Homework 5

Winston (Hanting) Zhang

Tuesday, October 18, by 23:00

General Instructions. The following assignment is meant to be challenging, and we anticipate that it will take most of you at least 10–15 hours to complete, so please allow yourself plenty of time to work on it. We highly recommend reading the entire assignment right away — you never know when inspiration will strike. Please provide a formal mathematical proof for all your claims, and present runtime guarantees for your algorithms using asymptotic (big-$O/\Omega/\Theta$) notation, unless stated otherwise. You may assume that all basic arithmetic operations (multiplication, subtraction, division, comparison, etc.) take constant time.

Collaboration. Please carefully check the collaboration policy on the course website. When in doubt, ask an instructor.

Consulting outside sources. Please carefully check the policy regarding outside sources on the course website. Again, when in doubt, ask an instructor.

Submission. Homework submission will be through the Gradescope system. Instructions and links have been provided through the course website and Piazza. The only accepted format is PDF. Starting with this homework, only typed solutions will be accepted, and we highly recommend producing your solutions using LaTeX (the text markup language we are also using for this assignment).

Recommended practice problems (do not hand in): KT Problems 6.2, 6.4, 6.6, 6.8, 6.11, 6.13, 6.16, 6.20, 6.24

Problem 1. [15]

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.)

Input: The board is given as a vector of integers $a_0, \ldots, a_n$, where $a_i = -1$ denotes a chute, $a_i = 0$ denotes a non-special block, and $a_i => i$ denotes a ladder from i to a_i.

Output: A vector of integers $r_1, \ldots, r_k$ – representing the shortest sequence of rolls that reaches square n.

Code:

```
#include <bits/stdc++.h>
using namespace std;

vector<int> chutes_and_ladders(vector<int> board) {
    int n = board.size() - 1;
    vector<int> ans;
    vector<pair<int, int>> prev(n+1, {-1, 0});
    vector<int> dp(n+1, INF);
    dp[0] = 0;
    // dp[i] represents the minimum number of rolls to get to i without dying
    for (int i = 0; i < n; ++i) {
        for (int r = 1; r <= 6; ++r) {
            if (i + r <= n) { // careful, don't roll out of bounds
                if (board[i + r] == 0 && dp[i] + 1 < dp[i + r]) { // update if non-special
                    dp[i + r] = dp[i] + 1;
                    prev[i + r] = {i, r};
                } else if (board[i + r] > 0 && dp[i] + 1 < dp[board[i]]) { // update if ladder
                    dp[board[i]] = dp[i] + 1;
                    prev[board[i]] = {i, r};
                }
            }
        }
    }
    // rebuild rolls
    int prev_ = prev[n].first;
    int roll = prev[n].second;
    while (prev_ != -1) { // while a previous step exists
        ans.push_back(roll);
        if (prev_ == 0) // if prev_ is 0 we're done, reverse and return
            return reverse(ans, ans.begin(), ans.end());
        prev_ = prev[prev_].first;
        roll = prev[prev_].second;
    }
    // if there is no previous step, then we return failure
    return vector<int>();
}
```

Proposition 1. *`chutes_and_ladders` outputs the shortest sequence of rolls that reaches square n.*

Proof. **Correctness:** We define OPT(i) to be the minimum number of rolls it takes to reach square i. We agree that OPT$(i) = \infty$ if it is not possible to reach square i without dying. Now consider all possible ways to reach some square i. Either we land on i through a direct roll from squares $\{i-6, i-5, \ldots, i-1\}$, or we arrive through a ladder that ends on i. Denote all ladder squares that end on i by end(i). With this analysis we can now write a reccurence for the optimal number of rolls it takes to reach square i:

$$\text{OPT}(i) = 1 + \min_{j \in \{i-6,\ldots,i-1\} \cup \text{end}(i)} \text{OPT}(j)$$

To prove the correctness of our algorithm, note that the given code can be split into two distinct parts. The first part is the dynamic programming which we claim computes OPT(i). The second part retraces a vector of previous data to recover the shortest sequence of rolls. Hence for correctness, we only need to show that the first part computes $dp[i] = \text{OPT}(i)$ correctly.

Indeed, we proceed by induction on n. The base case $k = 0$ is trivial, as we simply have $dp[0] = 0 = \text{OPT}(0)$.

Now assume for the sake of induction that we have that $dp[j] = \text{OPT}(j)$ for all $1 \leq j \leq k-1$. We want to show that $dp[k] = \text{OPT}(k)$. (We use k to avoid clashing with i in the code.)

