Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski

Today's topics:

- Recursion ~~Week~~ Fortnight continues!
- Today:
  - More *recursive backtracking* code examples:
    - Gift card spending optimization
    - Maze solving

Code Example #1

GIFT CARD SPENDING OPTIMIZATION

Gift card spending optimization

- You've been given a gift card for your birthday, yay!
- The store has a rule that you must use it in one trip, and any unused balance is forfeited
- You'll be given:
  - Set<int> itemsForSale: A set of prices of items for sale (assume only one of each item is in stock)
  - int giftCardAmt: The amount of the gift card
- Can you find a collection of items to buy that will sum to EXACTLY the amount on the gift card??
- Return:
  - bool: true if you can find such a collection, otherwise false

Gift card spending optimization

- You've been given a gift card for your birthday, yay!
- The store has a rule that you must use it in one trip, and any unused balance is forfeited
- You'll be given:
  - `int giftCardAmt`: The amount of the gift card
  - `Set<int> itemsForSale`: A set of prices of items for sale (assume only one of each item is in stock)
- **Task:** Can you find a collection of items to buy that will sum to EXACTLY the amount on the gift card?
- Return:
  - `bool`: true if you can find such a collection, otherwise false

Your Turn:

Help me write some test cases for this function. Come up with at least one basic correctness test, and a couple tricky/edge cases. **Submit yours at pollev.com/cs106b.** One test case per submission, you may submit multiple times.

Format example:

```
4, {1, 2, 5} = false
```

Backtracking template

Bookmark this slide!

bool backtrackingRecursiveFunction*(args)* {

- Base case test for success: return true
- Base case test for failure: return false
- Loop over several options for "what to do next":
  1. Tentatively "**choose**" one option
  2. if ("**explore**" with recursive call returns true) return true
  3. else That tentative idea didn't work, so "**un-choose**" that option,
     but don't return false yet!--let the loop explore the other options before giving up!
- None of the options we tried in the loop worked, so return false

}

Backtracking template: applied to Gift Card problem

bool backtrackingRecursiveFunction*(args)* {

What is success for this problem?

- Base case test for success: return true
- Base case test for failure: return false
- Loop over several options for "what to do next":
  1. Tentatively "**choose**" one option
  2. if ("**explore**" with recursive call returns true) return true
  3. else That tentative idea didn't work, so "**un-choose**" that option,
     but don't return false yet!--let the loop explore the other options before giving up!
- None of the options we tried in the loop worked, so return false

}

Backtracking template: applied to Gift Card problem

bool backtrackingRecursiveFunction*(args)* {

- Base case test for success: return true
- Base case test for failure: return false
- Loop over several options for "what to do next":
  1. Tentatively "**choose**" one option
  2. if ("**explore**" with recursive call returns true) return true
  3. else That tentative idea didn't work, so "**un-choose**" that option,
     but don't return false yet!--let the loop explore the other options before giving up!
- None of the options we tried in the loop worked, so return false

}

Exactly $0 left on card

Backtracking template: applied to Gift Card problem

bool backtrackingRecursiveFunction*(args)* {

- Base case test for success: return true
- Base case test for failure: return false
- Loop over several options for "what to do next":
  1. Tentatively "**choose**" one option
  2. if ("**explore**" with recursive call returns true) return true
  3. else That tentative idea didn't work, so "**un-choose**" that option,
     but don't return false yet!--let the loop explore the other options before giving up!
- None of the options we tried in the loop worked, so return false

}

Exactly $0 left on card

What is failure for this problem?

Backtracking template: applied to Gift Card problem

bool backtrackingRecursiveFunction*(args)* {

- Base case test for success: return true
  - Exactly $0 left on card
- Base case test for failure: return false
  - Overspend/negative balance, or no items left to choose.
- Loop over several options for "what to do next":
  1. Tentatively "**choose**" one option
  2. if ("**explore**" with recursive call returns true) return true
  3. else That tentative idea didn't work, so "**un-choose**" that option,
     but don't return false yet!--let the loop explore the other options before giving up!
- None of the options we tried in the loop worked, so return false

}

Backtracking template: applied to Gift Card problem

bool backtrackingRecursiveFunction*(args)* {

- Base case test for success: return true
- Base case test for failure: return false
- Loop over several options for “what to
  1. Tentatively “**choose**” one option
  2. if (“**explore**” with recursive call returns true) return true
  3. else That tentative idea didn’t work, so “**un-choose**” that option,
     but don’t return false yet!--let the loop explore the other options before giving up!
- None of the options we tried in the loop worked, so return false

}

Exactly $0 left on card

Overspend/negative balance, or no items left to choose.

