```

Movie Permutations

```
// Outputs all permutations of the given list of movies.
void printMovieOrders(const Vector<string>& allMovies) {
    Vector<string> chosen;
    Vector<string> moviesCopy = allMovies;
    printMovieOrders(moviesCopy, chosen);
}
```

Find/Print All Solutions

- **Base case:** is it a valid solution? If so, print it (or add to set of found solutions). If not valid, don't.
- **Recursive step:** Make all recursive calls. If you are returning a collection, add each recursive result to the collection.

Plan For Today

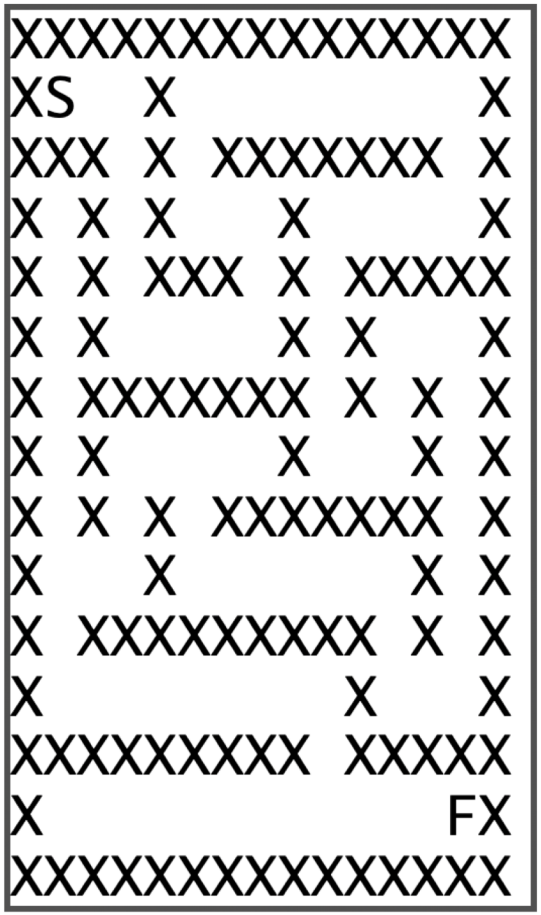
- **Recap:** Selection Problems
- **Recap:** Types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*
- Arms-length recursion
- Unit Testing

Maze Solving

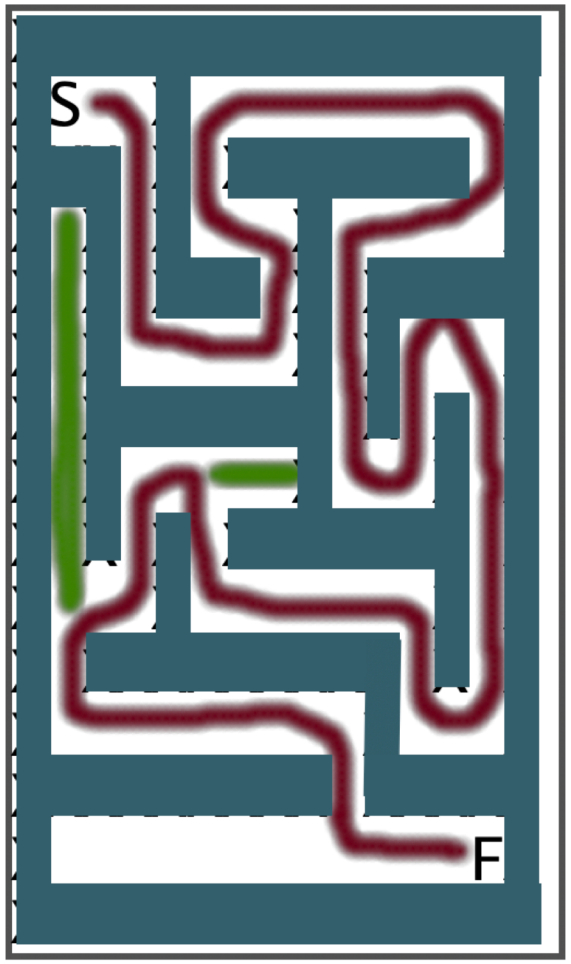
https://www.reddit.com/r/funny/comments/lejyk/billy_mays_maize_maze/

Maze Solving

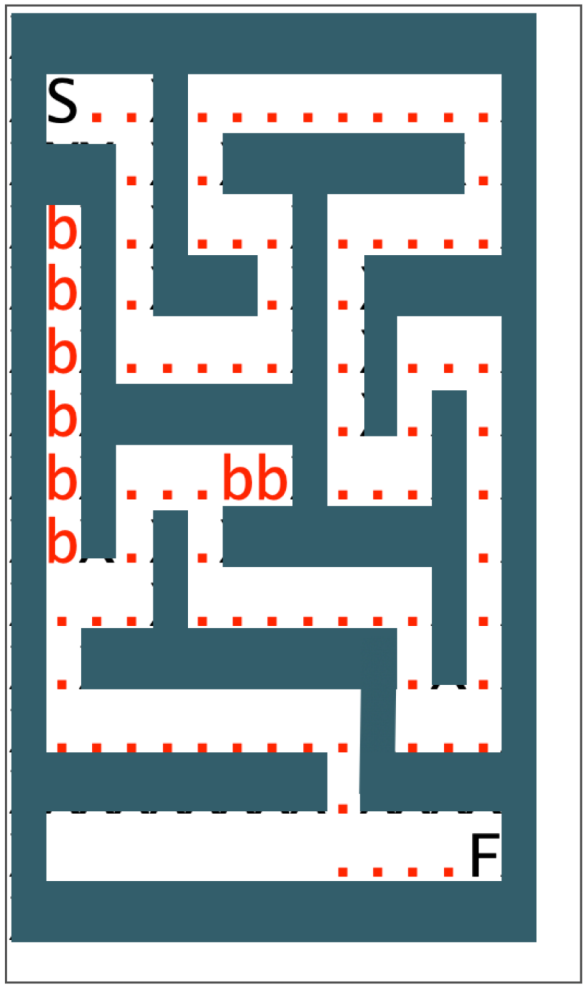
- The code for today's class includes a text-based recursive maze creator and solver.
- The mazes look like the one to the right
  - There is a Start (marked with an "S") and a Finish (marked with an "F").
  - The Xs represent walls, and the spaces represent paths to walk through the maze.



Maze Solving



- The program will put dots in the correct positions.
- But, it will also put lowercase b's when it goes in the wrong direction and has to backtrack.



Maze Solving