Indeed, let us consider all possible instructions within our program that assign a value to $dp[k]$. There are three places we make an assignment to dp. The first is the initialization to infinity. Then subsequently, we have two assignments in the non-special and ladder cases which occur if and only if $i + r = k$ or $i + r \in \text{end}(k)$ – we can justify the "if and only if" by noting that these are the only possible assignments that can affect $dp[k]$, and our loops attempt to make every possible assignment, so any assignment that can be made must be made. Furthermore, note that each assignment to dp is wrapped in an if statement that checks if the new proposed value is less than the current one, in other words, taking the minimum between the new and old value. Hence, we make assignments for $dp(i + r) = \min(dp[i] + 1, dp[i + r])$ and $dp[\text{board}[i]] = \min(dp[i] + 1, dp[\text{board}[i]])$. Now let's look through the notation: $i + r = k \iff i \in \{k-6, \ldots, k-1\}$ and $\text{board}[i] = k \iff i \in \text{end}(k)$. Rewriting with these equivalences, we are computing

$$dp[k] = \min(\infty, dp[k-6], \ldots, dp[k-1], \min_{i \in \text{end}(k)}(dp[i])) + 1.$$

Note that $\{k-6, \ldots, k-1\}$ and end(k) are all less than k by construction, so our induction hypothesis guarantees that $dp[k-r] = \text{OPT}(k-r)$, $1 \leq r \leq 6$ and $dp[i] = \text{OPT}(i)$, $i \in \text{end}(k)$. Then from above we have:

$$\begin{aligned} dp[k] &= \min(\infty, dp[k-6], \ldots, dp[k-1], \min_{i \in \text{end}(k)}(dp[i])) + 1 \\ &= 1 + \min_{j \in \{i-6,\ldots,i-1\} \cup \text{end}(i)} dp(j) \\ &= 1 + \min_{j \in \{i-6,\ldots,i-1\} \cup \text{end}(i)} \text{OPT}(j) \\ &= \text{OPT}(k) \end{aligned}$$

Thus our induction is complete. We conclude that at the end of execution, $dp[n]$ will contain the minimum number of rolls it takes to get to square n.

Termination: in the first part, we run a finite amount of iterations, which terminates. In the retracing step, by construction the pointers get smaller – i points to some j such that $j < i$. At some point we will retrace to $i = 0$, so this terminates as well.

Runtime: Refer to the code above. All of the initialization in the beginning, setting up vectors, etc, is $O(n)$. The main body consists of two for loops, the outer doing n iterations and inner doing exactly 6. The logic inside the loops consist only of comparisons and assignments, which runs in $O(1)$. Hence the runtime for the main body is $O(6n) = O(n)$. In the last part of the algorithm, we trace back with our previous pointers. By construction, this trace back visits at most all of the n squares, so this section also runs in $O(n)$. Hence the total runtime is just $O(n)$.

Output: We claim that the output is correct because it maintains the invariant that for every i, prev[i] consists of some square j and a roll r such that rolling r at j is the optimal last roll that reaches square i. This is clearly true, since the code only updates prev[i] when we find a better minimum for $dp[i]$. Thus prev[i] always maintains the best known roll to reach i from the dp. Thus when we reconstruct the rolls, at each trace back we are just taking the optimal roll until we hit square 0. Once we terminate, we simply end up with the best sequence of rolls from 0 to n (backwards, so we have to reverse it). □

Problem 2. [15]

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.)

Input: The strings x and y, as well as the values of A, B, c.

Output: An integer and vector of steps. The integer represents the minimum cost to transform x into y. The vector lists the necessary steps needed to perform the transformation.

