CSCI270 Homework 5

Jacob Ma

October 20, 2022

Problem 1

Imagine that you are playing a game resembling Chutes & Ladders with your sibling. It consists of $n + 1$ squares, labeled from 0 to n. You start on square 0. When you reach square n, you win. When it is your turn, you roll a 6-sided die, and move that number of squares forward. There are two types of special squares:

- If square i is a ladder, then it comes with a destination square $d[i] > i$ where the ladder leads. You immediately climb the ladder, and end up on square $d[i]$.
- If square i is a chute, you instantaneously die when you end up on square i by rolling a die. (However, if you climbed to square i using a ladder, you do not die.)
- Not all squares are special — some are neither chutes nor ladders.

When you reach a square i by climbing there, you do not make another move — that is, if you got to $i = d[j]$, and i is the start of a ladder, too, you do not follow that ladder, but instead stay there and roll a die on your next move. And if i is a chute, then — as stated above — you do not die.
To avoid annoying special cases, neither square 0 nor square n is a chute or a ladder, and no ladder leads to an undefined square $d[i] > n$.
You have practiced rolling dice in order to gain an advantage, and you have gotten so good at it that you can roll exactly the number you want. Using this skill, you now want to win the game. To do so, you want to calculate the sequence of die rolls that gets you to square n (the winning square) as quickly as possible. If it is impossible to get there (for instance, because all squares except $0, n$ are chutes), then you also want to diagnose that.
Give and analyze a polynomial-time algorithm for solving this problem. If you are following an approach based on recurrences, just the recurrence is not enough — you have to give a full pseudo-code implementation with correctness proof and running time analysis. (The running time analysis will likely be short.)

The algorithm is as follows: (Seems long since it is more a tabular implementaion, but can just read comments which outlines what I am doing)

```
    LinkedList<int> findPath(int[] board){
        int n = board.length - 1; // According to the definition given in the problem, consists of n+1 squares
        int[][] A = new int[n+1][2]; // Store the OPT(i) for any integer i in two cases, [0] enabling ladders,
            [1] disabling ladders
        int[][] B = new int[n+1][2]; // Store the path chosen along the way in two cases
```

```
Map<int, ArrayList<int>> ladder; // Store for current destination point i, what are the starting points of the ladders reaching i
Initialize the ladder from 0 to n with an empty ArrayList.
// Manually Initialize the Value from i = 0 to 6
A[0][0] = A[0][1] = 0; B[0][0] = B[0][1] = 0;
for (int i = 1; i <= 6; i++){
    if (i is chute) {A[i][0] = A[i][1] = n + 1; B[i][0] = B[i][1] = 0;}
    else {A[i][0] = 1; A[i][1] = 1; B[i][0] = i; B[i][1] = i;}
}

// Start Recurrence
for (int i = 6; i <= n; i++){
    if (board[i] == chute){
        minLadder = n + 1;
        minLadderStart = 0;
        for (int j = 0; j < ladder.get(i).length; j++){ // Iterate through all possible ladders reaching current i
            if (A[ladder.get(i)[j]][0] <= minLadder){
                minLadder = A[ladder.get(i)[j]][0];
                minLadderStart = j;
            }
        }
        if (n + 1 <= minLadder){ // Have to walk onto a chute, must die
            A[i][0] = n + 1; A[i][1] = n + 1; // Bigger than the largest possible steps we can obtain, which is n
            B[i][0] = 0; B[i][1] = 0; // Use 0 to indicate not reachable
        }
        else{ // Can walk till the chute with a ladder, the number of steps reaches the end of the ladder equals number of steps reaches the start of the ladder
            A[i] = minLadder;
            B[i] = minLadderStart - i; // Negative length of the ladder
        }
    }

    else { // i is a ladder or normal square
        if (board[i] == ladder){
            // Add the current ladder into the ladder map
            int dest = The Destination of Current Ladder Starting from i;
            ladder.get(dest).add(i);
        }

        // Case 1: Calculating A[i][0] - OPT enabling ladder
        minWalk = n + 1;
        minWalkStart = 0;
        for (int j = i-6; j < i; j++){ // Iterate all 6 possibilities directly rolling to i
            if (1 + A[j][0] <= minWalk){
                minWalk = 1 + A[j][0];
                minWalkStart = j;
            }
        }
        minLadder = n + 1;
        minLadderStart = 0;
        for (int j = 0; j < ladder.get(i).length; j++){ // Iterate through all possible ladders reaching current i
```

```java
                    if (A[ladder.get(i)[j]][1] <= minLadder){
                        minLadder = A[ladder.get(i)[j]][1]; // Could use ladder in i indicates we could not use ladder previously
                        minLadderStart = j;
                    }
                }

                if (minWalk <= minLadder){ // We prefer using rolling if minWalk == minLadder enabling further laddering
                    A[i][0] = minWalk;
                    B[i][0] = i - minWalkStart;
                }
                else{
                    A[i][0] = minLadder;
                    B[i][0] = minLadderStart - i;
                }

                // Case 2: Calculating A[i][1] - OPT disabling ladder
                minWalk = n + 1;
                minWalkStart = 0;
                for (int j = i-6; j < i; j++){ // Iterate all 6 possibilities directly rolling to i
                    if (1 + A[j][0] <= minWalk){
                        minWalk = 1 + A[j][0];
                        minWalkStart = j;
                    }
                }
                A[i][1] = minWalk;
                B[i][1] = i - minWalkStart;
            }
        }
        // Trace solutions
        if (A[n][0] >= n+1 && A[n][1] >= n+1){
            LinkedList<int> result = new LinkedList<int>();
            return result; // No possible solution, return an empty linked list
        }
        else {
            LiknedList<int> result = new LinkedList<int>();
            int i = n;
            boolean enableLadder = (A[n][0] <= A[n][1]);
            while (i > 0){ // Backtrace all the steps
                if (enableLadder){
                    if (B[i][0] < 0){ // Used ladder in this step
                        i += B[i][0]; // Return to the start point of ladder
                        enableLadder = false;
                    }
                    else { // Used roll in this step
                        result.addFirst(B[i][0]);
                        i -= B[i][0];
                    }
                }
                else {
                    result.addFirst(B[i][1]);
                    i -= B[i][1];
            }
```

```
                return result;
            }
        }
```