```
bool solveMazeRecursive(int row, int col, Grid<int>& maze) {
    if (maze[row][col] == 'X') return false;
    if (maze[row][col] == '.') return false;
    if (maze[row][col] == 'F') return true;

    maze[row][col] = '.';

    // north
    if (solveMazeRecursive(row-1,col,maze)) return true;
    // east
    if (solveMazeRecursive(row,col+1,maze)) return true;
    // south
    if (solveMazeRecursive(row+1,col,maze)) return true;
    // west
    if (solveMazeRecursive(row,col-1,maze)) return true;

    maze[row][col] = 'b';
    return false;
}
```

Maze Solving

```
bool solveMazeRecursive(int row, int col, Grid<int>& maze) {
    if (maze[row][col] == 'X') return false;
    if (maze[row][col] == '.') return false;
    if (maze[row][col] == 'F') return true;

    maze[row][col] = '.';

    // north
    return solveMazeRecursive(row-1,col,maze)); // bad idea ☹
    // east
    if (solveMazeRecursive(row,col+1,maze)) return true;
    // south
    if (solveMazeRecursive(row+1,col,maze)) return true;
    // west
    if (solveMazeRecursive(row,col-1,maze)) return true;

    maze[row][col] = 'b';
    return false;
}
```

Maze Solving

```
bool solveMazeRecursive(int row, int col, Grid<int>& maze) {
    if (maze[row][col] == 'X') return false;
    if (maze[row][col] == '.') return false;
    if (maze[row][col] == 'F') return true;

    maze[row][col] = '.';

    // north
    return solveMazeRecursive(row-1,col,maze)); // bad idea ☹
    // east
    if (solveMazeRecursive(row,col+1,maze)) return true;
    // south
    if (solveMazeRecursive(row+1,col,maze)) return true;
    //
    if

    ma
    re
}
```

Just because one recursive exploration fails doesn't mean we should give up hope! Have faith that another exploration might find a solution, and only return false if you've tried everything.

Maze Counting

https://www.reddit.com/r/funny/comments/lejyk/billy_mays_maize_maze/

Count Solutions

Given a maze represented as a Grid<bool> (true if you can go somewhere, false if it's a wall), and a start and end location, count the number of unique paths from the start to the end. We assume the maze is bordered by walls. (This problem is useful for e.g. ensuring there is only 1 solution!)

```
int countMazeSolutionsHelper(Grid<bool>& maze,
                             int startRow,
                             int startCol, int endRow,
                             int endCol);
```

Count Solutions

```
int countMazeSolutionsHelper(Grid<bool>& maze, int startRow,
                             int startCol, int endRow, int endCol) {
    if (!maze[startRow][startCol]) {
        return 0;
    }
    if (startRow == endRow && startCol == endCol) {
        return 1; //reached our goal
    }

    maze[startRow][startCol] = false; // choose
    int numSolutions = countMazeSolutionsHelper(maze, startRow + 1,
        startCol, endRow, endCol);
    numSolutions += countMazeSolutionsHelper(maze, startRow - 1,
        startCol, endRow, endCol);
    numSolutions += countMazeSolutionsHelper(maze, startRow,
        startCol + 1, endRow, endCol);
    numSolutions += countMazeSolutionsHelper(maze, startRow,
        startCol - 1, endRow, endCol);
    maze[startRow][startCol] = true; // unchoose
    return numSolutions;
}
```

Count Solutions

- Base case: is it a valid solution? If so, return 1. Otherwise, return 0.
- Recursive step: return the sum of all the recursive calls.

This approach is useful because sometimes we want to make sure that there is *exactly* one solution. For instance, a maze!

A Startling Observation

What nine-letter word can be reduced to a single-letter word one letter at a time by removing letters, leaving it a legal word at each step?

A Startling Observation

S	T	A	R	T	L	I	N	G

A Startling Observation

I

Are there other words with this property?

Shrinkable Words

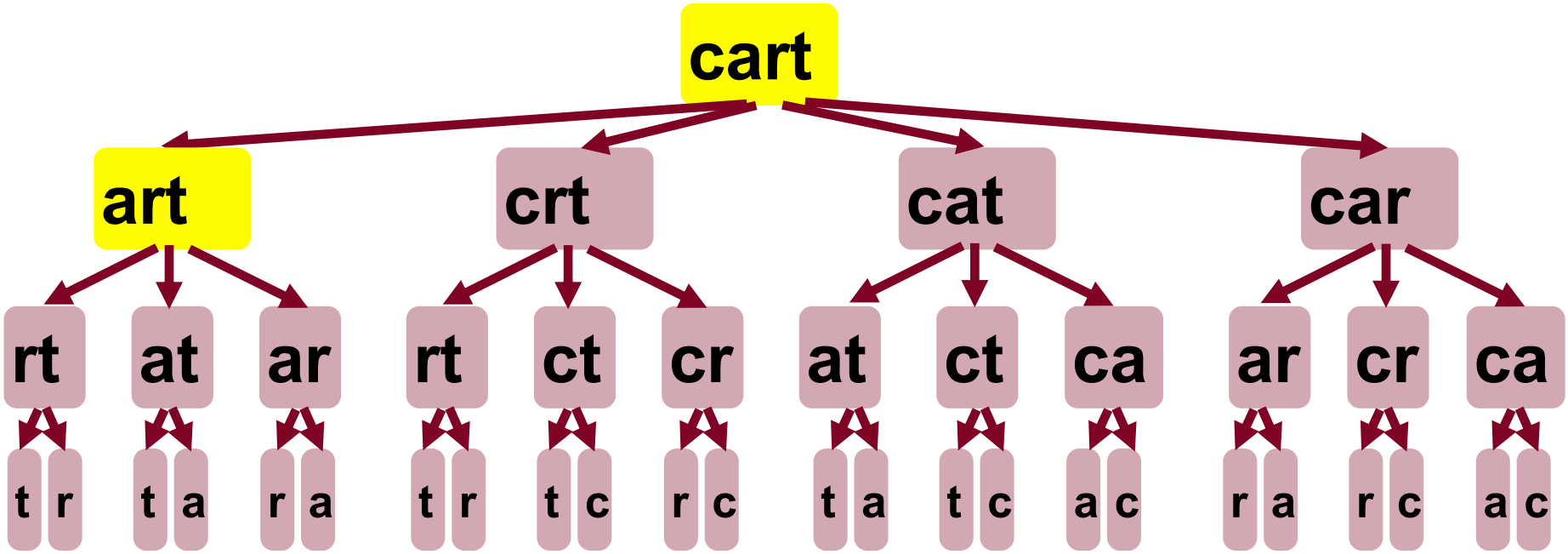
- Let's define a **shrinkable word** as a word that can be reduced down to one letter by removing one character at a time, leaving a word at each step.
- Given an English dictionary, how can we determine whether a word is shrinkable?

isShrinkable

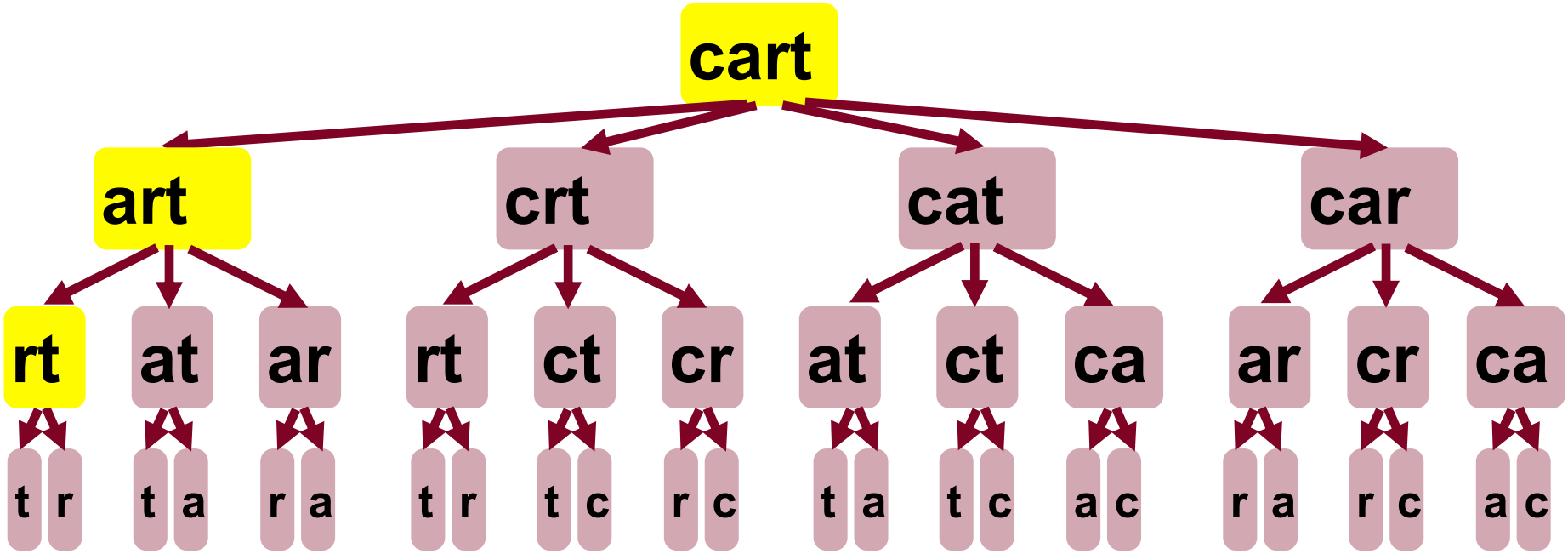
```
bool isShrinkable(const string& word, const Lexicon& english) {
    if (!english.contains(word)) return false;
    if (word.length() == 1) return true;

