Dynamic Programming (III)

Ethen Yuen {ethening}

2022-03-19

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Prerequisites

This lecture is about DP optimization.

If you are not familiar with dynamic programming, please refer to DP(I) and DP(II)

You should have knowledge on these topics:

- Optimization
- Recursion, Divide and Conquer
- Data Structure (I) - (III)

Beware that there are a lot of maths involved in this lecture. You have been warned.

Table of Contents

DP optimization

Why DP optimization?

Suppose you have come up with a correct DP formula

- State definition
- State transition
- Base case

Still TLE?

Time complexity is too high?

- Transition takes too much time
- O(N)?

Why DP optimization?

Four main ways to solve

- Explore non-DP solutions
- Write auxiliary DPs (DP2[][], DP3[][], …) to speed up
- Come up with alternative DP formula
- **Optimize DP transition** → **What we will explore today**

How to optimize DP transition?

Monotonicity

Monotonicity

non-decreasing / non-increasing

Useful property for DP optimization

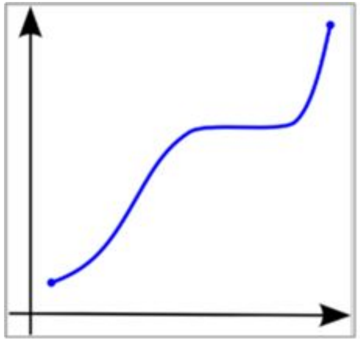


Figure 1 - A monotonically increasing function

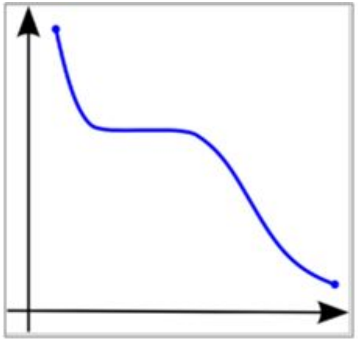


Figure 2 - A monotonically decreasing function

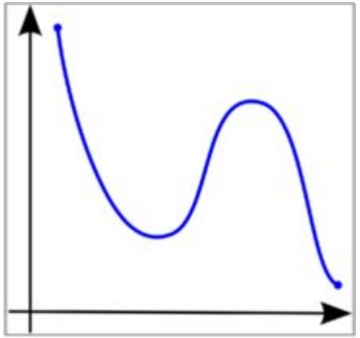


Figure 3 - A function that is not monotonic

Warm-up Optimization Question

Given an array A of length N, find the maximum element of every continuous interval of length K.

(i.e. A[0 .. K - 1], A[1 .. K], A [2 .. K + 1], ..., A[N - K .. N - 1]).

e.g. A = {3, 1, 4, 1, 5, 9, 2}, K = 3

max A[0 .. 2] = 4 max A[1 .. 3] = 4 max A[2 .. 4] = 5

max A[3 .. 5] = 9 max A[4 .. 6] = 9

Warm-up Optimization Question

Given an array A of length N, find the maximum element of every continuous interval of length K.

(i.e. A[0 .. K - 1], A[1 .. K], A [2 .. K + 1], ..., A[N - K .. N - 1]).

- O(NK) solution
- O(N lg N) solution
- O(N) solution

Warm-up Optimization Question

Given an array A of length N, find the maximum element of every continuous interval of length K.

(i.e. A[0 .. K - 1], A[1 .. K], A [2 .. K + 1], ..., A[N - K .. N - 1]).

- O(NK) solution - Naively loop through all the elements in each interval.
- O(N lg N) solution - Use any DS suitable (heap, segment tree, ...)
- O(N) solution

Warm-up Optimization Question

- O(N) solution - Monotonic Queue

e.g. A = {**3, 1**, **4**, 1, 5, 9, 2}, K = 3

Suppose we iterate through the elements one by one to consider them.

When we consider the 3rd element **(4)**, the previous elements **(3, 1) must not be candidates for further answers.** Why?

Any further interval that contains 1st or 2nd elements must contains 3rd element, and the 3rd element is larger.

Warm-up Optimization Question

- O(N) solution - Monotonic Queue

e.g. A = {3, 1, **4**, **1**, 5, 9, 2}, K = 3

Suppose we iterate through the elements one by one to consider them.

When we consider the 4th element **(1)**, the previous element **(4)** **may still be candidate for further answers.**

The 4th element **(1)** **may be a candidate for further answers** although the previous element **(4)** is larger because that will expire earlier.

Warm-up Optimization Question

- O(N) solution - Monotonic Queue

For the list of answer candidates stored in expiry order (quicker to expire put in front), we should maintain the list keeping their values **in descending order**.

Because not descending -> there are candidate that will never be the answer.