Code:

```
#include <bits/stdc++.h>
using namespace std;

typedef pair<pair<int, int>, string> step;

pair<int, vector<string>> min_distance(string x, string y, int A, int B, int c) {
    int n = x.length();
    int m = y.length();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
    vector<vector<step>> prev(n + 1, vector<step>(m + 1, {{-1, -1}, "none"}));

    for (int i = 1; i <= n; ++i) {
        if (A >= c) {
            dp[i][0] = A + c * (i - 1);
            prev[i][0] = {{0, 0},
                "delete␣at␣" + to_string(0) + ".." + to_string(i - 1)};
        } else {
            dp[i][0] = A * i;
            prev[i][0] = {{i - 1, 0},
                "delete␣at␣" + to_string(i - 1)};
        }
    }
    for (int j = 1; j <= m; ++j) {
        if (A >= c) {
            dp[0][j] = A + c * (j - 1);
            prev[0][j] = {{0, 0},
                "insert␣at␣" + to_string(0) + ".." + to_string(j - 1)};
        } else {
            dp[0][j] = A * j;
            prev[0][j] = {{0, j - 1},
                "insert␣at␣" + to_string(0) + ".." + to_string(j - 1)};
        }
    }

    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            pair<int, int> min_ij = {INF, INF};
            int min_cost = INF;
            // must be either "none", "insert", "delete", or "substitute"
            string operation = "";
            int cost = INF;
            for (int k = 1; k <= i; ++k) {
                cost = dp[i - k][j] + A + c * (k - 1);
                if (cost < min_cost) {
                    min_ij = {i - k, j};
                    min_cost = cost;
                    operation = "delete␣at␣" +
                        to_string(i - k) + ".." + to_string(i - 1);
                }
            }
            for (int k = 1; k <= j; ++k) {
                cost = dp[i][j - k] + A + c * (k - 1);
                if (cost < min_cost) {
                    min_ij = {i, j - k};
                    min_cost = cost;
                    operation = "insert␣at␣" +
                        to_string(j - k) + ".." + to_string(j - 1);
                }
            }
            cost = dp[i - 1][j - 1] + (B ? x[i - 1] == y[j - 1] : 0);
            if (cost < min_cost) {
                min_ij = {i - 1, j - 1};
                min_cost = cost;
                operation = "substitute␣" +
                    to_string(i - 1) + "␣->␣" + to_string(j - 1);
            }
            dp[i][j] = min_cost;
            prev[i][j] = {{min_ij.first, min_ij.second}, operation};
        }
    }

    vector<string> ans;
    int prev_x = prev[n][m].first.first;
    int prev_y = prev[n][m].first.second;
    string prev_descr = prev[n][m].second;
    // if we hit (-1, -1), then the previous iteration was
    // (0, 0) and we're done
    while (prev_x != -1 && prev_y != -1) {
        if (prev_descr != "none")
            ans.push_back(prev_descr);
        prev_descr = prev[prev_x][prev_y].second;
        int temp_x = prev_x;
        prev_x = prev[temp_x][prev_y].first.first;
        prev_y = prev[temp_x][prev_y].first.second;
    }
    reverse(ans.begin(), ans.end());
    return {dp[n][m], ans};
}
```

Proposition 2. *`min_distance` outputs the minimum cost sequence of operations that transforms x into y.*

Proof. **Correctness:** We define OPT(i,j) to be the minimum cost sequence of operations it takes to transform $x[0..i)$ into $y[0..j)$ (not inclusive at the endpoint, we agreed that $x[i..i)$ is the empty substring). The final answer we need is OPT(n,m). As we did for the original simpler version of this problem, for each subproblem, we can think in terms of alignment and conclude that there are three cases with their own subproblems:

1. $x[i]$ aligns with $y[j]$: $x[0..i-1]$ aligns $y[0..j-1]$ **optimally**.
2. For some k, $x[i-k..i]$ all align with blank: $x[0..i-k]$ aligns with $y[0..j]$ **optimally**.
3. For some k, $y[j-k..j]$ all align with blank: $x[0..i]$ aligns with $y[0..j-k]$ **optimally**.

Since these are all the subproblems we have, we can write a recurrence relation for OPT(i,j). There are 3 types of base cases. First, clearly we have OPT$(0,0)=0$. If $j=0$, then we are trying to match $x[0..i]$ to an empty substring, so the best we can do is just to delete everything for a cost of OPT$(i,0)=\min(A+c(i-1),Ai)$. Similarly if $i=0$, then we are matching an empty string to $y[0..j]$, so we just insert everything for a cost of OPT$(0,j)=\min(A+c(j-1),Aj)$. Now suppose $i,j>0$, then define the minimum for the second and third cases as:

$$\begin{aligned}\text{OPT}_m(i,j) &= \text{if } x[i]\neq y[j] \text{ then } B \text{ else } 0+\text{OPT}(i-1,j-1)\\ \text{OPT}_x(i) &= \min_{0\le k<i}(A+c(k-1)+\text{OPT}(i-k,j))\\ \text{OPT}_y(j) &= \min_{0\le k<j}(A+c(k-1)+\text{OPT}(i,j-k))\end{aligned}$$

Then we have
$$\text{OPT}(i,j)=\min(\text{OPT}_x(i),\text{OPT}_y(j),\text{OPT}_m(i,j)).$$

(Note this is really just a recurrence on OPT(i,j) – having OPT$_m(i,j)$, OPT$_x(i)$ and OPT$_y(j)$ just clears up the clutter.)

To prove the correctness of our algorithm, note that the given code can be split into two distinct parts. The first part is the dynamic programming which we claim computes OPT(i,j). The second part retraces a map of previous data to recover the shortest sequence of operations. Hence for correctness, we only need to show that the first part computes $dp[i][j]=\text{OPT}(i,j)$ correctly.

Indeed, we proceed by strong induction on $i+j$. The base case $i+j=0$ is trivial, as then we must have $i=j=0$, and from our setup $dp[0][0]=0=\text{OPT}(0,0)$.

