CS1010 Laboratory 09

Backtracking

Zhang Puyu

Group BD04

November 7, 2024

Plan of the Day

`sort.c`: Two Common Approaches

Approach 1: Small to Big.

1. Locate the index of the minimum element using either linear or binary search.
2. Put this minimum as the first element of output array.
3. Consider the left and right neighbours of the minimum:
   - If `left > right`, we put `right` into the next index of output array, and move one index to the right.
   - Otherwise, put `left` into the next index of output array, and move one index to the left.
4. When one side reaches the end, put whatever is left in the other side into the output array in ascending order.

`sort.c`: Two Common Approaches

Approach 2: Big to Small.

1. Initialise `left` at index 0 and `right` at index `len - 1`.
   - If `left > right`, we put `left` into the next index of output array, and move one index to the right.
   - Otherwise, put `right` into the next index of output array, and move one index to the left.
2. We stop when `left > right`. The output list now is reversely sorted.
3. Since we know the size of the output array, we can easily print it in reverse order.

`valley.c`

1. We will look for the minimum between `start` and `end`.
2. Compute the midpoint index `mid`.
   - If `mid + 1` is a bigger element, then we continue searching between `start` and `mid`.
   - Otherwise, we will search in between `mid + 1` and `end`.
3. The above terminates when the search space only contains a singleton.

`inversion.c`: How to Count Cumulatively

1. Let `i = 0` and `j = 0`. Assume the array is of size n.
2. For the i-th element, do:
   - If `arr[n - 1 - j] < arr[i]`, increment `j`.
   - Repeat until `arr[n - 1 - j] >= arr[i]`.
   - j is the number of elements which are inversions with respect to `arr[i]`.
   - Increment total count by j and move to the next element.
3. When $i = n - 1 - j$, we claim that `arr[i]` is the peak.
4. From `arr[i]` to `arr[n - 1]` there are $\frac{(n-i)(n-i-1)}{2}$ inversions.

Question: Why do we not reset j?

`mark.c`: Stable Sort

- Note that equal marks are already sorted based on their names in the input.
- So we only need to run counting sort on the marks, with the extra constraint that:
- The names appear in the exact same order as in the input.
- This is known as a **stable** sorting algorithm.

Definition (Stable Sort)

A sorting algorithm is **stable** if for any $a = b$ in the input such that a comes before b, in the sorted output a still comes before b.

`mark.c`: Stable Sort

- We can compute the frequency table of all marks.

$\cdots$	m_i	m_{i+1}	m_{i+2}	m_{i+3}	m_{i+4}	$\cdots$
$\cdots$	f_i	f_{i+1}	f_{i+2}	f_{i+3}	f_{i+4}	$\cdots$

- Simpler problem: how to transform the frequency table into a cumulative frequency table?

$$F_i = \sum_{j=0}^{i} f_j.$$

- The **cumulative frequency** table:

$\cdots$	m_i	m_{i+1}	$\cdots$
$\cdots$	$F_i = F_{i-1} + f_i$	$F_{i+1} = F_i + f_{i+1}$	$\cdots$

- Claim: If the cumulative frequency of m_k is F_k, then in the sorted array, the n-th occurrence of m_k is at index $F_k - f_k + n - 1$.

Recursion & Backtracking

- Previously, we have seen how to use recursion to arrive at $P(k)$ given $P(1)$ and $P(n) \mapsto P(n+1)$.
- It seems (erroneously) that recursion is always about "going backwards", but it is not always the case.
- A problem-solving paradigm:
  1. Suppose we have an **initial state** S.
  2. Our **goal** is to find all states where a **terminating condition** p holds.
  3. Upon entering each **current state** X, we first check if p is true. If it is, we **collect** X as a valid result.
  4. Else, we explore the **actions** that can be performed at state X to **transition** to a different state Y.
  5. For each new state Y generated this way, we enter it and repeat the process.
  6. After **exhausting** all possible actions, we **return** to the previous state.
- **Backtracking** is **searching by brute force**.

An Example: `stone.c` from past year PE

Basically, this asks you to **print out all binary sequences of size n in ascending order**.