What is “one step” for this problem?

What is “one step” in the Gift Card problem?

- We can imagine lining up all the items for sale, and our task is basically to make a binary yes/no decision for purchasing each item
  - The yes’es and no’s can come in any combination, we have to find a combination that sums to our gift card amount

Items:

$1

Y/N: ___

$5

Y/N: ___

$3

Y/N: ___

$2

Y/N: ___

$10

Y/N: ___

What is "one step" in the Gift Card problem?

- We can imagine lining up all the items for sale, and our task is basically to make a binary yes/no decision for purchasing each item
  - The yes'es and no's can come in any combination, we have to find a combination that sums to our gift card amount

Items:

$1	$5	$3	$2	$10
Y/N: **Y**	Y/N: ___	Y/N: ___	Y/N: ___	Y/N: ___

One step/decision

Delegate the rest to recursion

What is “one step” in the Gift Card problem?

- We can imagine lining up all the items for sale, and our task is basically to make a binary yes/no decision for purchasing each item
  - The yes’es and no’s can come in any combination, we have to find a combination that sums to our gift card amount

Items:





$1	$5	$3	$2	$10
Y/N: **Y**	Y/N: ___	Y/N: ___	Y/N: ___	Y/N: ___

One step/decision

If recursion comes back with the answer that no combination works for this set and the remaining funds, reconsider our Y on the banana.

What is “one step” in the Gift Card problem?

- We can imagine lining up all the items for sale, and our task is basically to make a binary yes/no decision for purchasing each item
  - The yes’es and no’s can come in any combination, we have to find a combination that sums to our gift card amount

Items:





$1	$5	$3	$2	$10
Y/N: **Y**	Y/N: ___	Y/N: ___	Y/N: ___	Y/N: ___

One step/decision

Conclusion: one step/decision has two options to “loop” over: Y and N (for one item).

Backtracking template: applied to Gift Card problem

bool backtrackingRecursiveFunction*(args)* {

- › Base case test for success: return true
- › Base case test for failure: return false
- › Loop over several options for "what to
  1. Tentatively "**choose**" one option
  2. if ("**explore**" with recursive call re
  3. else That tentative idea didn't wo
     but don't return false yet!--let the loop
- › None of the options we tried in the loop worked, so return false

}

Exactly $0 left on card

Overspend/negative balance, or no items left to choose.

Taking one item, "loop" over Y and N options for that item (we won't actually loop since Y and N are only two options, a loop is excessive)

If both Y and N options for an item fail, we've exhausted all possibilities, so return false.

Comparing our solution and the design template

```
bool canUseFullGiftCard(int giftCardAmt, Set<int>& itemsForSale, Set<int>& itemsToBuy)
{
    // base case success: card amount is spent down to 0 exactly
    if (giftCardAmt == 0) {
        return true;
    }
    // base case failure: we either overspent, or we need to spend more but there are
    //                    no more items for sale, so we can't succeed
    if (giftCardAmt < 0 || itemsForSale.size() == 0) {
        return false;
    }

    // recursive case: consider 1 next item
    int item = itemsForSale.first();
    Set<int> newItemsForSale = itemsForSale - item;

    // Y: buy this item
    itemsToBuy += item;
    if (canUseFullGiftCard(giftCardAmt - item, newI
        return true;
    }
    itemsToBuy -= item;
    // N: do not buy this item
    if (canUseFullGiftCard(giftCardAmt, newItemsForSale, itemsToBuy)) {
        return true;
    }
    return false;
}
```

bool backtrackingRecursiveFunction(*args*) {

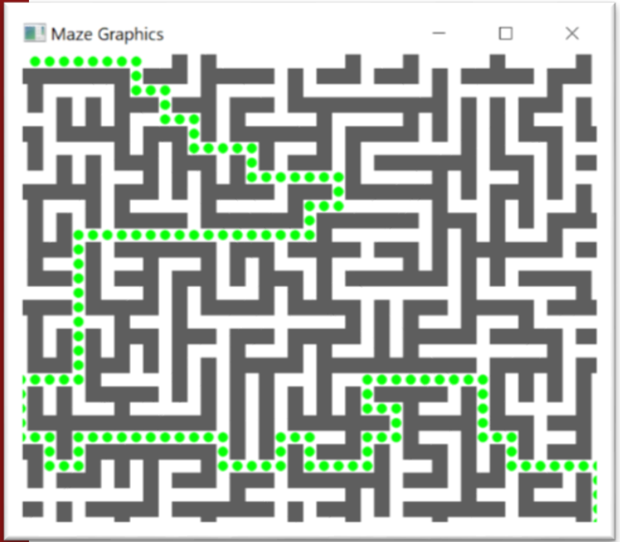
- Base case test for success: return true
- Base case test for failure: return false
- Try both Y and N ~~Loop~~ over several options for "what to do next":
    1. Tentatively "**choose**" one option
    2. if ("**explore**" with recursive call returns true) return true
    3. else That tentative idea didn't work, so "**un-choose**" that option,
       but don't return false yet!--let the loop explore the other options before giv
- None of the options we tried in the loop worked, so return false

