Example: Escape from a Maze

Consider a rectangular maze consisting of an array of squares some of which are occupied by large blocks of concrete:

![Maze Diagram](image)

Given the size of the maze and locations of the blocks, prisoner, and exit, how does the prisoner escape?

Maze Program (Incorrect)

```python
def solve_maze(row0, col0, maze):
    # Assume that MAZE is a rectangular 2D array (list of lists) where
    # maze[r][c] is true iff there is a concrete block occupying
    # column c of row r. ROW0 and COL0 are the initial row and column
    # of the prisoner. Returns true iff there is a path of empty
    # squares that are horizontally or vertically adjacent to each other
    # starting with (ROW0, COL0) and ending outside the maze."
    if row0 not in range(len(maze)) or col0 not in range(len(maze[row])):
        return True
    elif maze[row0][col0]:
        return False
    else:
        return solve_maze(row0+1, col0, maze) or solve_maze(row0-1, col0, maze)
        # What's wrong?
```

Maze Program (Corrected)

To fix the problem, remember where we've been:

```python
def solve_maze(row0, col0, maze):
    # Assume that MAZE is a rectangular 2D array (list of lists) where
    # maze[r][c] is true iff there is a concrete block occupying
    # column c of row r. ROW0 and COL0 are the initial row and column
    # of the prisoner. Returns true iff there is a path of empty
    # squares that are horizontally or vertically adjacent to each other
    # starting with (ROW0, COL0) and ending outside the maze."
    visited = set() # Set of visited cells
cols, rows = range(len(maze[0])), range(len(maze))
def escapep(r, c):
    # True iff is a path of empty, unvisited cells from (R, C) out of maze."
    if r not in rows or c not in cols:
        return True
    elif maze[r][c] or (r, c) in visited:
        return False
    else:
        visited.add((r,c))
        return escapep(r+1, c) or escapep(r-1, c) or escapep(r, c+1) or escapep(r, c-1)
    return escapep(row0, col0)
```

Example: Making Change

```python
def count_change(amount, denoms = (50, 25, 10, 5, 1)):
    # The number of ways to change AMOUNT cents given the
    # denominations of coins and bills in DENOMS.
    if amount == 0: return 1
    elif len(denoms) == 0: return 0
    elif amount >= denoms[0]:
        return count_change(amount-denoms[0], denoms) + count_change(amount, denoms[1:])
    else:
        return count_change(amount, denoms[1:])
```

Avoiding Redundant Computation

- In the (tree-recursive) maze example, a naive search could take us
  in circles, resulting in infinite time.
- Hence the visited set in the escapep function.
- This set is intended to catch redundant computation, in which re-
  processing certain arguments cannot produce anything new.
- We can apply this idea to cases of finite but redundant computation.
- For example, in count_change, we often revisit the same subprob-
  lem:
  - E.g., Consider making change for 87 cents.
  - When choose to use one half-dollar piece, we have the same sub-
    problem as when we choose to use no half-dollars and two quar-
    ters.
- Saw an approach in Lecture #16: memoization.
Memoizing

- Idea is to keep around a table ("memo table") of previously computed values.
- Consult the table before using the full computation.
- Example: `count_change`:

```python
def count_change(amount, denoms = (50, 25, 10, 5, 1)):
    memo_table = {} # Indexed by pairs (row, column)
    # Local definition hides outer one so we can cut-and-paste
    # from the unmemoized solution
    def count_change(amount, denoms):
        if (amount, denoms) in memo_table:
            return memo_table[(amount, denoms)]
        # Memoization is additive (returns only):
        # count_change(57) (returns only):
        # memo_table[57] = a
        = full_count_change(amount, denoms)
        return memo_table[(amount, denoms)]
    def full_count_change(amount, denoms):
        # Memoized original solution goes here verbatim
        return count_change(amount, denoms)
    # Question: how could we test for infinite recursion?

