Overview

• Graphics module “turtle”
• Visualizing recursion
• Dynamic programming
Objective

- Learn how to use “turtle” for visualization
- Learn how to use recursion to implement a game
- Be apply to apply dynamic programming to solve optimization problems

Visualization of Recursion

- Use turtle tool for visualization
- Turtle metaphor:
  - Move forward, back, turn left/right etc.
  - Tail up/down; if down => draws a line
  - Change width and color of tail
Visualization of Recursion

import turtle

myTurtle = turtle.Turtle()
myWin = turtle.Screen()

def drawSpiral(myTurtle, lineLen):
    if lineLen > 0:
        myTurtle.forward(lineLen)
        myTurtle.right(90)
        drawSpiral(myTurtle,lineLen-5)

drawSpiral(myTurtle,100)
myWin.exitonclick()

• Draw a fractal tree
• Fractals: same basic shape now matter how much it is magnified (self-similarity)
  • Snowflakes, fern, nautilus
• Used in computer graphics to generate realistic scenes
Visualization of Recursion

- Generate a fractal tree
- Small twig has same shape as tree
  - Tree is trunk with smaller trees going of to the left and the right
  - Apply recursion to both smaller left and right trees

```python
import turtle

def tree(branchLen, t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(20)
        tree(branchLen-15, t)
        t.left(40)
        tree(branchLen-15, t)
        t.right(20)
        t.backward(branchLen)

def main():
    t = turtle.Turtle()
    myWin = turtle.Screen()
    t.left(90)
    t.up()
    t.backward(100)
    t.down()
    t.color("green")
    tree(75, t)
    myWin.exitonclick()

main()
```
Exploring a Maze

- Important application for robotics
- Problem to solve: find way out of maze
- Assume maze is divided up in squares
  - Open or occupied
  - Turtle can only pass through open squares
  - If it bumps in wall => needs to find different direction

Exploring a Maze

Procedure:

- From starting position, first try going North one square and then recursively try procedure from there.
- If not successful by trying Northern path as first step then take a step to the South and recursively repeat procedure.
- If South does not work then try a step to the West as first step and recursively apply procedure.
- If North, South, and West have not been successful then apply the procedure recursively from a position one step to the East.
- If none of these directions works then there is no way to get out of the maze and we fail.
Exploring a Maze

• Must remember where turtle has been to avoid infinite loops
• Brothers Grimm to the rescue: Bread crumbs!
  • If step back is taken and bread crumb is already there, back up further
  • Try next direction
  • Backing up as easy as returning from recursive call

Base cases:
1. The turtle has run into a wall. Since the square is occupied by a wall no further exploration can take place.
2. The turtle has found a square that has already been explored. We do not want to continue exploring from this position or we will get into a loop.
3. We have found an outside edge, not occupied by a wall. In other words we have found an exit from the maze.
4. We have explored a square unsuccessfully in all four directions.
Exploring a Maze - Representation

Represent maze:
- **__init__** Reads in data file representing a maze, initializes the internal representation of the maze, and finds the starting position for the turtle.
- **drawMaze** Draws the maze in a window on the screen.
- **updatePosition** Updates the internal representation of the maze and changes the position of the turtle in the window.
- **isExit** Checks to see if the current position is an exit from the maze.

```python
def searchFrom(maze, startRow, startColumn):
    # try each of four directions from this point until we find a way out.
    # base Case return values
    # 1. We have run into an obstacle, return false
    maze.updatePosition(startRow, startColumn)
    if maze[startRow][startColumn] == OBSTACLE:
        return False
    # 2. We have found a square that has already been explored
    if maze[startRow][startColumn] == TRIED or maze[startRow][startColumn] == DEAD_END:
        return False
    # 3. We have found an outside edge not occupied by an obstacle
    if maze.isExit(startRow, startColumn):
        maze.updatePosition(startRow, startColumn, PART_OF_PATH)
        return True
    maze.updatePosition(startRow, startColumn, TRIED)
    # Otherwise, use logical short circuiting to try each direction
    # in turn (if needed)
    found = searchFrom(maze, startRow-1, startColumn) or \
             searchFrom(maze, startRow+1, startColumn) or \
             searchFrom(maze, startRow, startColumn-1) or \
             searchFrom(maze, startRow, startColumn+1)
    if found:
        maze.updatePosition(startRow, startColumn, PART_OF_PATH)
    else:
        maze.updatePosition(startRow, startColumn, DEAD_END)
    return found
```
Dynamic Programming

- Strategy to solve optimization problems
- Example: making change using fewest coins
  - Customer puts 1 dollar for 37 cents item in vending machine
  - Smallest number of coins for change = six: 2 quarters, 1 dime, 3 pennies
- How was result derived?

