ICPC Workshop T3 W2

Dynamic Programming

Jonathan Lam and Angus Ritossa

Table of contents

Background: A quick refresher on recursion

- You might remember about recursion from COMP1511.
- Example: Write a program which returns the nth Fibonacci number, F_n.
  Definition: $F_0 = 1$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$.
- The main idea we can construct larger Fibonacci numbers by adding up smaller Fibonacci numbers.
- Dynamic Programming builds upon recursion (but they aren't necessarily the same thing)

What is Dynamic Programming (DP)?

- A technique for solving complicated problems by breaking it down into many simpler subproblems.
- We can construct a solution for a subproblem from the solutions of other subproblems – using recursion

Strategy for solving DP problems

When should we use DP?

There's no definitive rule on when a particular technique works, but you can try DP when:

- Greedy/other techniques does not work
- Counting small cases is easy but the technique does not scale up to larger cases
- You want to find an optimal solution
- You want to count the number of solutions

Solving DP problems

1. Define your state and subproblem
2. Identify your base cases
3. Write down recurrence between subproblems
4. Answer the original problem

Classic Problems

We won't go through these in the interest of time as these are fairly well known.
COMP3121 students may be interested in revising these problems and their variations.

- Longest increasing subsequence
- Longest common subsequence
- Edit distance
- Knapsack

1 Dimensional DP

Determine the number of binary strings of length n that does not have three consecutive 0s.

Solution Ideas

- This seems like a high school maths combinatorics problem.
  - We can arrange n objects in a line in $n!$ ways
  - Subtract the cases where three or more zeroes are together?
  - Account for repetition?
  - But how do we put this all together?
- It seems like a direct counting approach is too hard.
- What if we consider smaller cases?

  $n = 1$: 2 ways

  $n = 2$: 4 ways

  $n = 3$: 7 ways
- We can use our existing solutions to construct solutions for higher n. This is the main idea of DP.

Solution

I find it helpful to go through the steps to make sure I don't miss anything.

- State:
  The length of the string
- Subproblem:
  Let c_n be the number of strings that have exactly n digits.
- Base cases:
  $c_1 = 2$
  $c_2 = 4$
  $c_3 = 7$
- Original Problem:
  We want to find c_n

Solution

- Recurrence:
  For $n \geq 4$, a string with n digits ends with `1` or `10` or `100`.

  We have the following cases

  - The string ends in `1`
    Then the string that remains when the end digit is removed must also work. So the number of strings that end in `1` and have exactly n digits equals c_{n-1}.
  - The string ends in `10`
    Then the string that remains when the last 2 digits are removed must also work. So the number of strings that end in `10` and have exactly n digits equals c_{n-2}.
  - The string ends in `100`
    Then the string that remains when the last 3 digits are removed must also work. So the number of strings that end in `100` and have exactly n digits equals c_{n-3}.

  Hence
  $$c_n = c_{n-1} + c_{n-2} + c_{n-3} \text{ for } n \geq 4.$$

Implementation

```
D[1] = 2;
D[2] = 4;
D[3] = 7;
for(i = 4; i <= n; i++)
    D[i] = D[i-1] + D[i-2] + D[i-3];
```

Aside:

Another recurrence exists:

$$c_n = 2c_{n-1} - c_{n-4}$$

Can you work out why this works?

Optimisations

- Our solution takes $O(n)$ time and $O(n)$ memory.
  We can reduce this to $O(1)$ memory by recognising that we only need to store the previous three terms.

```
D[1] = 2;
D[2] = 4;
D[3] = 7;
for(i = 4; i <= n; i++)
    D[i%3] = D[(i-1)%3] + D[(i-2)%3] + D[(i-3)%3];
```

- What happens if n is huge?
  We can solve recurrences using matrices and fast exponentiation (if you are interested, you can research this in your own time). This will solve the problem in $O(\log n)$ time.

2D DP

In the previous problem, each state depended on one parameter only. We call this a 1D problem.

We don't need to restrict ourselves to one variable for our states, as seen in the next problem.