1. We start with our sequence $s = \varnothing$ and number of digits $k = 0$.
2. At every stage, check: is $k = n$?
   - If yes, then this is a valid sequence, we will print it and return. (**Collect** a valid result and **backtrack** to the previous state.)
   - If no, then we have 2 choices to proceed to the next stage:
     1. Append 0 to the end of s; or
     2. Append 1 to the end of s.

An Example: `stone.c` from past year PE


```c
void print_binary(long k, long n) {
    // Is [s, k] a goal state?
    if (k == n) {
        // Yes, collect the result.
        cs1010_println_string("");
        // Backtrack to the previous state.
        return;
    }
    // No, explore the possible actions.
    // Action 1: append 0.
    // Transition 1: [s, k] -> [s0, k + 1].
    putchar('0');
    print_binary(k + 1, n);
    // Action 2: append 1.
    // Transition 2: [s, k] -> [s1, k + 1].
    putchar('1');
    print_binary(k + 1, n);
}

int main() {
    long n = cs1010_read_long();
    print_binary(0, n); // Initial state: ["", 0].
    return 0;
}
```

Graph Traveral: Depth-First Search

A graph consists of vertices and edges. An edge $x \rightarrow y$ allows you to move from vertex x to vertex y.

- Given two vertices v and u, we wish to determine whether they are connected by a path.
- Identify the following:
  1. **Current Stage:** The current vertex x (initially v).
  2. **Terminating Condition:** $x = u$.
  3. **State Transitions:** Move to any unvisited neighbour of x.
- DFS at VisuAlgo.

Other Interesting Applications of Backtracking

- Writing a parser and compiler.
- Goal-Oriented Action Planning in game AI.
- Generating permutations.
- Building a dependency tree.

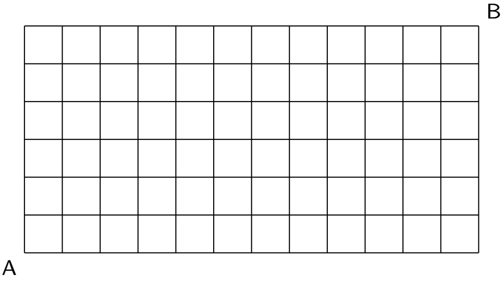
walk.c

- This is a graph traversal problem in a grid. So each vertex can be uniquely represented by lattice point coordinates.
- Identify the following:
  1. **Current Stage:** The current coordinates (a, b) (initially $(0, 0)$).
  2. **Terminating Condition:** $(a, b) = (x, y)$.
  3. **State Transitions:**
     1. $(a, b) \mapsto (a + 1, b)$
     2. $(a, b) \mapsto (a, b + 1)$

In fact this is a **troll question...** The phrasing seems that $\mathcal{O}(xy)$ is optimal, but someone trained in combinatorics quickly realises that we can do this in $\mathcal{O}(\min\{x, y\})$ (and using a single loop only).

`walk.c`: The Uninteresting Solution

How to go from A to B (x steps to the right and y steps to the top)?



Let $P(x, y)$ be the number of paths from A to (x, y). Observe that

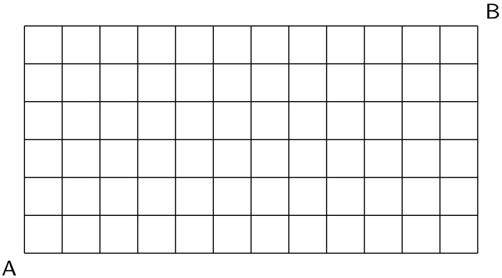
$$P(x,y) = \begin{cases} 1 & \text{if x} = 0 \text{ or y} = 0 \\ P(x-1,y) + P(x,y-1) & \text{otherwise} \end{cases}.$$

walk.c: The Uninteresting Solution


```
long move_to(long m, long n) {
    if (m == 0 || n == 0) {
        return 1;
    }
    return move_to(m - 1, n) +
           move_to(m, n - 1);
}
```

Have you spotted the duplicated computation here?

`walk.c`: The Uninteresting Solution



Note that the above grid can be seen as a **2D array**! Each vertex is an entry, in which we will store the number of ways to get to that vertex from A.

