CS1010 Tutorial 11 (Final)

Slides by Ryan Tan Yu

Agenda for Today

- Assignment 8
- Assignment 9

Assignment 8

Question 1: Walk

- Given an input (x, y), find out how many ways we can reach (x, y) from the coordinates $(0, 0)$
- Let $F(x, y)$ calculate the desired value
- $F(1, k) = F(k, 1) = 1$ for all k
- Draw out a small example to find out the pattern

Formula

- We can represent the recursion using the following recurrence relation

$$F(x,y) = \begin{cases} 1, & \text{if } x = 0 \text{ or } y = 0 \\ F(x-1,y) + F(x,y-1), & \text{otherwise} \end{cases}$$

- The code is immediate with the recurrence relation

```
algorithm RecursiveWalk(x, y):
    if x == 0 || y == 0:
        return 1
    return RecursiveWalk(x - 1, y) + RecursiveWalk(x, y - 1)
```

Naive Recursion Run-time Analysis

- For high values of x and y, the run-time of the recursion is very slow, definitely not $O(xy)$

Input	Time
F(15,15)	0.6s
F(16,16)	3.0s
F(17,17)	10.3s
F(18,18)	41.4s
F(19,19)	157.7s
F(20,20)	635s

Naive Recursion Run-time Analysis (cont.)

- We can represent the time complexity by the following recurrence
- Note that $T(1, k) = T(k, 1) = 1$ for all k, then

$$T(x, y) = T(x-1, y) + T(x, y-1)$$

- Solving the recurrence, we get that

$$T(x, y) = \binom{x+y}{x} = \frac{(x+y)!}{x!((x+y)-x)!} = \frac{(x+y)!}{x!y!}$$

- The algorithm is factorial in x, y

Dynamic Programming

- To a trained eye, this is a *dynamic programming* problem
- To hit $O(xy)$ time, represent the paths with a $x \times y$ matrix, M
- Every $M[i][j]$ represents the solution to $F(i, j)$
- Initialise all $M[0][k] = M[k][0] = 1$ for all k
- ***Algorithm*** Simply set each $M[i][j] = M[i-1][j] + M[i][j-1]$ using a double `for` -loop
- Clearly, such an algorithm runs in $O(xy)$

$$
\begin{tikzcd}
1 \arrow[r, "+"] & 5 & \cdots & & \\
1 \arrow[r, "+"] & 4 \arrow[u, "+"'] \arrow[r, "+"] & \cdots & \cdots & \\
1 \arrow[r, "+"] & 3 \arrow[u, "+"'] \arrow[r, "+"] & 6 \arrow[u, "+"'] \arrow[r, "+"] & \cdots & \cdots \\
1 \arrow[r, "+"] & 2 \arrow[u, "+"'] \arrow[r, "+"] & 3 \arrow[u, "+"'] \arrow[r, "+"] & 4 \arrow[u, "+"'] \arrow[r, "+"] & 5 \\
1 & 1 \arrow[u, "+"'] & 1 \arrow[u, "+"'] & 1 \arrow[u, "+"'] & 1 \arrow[u, "+"']
\end{tikzcd}
$$

Recursive Dynamic Programming

- We can refrain from using loops to solve the problem, by using the same recursive approach but with the matrix M
- The loop-version is normally called *Bottom-Up Dynamic Programming*
- The recursive-version is called *Memoization* or *Top-down Dynamic Programming*

```
// M[0...x][0...y] is ZERO-INDEXED in this case, and all its values
// are initialized to -1
algorithm RecursiveWalkMemoization(M[0...x][0...y], x, y):
    if x == 0 || y == 0:
        M[x][y] = 1
        return 1
    if M[x][y] != -1:
        return M[x][y]

    M[x - 1][y] = RecursiveWalkMemoization(M, x - 1, y)
    M[x][y - 1] = RecursiveWalkMemoization(M, x, y - 1)
    return M[x - 1][y] + M[x][y - 1]
```

Question 2: Maze

- Given an $m \times n$ maze M, print out a maze-solving algorithm to the screen
- A relatively tough question, second-hardest after Social (imo)
- Goal: Practice **Recursion with Backtracking**

Some Housekeeping

1. We need a function that prints the maze with the number of steps to the screen
    i. This is provided in the skeleton
2. We need a function to return the x, y co-ordinates of `USER`. We can use a double `for`-loop and use pointers to return respective x and y values

Given Algorithm

> She follows **strictly** the following strategy to find a way through the maze starting from her initial position. At each time step,
>
> 1. She looks for an empty adjacent cell that has never been visited yet, in the sequence of up/right/down/left to the current cell she is at. If there is an empty adjacent cell, she moves to that cell. The cell she moves to is now visited.
> 2. If no empty, unvisited, adjacent cell exists, she backtracks on the path that she comes from, moving one step back, and repeat 1 again.