Now assume for the sake of induction that $dp[i][j]=\text{OPT}(i,j)$ whenever we have i',j' such that $i'+j'<i+j$. Now we can do casework on the values of i,j:

1. If $j=0$, then our algorithm simply sets $dp[i][0]=\min(A+c(i-1),Ai)=\text{OPT}(i,0)$ (base case).
2. If $i=0$, then our algorithm simply sets $dp[0][j]=\min(A+c(j-1),Aj)=\text{OPT}(0,j)$ (base case).
3. If $i,j>0$, then our algorithm first attempts to remove $1\le k<i$ characters from x and use an optimal alignment between $x[0..i-k]$ and $y[0..j]$ by applying our induction hypothesis – note that $i-k+j<i+j$. Hence $dp[i-k][j]=\text{OPT}[i-k][j]$, and at the end of the first loop we have the value:

$$\min_{1\le k<i}(A+c(k-1)+dp[i-k][j])=\min_{1\le k<i}(A+c(k-1)+\text{OPT}[i-k][j])=\text{OPT}_x(i)$$

In the second loop, similarily we attempt to insert k characters into x from y and use the optimal alignment between $x[0..i]$ and $y[0..j-k]$. Again we apply our induction hypothesis ($i+j-k < i+j$), and at the end of the second loop we have:

$$\begin{aligned}
&\min(\text{OPT}_x(i), \min_{1\le k<j}(A + c(k-1) + dp[i][j-k])) \\
&= \min(\text{OPT}_x(i), \min_{1\le k<j}(A + c(k-1) + \text{OPT}[i][j-k])) \\
&= \min(\text{OPT}_x(i), \text{OPT}_y(j))
\end{aligned}$$

In the final step, we take a final minimum with if $x[i] \neq y[j]$ then B else $0 + dp[i-1][j-1]$. Again, $i-1+j-1 < i+j$, so our induction hypothesis guarantees

$$\text{if } x[i] \neq y[j] \text{ then } B \text{ else } 0 + dp[i-1][j-1] = \text{OPT}_m(i,j).$$

Thus our final computed cost is

$$dp[i][j] = \min(\text{OPT}_x(i), \text{OPT}_y(j), \text{OPT}_m(i,j)) = \text{OPT}(i,j).$$

Hence in all cases, we have $dp[i][j] = \text{OPT}(i,j)$. This completes the induction.

Termination: in the first part, we run a finite amount of iterations, which terminates. In the retracing step, by construction the pointers get smaller – i, j points to some i', j' such that $i' + j < i + j$. At some point we will retrace to $(i,j) = (0,0)$, so this terminates as well.

Runtime: Refer to the code above. All of the initialization in the beginning, setting up vectors, etc, is $O(nm)$. The main body has two top for loops, the outer doing n iterations and inner doing m. Within the two inner loops, we run two more loops in sequence, the first doing n iterations and the second doing m iterations. Everything inside these two loops and within the main body consist only of comparisons and assignments, which runs in $O(1)$. Hence the runtime of the inner body is $O(n+m)$, and then adding the two outer loops gives $O(nm(n+m))$. In the last part of the algorithm, we trace back with our previous pointers. By construction, this trace back visits at most all of the nm dp subproblems, so this section also runs in $O(nm)$. Hence the total runtime of the entire piece of code is $O(nm) + O(nm(n+m)) + O(nm) = O(nm(n+m))$.

Output: We claim that the output is correct because it maintains the invariant that for every i, j, prev$[i][j]$ consists of the squares i', j' and a operation t such that doing t at i', j' is the optimal operation that aligns $x[0..i]$ with $y[0..j]$. This is true since the code only updates prev$[i][j]$ when we find a better minimum for $dp[i][j]$. Thus prev$[i][j]$ always maintains the best known operation to align $x[0..i]$ to $y[0..j]$ from the dp. Thus when we reconstruct the operations, at each trace back we are just taking the optimal operation until we can no longer go back more. Once we terminate, we simply end up with the best sequence of operations transforming x to y (once again backwards, so we must reverse). □

Chocolate Problem: 1 chocolate bar

Reminder: If you solve a chocolate problem (which you can do in groups of size up to 3), please e-mail David with the solution — do not submit it on Gradescope. Also, feel free to list preferences or dietary restrictions for/against particular types of chocolate.

This problem is a significant generalization of the problem that you saw in discussion section a week ago. You want to tile the hallway of your home with rectangular tiles of size 2×1. You can use them horizontally or vertically. Your hallway is of size $k \times n$, and your tiling must not let any of those kn squares be uncovered by a tile. To illustrate tilings, two valid tilings of a 3×4 rectangle are given in Figure **??**.

Your goal is to find out how many *distinct* legal tilings there are for your hallway. If two tilings are mirrors or rotations of each other, we still count them as distinct. Give an algorithm with running time $f(k) \cdot p(n)$ for this problem, where $p(n)$ must be a polynomial, while $f(k)$ is allowed to be (singly) exponential.

(Hint: start by thinking about small values of k, like $k = 3$ or $k = 4$. $k = 2$ can also be helpful, but it is perhaps too special.)