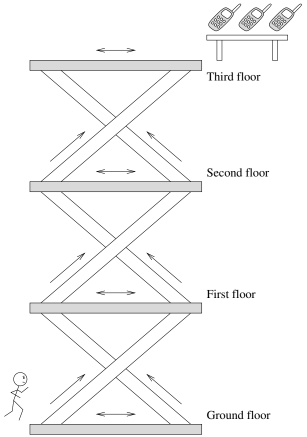
AIO 2007: Superphone

It is almost time. After months of waiting and media hype, the new all-in-one mobile phone, music player and electric toothbrush is about to be released. You have been waiting outside the tall store for the last twelve hours hoping to be the first to get your hands on one of these new phones. In just a few minutes, the store is going to open.

Unfortunately, a large number of other people have also been milling about outside the store, they too wanting to be the first to purchase one of these new phones. To ensure that they don't beat you to it, you will not only have to sprint to the sales desk of the store, but you will need to ensure that you take the absolute fastest route to get there.

The store is a large building consisting of several floors. Each floor has a pair of escalators that connect to the floor above, as shown in the figure below. The left-hand side of the building has an escalator to the right-hand side of the floor above, and the right-hand side of the building has an escalator to the left-hand side of the floor above. All of these escalators move up. Also, on each floor you are able to (if you wish) sprint to the other side of the building to get to the escalator on the other side.

The doors of the store, where you are currently waiting, are located to the left of bottom floor. The sales desk, where you must purchase your phone, is located on the right of the top floor.



at different speeds, and some floors are more difficult to run through than others. Fortunately, you have carefully recorded precisely how many seconds it takes you to run up each individual escalator, as well as how many seconds it takes you to run from one side of each individual floor to the other.

Your task is to determine the minimum amount of time required to reach the sales desk of the building from the doors of the building.

From your research, you know that different escalators move

Formal/Short Statement

A key part of competitive programming is solving problems fast, which means skimming over the unneceesary details. What is this question actually asking us to do?

Given an input corresponding to the times to traverse each elevator/floor, find the shortest time to travel from the bottom to the top.

Solution Ideas

- We could try to tackle this with a greedy algorithm. For example, choose the fastest route at each possible point.
  This doesn't work: What if your first elevator was really quick but every elevator after that took long?
- What about choosing the *set* of routes which takes me to the next floor the fastest?
  Also doesn't work: You could end up on either the left or right side, which might affect future decisions.
- This is the key observation: We need to consider how long it takes to reach both the left and right sides, and move upstairs from that.
- That is, we are going to construct our solution based on a bunch of *states*. This is exactly what DP is about.

DP solution

- What is your state?
  The floor you are on, and whether you are on the left or right.
  That is, your state depends on two parameters, so this is 2D-DP.
- What is your subproblem?
  The subproblem is assigning a value to your state.
  Let $DP(f, s)$ be the minimum time it takes to reach floor f on side s (where $s \in \{L, R\}$).
- Base cases
  The base cases are the times to reach the left and right positions of the first (ground) floor.
  $DP(1, L) = 0$ (since this is where we are starting)
  $DP(1, R) = \text{floor}_1$ (time to run across the floor on level 1)
- What is the original problem?
  We want to reach the nth floor on the right side. So we want to find $DP(n, R)$.

DP Solution

- Recurrence between the subproblems?

$$\begin{aligned} DP(i,L) &= \min(DP(i-1,R) + \text{right}_{i-1}, \\ &\qquad DP(i-1,L) + \text{left}_{i-1} + \text{floor}_i) \\ DP(i,R) &= \min(DP(i-1,L) + \text{left}_{i-1}, \\ &\qquad DP(i-1,R) + \text{right}_{i-1} + \text{floor}_i) \end{aligned}$$

Mortal Kombat Tower

Source: https://codeforces.com/contest/1418/problem/C

You and your friend take turn fighting n bosses which can be easy or hard. At each turn, the current player can kill one or two bosses. Your friend is not good enough to kill hard bosses, but can use one skip point to kill one hard boss.

Determine the minimum number of skip points needed to kill all n bosses in the given order.

Input:

n, the number of bosses and
a, an array of length n where $a_i = 0$ if the boss is easy and $a_i = 1$ if the boss is hard.

Output:

The minimum number of skip points needed.

