CS1010 Tutorial 9

Slides by Ryan Tan Yu

Agenda for Today

- Problem Set 25
- Problem Set 26
- Problem Set 27
- Past Year PE2 Questions
- Assignment 7 Comments
- Assignment 8 Hints

Problem Set 25

Problem 25.1

- Add a new restriction to the Tower of Hanoi puzzle.
    - Disks are on Peg A to begin with
    - Move disks to Pec C
    - Only allowed to move a disk either
        - To Peg B from another peg
        - From Peg B to another peg
        - i.e **cannot move the disks between A and C directly**

Change the recursive algorithm above to solve the Tower of Hanoi with this new restriction. How many steps (use big-O notation) are needed now?

Problem 25.1

- Original algorithm is given below

```
void tower_of_hanoi(long k, long source, long dest, long placeholder) {
    if (k == 1) {
        print(k, source, dest);
        return;
    }
    solve_tower_of_hanoi(k - 1, source, placeholder, dest);
    print(k, source, dest);
    solve_tower_of_hanoi(k - 1, placeholder, dest, source);
}
```

Problem 25.1

- **Think recursively**
  - We only need to think about two "objects"
    - The largest disk, and
    - The first $k - 1$ disks
  - The rest comes from recursion
- Think about how you would solve the problem in the case of
  - 1 disk (base case)
  - 2 disks (representing the first $k - 1$ disks, and the largest disk)

Problem 25.1

- Label leftmost peg as A, middle peg as B, rightmost peg as C
- ***Base case***: 1 disk
  - Move the disk from $A \to B \to C$
- ***Recursive Case***: 2 disks
  - Move $k-1$ disks from $A \to B \to C$
  - Move k-th disk from $A \to B$
  - Move $k-1$ disks from $C \to B \to A$
  - Move k-th disk from $B \to C$
  - Move $k-1$ disks from $A \to B \to C$

Problem 25.1

```
void solve(long k, long source, long dest, long placeholder) {
   if (k == 1) {
      move(k, source, placeholder);
      move(k, placeholder, dest);
      return;
   }
   solve(k - 1, source, dest, placeholder);
   move(k, source, placeholder);
   solve(k - 1, dest, source, placeholder);
   move(k, placeholder, dest);
   solve(k - 1, source, dest, placeholder);
}
```

Problem 25.1 - Runtime Analysis

```
void solve(long k, long source, long dest, long placeholder) {
    if (k == 1) {
        move(k, source, placeholder);
        move(k, placeholder, dest);
        return;
    }
    solve(k - 1, source, dest, placeholder);
    move(k, source, placeholder);
    solve(k - 1, dest, source, placeholder);
    move(k, placeholder, dest);
    solve(k - 1, source, dest, placeholder);
}
```

- There are 3 recursive calls, each of $T(n-1)$
- There are 2 calls to `move`, both of which are $O(1)$
- $T(n) = 3T(n-1) + 2$

Problem 25.1

$$
\begin{aligned}
T(n) &= 3T(n-1) + 2 \\
&= 3(3T(n-2) + 2) + 2 = 9T(n-2) + 8 \\
&= 9(3T(n-3) + 2) + 8 = 27(n-3) + 26 \\
&= \ldots \\
&= 3^n \times T(1) + O(1) \in O(3^n)
\end{aligned}
$$

Problem Set 26

Problem 26.1

- Modify the algorithm below to avoid considering duplicate permutations if there are repeated letters in the input

```
void permute(char a[], long len, long curr) {
    if (curr == len - 1) {
        cs1010_println_string(a);
        return;
    }

    permute(a, len, curr + 1);
    for (long i = curr + 1; i < len; i += 1) {
        if (              ) { // Add a condition here
            swap(a, curr, i);
            permute(a, len, curr + 1);
            swap(a, i, curr);
        }
    }
}
```

Problem 26.1

- Consider the string $a\ b_0\ c\ b_1$
- Permute for 6 iterations...
  - $a\ b_0\ c\ b_1$ - Δ
  - $a\ b_0\ b_1\ c$ - Φ
  - $a\ c\ b_0\ b_1$ - Σ
  - $a\ c\ b_1\ b_0$ - Σ
  - $a\ b_1\ c\ b_0$ - Δ
  - $a\ b_1\ b_0\ c$ - Φ
- Already have three duplicates!