Outline

- Sounds simple, but implementation and animation is hard
- ***Base Cases***
  - If `USER` walks to the outer-most perimeter of the maze - `USER` has escaped
  - If `USER` walks to a cell that has already been visited
  - If `USER` walks into a wall
- Most are simple index-checking, except for no.2
- To check for no.2
  - Initialise another $m \times n$ array, call it V (for `visited`) and initialise all values to `false`
  - Whenever a cell at (i, j) is visited, label $V[i][j]$ as `true`

Outline (cont.)

```
algorithm solve(maze, visited, m, n, prev_x, prev_y, x, y):
    if escaped:
        return TRUE
    if visited[x][y] or maze[x][y] is a wall:
        return FALSE

    visit cell (x,y) // we don't know how to do this
    if (go_up || go_right || go_down || go_left) // or this
        return TRUE
    backtrack // or this

    return FALSE
```

- The use of `||` is importable
- We exploit short-circuiting of `||` in order to explore only as much as needed

Visiting a Cell

- Simple, just "move" the user in the maze M
- Can use a `swap` function as well

```
maze[prev_x][prev_y] = EMPTY
maze[x][y] = USER
visited[x][y] = true
```

Exploration

- Recursively call `solve` with different parameters for `prev_x`, `prev_y`, etc
- `prev_x` and `prev_y` are just the current values of `x` and `y`
- `x` and `y` will be the respective directions from the current `x` and `y`

```
algorithm solve(maze, visited, m, n, prev_x, prev_y, x, y):
    // ...
    if (solve(maze, visited, m, n, x, y, x - 1, y) // Up
             || solve(maze, visited, m, n, x, y, x, y + 1) // Right
             || solve(maze, visited, m, n, x, y, x + 1, y) // Down
             || solve(maze, visited, m, n, x, y, x, y - 1)): // Left
        return TRUE
```

Backtracking

- Backtracking can be done by "resetting" the visiting step from earlier in the function

```
maze[x][y] = EMPTY
maze[prev_x][prev_y] = USER
```

(Nearly) Complete Algorithm

```
algorithm solve(maze, visited, m, n, prev_x, prev_y, x, y):
    if escaped:
        return TRUE
    if visited[x][y] or maze[x][y] is a wall:
        return FALSE

    // visit the cell
    maze[prev_x][prev_y] = EMPTY
    maze[x][y] = USER
    visited[x][y] = true

    // visit up, right, down, left
    if (solve(maze, visited, m, n, x, y, x - 1, y)
            || solve(maze, visited, m, n, x, y, x, y + 1)
            || solve(maze, visited, m, n, x, y, x + 1, y)
            || solve(maze, visited, m, n, x, y, x, y - 1))
        return TRUE

    // backtrack
    maze[x][y] = EMPTY
    maze[prev_x][prev_y] = USER

    return FALSE
```

Assignment 9

The Final Assignment

Digits

- Using an AI/ML algorithm (k-nearest neighbours), implement a program that can detect which digit an input, handwritten digit is
- Put (nearly) everything you've learnt into one massive program
- You're expected to write at least 200-300 lines of code

The Algorithm

- Given a list of all test digits L_q, and a list of training digits L_t
- For each $i \in |L_q|$
  - Let $q \leftarrow L_q[i]$
  - Create a distance list D_t of size $|L_t|$ where each $D_t[j] = d(q, L_t[j])$
    - where d is the distance function
  - Partial sort D_t to get the k nearest neighbors
  - Look at all k neighbors, determine which digit q is most likely to be, call it t
  - Print out the digit q and the likely digit t
- It shouldn't get it right everytime

Tips

- As mandated in the question, create a `struct digit` and `struct neighbour`

```
struct digit { long label; char **digit; };
struct neighbor { long neighbor; long distance; /* possible additional fields */ };
```

- Create a function that compares a test digit q with all training digits t_i, then returns what digit q is most likely to be
    - Use the given k-nearest neighbor algorithm
    - Within this function, call other functions...
- To "partial-sort" k elements of a list, use $SelectionSort$, but only iterate k times
- Create a function `distance_between` that finds the distance between two digits