Warm-up Optimization Question

e.g. A = {4, 1, 3, 2, 5}, K = 3

CandidateList = {**A[0] = 4**}

CandidateList = {A[0] = 4, **A[1] = 1**}

CandidateList = {A[0] = 4, **~~A[1] = 1~~ (<= 3)**, **A[2] = 3**}

CandidateList = {**~~A[0] = 4~~ (expired)**, A[2] = 3, **A[3] = 2**}

CandidateList = {**~~A[2] = 3, A[3] = 2~~ (<= 5)**, **A[4] = 5**}

The front not-deleted element is the answer: {4, 3, 5}

Monotone Queue Optimization

Monotone Queue Optimization

Queue where the elements from the front to the end is either **increasing** or **decreasing**

Useful in many situations, not only DP problems

Usually implemented with **deque** (doubly ended queue)

- std::deque
- push_back(), push_front(), pop_back(), pop_front()

Monotone Queue Optimization

The basic form of DP formula:

$dp[i] = max_{L(i)<=j<i}\ (dp[j]) + f(i)$

L(i) is increasing

- e.g. $dp[i] = max_{i-k<=j<i}\ (dp[j]) + f(i)$

  ^^*This is the same with the warm-up question*^^

- *not increasing -> RMQ using segment tree*
- [DS (III)]

Monotone Queue Optimization

The basic form of DP formula:

$dp[i] = max_{L(i)<=j<i}\ (dp[j]) + f(i)$

May replace dp[j] by any function depending on j

- e.g. `g(j) = dp[j] * 2 − j`
- $dp[i] = max_{L(i)<=j<i}\ (dp[j] * 2 - j) + f(i)$

Monotone Queue Optimization

Naïve implementation: $O(N^2)$

```
for i from 1 to N
    dp[i] = -INF
    for j from L(i) to i - 1
        dp[i] = max(dp[i], f(i) + g(j))
```

Can be optimize using **Monotone Queue**!

Bowling for Numbers ++

CCC 2007 Stage 2 Problem

You have **N** (N<=10000) bowling pins and **K** (K<=500) bowling balls, each ball has width **w** (w<=100)

Each pin has a score s[i] from **-10000** to **10000**

You are allowed to **miss**

Find the maximum achievable score

Bowling for Numbers ++

Sample (N = 9, K = 4, w = 3)

```
2 8 -5 3 5 8 4 8 -6
X X -5 3 5 8 4 8 -6 (ball 1, score = 10), avoid -5
_ _ -5 X X X 4 8 -6 (ball 2, score = 26)
_ _ -5 _ _ X X X -6 (ball 3, score = 38), avoid -6
_ _ -5 _ _ _ _ _ -6 (ball 4, score = 38), miss completely
```

Answer = 38

Bowling for Numbers ++

Order of balls are **not important**

Consider balls thrown from left to right

What if all pins have **non-negative values**?

- Better to hit more pins than to miss

dp[i][j] = max. Score if we use i balls for pins $1..j$

transition?

Sample (N = 9, K = 4, w = 3)

```
2 8 -5 3 5 8 4 8 -6
X X -5 3 5 8 4 8 -6 (ball 1, score = 10)
_ _ -5 X X X 4 8 -6 (ball 2, score = 26)
_ _ -5 _ _ X X X -6 (ball 3, score = 38)
_ _ -5 _ _ _ _ _ -6 (ball 4, score = 38)
```

Bowling for Numbers ++

For state (i, j), consider either **throw a ball** or **not**, roughly

`dp[i][j] = max(`**`dp[i][j-1]`**`, `**`dp[i-1][j-W]`**` + sum(s[j-W+1]..s[j]))`

`sum(s[j-W+1]..s[j])` can be pre-computed and obtained in O(1)

- Prefix Sum (Optimization)

Time complexity: O(NK)

CCC 2007 Stage 1 Senior Q5

dp[i][j] = max. Score if we use `i` balls for pins `1..j`

Bowling for Numbers ++

Each pin has a score s[i] from **-10000** to **10000**

~~What if all pins have **non-negative values**?~~

- ~~Better to hit more pins than to miss~~

Sometimes, we want to hit less pins to “avoid” those negatives value

```
_ _ -5 X X X 4 8 -6 (ball 2, score = 26)
_ _ -5 _ _ X X X -6 (ball 3, score = 38)
```

How?