Problem 26.1

- ***Key Idea*** When considering some `a[i]`
  - **Do not permute** if `a[i]` has occured somewhere before `a[curr]...a[i]`
  - If `a[i]` has appeared in the this range, then it has already been considered as the "first character" in the permutation algorithm
  - Considering it will lead to duplicate permutations
- If you don't get it - *spend some time tracing the recursion for a small example* and convince yourself that it works

```c
// Checks if the character at a[i] has appeared from a[k]...a[i-1]
bool has_appeared_before(char a[], long k, long i)
{
    for (long j = k; j <= i - 1; j += 1)
        if (a[j] == a[i])
            return true;
    return false;
}

void permute(char a[], long len, long curr)
{
    if (curr == len - 1) {
        cs1010_println_string(a);
        return;
    }
    permute(a, len, curr + 1);
    for (long i = curr + 1; i < len; i += 1) {
        if (!has_appeared_before(a, curr, i)) { // Check here
            swap(a, curr, i);
            permute(a, len, curr + 1);
            swap(a, i, curr);
        }
    }
}
```

Problem Set 27

Problem 27.1

- See code for Approach 2
  - Check if queens placed on rows `0` to `row` threaten each other
  - Call `nqueens` recursively if these queens do not threaten each other
- Identity repetitive work done in the calls to `threaten_each_other_diagonally` and suggest a way to remove the repetitive work

Problem 27.1

```
void nqueens(char queens[], long n, long row) {
    if (row == n-1) {
        if (!threaten_each_other_diagonally(queens, n)) {
            cs1010_println_string(queens);
        }
        return;
    }

    if (!threaten_each_other_diagonally(queens, row)) {
        nqueens(queens, n, row + 1);
    }

    for (long i = row + 1; i < n; i++) {
        swap(queens, row, i);
        if (!threaten_each_other_diagonally(queens, row)) {
            nqueens(queens, n, row + 1);
        }
        swap(queens, row, i);
    }
}
```

Problem 27.1

- The function `threaten_each_other_diagonally` checks whether all queens threatens each other
- But, if we recurse with `nqueens(queens, n, row + 1)`, we already know that the queens on row `0` to `row - 1` do not threaten each other
- **We only need to check if a newly added queen on row `row` threatens any of the queens on the previous row**

Problem 27.2

- Consider the code to generate all permutations of a string from Problem 26.1
- Suppose we restrict the permutations to those where the same character does not appear next to each other.
- Modify the solution to Problem 26.1 to prune away permutations where the same character appears more than once consecutively.
- ***Example*** Consider the string `aabc`
  - We don't want to generate `aabc` , `aacb` , `baac` , `caab` , `bcaa` , `cbaa`
- This transforms the Permutation problem into a search-and-prune problem

Problem 27.2

- ***Key Idea*** Have a check for consecutive characters at three places
  - Before printing the string
  - Before recursively calling `permute`
  - Before fixing a "first character"

Problem 27.2

```c
void permute(char a[], long len, long curr) {
    if (curr == len - 1 && a[curr] != a[curr - 1]) {
        cs1010_println_string(a);
        return;
    }
    if (a[curr] != a[curr - 1]) {
        permute(a, len, curr + 1);
    }
    for (long i = curr + 1; i < len; i++) {
        // Check if the "first character" is same as the character being
        // considered to be put as the next "first character"
        if (!has_appeared_before(a, curr, i) && a[i] != a[curr - 1]) {
            swap(a, curr, i);
            permute(a, len, curr+1);
            swap(a, curr, i);
        }
    }
}
```

Past Year PE2 Questions

General Comments

- It's tough - typical Prof Ooi paper
- Most students probably not expected to go past Question 3

Question 1: TicTacToe

- Given a tic-tac-toe game, determine whether `X` or `O` has won, or neither
- Supposed to be a give-away
- Testing whether you know the syntax and how to loop through a 2-D array
- You need to check
  - All rows
  - All columns
  - Both diagonals
  - … You need to write a lot of loops

Question 1 - TicTacToe [Example]

```
// Check rows whether player has won
for (long i = 0; i < 3; i++) {
    bool won = true;
    for (long j = 0; j < 3; j++) {
        if (board[i][j] != player) {
            won = false;
        }
    }
    if (won) {
        return true;
    }
}
return false;
```