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i+1);
        if (isShrinkable(shrunken, english)) {
            return true;
        }
    }

    return false;
}
```

Does a Solution Exist?

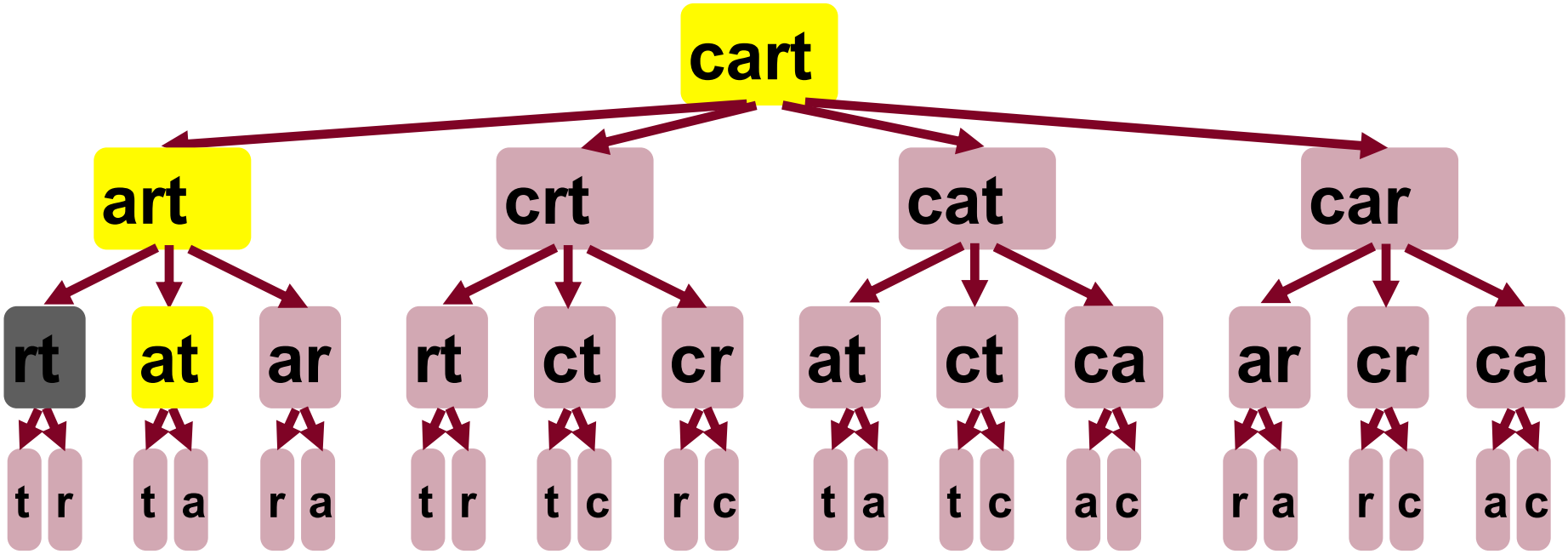
- Should return a boolean
- Base case: validate the solution so far; return **true** if valid
- Recursive step: check all potential choices. If one returns **true**, you return **true**. Otherwise, return **false**.


cart
art
crt
cat
car
rt
at
ar
rt
ct
