Princeton Competitive Programming

Dynamic Programming II

Pedro Paredes

October 14, 2022

Outline

Recap of Lecture I

What is DP?

Algorithm design technique based on breaking problems into simpler subproblems

What is DP?

Algorithm design technique based on breaking problems into simpler subproblems

Usual workflow:

- Break the problem into overlapping subproblems

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

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

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

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

Warm Up DP

Longest Increasing Subsequence

Problem

Given an array $a_1, \ldots a_n$ with n elements, find the length of the longest increasing subsequence.

Longest Increasing Subsequence

Problem

Given an array $a_1, \ldots a_n$ with n elements, find the length of the longest increasing subsequence.

- $a = 1, 2, 3$ then $\text{LIS}(a) = 3$, the whole sequence is increasing
- $a = 1, 2, 1, 4, 3, 4$ then $\text{LIS}(a) = 4$, take $[1, 2, 3, 4]$
- $a = 5, 4, 3, 2, 1$ then $\text{LIS}(a) = 1$, take $[1]$

Longest Increasing Subsequence

First possible state:

dp(i) is the longest increasing subsequence within $a_1, \ldots, a_i$

Longest Increasing Subsequence

First possible state:

> $\mathrm{dp}(i)$ is the longest increasing subsequence within $a_1, \ldots, a_i$

It's not clear how to write a recursion on this...

Longest Increasing Subsequence

Another suggestion:

dp(i) is the longest increasing subsequence within $a_1, \ldots, a_i$ that ends at a_i

Longest Increasing Subsequence

Another suggestion:

dp(i) is the longest increasing subsequence within $a_1, \ldots, a_i$ that ends at a_i

But now:

$$\mathrm{dp}(i) = \max(1, \max_{\substack{j<i \\ a_j<a[i]}} (1 + \mathrm{dp}(j)))$$

```java
public static int lis(int i) {
  if (dp[i] != -1) {
    return dp[i];
  }

  int res = 1;
  for (int j = 0; j < i; j++) {
    if (a[j] < a[i]) {
      res = max(res, 1 + lis(j));
    }
  }

  return dp[i] = res;
}
```

```java
public static int lis(int i) {
  if (dp[i] != -1) {
    return dp[i];
  }

  int res = 1;
  for (int j = 0; j < i; j++) {
    if (a[j] < a[i]) {
      res = max(res, 1 + lis(j));
    }
  }

  return dp[i] = res;
}
```

How to get answer?

```java
public static int lis(int i) {
  if (dp[i] != -1) {
    return dp[i];
  }

  int res = 1;
  for (int j = 0; j < i; j++) {
    if (a[j] < a[i]) {
      res = max(res, 1 + lis(j));
    }
  }

  return dp[i] = res;
}
```

How to get answer?

```java
int mx = 0;
for (int i = 0; i < n; i++)
  mx = Math.max(mx, lis(i));
System.out.println(mx);
```

```
public static int lis(int i) {
  if (dp[i] != -1) {
    return dp[i];
  }

  int res = 1;
  for (int j = 0; j < i; j++) {
    if (a[j] < a[i]) {
      res = max(res, 1 + lis(j));
    }
  }

  return dp[i] = res;
}
```

How to get answer?

```
int mx = 0;
for (int i = 0; i < n; i++)
  mx = Math.max(mx, lis(i));
System.out.println(mx);
```

What is the time complexity?

```java
public static int lis(int i) {
  if (dp[i] != -1) {
    return dp[i];
  }

  int res = 1;
  for (int j = 0; j < i; j++) {
    if (a[j] < a[i]) {
      res = max(res, 1 + lis(j));
    }
  }

  return dp[i] = res;
}
```

How to get answer?

```java
int mx = 0;
for (int i = 0; i < n; i++)
  mx = Math.max(mx, lis(i));
System.out.println(mx);
```

What is the time complexity? It's $O(n^2)$.