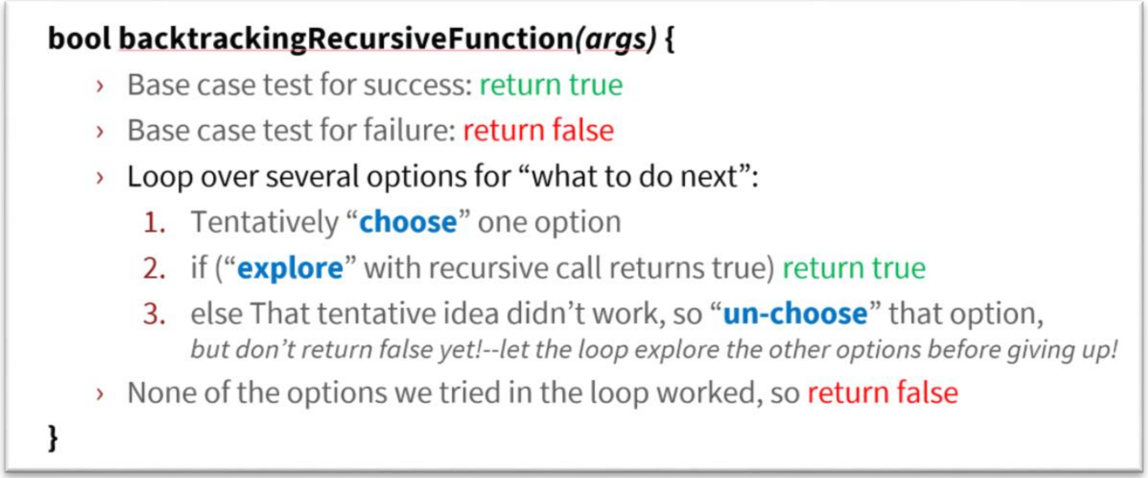
}

Code Example #2

SAY HELLO AGAIN TO YOUR FRIEND, ASSIGNMENT 2 MAZE

Backtracking template: applied to Maze problem





- If at the exit, return true *(no false base case needed)*
- Loop over N, W, E, S directions that are valid moves
  1. Choose: add that move to our path
  2. Recursively explore from there (maybe return true)
  3. Unchoose: remove that move from path
- If no valid move reached end, return false

Comparing our solution and the design template

```
bool solveMazeHelper(Grid<bool>& maze, Stack<GridLocation>& path, GridLocation cur) {
    MazeGraphics::highlightPath(path, "blue");
    GridLocation exitLoc = {maze.numRows() - 1, maze.numCols() - 1};

    // Base case: we are at the exit
    if (cur == exitLoc) {
        MazeGraphics::highlightPath(path, "green");
        return true;
    }

    // Mark that we have visited this place so we don't retrace steps
    maze[cur] = false;

    // We get valid neighbors (as applicable) sorted as: N
    Set<GridLocation> validNeighbors = generateValidMoves(
    for (GridLocation cell : validNeighbors) {
        // Choose
        path.push(cell);
        // Explore
        if (solveMazeHelper(maze, path, cell)) {
            return true;
        }
        // Unchoose (undo)
        path.pop();
    }

    // Unmark
    maze[cur] = true;

    return false;
}
```

