Princeton Competitive Programming

Dynamic Programming I

Pedro Paredes
September 30, 2022
Outline

1. Introduction
2. 1D DP
3. 2D DP
4. Interval DP
5. Bitmask DP
Introduction
What is DP?

Algorithm design technique based on breaking problems into simpler subproblems

Usual workflow:

- Break the problem into overlapping subproblems
- Solve each subproblem and store the answer
- Combine the subproblems into a solution to the main problem

Finding the right subproblem break down is part art part science

Can only be learned by looking at lots of examples
1D DP
The well-known Fibonacci series is defined as follows:

\[ F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \]

This is often used as an example of recursion like so:

```java
public static int fib(int n) {
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}
```
Fibonacci

How efficient is this solution?

This is the recursive tree:

Source: https://textbooks.cs.ksu.edu/cc310/6-recursion/6-example-fibonacci-numbers/

Tip: notice how the time complexity of this solution is related to the number of leaves, which in turn is related to the actual value of $F_n$.
To avoid repeating calculations we memoize (i.e. store) the values of each $F_n$ after computing it:

```java
public static int[] dp = new int[N];

public static int fib(int n) {
    if (n <= 1)
        return 1;
    if (dp[n] > 0)
        return dp[n];
    return dp[n] = fib(n - 1) + fib(n - 2);
}
```

This is clearly much faster, but by how much?
Suppose we use the previous code to compute $F_n$, what is the time complexity in terms of $n$?

What happens when we call $\text{fib}(i)$:

- if we computed $\text{fib}(i)$ before we return it, which is $O(1)$
- if not, we compute it and store it, which is $O(1)$

Since we compute each input once, the total time is $O(n)$
In the Fibonacci example the problem subdivision was obvious: it was just the definition itself. So let’s look at a more interesting one

**Problem**

Given an integer $n$, find the number of ways to fill a $3 \times n$ board with $1 \times 2$ dominoes

Example source http://web.stanford.edu/class/cs97si/04-dynamic-programming.pdf
We need to find a recurrence that breaks problem into subproblems.

Let $D_n$ be the number of ways of tiling a $3 \times n$ grid.

Let $A_n$ be the number of ways of tiling a $3 \times n$ grid with a “hole” on the top-right corner.
Tiling

Let’s try to use the previous subproblems and define a recurrence based on trying to till the last column

\[ D_n = D_{n-2} + A_{n-1} + A_{n-1} \]

\[ A_n = D_{n-1} + A_{n-2} \]
Based on the previous here is the recurrence:

```java
public static int D(int n) {
    if (n == 0)
        return 1;
    if (n == 1)
        return 0;
    return D(n - 2) + 2 * A(n - 1);
}

public static int A(int n) {
    if (n <= 1)
        return 1;
    return D(n - 1) + A(n - 2);
}
```

Exercise: memoize the above code to make it $O(n)$
2D DP
Paths on a Grid

Problem

Given an \( n \) by \( n \) integer matrix, find a path from the upper-left corner to the lower-right corner with the lowest sum. The path can only move down or right.

The recurrence is the following:

\[
dp(x, y) = \min(dp(x - 1, y), dp(x, y - 1)) + grid_{x,y}
\]
Based on the previous here is the solution:

```java
int[][] dp = new int[n][n];

for (int y = 1; y <= n; y++) {
    for (int x = 1; x <= n; x++) {
        dp[y][x] = Math.min(dp[y][x - 1], dp[y - 1][x]) + grid[y][x];
    }
}
```

This is $O(n^2)$ since we have two nested for loops
Interval DP
Problem

Given a sequence of $n$ characters $x_1x_2\ldots x_n$, find the minimum number of characters we need to add to make it a palindrome.

If $x = abga$ we can add one $b$ and make it abgba, so the answer is 1.

We need to think about how to define some recurrence that is easy to compute.
Palindromic Edit Distance

Let $D_{ij}$ be the minimum number of characters that need to be inserted to make $x_i \ldots x_j$ into a palindrome.

So the solution is given by $D_{1n}$

$$D_{ij} = \begin{cases} 
1 + \min\{D_{i+1,j}, D_{i,j-1}\} & x_i \neq x_j \\
D_{i+1,j-1} & x_i = x_j
\end{cases}$$

Exercise: implement that to make it $O(n^2)$
Bitmask DP
Traveling Salesman Problem

**Problem**
Given a complete graph with $n$ vertices, the cost between each pair of vertices $u, v$ is a positive integer $c_{u,v}$. Find the minimum sum cycle that visits each vertex once.

Note that since we are looking for a cycle with all vertices, without loss of generality we can look for cycles that start at vertex 0 and end at vertex 0.

**Recursion idea**
Suppose we have some partially built path that is currently at a vertex $v$ and has visited all the vertices in a set $S$.

Let $tsp(x, S)$ be cheapest way to complete this path, i.e. cheapest path that starts at $x$, visits all vertices in $V \setminus S$ and ends at vertex 0.
Traveling Salesman Problem

Note that we can define the following recursion to compute \( \text{tsp}(x, S) \):

\[
\text{tsp}(x, S) = \min_{y \notin S} \{ \text{tsp}(y, S \cup \{y\}) + c_{xy} \}
\]

And we have a base case when \( S \) contains all vertices, i.e. \( S = V \):

\[
\text{tsp}(x, V) = c_{x0}
\]

Since here we have visited everyone so we have to return to 0
How do we memoise over sets?

Note that $n$-bit integers represent sets: $100100 \equiv \{0, 3\}$, $000001 \equiv \{5\}$

So we can use 32-bit integers to represent sets of up to 32 elements

Some bitwise operations on sets (suppose $b$ is a bitmask representing a set $S$ and $i$ is a vertex index):

- $1 \ll i$ is the set $\{i\}$
- $(1 \ll N) - 1$ is the set $\{0, \ldots, N-1\}$ (all numbers up to $N$)
- $b \& (1 \ll i)$ is 0 if $i \notin S$ and positive otherwise
- $b | (1 \ll i)$ adds $i$ to $S$, so it is $S \cup \{i\}$
Traveling Salesman Problem

```java
public static int[][] dp = new int[N][1 << N];

public static int tsp(int i, int S) {
    if (S == ((1 << N) - 1)) {
        return c[i][0];
    }
    if (dp[i][S] != -1) {
        return dp[i][S];
    }
    int res = Integer.MAX_VALUE;
    for (int j = 0; j < N; j++) {
        if ((S & (1 << j)) > 0)
            continue;
        res = Math.min(res, c[i][j] + tsp(j, S | (1 << j)));
    }
    return dp[i][S] = res;
}
```

The solution to the overall problem is `tsp(0, 1<<0)`

This is $O(2^n \cdot n^2)$