- Do the same for columns and diagonals

Question 1 - TicTacToe [Meme Solution]

- *Note* A solution like this will probably get you penalised

```
bool check_three(char c1, char c2, char c3, char player)
    return c1 == player && c2 == player && c3 == player;

bool has_won(char **board, char player)
    // Check rows
    return check_three(board[0][0], board[0][1], board[0][2], player)
            || check_three(board[1][0], board[1][1], board[1][2], player)
            || check_three(board[2][0], board[2][1], board[2][2], player)

            // Check columns
            || check_three(board[0][0], board[1][0], board[2][0], player)
            || check_three(board[0][1], board[1][1], board[2][1], player)
            || check_three(board[0][2], board[1][2], board[2][2], player)

            // Check diagonals
            || check_three(board[0][0], board[1][1], board[2][2], player)
            || check_three(board[0][2], board[1][1], board[2][0], player);
```

Question 2: Sun

- ***Conjecture*** Any number, except the special case 216, can be represented as the sum $p + t$ where p is a prime and t is a triangle number
- ***Triangle Number*** A triangle number is a number n that, for some x, can be represented as

$$n = \frac{x(x+1)}{2}$$

- The first few triangle numbers are 1, 3, 6, 10, 15, 21
- For any n, there are $O(\sqrt{n})$ triangle numbers less than n
- Goal: Print all combinations of p and t where p is in increasing order
- For full efficiency marks, you are to give an $O(n)$ algorithm

Question 2: Sun [Naive Algorithm]

- Define two functions
  - `is_prime(n)` that checks whether any n is prime
  - `is_triangle(n)` that checks whether any n is a triangle number

```
for i in range [0, n - 1]:
    if (is_prime(i) or i == 0) and (is_triangle(n - i)):
        print i, n - i
```

- The outer loop runs $O(n)$ steps
- Primality checking and triangle number checking both run in $O(\sqrt{n})$
- The algorithm runs in $O(n\sqrt{n})$

Question 2: Sun [Efficient]

- **Key Question:** How many possible combinations of p, t can exist in the output?
- Since there are $O(\sqrt{n})$ possible triangle numbers less than n, the number of combinations must be $O(\sqrt{n})$ as well
- Strategy: **Loop through all triangle numbers**, let each one be t, and check if $n - t$ is a prime number

Question 2: Sun [Efficient]

```
algorithm print_pairs(n):
    // returns i, corresponding to the i-th triangle number that is
    // greater than or equal to n
    i = which_triangle_number_greater_than_or_equal_to(n)

    tri = -1
    do
        tri = triangle(i) // returns the i-th triangle number
        possible_prime = n - tri
        if possible_prime == 0 or is_prime(possible_prime):
            print possible_prime, tri
        decrement i by 1
    while tri >= 1
```

Question 2: Sun [Run-time Analysis]

- We are looping through roughly $\sqrt{n}$ number of triangle numbers
- In the worst case - for each triangle number, we have to do an $O(\sqrt{n})$ primality tests
- The algorithm runs in $O(\sqrt{n} \cdot \sqrt{n}) = O(n)$

Question 3: Replace

- Given a string h, and k number of pairs (s_1, s_2)
  - Iteratively perform "search-and-replace" on h, where each occurrence of s_1 is replaced with s_2 in h, repeat for k number of pairs
- **This problem is similar in intention to Add from Assignment 6** - the algorithm is easy to see, but difficult to manage

Question 3: Replace [Housekeeping]

- Define two helper functions to help us
- `string_equal(h, s, i, j)` that returns true if the substring $h[i] \dots h[j-1]$ is equal to s, false otherwise
  - To make our lives easier, we use `assert` to check that $j - i + 1 = |s|$
- `copy_to(source, dest, dest_i)` that copies `source` to `dest`, starting from index `dest_i` in `dest`
  - Returns the index after the last character copied to `dest`
  - e.g `copy_to("abc", "012345", 1)` modifies `dest` to `"0abc45"` and returns `4`

Question 3: Replace [Algorithm]

```
algorithm replace(h, s1, s2):
    ret = ""
    r = 0

    i = 0
    for i in range[0, |h| - |s1|]
        if string_equal(h_i, s1, i, i + |s1|):
            r = copy_to(s2, ret, r)
            i += |s1|
        else:
            ret[r] = s[i]
            r += 1
            i += 1

    while i < |s|:
        ret[r] = s[i]
        r += 1
        i += 1
```