bool backtrackingRecursiveFunction(*args*) {

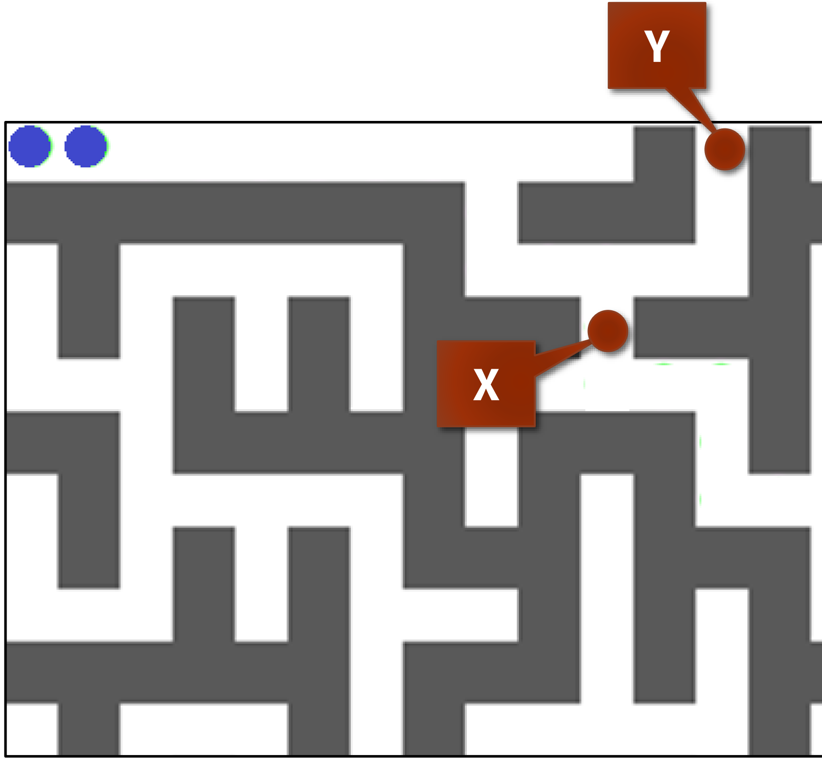
- Base case test for success: return true
- ~~Base case test for failure: return false~~
- Loop over several options for "what to do next":
  1. Tentatively "**choose**" one option
  2. if ("**explore**" with recursive call returns true) return true
  3. else That tentative idea didn't work, so "**un-choose**" that option,
     but don't return false yet!--let the loop explore the other options before giv
- None of the options we tried in the loop worked, so return false

}

Your Turn: tracing the recursion in DFS maze-solver

Assume that the `generateValidMoves` *function we use provides the valid moves (as applicable) sorted in this order:* N, W, E, S.

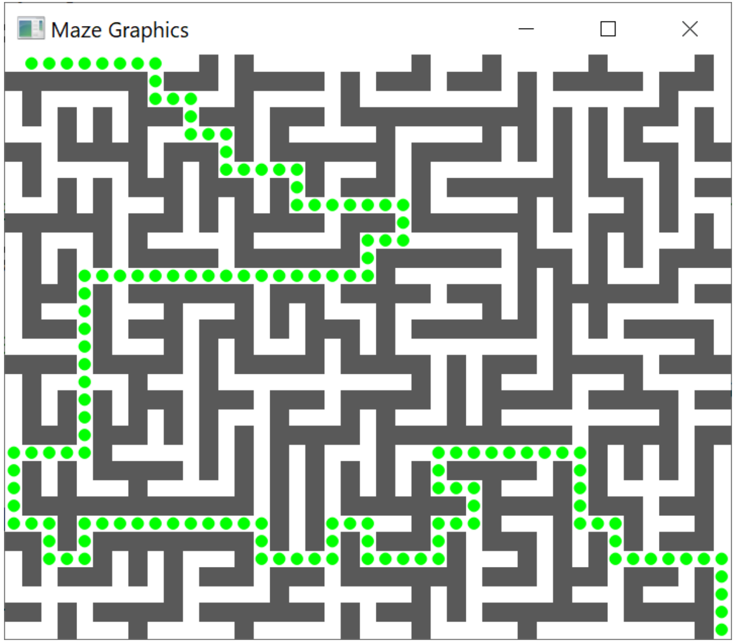
- In which order does the **DFS** visit the points marked X and Y?
  A. visits X before Y
  B. visits Y before X
  C. doesn't visit both X and Y
- In which order does the **BFS** (like your homework) visit the points marked X and Y?
  A. visits X before Y
  B. visits Y before X
  C. doesn't visit both X and Y



Your Turn: tracing the recursion in DFS maze-solver

Imagine the recursive call stack as we push/pop (call/return) in our recursive function as we solve this maze

- What is the **<u>most</u> number of stack frames** on the stack at any point?
  - A. Equal to the number of cells in the maze
  - B. Equal to the number of “forks in the road” we encounter as we explore
  - C. Equal to the length of the path at its longest in our exploration
  - D. Equal to the length of the final solution path



Depth-first vs. Breadth-first (DFS vs BFS)

- There's no one universal winner in terms of efficiency
  - We can design a maze that is instantly solvable with BFS, but where DFS would take a very long time, and vice versa
  - DFS heads off boldly in one direction
    - If that turns out to be right, very fast!
    - If it's wrong, may take a long time to course correct
- BFS has one key advantage: it is guaranteed to find the shortest path
  - DFS just finds any working path (which can sometimes make it faster)