Bowling for Numbers ++

dp[i][j] = max. Score if we use i balls and the **rightmost hit pin is j**

Consider balls thrown from left to right

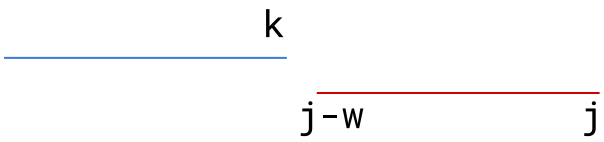
```
dp[0][0] = 0 (pins are 1-based)
dp[0][i] = -INF for i > 0
```

Two cases:

1. The i^{th} ball **does not overlap** with the $(i-1)^{th}$ ball
2. The i^{th} ball **overlaps** with the $(i-1)^{th}$ ball

Bowling for Numbers ++

Ps[i] = s[1] + s[2] + … + s[i] (prefix sum)



1. The i^{th} ball **does not overlap** with the $(i-1)^{th}$ ball

$$M1 = \max_{0<=k<=j-w}(dp[i-1][k] + ps[j] - ps[j-w])$$
$$= \max_{0<=k<=j-w}(dp[i-1][k]) + ps[j] - ps[j-w]$$

Precompute dp2[i-1][k] = max(dp[i-1][0], ..., dp[i-1][k])

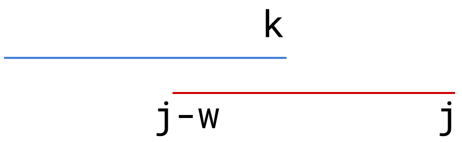
$$\max_{0<=k<=j-w}(dp[i-1][k]) = dp2[i-1][j-w]$$

O(1) for transition

dp[i][j] = max. Score if we use i balls and the rightmost hit pin is j

Bowling for Numbers ++

2. The i^{th} ball **overlaps** with the $(i-1)^{th}$ ball



$$M2 = \max_{j-w<k<j}(dp[i-1][k] + ps[j] - ps[k])$$
$$= \max_{j-w<k<j}(dp[i-1][k] - ps[k]) + ps[j]$$
$$dp[i][j] = \max(M1, M2)$$

O(w) for each transition

- Time complexity: **O(NKw)**

Optimize?

dp[i][j] = max. Score if we use i balls and the rightmost hit pin is j

Bowling for Numbers ++

The basic form of DP formula:

$dp[i] = max_{L(i)<=j<i}\ (dp[j]) + f(i)$

```
M2 = max_{j-w<k<j}(dp[i-1][k] - ps[k]) + ps[j]
L(j) = j-w, increasing
g(k) = dp[i-1][k] - ps[k]
f(j) = ps[j]
```

Monotone Queue optimization!

Bowling for Numbers ++

M2 = $\max_{j-w<k<j}$(dp[i-1][k] - ps[k]) + ps[j]

L(j) = j-w, increasing

g(k) = dp[i-1][k] - ps[k]

f(j) = ps[j]

Basic Idea:

k1 < k2 (expire earlier)

g(k1) < g(k2) (value is smaller)

k1 can never be optimal candidate, **can remove k1** from queue!

Bowling for Numbers ++

We maintain a queue (in fact deque) of indices s.t.

- `Q[j] < Q[j+1]` (indices are increasing)
- `g(Q[j]) >= g(Q[j+1])` (values are decreasing)

for all `j`

We can use std::deque or array to implement it

Bowling for Numbers ++

We use an array `Q[]` and two pointers `l` and `r` to represent the deque

`Q[l]` is the head of the deque

`Q[r]` is the tail of the deque

Deque is empty iff `l = r + 1`

Initially, `l = 1, r = 0` (i.e. deque is empty)

Monotone Queue: step by step

Step 1: Pop elements in the front that are “out of bounds”

```
while (l <= r) and (Q[l] < L(i))
    l++;
```

$$dp[i] = max_{L(i)<=j<i}\ g(j) + f(i)$$

Monotone Queue: step by step

Step 2: Update answer using Q[l]

```
if (l <= r)
    dp[i] = f(i) + g(Q[l]);
```

$$dp[i] = \max_{L(i) <= j < i} g(j) + f(i)$$

Monotone Queue: step by step

Step 3: Pop elements at the back that have small values

```
while (l <= r) and (g(Q[r]) < g(i))
    r--;
```

$$dp[i] = max_{L(i)<=j<i}\ g(j) + f(i)$$

Monotone Queue: step by step

Step 4: Insert `i` at the back

```
r++;
Q[r] = i;
```

$$dp[i] = \max_{L(i)<=j<i} g(j) + f(i)$$

Monotone Queue: step by step

```
1. while (l <= r) and (Q[l] < L(i))
     l++;
2. if (l <= r)
     dp[i] = f(i) + g(Q[l]);
3. while (l <= r) and (g(Q[r]) < g(i))
     r--;
4. r++;
     Q[r] = i;
```

$$dp[i] = max_{L(i)<=j<i}\ g(j) + f(i)$$

Bowling for Numbers ++

Apply monotone queue for each i

~~O(w)~~ O(1) transition for each state

Time complexity: O(NK)

dp[i][] depends only dp[i - 1][]

Rolling array to reduce space complexity

O(N)