Correctness Proof. There are two part of the algorithm: Fining the $\min(A[i][0], A[i][1])$, which is the minimum step to get to i ; and tracing how we rolled along the road. Thus, we prove them separately. □

Lemma 0.0.1

$A[i][0] = \text{OPT}[i]$ for $0 \leqslant i \leqslant n$ while i enables ladders, $A[i][1] = \text{OPT}[i]$ for $0 \leqslant i \leqslant n$ while i disables ladders.

Proof. We prove by strong induction.

Base Case: $i = 0$ to $i = 6$, $A[i][0] = A[i][1] = \text{OPT}[i]$. To $i = 0$, 0 steps is needed. To $1 \leqslant i \leqslant 6$, just needs to roll once, holds for both enabling and disabling ladders.

Induction Hypothesis: for an arbitrary $k \leqslant i - 1$, the lemma above holds.

Induction Step: We have three big cases, and for each big case we have several sub cases.

(1) **Case 1:** i is a chute. Two sub cases:

(a) There aren't ladders reaching i. Thus, i will not be reachable. We have $A[i][0] = A[i][1] = n + 1$ and $B[i][0] = B[i][1] = 0$ indicating this is not possible. Thus, holds for both enabling and disabling ladders for i.

(b) There are ladders reaching i. Thus, the $OPT[i][0] = OPT[i][1]$ will be the minimum out of all the $\text{OPT}[\text{Starting Points of Ladders to } i]$. Since all the starting points j must be $< i$, thus, $A[j] = \text{OPT}[j]$ for all j. Thus,

$$\begin{aligned}\text{OPT}(i) &= \min(\forall\ \text{OPT}[\text{Starting Points } j]) \\ &= \min(\forall \min(A[\text{Starting Points } j][0], A[j][1])) \\ &= A[i][0] = A[i][1]\end{aligned}$$

Thus, holds for both enabling and disabling ladders for i.

(2) **Case 2:** i is a ladder.

(a) We first consider enabling the ladder. The optimal way to get to current i must be chosen from either rolled here within 6 steps, or a ladder directly lead here. Thus, we know

$$\text{OPT}(i) = \min(\text{OPT}(i-1) + 1, \cdots, \text{OPT}(i-6) + 1, \text{OPT}(j_1), \cdots, \text{OPT}(j_k))$$

where j_1 to j_k are all possible starting point of a ladder connecting to i.