Now we can use this 2D array as a look-up table to build a bottom-up solution. (So this is what "think recursively but solve iteratively" supposed to mean...)

We will need to fill up the matrix completely before we know the value of `mat[m][n]`, so it is $\mathcal{O}(mn)$.

`walk.c`: The Legit Solution

Suppose that 0 represents a step rightwards and 1 represents a step upwards, then every path can be uniquely mapped to a binary sequence with m 0's and n 1's. (Bijection)

Now we are dealing with a simpler question: **how many distinct binary sequences are there with m 0's and n 1's?**

The way to create such a sequence: **list down $(m+n)$ 0's** and **toggle n of them to be 1.**

Using some maths, we know that the answer is

$$\binom{m+n}{n} = \frac{(m+n)(m+n-1)(m+n-2)\cdots(m+1)}{1\cdot 2\cdot 3\cdot\cdots\cdot n}.$$

`walk.c`: The Legit Solution

```
long x_choose_y(long x, long y) {
    long top = 1;
    long bottom = 1;
    for (long i = 0; i < y; i += 1) {
        top *= (x - i);
        bottom *= (i + 1);
    }
    return top / bottom; // Guaranteed to be an integer
}

int main() {
    long m = cs1010_read_long();
    long n = cs1010_read_long()
    cs1010_println_long(x_choose_y(m + n, m > n ? n : m));
}
```

You are wrong if you think this is correct. (Why?)

Both `top` and `bottom` are factorials, which easily gets too big for `long` or even `long long`.

`walk.c`: The Legit Solution

Observe that

$$\binom{x}{y} = \frac{x-y+1}{y}\binom{x}{y-1}$$

This implies that we can start from $c = \binom{x}{1} = x$, at each iteration, we update by `c = c * (x - i) / (i + 1)`. This number will be small at each iteration, so we can use `long` again!

`walk.c`: The Legit Solution


```c
long x_choose_y(long x, long y) {
    long c = x; // x choose 1
    for (long i = 1; i < y; i += 1) {
        // Guaranteed to be an integer
        c = c * (x - i) / (i + 1);
    }
    return c;
}

int main() {
    long m = cs1010_read_long();
    long n = cs1010_read_long()
    cs1010_println_long(x_choose_y(m + n, m > n ? n : m));
}
```

This will be $\mathcal{O}(\min\{m, n\})$ (WAY faster than proposed “optimal” solution).

`maze.c`: A First Encounter with DFS

- This is a real graph traversal problem which does require recursion and DFS.
- Identify the following:
  1. **Current Stage:** The current coordinates (a, b) (initially at player spawn point).
  2. **Terminating Condition:** (a, b) is at the boundary.
  3. **State Transitions:** Go to any **unvisited** neighbouring point which is not a wall.

`maze.c`: A First Encounter with DFS

We are given a maze (a 2D array) and a birth point at (x, y).

- We wish to know if we can reach the exit from (x, y).
- Step 1: Check **whether (x, y) is already an exit**.
- Step 2: If we are not at the exit yet, we iterate the neighbouring cells. For each neighbour (x', y'):
  - If (x', y') is visited, skip.
    - We can use another 2D array (the "map") to store the status of the cells!
    - Note that we actually waste a lot of memory space here. (A more space-efficient solution requires a hash table.)
  - Else, set starting point to (x', y') and run the same processes.
- Step 3: If we cannot get to the exit from any neighbouring cell, it means the maze is unsolvable.
- How do we keep track of step count? **Use a pointer** so that we can update its value in any function call!

maze.c: A First Encounter with DFS

```
bool can_escape(char **maze, bool **visited, long x, long y,
    long nrows, long ncols, long *count) {
  visited[x][y] = true; // Current cell becomes visited
  // TODO: Implement is_exit
  if (is_exit(x, y, nrows, ncols)) {
    return true; // Escaped!
  }
  // Try going up, TODO: Implement can_go_there and move_to
  if (can_go_there(x - 1, y, maze, visited)) {
    // Go there
    move_to(maze, x - 1, y, x, y, count, nrows);
    if (can_escape(maze, visited, x - 1, y,
        nrows, ncols, count)) {
      return true; // That cell is connected to an exit!
    }
    // Think: What does it mean if we ever reach this line?
    move_to(maze, x, y, x - 1, y, count, nrows); // Step back
  }
  // TODO: go right, go down, and go left
  return false; // No exit is reachable
}
```