cr
at
ct
ca
ar
cr
ca
t r
t a
r a
t r
t c
r c
t a
t c
a c
r a
r c
a c

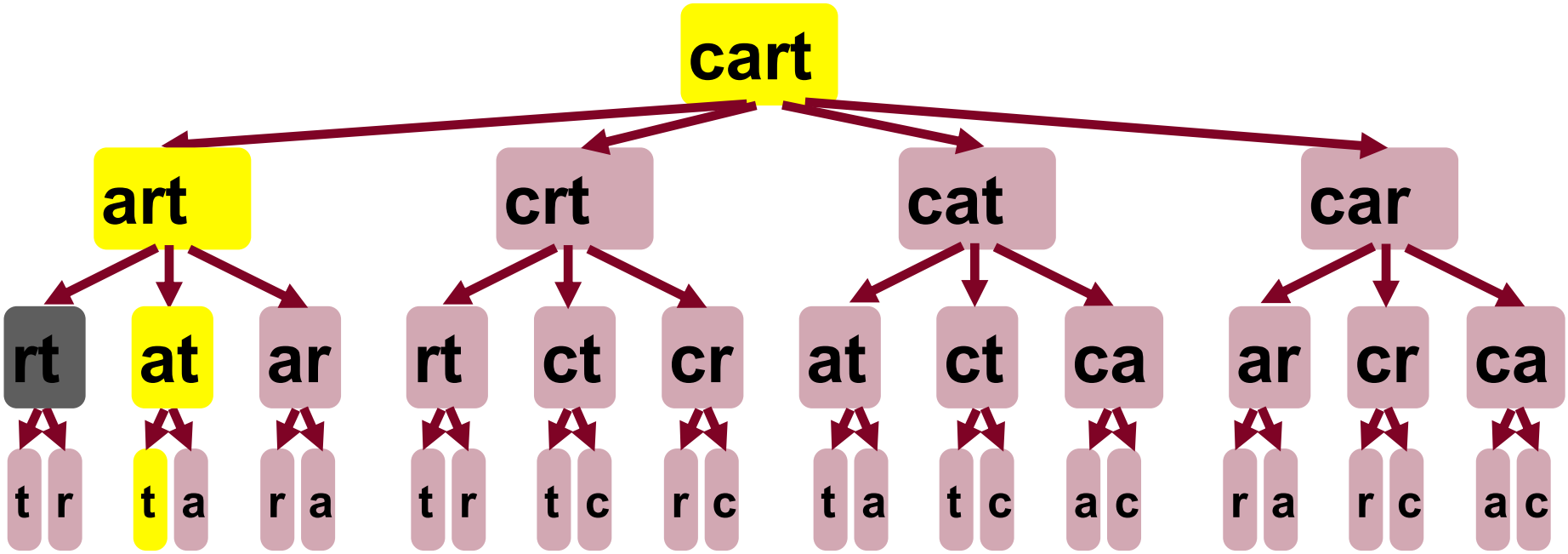
art: is a word


cart
art
crt
cat
car
rt
at
ar
rt
ct
cr
at
ct
ca
ar
cr
ca
t r
t a
r a
t r
t c
r c
t a
t c
a c
r a
r c
a c

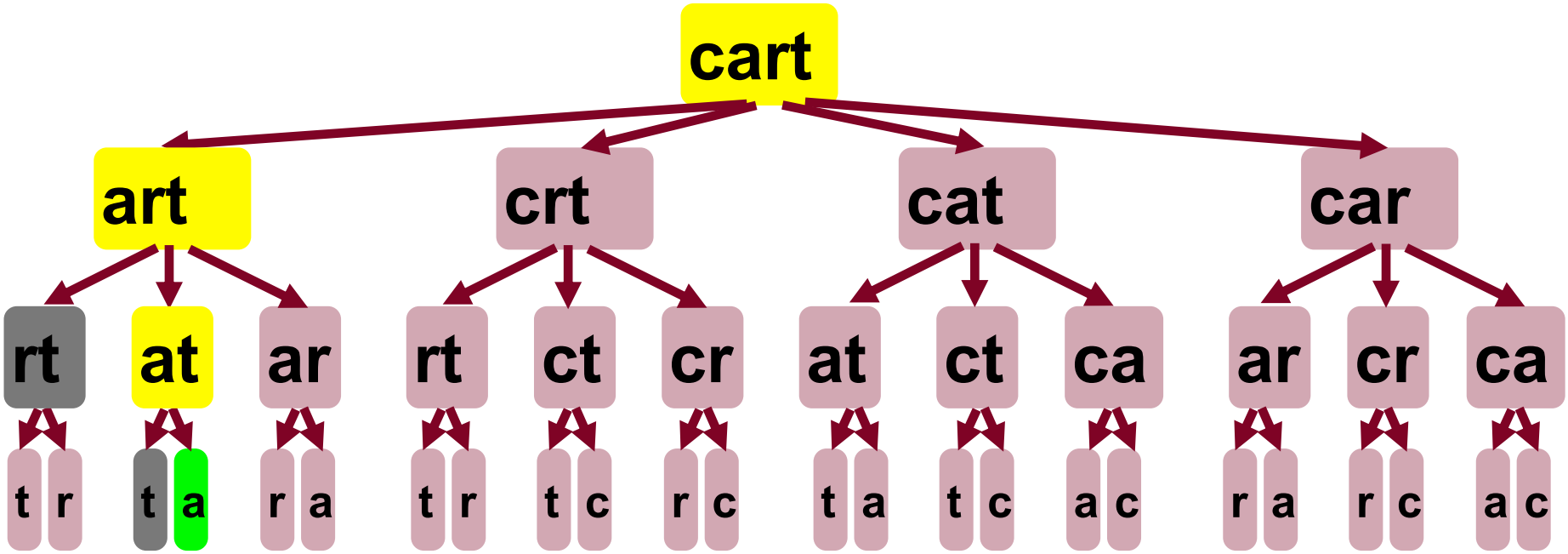
rt: not a word


cart
art
crt
cat
car
rt
at
ar
rt
ct
cr
at
ct
ca
ar
cr
ca
t r
t a
r a
t r
t c
r c
t a
t c
a c
r a
r c
a c

at: is a word

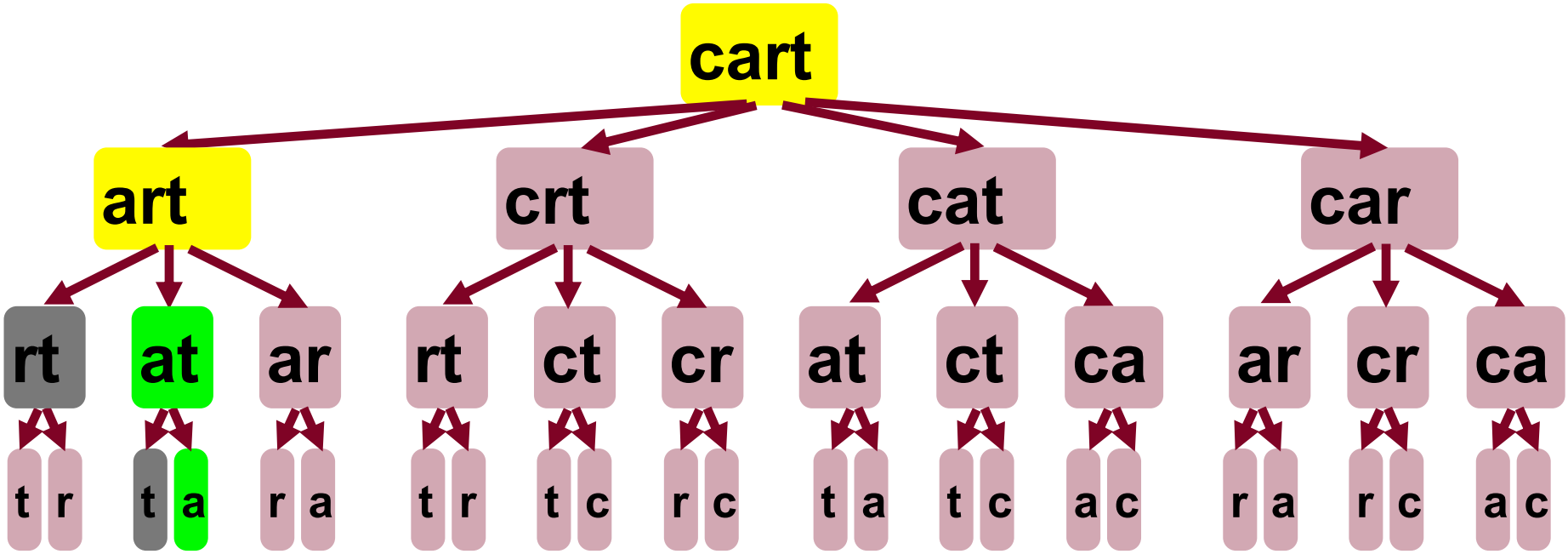

cart
art
crt
cat
car
rt
at
ar
rt
ct
cr