Note that, if we are using a ladder, the OPT for j_1 to j_k will apply to the situation that disabling ladders since we cannot use ladders in consecutive times. Thus we have

$$\begin{aligned}\text{OPT}(i) &= \min(A[i-1][0] + 1, \cdots, A[i-6][0] + 1, A[j_1][1], \cdots, A[j_k][1]) && \text{By IH} \\ &= A[i][0] && \text{By Algorithm}\end{aligned}$$

Thus holds while enabling the ladder.

(b) Consider disable the ladder. The optimal way to get the current i can only be rolling a dice till here. Since we are disabling the ladder in i, we will be able to enable the ladder in the previous step. Thus,

$$\begin{aligned}\text{OPT}(i) &= \min(\text{OPT}(i-1)+1, \cdots, \text{OPT}(i-6)+1) \\ &= \min(A[i-1][0], \cdots, A[i-6][0]) \qquad \text{By IH} \\ &= A[i-1][1] \qquad \text{By algorithm}\end{aligned}$$

Thus, holds while disabling the ladder.

(3) **Case 3:** i is neither a ladder nor chute. This situation will be identical to Case 2 since we are only considering how to get to i instead of starting from i. Thus we do not really care if i is a ladder or not. The only difference is that in Case 2 when i is a ladder, we need to add the i to the map at location j, where j is the destination of the ladder starting from i. This will enable further use of the ladder. But the proof will be identical to Case 2, thus holds.

□

Lemma 0.0.2

The algorithm will terminate.

Proof. This is trivial in the first section while establishing A and B, since from line 1 to line 83 are all for loops with finite number of iterations. Thus it will be suffices to show the back trace algorithm successfully find the path along the way and terminates, which is shown below. □

Lemma 0.0.3

The back trace algorithm will find the correct path and terminates.

Proof. The algorithm will firstly terminate. Since $j = n$ is a finite number, and in every iteration, j will be minus a positive number $B[j][0]$ or $B[j][1]$, thus it will definitely terminate in finite steps, in fact at most n steps.
If $B[j][0] = B[j][0] = 0$, it indicates that $A[n][0] \geqslant n+1$ and same for $A[n][1]$, meaning the algorithm does not have a legal output, will immediately terminates.
Since we have shown that $A[j][0]$ as well as $A[j][1]$ correctly records the optimal number of steps to get to i, $B[j][0/1]$ also records the number of rolls from the previous step to the current step j in different situations. We know that for the whole array of length n, we will enable the use of the ladder, thus we will back trace starting from $B[j][0]$. The algorithm will shift to adds $B[j][1]$ if we used the ladder in the previous step, and choose $B[j][0]$ otherwise, which is the optimal solution shown in the previous lemma. Thus, by appending these numbers at the front of the Array List, we will get the correct sequence. □

Final Step: Note that we are discussing the whole Chutes & Ladders game with size n while enabling the use of the ladder at the very beginning, thus back tracing from $A[n][0]$ will correctly outputs the sequence according to the lemmas above.

Runtime Analysis. From line 1 to line 5 are constant runtime. Line 6 will be a $\Theta(n)$ runtime initializing the map. The initialization from line 7 to line 13 will take constant time since we know there will be 6 iterations with constant operations within all of them.

From line 16 to line 82, the big for loop will iterate $n - 6 + 1$ times, which is $O(n)$ iterations.

In the first if block from 17 to 35, we have a for loop. However we know that it is actually iterating through all the ladders, which takes at most $O(n)$ iterations in total, with $O(\log n)$ runtime inside if considering the get method of map as $O(\log n)$ operation. Thus is for loop takes $O(n \log n)$ in total. Line 26 to line 34 are all constant time operations. Thus the 17 to 34 line takes $O(n \log n)$ after iterating from $i = 6$ to $i = n$.

Consider the else block. 37 to 41 takes $O(\log n)$. Line 44 to line 82 are all constant operations, with two for loops with constant iterations.

Thus, from line 16 to line 82, it will take at most $O(n \log n)$.

From line 84 to line 107, the worst case is iterate through all n and operate the constant compare and append operations, which takes $O(n)$ in total.

Thus, the algorithm takes $O(n \log n)$. □

Problem 2

In class, we saw the algorithm for computing Edit Distance. We motivated it by spell checking, and also briefly mentioned that it is a reasonable (though far from perfect) approach for plagiarism detection. One thing that it is missing is that deletions often happen in chunks. When someone copies text, they may add or leave out whole words, sentences, or paragraphs. Even when typing, we might be more likely to be adding or leaving out multiple letters.