`fill.c`: A Maze with No Walls

The key to solving this question is to be able to identify it as an **implicit graph traversal question!**

- The coordinates at which we drop the colour bucket is essentially our player spawn point.
- Instead of finding an exit, this time our task is to **visit every reachable cell with the same colour as the spawn point**.
- Note that as compared the `maze.c`, now each cell contains additional information: a 3-tuple (r, g, b) indicating its colour!
  - This is a 3D array! How do we process it?
  - Approach 1: Fake 3D array by `ncols = 3 * n`. To visit the cell at row x and column y, use `canva[x][y * 3]`.
  - Approach 2: Use a real 3D array `long (**canva)[3]`.
  - Approach 3: Use a real 3D array `long **canva[3]`.

`fill.c`: A Maze with No Walls

Pseudo-code:

- We wish to pour the colour bucket containing (R', G', B') at (x, y) which has colour (R, G, B).
- If $(R, G, B) = (R', G', B')$, there's nothing to colourise.
- Else, we will change the colour of (x, y)
- For the four neighbours:
  - If the neighbour is out of boundary of the canva, skip.
  - If the neighbour has a different colour from (R, G, B), skip.
  - Else, spread the colour (R', G', B') to the neighbour.

fill.c: A Maze with No Walls


```
void colour(
    long (**canva)[3], long x, long y,
    long old_colour[3], long new_colour[3],
    long nrows, long ncols
) {
  if (!is_within_boundary(x, y, nrows, ncols) ||
      are_the_same_colour(old_colour, new_colour)) {
    return;
  } // TODO: Implement these two boolean functions

  // TODO: Change the colour of current pixel
  change_colour(canva, x, y, old_colour, new_colour);

  // Now go up
  colour(canva, x - 1, y, old_colour, new_colour,
      nrows, ncols);

  // TODO: Implement go-right, go-down and go-left cases
}
```

`sudoku.c`: Intimidating, but the Idea Is Easy

As human beings, how do we solve a Sudoku puzzle?

By **trial and error!** We randomly put a number into the blank and proceed, if it works out we have a solution, else we erase the number and try a different one.

We only need to mimic this behaviour with code!

(By the way, the wording of the question statement was very ambiguous and I am personally not a fan of this problem.)

`sudoku.c`: Intimidating, but the Idea Is Easy

- We will run through the puzzle from top to bottom and left to right.
- At the x-th row and y-th column, print the current state of the puzzle and check:
  - If `x > NROWS`, we have successfully run through the puzzle and nothing wrong happens, so the puzzle is solvable.
  - Else if `y > NCOLS`, we should move to the next row.
  - Else if (x, y) is not empty, we should move to the next column.
  - Else if (x, y) is empty, for $i = 1, 2, \cdots, 9$:
    - If placing i here is safe, set $(x, y) = i$ and move on.
    - If a solution can be found from here, the puzzle is solvable.
    - Else, reset (x, y) and try the next valid i.
  - If none of $1, 2, \cdots, 9$ leads to a solution, the puzzle is not solvable with the current state.

`sudoku.c`: Anything Wrong Here?


```c
bool can_solve(char puzzle[9][9], long x, long y) {
    print_sudoku(puzzle); // Print current state of the puzzle
    if (x > 8) {
        // Have checked every cell and nothing went wrong
        return true;
    }
    if (y > 8) {
        return can_solve(puzzle, x + 1, 0); // To next row
    }
    if (puzzle[x][y] != EMPTY) {
        return can_solve(puzzle, x, y + 1); // To next column
    }
    for (char guess = '1'; guess <= '9'; guess += 1) {
        // TODO: Implement is_valid
        if (is_valid(guess, x, y, puzzle)) {
            puzzle[x][y] = guess;
            if (can_solve(puzzle, x, y + 1)) { return true; }
            puzzle[x][y] = EMPTY; // Didn't work, guess again
        }
    }
    return false; // Tried everything, didn't find a solution
}
```