Dynamic Programming

- Greedy method:
  - Start with largest coin and use as many of those
  - Then next smaller one, and so on
- What if a 21 cents coin exists in addition to 1, 5, 10, and 25?
  - Greedy solution => 6 coins
  - Correct answer => 3 (21 cents) coins
Dynamic Programming

Recursive approach:

\[ numCoins = \min \begin{cases} 
1 + \text{numCoins}(\text{originalamount} - 1) \\
1 + \text{numCoins}(\text{originalamount} - 5) \\
1 + \text{numCoins}(\text{originalamount} - 10) \\
1 + \text{numCoins}(\text{originalamount} - 25) 
\end{cases} \]

```python
def recMC(coinValueList, change):
    minCoins = change
    if change in coinValueList:
        return 1
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recMC(coinValueList, change-i)
            if numCoins < minCoins:
                minCoins = numCoins
        return minCoins

print(recMC([1,5,10,25],63))
```
Dynamic Programming

- Algorithm is extremely inefficient
- Takes 67,716,925 recursive calls to find solution
- Each node in the following graph corresponds to \( \text{recMC}(\) 
- Label in node indicates amount of change for calculation
- Arrows indicate coin just used
- Lot of redundancy
  - E.g., make change for 15 cents done 3 times
Dynamic Programming

• Key component to cutting down computational overhead:
  • Remember past results
  • Avoid re-computing already known results
• Store results for minimum number of coins in a table when found
• To compute minimum, first check in table:
  • If found, use result from table
  • Else compute

```python
def recMC(coinValueList, change):
    minCoins = change
    if change in coinValueList:
        return 1
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recMC(coinValueList, change-i)
            if numCoins < minCoins:
                minCoins = numCoins
        return minCoins

print(recMC([1, 5, 10, 25], 63))
```
Dynamic Programming

- Line 6: added test to check if table contains minimum number of coins
  - If NOT, recursively compute minimum and store result in table
  - Reduces number of recursive calls to 221!
  - There are still holes in the table
  - What we have done so far is called “memoization” or “caching”!

Dynamic Programming

- More systematic approach for true Dynamic Programming algorithm
  - Start with making change for 1 cent
  - Work our way up to amount of change we require
- Guarantees:
  - At each step of algorithm minimum number of coins to need to make change for any smaller amount already known
Dynamic Programming

Change to Make

Step of the Algorithm

Dynamic Programming

123412345123451
123412345123451

11-1

11-5

11-10
Dynamic Programming

```python
def dpMakeChange(coinValueList, change, minCoins):
    for cents in range(change+1):
        coinCount = cents
        for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
                coinCount = minCoins[cents-j]+1
        minCoins[cents] = coinCount
    return minCoins[change]
```

• Not a recursive function

• For loop in line 4:
  • Consider using all possible coins to make change for the amount specified by `cents`
  • Store minimum value in `minCoins`

Dynamic Programming

• Algorithm doesn’t help to make change since it keep track of used coins

• Extend `dpMakeChange()` to keep track of used coins:
  • Remember last coin added for each entry in `minCoins`
  • If we know last coin added, simply subtract the value of the coin to find a previous entry in the table
  • Tells us last coin added to make that amount.
  • Keep tracing back through table until we get to the beginning.
Dynamic Programming

```python
def dpMakeChange(coinValueList, change, minCoins, coinsUsed):
    for cents in range(change+1):
        coinCount = cents
        newCoin = 1
        for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
                coinCount = minCoins[cents-j]+1
                newCoin = j
        minCoins[cents] = coinCount
        coinsUsed[cents] = newCoin
    return minCoins[change]
```

• Not a recursive function
Next Steps

- Next lecture on Tuesday: Greedy Algorithms
- HW3 and Project 1 posted today