Here, we will consider a refined cost model. You will still be comparing a string x of length n with a string y of length m. Replacing a single character still costs B. But now, we have that for each k, the deletion of k characters in a row in a position costs $A + c \cdot (k - 1)$, and the insertion of any k characters in a row in a position costs $A + c \cdot (k - 1)$. So the previous model was the special case when only $k = 1$ was allowed, but now, we allow the operation for any k.

The goal is now again to find a minimum-cost sequence of operations that turns x into y, except that now, there are more different operations available.

Give and analyze a polynomial-time algorithm for solving this problem. If you are following an approach based on recurrences, just the recurrence is not enough — you have to give a full pseudo-code implementation with correctness proof and running time analysis. (The running time analysis will likely be short.)

Algorithm:

```
    Stack<Operation> findCheapestOperations(A, B, c, x, y){
        enum Operation{Replace, Insert, Delete};
        Stack<Operation> operations;
        c = min(A,c); // Automatically choose delete word by word or in a row by minimizing c
        int m = x.size(); int n = y.size();
        int[][] a = int[m][n]; Initialize it with all starting value MAX_INTEGER
        int[][] kSeq = int[m][n]; Initialize them with all zeroes
        Operation[][] seq = int[m][n]; Initialize it with all starting value Replace
        // Initialize all base cases
        for (int i = 1; i <= m; i ++) a[i][0] = A + (i - 1) * c; seq[i][0] = Insert;
        for (int j = 1; j <= n; j ++) a[0][j] = A + (j - 1) * c; seq[j][0] = Delete;
        a[0][0] = 0;
        // Exhaust all possibilities for each pair (i,j)
        for (int i = 1; i <= m; i ++) {
            for (int j = 1; j <= n; j ++) {
                int kTop = -1;
                for (int k = 1; k <= min(m,n); k++){
                    if (x[i] == y[i]){
                        a[i][j] = min(a[i][j], a[i-1][j-1], a[i-k][j]+A+(k-1)*c, a[i][j-k]+A+(k-1)*c);
                        if (a[i][j] got updated to a smaller value) {kTop = k;}
                    }
                    else {
                        a[i][j] = min(a[i][j], a[i-1][j-1]+B, a[i-k][j]+A+(k-1)*c, a[i][j-k]+A+(k-1)*c);
                        if (a[i][j] got updated to a smaller value) {kTop = k;}
                    }
                }
                // seq[m][n] = Replace; by default
                if (a[i][j] == a[i-kTop][j]+A+(kTop-1)*c){ seq[m][n] = Delete; kseq[m][n] = kTop;}
                if (a[i][j] == a[i][j-kTop]+A+(kTop-1)*c){ seq[m][n] = Insert; kseq[m][n] = kTop;}
            }
        }
        // Backtrace Operations
        int i = m-1; int j = n-1;
        while (i > 0 || j > 0){
            if (seq[i][j] == Replace){i--; j--; operations.add(Replace);}
            if (seq[i][j] == Insert){i -= kseq[i][j]; operations.add(Insert for kseq[i][j] times);}
            if (seq[i][j] == Delete){j -= kseq[i][j]; operations.add(Delete for kseq[i][j] times);}
        }
    }
```

Correctness Proof. Firstly, we want to prove $a[m][n] = \text{OPT}(n, m)$, and the algorithm returns a correct sequence of operations.

We conduct an induction on $i + j$, to ensure that whenever we need a value, it has a smaller "index" in the induction proof.

Base Case: $i + j = 0 \implies i = 0, j = 0 \implies$ algorithm sets $a[0][0] = 0 = \text{OPT}(0, 0)$. The algorithm will only return $seq[0][0]$ which is replace (means alignment with/without replacement based on situation), which is the only possibility with two size 1 string.

Induction Hypothesis: Assume that a[i'][j'] = OPT(i',j') whenever i'+j' < i+j [strong induction]

Induction Step: We prove $a[i][j] = \text{OPT}(i, j)$.