DP on trees

Tree Diameter

Problem

Given a tree T of n nodes, calculate longest path between any two nodes

Tree Diameter

Problem

Given a tree T of n nodes, calculate longest path between any two nodes

First things first: root the tree at an arbitrary node

Tree Diameter

Problem

Given a tree T of n nodes, calculate longest path between any two nodes

First things first: root the tree at an arbitrary node

Now note:

- The longest path can be entirely on one of the root's subtrees

Tree Diameter

Problem

Given a tree T of n nodes, calculate longest path between any two nodes

First things first: root the tree at an arbitrary node

Now note:

- The longest path can be entirely on one of the root's subtrees
- The longest path can start in a subtree of root, go through root and end at a different subtree

Tree Diameter

Define:

- longest(v) is the longest path contained in v's subtree

Tree Diameter

Define:

- longest(v) is the longest path contained in v's subtree
- height(v) is the longest path contained in v's subtree that ends at v

Tree Diameter

Define:

- longest(v) is the longest path contained in v's subtree
- height(v) is the longest path contained in v's subtree that ends at v

Tree Diameter

Define:

- longest(v) is the longest path contained in v's subtree
- height(v) is the longest path contained in v's subtree that ends at v

Now note:

$$\text{height}(v) = 1 + \max_{u \text{ child of } v} \text{height}(v)$$

Tree Diameter

Define:

- longest(v) is the longest path contained in v's subtree
- height(v) is the longest path contained in v's subtree that ends at v

Now note:

$$\text{height}(v) = 1 + \max_{u \text{ child of } v} \text{height}(v)$$

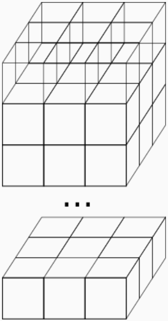
$$\begin{aligned}\text{longest}(v) = \max(&\max_{u \text{ child of } v} \text{longest}(v), \\ &1 + \max_{u \text{ child of } v} \text{height}(v) + \text{second} \max_{u \text{ child of } v} \text{height}(v))\end{aligned}$$

The Twin Tower

The Twin Tower

Problem

Consider a tower of $3 \times 3 \times n$ blocks, count how many ways there are to tile it using $1 \times 1 \times 2$ blocks.



The Twin Tower

Main ideas:

- Suppose we define the obvious: $\mathrm{dp}(n)$ is the number of ways to tile a $3 \times 3 \times n$ tower

The Twin Tower

Main ideas:

- Suppose we define the obvious: $\text{dp}(n)$ is the number of ways to tile a $3 \times 3 \times n$ tower
- Now we can try to setup a recursion where we try to tile the topmost 3×3 layer, but this doesn't quite work

The Twin Tower

Main ideas:

- Suppose we define the obvious: $\mathrm{dp}(n)$ is the number of ways to tile a $3 \times 3 \times n$ tower
- Now we can try to setup a recursion where we try to tile the topmost 3×3 layer, but this doesn't quite work
- Key idea: when we fill the nth layer we might fill some of the $(n-1)$th layer, so we define a state like: $\mathrm{dp}(n, S)$ is the number of ways to tile a $3 \times 3 \times n$ tower where the tiles in S are prefilled

The Twin Tower

Main ideas:

- Suppose we define the obvious: $\mathrm{dp}(n)$ is the number of ways to tile a $3 \times 3 \times n$ tower
- Now we can try to setup a recursion where we try to tile the topmost 3×3 layer, but this doesn't quite work
- Key idea: when we fill the nth layer we might fill some of the $(n-1)$th layer, so we define a state like: $\mathrm{dp}(n, S)$ is the number of ways to tile a $3 \times 3 \times n$ tower where the tiles in S are prefilled

The Twin Tower

Main ideas:

- Suppose we define the obvious: $\mathrm{dp}(n)$ is the number of ways to tile a $3 \times 3 \times n$ tower
- Now we can try to setup a recursion where we try to tile the topmost 3×3 layer, but this doesn't quite work
- Key idea: when we fill the nth layer we might fill some of the $(n-1)$th layer, so we define a state like: $\mathrm{dp}(n, S)$ is the number of ways to tile a $3 \times 3 \times n$ tower where the tiles in S are prefilled

But wait, don't we have to handle a huge number of cases?

The Twin Tower

Instead of handling cases manually, we can determine the recursion for each possible case by computing it!

The Twin Tower

Instead of handling cases manually, we can determine the recursion for each possible case by computing it!

Recall the idea of bitmasks: n-bit integers represent sets: $100100 \equiv \{0, 3\}$, $000001 \equiv \{5\}$

The Twin Tower

Instead of handling cases manually, we can determine the recursion for each possible case by computing it!

Recall the idea of bitmasks: n-bit integers represent sets: $\texttt{100100} \equiv \{0, 3\}$, $\texttt{000001} \equiv \{5\}$

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

The Twin Tower

Instead of handling cases manually, we can determine the recursion for each possible case by computing it!