at
ct
ca
ar
cr
ca
t r
t a
r a
t r
t c
r c
t a
t c
a c
r a
r c
a c

t: not a word



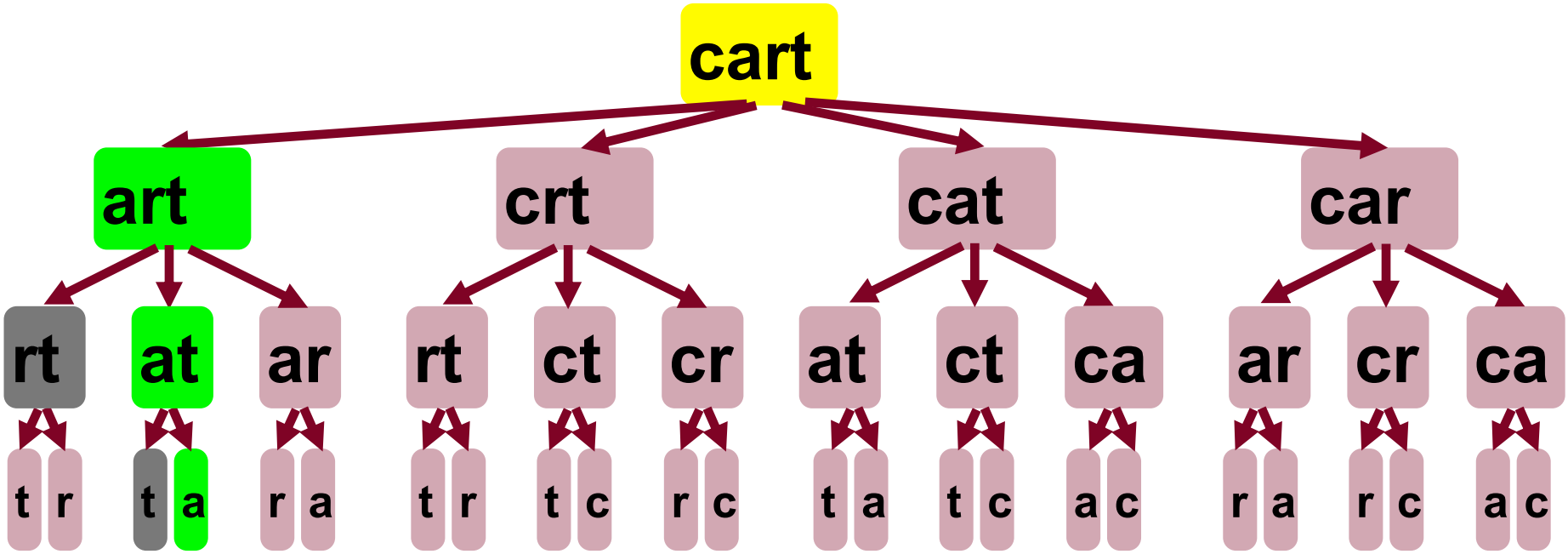
a: is a word

there is a solution!



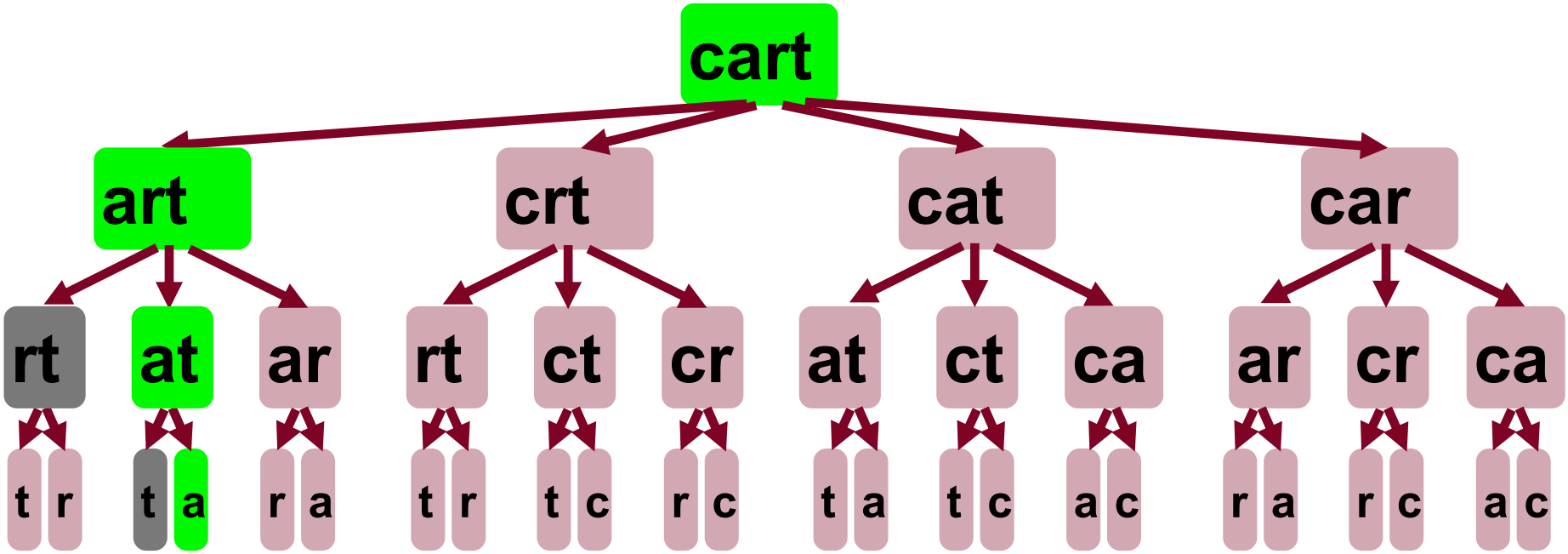
a: is a word

there is a solution!



a: is a word

there is a solution!



a: is a word

there is a solution!

Plan For Today

- **Recap:** Selection Problems
- **Recap:** Types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*
- Arms-length recursion
- Unit Testing

isShrinkable