Mortal Kombat Tower

Sample Input:

8

1 0 1 1 0 1 1 1

Optimal game plan:

- your friend kills two first bosses, using one skip point for the first boss
- you kill the third and the fourth bosses
- your friend kills the fifth boss
- you kill the sixth and the seventh bosses
- your friend kills the last boss, using one skip point

Hence the minimum skip points needed is 2.

Mortal Kombat Tower

- This is a decision making problem where at each step, the player has to choose whether to kill 1 or 2 bosses.
- We don't know ahead of time which player will kill a given boss. So our DP solution will need to consider both players killing each boss.
- The DP value is the number of skip points used so far (the number of hard bosses your friend kills) which we want to minimise.
- Then for each player, we will consider what happens if they kill 1 or 2 bosses. Then choose the optimum out of that.

Mortal Kombat Tower

- **State:**
  The index (x) of the boss we are fighting and the person (p) who's turn it is.
- **Subproblem:**
  The minimum number of skip points needed to kill the bosses from $[x, n]$ with current player p.
- **Original problem:**
  You want to find $DP(0, 0)$.
- **Base cases:**
  If it is my turn and there are only 1 or 2 bosses left, then I can just kill them and no skip points are needed.
  If it is my friend's turn and there is one boss left, they are forced to kill this boss. So the value depends on whether the last boss is easy or hard.

Mortal Kombat Tower

Recurrence:

- If it is my turn, I can either kill 1 or 2 bosses. Then the next turn will be my friend's. No skip points are used so I don't need to add on the values. That is,

$$\begin{aligned} DP(x,1) = \min(&DP(x+1,0),\\ &DP(x+2,0)) \end{aligned}$$

- If it is my friend's turn, I can either kill 1 or 2 bosses. Then the next turn is mine. This time, we need to add on the amount of skip points used.

$$\begin{aligned} dp(x,0) = \min(&a_x + DP(x+1,1),\\ &a_x + a_{x+1} + DP(x+2,1)) \end{aligned}$$

Mortal Kombat Tower

Implementation notes

- I will implement this top–down as our pseudocode translates almost directly into C++ code.
- We will need to cache our values.
- If we have already seen a state, we can just retrieve the value from the cache rather than recalculating it.
- Since our DP values could be 0, we should initialise our cache for -1 for ‘not seen’.
- Whenever we find a new value (via recursion), we should store the newly found value in the cache.

Recursion vs Iteration

- We have seen examples where we have implemented our DP both iteratively and recursively.
- In most DP problems, recursive and iterative solutions both work and its a matter of preference. There are a few cases where one is preferred over the other.
- In problems where memory optimisations are needed, iterative is generally better
- In problems where there are many unreachable states, recursive is generally better because it will not visit these states.
- With iterative solutions, make sure you progress to your next states in a correct order.

Common DP Patterns

Deciding on your subproblems and determining their recurrence can be quite tricky. Here are some common patterns you might encounter.

- 1-Dimension DP
  - Find the number of ways to reach a given target.
    Strategy: Add all the possible ways to reach the current state from previously found states.
    $$\text{ways}_i = \text{ways}_{i-1} + \text{ways}_{i-2} + \cdots$$
  - Find the $\begin{Bmatrix} \text{min} \\ \text{max} \end{Bmatrix}$ $\begin{Bmatrix} \text{cost} \\ \text{sum} \end{Bmatrix}$ to reach a given target.
    Strategy: Choose the min/max cost among all possible paths before the current state, then add the value for the current state.
    $$\text{path}_i = \min\left(\text{path}_{i-1}, \text{path}_{i-2}, \ldots\right) + \text{cost}_i$$

Common DP Patterns

- 2-Dimension DP
  - Decide whether or not to include the current state.
    Strategy: Let the state be (p, d) where p is the current position and $d \in \{0, 1\}$ is whether you decide to include or exclude the current state.
  - Does a subset of a set achieve a given target?
    Strategy: Set the state to be (i, t) where we determine if the subset $[1, i]$ can achieve t.
- Interval DP
  Consider an interval $[i, j]$. For recurrences, you may consider the intervals $[i, j + 1]$ and $[i - 1, j]$. Base cases when $i = j$.