พีรณัฐ กิตติวิทยากุล

Band Protocol assignment (Software engineer) Summary Report

Problem 1: Boss Baby's Revenge

Concept

This problem is string processing problem. I used C++ for-loop to process the problem. The conditions we need to consider are:

1. If first character is R, Boss Baby initiate shots at the neighborhood kids first. => “Bad Boy”
2. If last character is S, there are shots which have not been revenged => “Bad Boy”
3. Any R can only be used to avenge its prior S, it cannot be used to avenge S that comes after it.
4. But any S can be neutralized by any R after it.

From 1 and 2, we can write exiting conditions from them. From 3 and 4, we can write logic that acts differently. The most concerning condition is when there is an S after R. No more R can be carried to the next iteration, so we need to set the count of R to 0 after do the calculation how many S left after minus all Rs.

Time Complexity

The main part of the process is a single loop that iterates over each character in the string. Every functions except loop takes constant time O(1). Therefore, the time complexity of the function is O(n) where n is the length of the input string.

Problem 2: Superman's Chicken Rescue

Concept

This problem can work with arrayList in Java or vector in C++ since the number of chickens is defined from user input. I also used C++ to solve this problem again. I have done 2 approaches of this problem.

First approach uses divide and conquer concept since we can divide the vector problem in half and solve from smallest problem, then conquer the answer until we got the original question.

The second approach is using two variables to point at index of the vector and iterate until the end of the vector.

Time Complexity

Divide and conquer approach

```cpp
int saved_chicken_number(int k, vector<int> &chicken_pos, int start, int stop) {
    // base case
    if (start == stop) return 1;

    // mid-point
    int m = (start+stop)/2;

    // find maximum chicken number saved from left and right halves
    int left = saved_chicken_number(k, chicken_pos, start, m);
    int right = saved_chicken_number(k, chicken_pos, m+1, stop);

    // find maximum chicken number saved in case we place the roof cover the mid-point
    int max_mid = 1;
    for (int i = m; i >= start; i--)
    {
        int j = m+1;
        if (chicken_pos[i] + k - 1 < chicken_pos[j]) continue;

        while (j < stop && chicken_pos[i] + k - 1 >= chicken_pos[j+1]) j++;
        max_mid = max(max_mid, j - i + 1);
    }

    return max(max(left,right),max_mid);
}
```

For-loop brute force approach

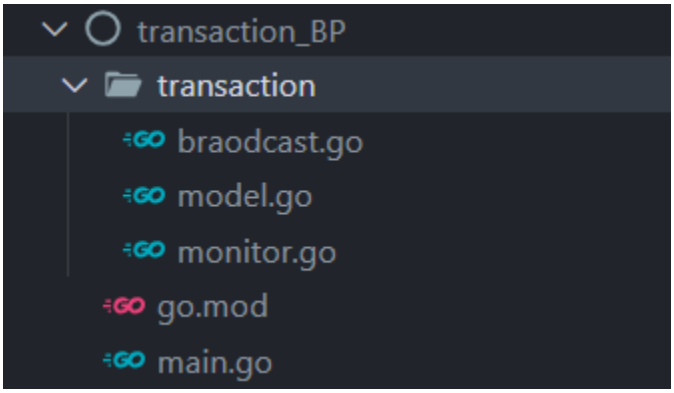
```cpp
int saved_chicken_number(int k, vector<int> &chicken_pos, int start, int n) {
    int max_chicken = 0;
    for (int trav = 0; trav < n; trav++) {
        while (start < n && chicken_pos[start] + k <= chicken_pos[trav]) start++;
        max_chicken = max(max_chicken, trav - start + 1);
    }
    return max_chicken;
}
```

If we look at the function roughly, the second approach looks like it has 2 loops, outer and inner loop, thus it should be n*n => O(n^2) while the first approach which is divide and conquer guarantees to divide in a half until there is 1 element vector is sub-problem, which results in T(n) = 2T(n/2) +O(n). So it is O(nlogn).

In the second approach using brute force, the function iterates on outer loop using ‘trav’ variable, which runs from 0 to n-1. This gives us an O(n) contribution to the complexity. In the inner loop, the ‘start’ variable only increments and never decrements or reassign at any point of program, so it only moves forward, and does not go back to a previous position. The maximum number of iterations for ‘start’ can be at most n (each chicken position is checked only once). Thus, the overall time complexity is O(n + n) = O(2n) = O(n).

So the second approach is more efficient for this type of problem.

Problem 3: Transaction Broadcasting and Monitoring Client



The project is in transaction_BP directory. The client module is implemented in transaction package with

- model.go contains important structs in this project
- broadcast.go contains broadcasting handling function
- monitor.go contains monitoring function and get status function which is helper function

The URL and paths to send requests are already stored in both broadcast.go and monitor.go.

```
12  func (client *Client) BroadcastTransaction(request TransactionRequest) (string, error)
13      url := "https://mock-node-wgqbnxruha-as.a.run.app/broadcast"
```

```
12  func (client *Client) CheckTransactionStatus(txHash string) (string, error) {
13      url := fmt.Sprintf("https://mock-node-wgqbnxruha-as.a.run.app/check/%s", txHash)
```

This is not the best practice since we can assign domain name or url to one variable and use anywhere, but since there are only two handling function and two routes to implement, I place them there to reduce code compactness in main.go.

How to use it

```
12      // Transaction Data Sample
13      reqBody := transaction.TransactionRequest{
14          Symbol: "BTC",
15          Price:  50000,
16      }
```

If you want to broadcast transaction, only thing you need to do is changing the payload data to your data, both symbol and price and press run or go to your project and type:

go run main.go

You can also select how long you want the program to wait before issuing another request by changing second parameter of MonitorTransaction() function.

```
27      // Monitor the Transaction Status
28      status, err := client.MonitorTransaction(txHash, 5*time.Second)
```

(In this case, it is 5 seconds.)

Output

The program will keep sending requests and notify user until the transaction status is either "CONFIRMED", "FAILED" or "DNE".

```
[Running] go run "d:\working-project\transaction_BP\main.go"
Transaction broadcasted with hash: ed60fbe926cc03179aae6defdab9ea429bd4a045e69ca33d55563e14f8d2d1e8
Current status: PENDING
Current status: PENDING
Current status: PENDING
Current status: PENDING
Current status: CONFIRMED
Current status: CONFIRMED
Transaction finalized with status: CONFIRMED
```