```
bool isShrinkable(const string& word, const Lexicon& english) {
    if (!english.contains(word)) return false;
    if (word.length() == 1) return true;

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i+1);
        if (isShrinkable(shrunken, english)) {
            return true;
        }
    }

    return false;
}
```

isShrinkable

```
bool isShrinkable(const string& word, const Lexicon& english) {
    if (!english.contains(word)) return false;
    if (word.length() == 1) return true;

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i+1);
        if (isShrinkable(shrunken, english)) {
            return true;
        }
    }

    return false;
}
```

What if we want to actually get back the shrinkable sequence of words?

shrinkableSequence

```
Vector<string> shrinkableSequence(const string& word,
                                  const Lexicon& english) {
    if (!english.contains(word)) return {};
    if (word.length() == 1) return { word };

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i+1);
        Vector<string> words = shrinkableSequence(shrunken, english);
        if (words.size() > 0) {
            words.insert(0, word);
            return words;
        }
    }

    return {};
}
```

shrinkableSequence

```
Vector<string> shrinkableSequence(const string& word,
                                  const Lexicon& english) {
    if (!english.contains(word)) return {};
    if (word.length() == 1) return { word };

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i+1);
        Vector<string> words = shrinkableSequence(shrunken, english);
        if (words.size() > 0) {
            words.insert(0, word);
            return words;
        }
    }
}
```

Here, it's easy for us to check whether this recursive call found a solution. But sometimes it's hard (e.g. trying to solve a Sudoku puzzle; how can you communicate back that it was/wasn't solved?).

shrinkableSequence

```
Vector<string> shrinkableSequence(const string& word,
                                  const Lexicon& english) {
    if (!english.contains(word)) return {};
    if (word.length() == 1) return { word };

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i+1);
        Vector<string> words = shrinkableSequence(shrunken, english);
        if (words.size() > 0) {
            words.insert(0, word);
            return words;
        }
    }

    return {};
}
```

Sometimes, it's easier to pass what we are trying to fill in *by reference*, and return a boolean indicating whether or not we found a solution.

shrinkableSequence

```
Vector<string> shrinkableSequence(const string& word,
                                  const Lexicon& english) {
    Vector<string> sequence;
    shrinkableSequence(word, english, sequence);
    return sequence;
}
```

shrinkableSequence

```
bool shrinkableSequence(const string &word,
                        const Lexicon &english, Vector<string>& words) {
    if (!english.contains(word)) return false;
    if (word.length() == 1) {
        words.insert(0, word);
        return true;
    }

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i+1);
        if (shrinkableSequence(shrunken, english, words)) {
            words.insert(0, word);
            return true;
        }
    }