Question 3: Replace [Run-time Analysis]

- The function `replace` runs roughly n iterations, each iteration checking for substring equality of length k
- Therefore, the run-time is $O(nk)$
- We can achieve $O(n + k)$ by using the ***Knuth-Morris-Pratt (KMP) Algorithm***
  - This is how you get bonus 2 marks for efficiency

Question 4: Soil

- Given a 2-D integer array with the following properties
  - In each row, integers are always increasing
  - In each column, integers are always increasing
  - All integers in the 2-D array are unique
- Print out the (x, y) coordinates of a query q, where q is some integer in the array
- For full efficiency marks, the algorithm must run in $O(m + n)$ time
- A trivial $O(mn)$ solution gets 0 efficiency marks

Question 4: Soil [Exam Strategy]

- If you don't know the trick, it will come down to your problem-solving and pattern-recognition skills
- **If you can't figure out the efficient algorithm, write the trivial algorithm first** to secure correctness, memory management and style marks
  - This applies to all questions

Question 4: Soil [Naive Algorithm]

- Simply loop through all $m \times n$ cells to find the query and print the two indices

```
for i in range[0, m - 1]:
    for j in range[0, n - 1]:
        if land[i][j] == q:
            print i, j
```

- A solution like this will get you 5 out of 11 marks, pretty good.

Question 4: Soil [Efficient]

- Notice that we can use the sorted property of the matrix to make decisions on how to traverse the array
- ***Key Idea*** Start traversal from the **bottom-left** corner
- If the current element is greater than q
  - q must be somewhere in the matrix above, so move up
- If the current element is less than q
  - q cannot exist in that column, so move right
- Obviously, if the current element equals q
  - Then just print the indices
- Stop when we go out-of-bounds of the matrix

Question 4: Soil [Run-time Analysis]

- What is the run-time of the algorithm?
- At any point, we either move up or move right
- Every time we move, we "remove" a row or column from the set of cells that needs to be checked
- If we always move up, we visit m cells
- If we always move right, we visit n cells
- In the worst case, we alternate between moving up and right
  - We visit a total of $O(m+n)$ cells
- The algorithm has run-time of $O(m+n)$
- ***Optional Note*** This algorithm has a name, known as a "Saddleback Search"

Question 5: Substring

- This is a common algorithmic question in coding interviews
- Given a string h of length n, print all possible *ordered subsequences* of length k
  - "Substring" is a misnomer, and implies contiguous elements in h
- ***Example*** Given the string `abcde`, the possible ordered subsequences of length 3 is
  - `abc`, `abd`, `abe`, `acd`, `ace`, `ade`, `bcd`, `bce`, `bde`, `cde`

Question 5: Substring [Algorithm]

- While the question does not state it, **this is a difficult recursion problem**
- ***Strategy*** You just have to be able to see the recursion, no other way to go about it
- This question is to differentiate the A from the A+
  - Don't feel bad if you can't do it even after awhile :-)
  - I was given this as a homework question in higher level algorithm classes

Question 5: Substring [Algorithm]

```
algorithm print_subsequences(s, subseq, k, subseq_i, i):
    if subseq is of length k:
        print subseq
        return
    for j in range [i, |s|]:
        subseq[subseq_i] = s[j]
        print_subsequences(s, subseq, k, subseq_i + 1, j + 1)
```

We can call this algorithm with

```
print_subsequences(s, subseq, k, 0, 0)
```

Assignment 7 Comments

I haven't finished marking yet, however, the document explaining the questions is released here: https://ryanytan.github.io/cs1010_2021_s2_TA/

Assignment 8 Hints

Question 1: Walk

- **Draw out an example and try to find a pattern between cells**
- The problem has a 3-line recursive solution but is *very* slow
- Use an example to figure out how to do the problem iteratively to hit the $O(xy)$ run-time complexity
- Hint: *Dynamic Programming*

Question 2: Maze

- The solution is **recursion with backtracking**
- Make sure to do the backtracking
- You can try Googling for Maze solving algorithms, but none of them contain the "full solution"
  - Trust me, I tried as a student
- My best advice is to **follow the prescribed algorithm exactly** and remember to **backtrack**
- But you still need to do some faffing about with indexing, etc