Recall the idea of bitmasks: n-bit integers represent sets: $\texttt{100100} \equiv \{0, 3\}$, $\texttt{000001} \equiv \{5\}$

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 << i` is the set $\{i\}$

The Twin Tower

Instead of handling cases manually, we can determine the recursion for each possible case by computing it!

Recall the idea of bitmasks: n-bit integers represent sets: $\texttt{100100} \equiv \{0, 3\}$, $\texttt{000001} \equiv \{5\}$

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):

- $\texttt{1} << \texttt{i}$ is the set $\{i\}$
- $(\texttt{1} << \texttt{N}) - \texttt{1}$ is the set $\{0, \ldots, N-1\}$ (all numbers up to N)

The Twin Tower

Instead of handling cases manually, we can determine the recursion for each possible case by computing it!

Recall the idea of bitmasks: n-bit integers represent sets: $\texttt{100100} \equiv \{0, 3\}$, $\texttt{000001} \equiv \{5\}$

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):

- $\texttt{1} << \texttt{i}$ is the set $\{i\}$
- $(\texttt{1} << \texttt{N}) - \texttt{1}$ is the set $\{0, \ldots, N-1\}$ (all numbers up to N)
- $\texttt{b}\&(\texttt{1} << \texttt{i})$ is 0 if $i \notin S$ and positive otherwise

The Twin Tower

Instead of handling cases manually, we can determine the recursion for each possible case by computing it!

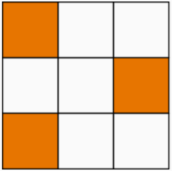
Recall the idea of bitmasks: n-bit integers represent sets: $\texttt{100100} \equiv \{0, 3\}$, $\texttt{000001} \equiv \{5\}$

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

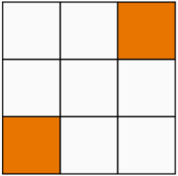
Some bitwise operations on sets (suppose b is a bitmask representing a set S and i is a vertex index):

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

The Twin Tower



$$= \begin{matrix}100\\001\\100\end{matrix} = 100001100 = 268$$



$$= \begin{matrix}001\\000\\100\end{matrix} = 001000100 = 68$$

We represent 3×3 tiles as a bitmask like so

Useful functions to use:

```
public static int isNotPresent(int x, int y, int mask) {
  return (mask & (1 << (y * 3 + x))) == 0;
}

public static int add(int x, int y, int mask) {
  return (mask | (1 << (y * 3 + x)));
}

public static int remove(int x, int y, int mask) {
  return (mask ^ (1 << (y * 3 + x)));
}
```

```java
public static void fill(int maskN, int maskN1, int origMask) {
    if (maskN == (1 << 9) - 1) {
        transition[origMask][maskN1]++;
        return;
    }

    int x = 0, y = 0;
    // find first unused on last layer
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (isNotPresent(i, j, maskN)) {
                x = i;
                y = j;
                break;
            }

    maskN = add(x, y, maskN);
    // horizontal
    if (x > 0 && isNotPresent(x - 1, y, maskN))
        fill(add(x - 1, y, maskN), maskN1, origMask);
    // vertical
    if (y < 2 && isNotPresent(x, y + 1, maskN))
        fill(add(x, y + 1, maskN), maskN1, origMask);
    // down
    if (isNotPresent(x, y, maskN1))
        fill(maskN, add(x, y, maskN1), origMask);
}
```

```java
public static int calc(int n, int mask) {
    //
    // Base Case
    // ?????????
    //

    if (dp[n][mask] != -1)
        return dp[n][mask];

    int res = 0;
    for (int i = 0; i < (1 << 9); i++) {
        if (transition[mask][i] == 0) continue;
        res += transition[mask][i] * calc(n - 1, i);
    }

    return dp[n][mask] = res;
}
```

```java
public static int calc(int n, int mask) {
    if (n == 0 && mask == 0)
        return 1;
    if (n == 0)
        return 0;

    if (dp[n][mask] != -1)
        return dp[n][mask];

    int res = 0;
    for (int i = 0; i < (1 << 9); i++) {
        if (transition[mask][i] == 0) continue;
        res += transition[mask][i] * calc(n - 1, i);
    }

    return dp[n][mask] = res;
}
```