```

Order of Calls

- Going one step further, we can analyze the order in which our program ends up filling in the table.
- So consider adding some tracing to our memoized `count_change` program:

```python
    def count_change(amount, denoms):
        memo_table = {}
        ... full_count_change(amount, denoms) ...
        return memo_table[amount,denoms]
    @trace
    def full_count_change(amount, denoms):
        if amount == 0: return 1
        elif not denoms: return 0
        elif amount >= denoms[0]:
            return count_change(amount, denoms[1:]) \  
            + count_change(amount-denoms[0], denoms)
        else:
            return count_change(amount, denoms)
```

Dynamic Programming

- Now rewrite `count_change` to make the order of calls explicit, so that we needn't check to see if a value is memoized.
- Technique is called dynamic programming (for some reason).
- We start with the base cases, and work backwards.

```python
    def full_count_change(amount, denoms):
        # How often is this called?
        ... (calls count_change for recursive results)
        for a in range(amount+1):
            memo_table[a][0] = full_count_change(a, ())
        for k in range(len(denoms) + 1):
            for a in range(amount+1):
                memo_table[a][k] = full_count_change(a, denoms[k:])**
                return count_change(amount, denoms)
```

Optimizing Memoization

- Used a dictionary to memoize `count_change`, which is highly general, but can be relatively slow.
- More often, we use arrays indexed by integers (lists in Python), but the idea is the same.
- For example, in the `count_change` program, we can index by `amount` and by the portion of `denoms` that we use, which is always a slice that runs to the end.

```python
    def count_change(amount, denoms = (50, 25, 10, 5, 1)):
        memo_table = [-1] * (len(denoms)+1) for i in range(amount+1)
        def count_change(amount, denoms):
            if (amount, denoms) in memo_table:
                return memo_table[(amount, denoms)]
            # Memoizing is additive (returns only):
            # count_change(57) (returns only):
            # memo_table[57] = a
            = full_count_change(amount, denoms)
            return memo_table[(amount, denoms)]
        # Question: how could we test for infinite recursion?
```

New Topic: Tree-Structured Data

- 1 Linear-recursive and tail-recursive functions make a single recursive call in the function body. Tree-recursive functions can make more.
- Linear recursive data structures (think rlists) have single embedded recursive references to data of the same type, and usually correspond to linear- or tail-recursive programs.
- To model some things, we need multiple recursive references in objects.
- In the absence of circularity (paths from an object eventually leading back to it), such objects form data structures called trees:
  - The objects themselves are called nodes or vertices.
  - Tree objects that have no (non-null) pointers to other tree objects are called leaves.
  - Those that do have such pointers are called inner nodes, and the objects they point to are children (or subtrees or (uncommonly) branches).
  - A collection of disjoint trees is called a forest.

```
    Result of Tracing

    • Consider count_change(57) (returns only):
      full_count_change(57, (50, 25, 10, 5, 1)) -> 62
      full_count_change(57, (25, 10, 5, 1)) -> 60
      full_count_change(57, (10, 5, 1)) -> 58
      full_count_change(57, (5, 1)) -> 56
      full_count_change(57, (1,)) -> 54
```

```
    • Order of Calls

    • Going one step further, we can analyze the order in which our program ends up filling in the table.
    • So consider adding some tracing to our memoized count_change program:
```

```
    • Dynamic Programming

    • Now rewrite count_change to make the order of calls explicit, so that we needn't check to see if a value is memoized.
    • Technique is called dynamic programming (for some reason).
    • We start with the base cases, and work backwards.
      full_count_change(0, ()) -> 1
```
Example: Expressions

- An expression (in Python or other languages) typically has a recursive structure. It is either
  - A literal (like 5) or symbol (like x)—a leaf—or
  - A compound expression consisting of an operator and zero or more operands, each of which is itself an expression.
- For example, the expression \( x + (y+2)*(z+10) \) can be thought of as a tree (what happened to the parentheses?):

Expressions as Tuples or Lists

- We can represent the abstract structure of the last slide with Python objects we’ve already seen:

Class Representation

- …or we can introduce a Python class:

A General Tree Type

- Trees don’t quite lend themselves to being captured with standard syntax like tuples or lists, because they get accessed in various ways, with slightly varying interfaces.
- To start with, we’ll use this type, which has no empty trees:

A General Tree Type: Accessors

- Since trees are recursively defined, recursion generally figures in algorithms on them.
- Example: number of leaf nodes.

A Simple Recursion

- How long does this take (for a tree with \( N \) leaves)?