    return false;
}
```

Find a Solution

- **Base case:** validate the solution so far; return the solution if it's valid, or an empty solution otherwise.
- **Recursive step:** check all potential choices. If one returns a valid solution, return that. Otherwise, return an empty solution.
- Consider passing the solution by reference, and returning a boolean indicating whether a solution was found.

Announcements

- **Homework 4** goes out at the end of the day today, and is due at week from Monday at 11AM.
- **Optional pair assignment**
- Super cool recursive backtracking problems!

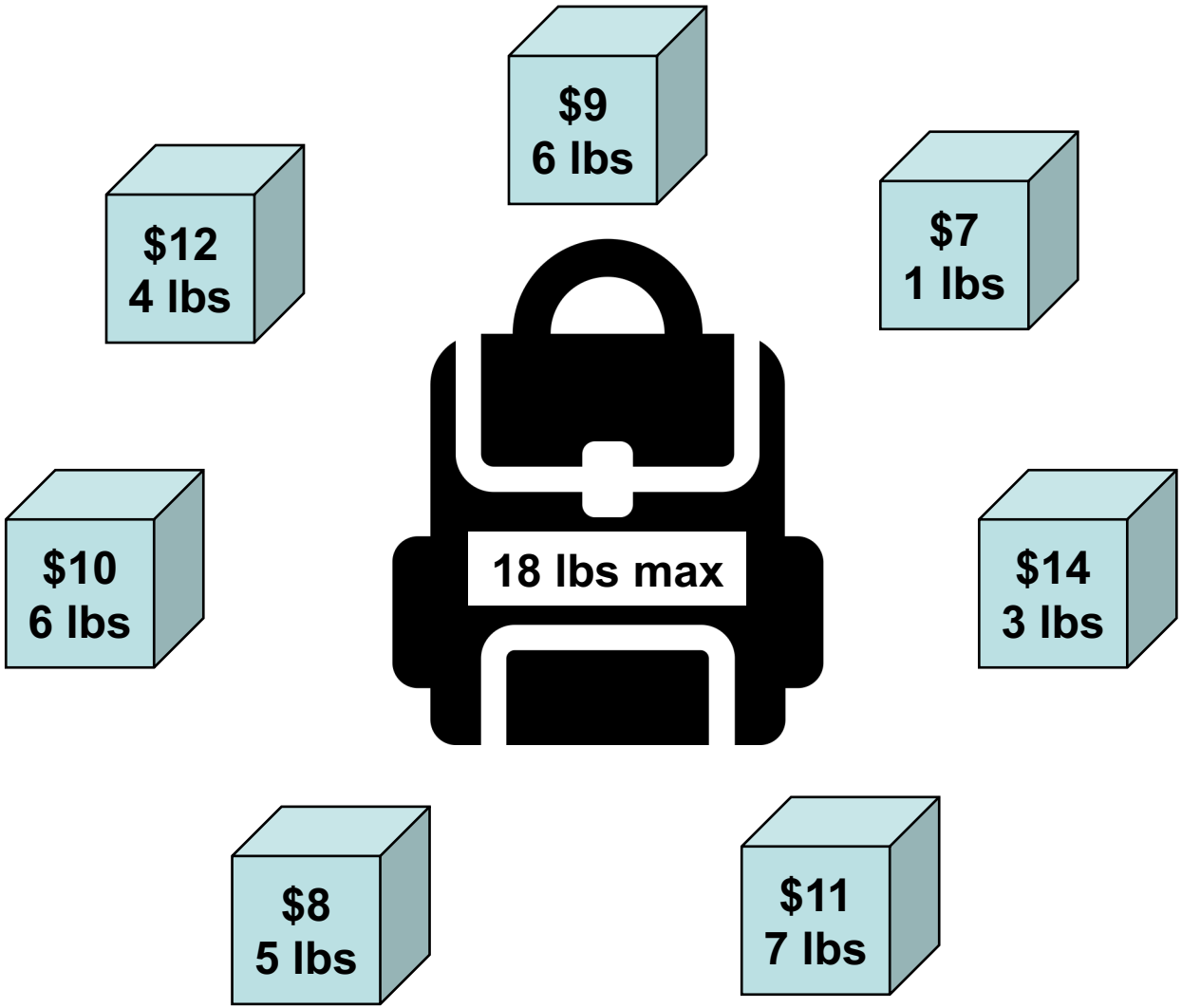
Plan For Today

- **Recap:** Selection Problems
- **Recap:** Types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*
- Arms-length recursion
- Unit Testing

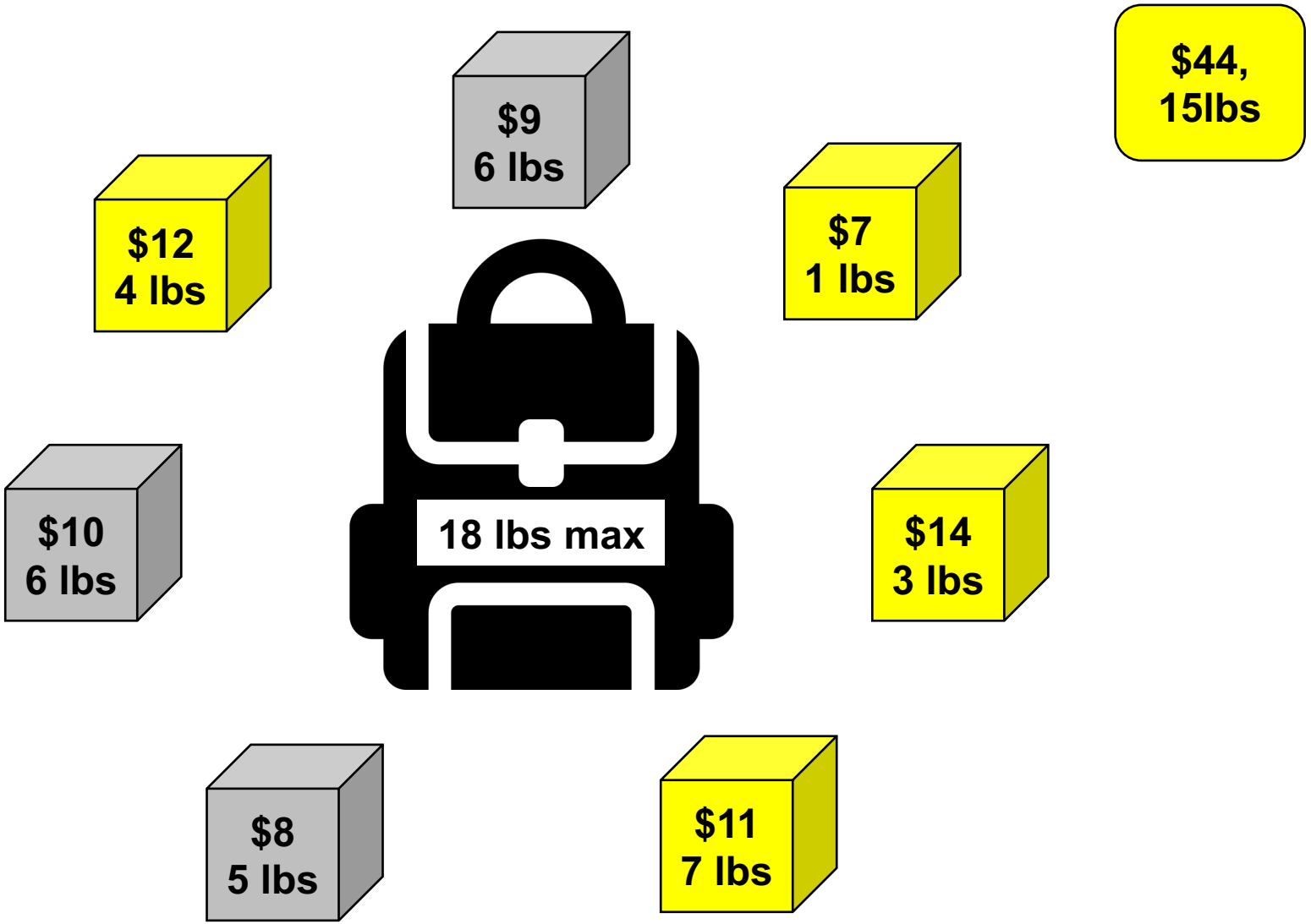
The Knapsack Problem

If I have a backpack that can hold at most ***W*** pounds, and a collection of objects with weights and values, which objects should I put in my bag to maximize its total value?

The Knapsack Problem



The Knapsack Problem



The Knapsack Problem

For this problem, let's represent an object with the following struct:

```
struct WeightedObject {
    int weight; // assume greater than or equal to 0
    int value;  // assume greater than or equal to 0
};
```

Let's write the function:

```
int fillKnapsack(const Vector<WeightedObject>& objects,
                 int targetWeight)
```

that considers all possible combinations of objects (such that the sum of their weights is less than or equal to targetWeight) and returns the maximum possible sum of object values.

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 Knapsack Problem

```
int fillKnapsackHelper(Vector<WeightedObject> &objects,
                       int targetWeight) {
    if (targetWeight == 0 || objects.isEmpty()) return 0;

    WeightedObject originalObject = objects.removeBack();

    // Attempt 1: try excluding this from the bag
    int best = fillKnapsackHelper(objects, targetWeight);

    // Attempt 2: try including this in the bag, if it fits
    if (originalObject.weight <= targetWeight) {
        int with = originalObject.value +
                   fillKnapsackHelper(objects,
                   targetWeight - originalObject.weight);
        best = max(best, with);
    }

    objects.add(originalObject);
    return best;
}
```

The Knapsack Problem

```
// Returns the max value of all items at most the given weight
int fillKnapsack(const Vector<WeightedObject>& objects,
                 int targetWeight) {
    Vector<WeightedObject> objectsCopy = objects;
    return fillKnapsackHelper(objectsCopy, targetWeight);
}
```

Find the Best Solution

- Base case: is it a valid solution? If so, return it. Otherwise, return a default/empty solution.
- Recursive step: check all potential choices, then output the "best" of all of them.

不同choices之间出现的结果要比较找best

The Knapsack Problem

- This problem is a key algorithm in computer science, particularly in resource allocation and other types of problems.
- Take CS161 for more details!

Plan For Today

- **Recap:** Selection Problems
- **Recap:** Types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*
- Arms-length recursion
- Unit Testing

Revisiting the Maze

```
int countMazeSolutionsHelper(Grid<bool>& maze,
                             int startRow,
                             int startCol, int endRow,
                             int endCol);
```

Running Into Walls

```
int countMazeSolutionsHelper(Grid<bool>& maze, int startRow,
                             int startCol, int endRow, int endCol) {
    if (!maze[startRow][startCol]) {
        return 0;
    }
    if (startRow == endRow && startCol == endCol) {
        return 1; //reached our goal
    }

    maze[startRow][startCol] = false; // choose
    int numSolutions = countMazeSolutionsHelper(maze, startRow + 1,
        startCol, endRow, endCol);
    numSolutions += countMazeSolutionsHelper(maze, startRow - 1,
        startCol, endRow, endCol);
    numSolutions += countMazeSolutionsHelper(maze, startRow,
        startCol + 1, endRow, endCol);
    numSolutions += countMazeSolutionsHelper(maze, startRow,
        startCol - 1, endRow, endCol);
    maze[startRow][startCol] = true; // unchoose
    return numSolutions;
}
```

Running Into Walls

```
int countMazeSolutionsHelper(Grid<bool>& maze, int startRow,
                             int startCol, int endRow, int endCol) {
    if (startRow == endRow && startCol == endCol) {
        return 1; //reached our goal
    }

    maze[startRow][startCol] = false; // choose
    int numSolutions = 0;
    if (maze[startRow + 1][startCol]) {
        numSolutions += countMazeSolutionsHelper(maze, startRow + 1,
                                                 startCol, endRow, endCol);
    }
    if (maze[startRow - 1][startCol]) {
        numSolutions += countMazeSolutionsHelper(maze, startRow - 1,
                                                 startCol, endRow, endCol);
    }
    ...

    maze[startRow][startCol] = true; // unchoose
    return numSolutions;
}
```

"Arm's Length" Recursion

- Arm's length recursion: a poor style where unnecessary tests are performed before performing recursive calls
- Typically, the tests try to avoid making a call into what would otherwise be a base case
- Can lead to **functionality bugs** as well as **less readable code**
- Applies to all recursive code but **especially backtracking**

Running Into Walls

```
int countMazeSolutionsHelper(Grid<bool>& maze, int startRow,
                             int startCol, int endRow, int endCol) {
    if (startRow == endRow && startCol == endCol) {
        return 1; //reached our goal
    }

    maze[startRow][startCol] = false; // choose
    int numSolutions = 0;
    if (maze[startRow + 1][startCol]) {
        numSolutions += countMazeSolutionsHelper(maze, startRow + 1,
                                                 startCol, endRow, endCol);
    }
    if (maze[startRow - 1][startCol]) {
        numSolutions += countMazeSolutionsHelper(maze, startRow - 1,
                                                 startCol, endRow, endCol);
    }
    ..

    maz
    return numSolutions;
}
```

This implementation no longer handles the case where the start position is in an invalid location!

isShrinkable

```
bool isShrinkable(const string& word, const Lexicon& english) {
    if (!english.contains(word)) return false;
    if (word.length() == 1) return true;

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i+1);
        if (isShrinkable(shrunken, english)) {
            return true;
        }
    }

    return false;
}
```

isShrinkable

```
bool isShrinkable(const string& word, const Lexicon& english) {
    if (!english.contains(word)) return false;

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i+1);
        if (shrunken.length() == 1 ||
            isShrinkable(shrunken, english)) {
            return true;
        }
    }

    return false;
}
```

isShrinkable

```
bool isShrinkable(const string& word, const Lexicon& english) {
    if (!english.contains(word)) return false;

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i+1);
        if (shrunken.length() == 1 ||
            isShrinkable(shrunken, english)) {
            return true;
        }
    }

    return false;
}
```

This implementation no longer handles the case where the word is length 1!

Plan For Today

- **Recap:** Selection Problems
- **Recap:** Types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*
- Arms-length recursion
- Unit Testing

Unit Testing

Unit Testing

- **Unit Testing** is a method for testing small pieces or ”units” of source code in a larger piece of software.
- Each test is usually represented as a single function.
- **Key idea:** each test should examine one portion of functionality that is as narrow and isolated as possible.
- Each test has a way of indicating pass or failure.
- **Benefits:**
  - Limits your code to only what is necessary
  - Finds bugs early
  - Preserves functionality when code is changed

Unit Testing

- **expect()** ; displays error when condition inside this statement is false, but other tests continue to run.

Unit Testing

- **Example:** We are given a black-box function that takes a vector of ints and a number. It is supposed to return the largest sum we can make using elements in the vector without exceeding the number.

Vector	Target Number	Return
{}	1	0
{10, 2, 5, 1}	9	8
{1, 2, 3, 4}	20	10

Recap

- **Recap:** Selection Problems
- **Recap:** Types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*
- Arms-length recursion
- Unit Testing