(1) **Case 1:** $x[i] = y[j]$, and $i > 0$, $j > 0$.

$$a[i][j] = \min\Big(a[i-1][j-1], \underbrace{a[i-1][j] + A, a[i-2][j] + A + c, \cdots, a[0][j] + A + c \cdot (i-1)}_{i \text{ possibilities}},$$

$$\underbrace{a[i][j-1] + A, a[i][j-2] + A + c \cdot 2, \cdots, a[i][0] + A + c \cdot (j-2)}_{j \text{ possibilities}}\Big)$$

$$= \min\Big(\text{OPT}[i-1][j-1], \underbrace{\text{OPT}[i-1][j] + A, \text{OPT}[i-2][j] + A + c, \cdots, \text{OPT}[0][j] + A + c \cdot (i-1)}_{i \text{ possibilities}},$$

$$\underbrace{\text{OPT}[i][j-1] + A, \text{OPT}[i][j-2] + A + c \cdot 2, \cdots, \text{OPT}[i][0] + A + c \cdot (j-2)}_{j \text{ possibilities}}\Big) \qquad \text{By Induction Hypothesis}$$

$$= \text{OPT}(i,j) \qquad \text{By Recurrence Relationship}$$

Because the $OPT(i,j)$ is get by one of the minimum operations above, which outputs a correct operations sequence by induction hypnosis, the new operation we are adding will not influence the correctness of previous operations, and leads to the current operation sequence. Thus it will outputs a correct operation sequence.

(2) **Case 2:** $x[i] \neq y[j]$, and $i > 0$, $j > 0$.

$$a[i][j] = \min\Big(a[i-1][j-1] + B, \underbrace{a[i-1][j] + A, a[i-2][j] + A + c, \cdots, a[0][j] + A + c \cdot (i-1)}_{i \text{ possibilities}},$$

$$\underbrace{a[i][j-1] + A, a[i][j-2] + A + c \cdot 2, \cdots, a[i][0] + A + c \cdot (j-2)}_{j \text{ possibilities}}\Big)$$

$$= \min\Big(\text{OPT}[i-1][j-1] + B, \underbrace{\text{OPT}[i-1][j] + A, \text{OPT}[i-2][j] + A + c, \cdots, \text{OPT}[0][j] + A + c \cdot (i-1)}_{i \text{ possibilities}},$$

$$\underbrace{\text{OPT}[i][j-1] + A, \text{OPT}[i][j-2] + A + c \cdot 2, \cdots, \text{OPT}[i][0] + A + c \cdot (j-2)}_{j \text{ possibilities}}\Big) \qquad \text{By Induction Hypothesis}$$

$$= \text{OPT}(i,j) \qquad \text{By Recurrence Relationship}$$

Operation sequence is correct the same as above case.

(3) **Case 3:** $i = 0$. The algorithm writes a[0][j] = A * j = OPT(0,j). Operation is a big insertion operations, which is optimal.

(4) **Case 4:** $j = 0$. The algorithm writes a[i][0] = A * i = OPT(i,0). Operation is a big deletion operations, which is optimal.

Lemma 0.0.4

The algorithm terminates.

Proof. The only while loops is form line 34 to line 38. In each iterations, either i or j or both will be minuses a number $\geqslant 1$ since the seq only contains Replace, Insert, Delete. Thus, the while loop will at most have $\max(m,n)$ iterations, which is a linear $O(n)$ algorithm. □

Final Step: Since the algorithm terminates and we have $m, n > 0$, apply m, n to the proof above, the algorithm will yields $a[m][n] = \text{OPT}(m, n)$ and outputs correct operation sequence. □

Runtime Anaylisis. All the operations from line 1 to line 5 are $O(1)$ operations. Both initialization on line $6-8$ will take $O(mn)$ time. Initialization from line 9 to line 12 takes to for loop which has $m+n$ iterations, each has a constant time operation. Thus, $O(m+n) = O(n)$.

Consider the nested for loop from line 13 to line 29, in the inner-most for loop, all the operations are just $O(1)$ operations. The three while loops has $m \cdot n \cdot \min(m, n)$ in total, indicating the overall runtime would be $\min\left(O\left(m^2 n\right), O\left(n^2 m\right)\right)$. Both are polynomial.

The backtracing operations would take at most $m+n$ iterations, with constant operations in each iterations. Thus $O(m+n)$.

Thus, we know that $O\left(m^2 n\right)$ or $O\left(n^2 m\right)$ would dominate others. So the overall runtime would be $O\left(m^2 n\right)$ if $m \leqslant n$, $O\left(n^2 m\right)$ if $n \leqslant m$. □