Bowling for Numbers ++

N = 2, K = 1, w = 2

-1 1

Answer = 1, can’t obtained from dp :(

Solution: add `(w-1)` copies of `0`s at the end

Convex Hull Trick (CHT)

Convex Hull Trick

[Computational Geometry](#)

Nothing to do with convex hull algorithm

Maintain lower / upper hull

Query max / min values at some x

Find the best transition quickly

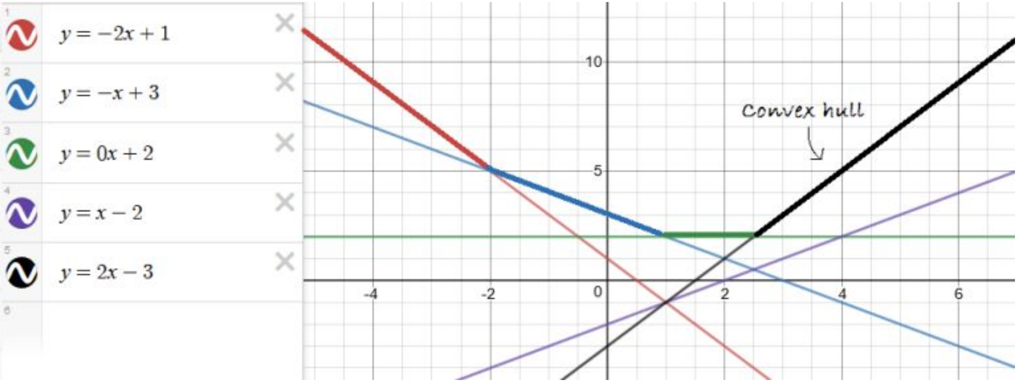


Figure from https://codeforces.com/blog/entry/63823

Sounds scary :O

Today we will use to easier way to learn it :)

Convex Hull Trick

IOI 2002 “Batch Scheduling”

- First(?) CHT task in IOI
- 11 contestants got full scores :o

Other CHT tasks in big competitions

- APIO 2010 Commando
- APIO 2014 Split the Sequence
- IOI 2016 Aliens (60 points), slide

Convex Hull Trick

Useful technique for DP optimization

The basic form of DP formula:

$$dp[i] = \max_{j<i} (dp[j] + f[i] * g[j])$$

Intuitively looks like y = mx + c, a line on the plane

May apply CHT if g is **monotone**

- Easier if f is also monotone

Kalila and Dimna in the Logging Industry

[CF189C Kalila and Dimna in the Logging Industry]

Simplified problem statement:

Given `N`, `a[i]`, `b[i]`, find indices $p_1, ..., p_k$ such that $p_1 = 1, p_k = N$, $p_i < p_{i+1}$ for all `i`, and `sum(a[`p_{i+1}`] * b[`p_i`])` is minimal

Output that minimal sum

$a_1 < a_2 < ... < a_n$ (*** a[] is strictly increasing ***)

$b_1 > b_2 > ... > b_n$ (*** b[] is strictly decreasing ***)

Kalila and Dimna in the Logging Industry

`N = 6, a[] = {1, 2, 3, 10, 20, 30}, b[] = {6, 5, 4, 3, 2, 0}`

If choose `p[] = {1, 2, 4, 6}`

- `sum = a[2] * b[1] + a[4] * b[2] + a[6] * b[4] = 152`

If choose `p[] = {1, 3, 6}`

- `sum = a[3] * b[1] + a[6] * b[3] = 138`
- which is minimal

Kalila and Dimna in the Logging Industry

dp[i] = **minimum sum** obtainable by choosing p[] where **the last index is i**

answer = dp[n]

Base case: `dp[1] = 0`

Transition: $\texttt{dp[i] = min}_{\texttt{j<i}}\texttt{(dp[j] + a[i] * b[j])}$

Naïve implementation: $O(N^2)$

Speed up using CHT!

Convex Hull Trick

$dp[i] = min_{j<i}(dp[j] + a[i] * b[j])$

Consider two indices j, k (1 <= **j < k** < i)

When choose indices j instead of k to update dp[i]? Or vice versa?

Assume we want to choose index k instead of j

- index k gives a better value
- we want to minimize the sum
- dp[j] + a[i] * b[j] > dp[k] + a[i] * b[k]

Convex Hull Trick

a[] is strictly increasing, b[] is strictly decreasing

index k is better than j (j < k)

- dp[j] + a[i] * b[j] > dp[k] + a[i] * b[k]
- dp[j] - dp[k] > a[i] * (b[k] - b[j])
- (dp[j] - dp[k]) / (b[j] - b[k]) > -a[i]

looks like a slope function $(y_j - y_k) / (x_j - x_k)$

Let m(j, k) = (dp[j] - dp[k]) / (b[j] - b[k])

Index k is better than j ⇔ **m(j, k) > -a[i]**

Convex Hull Trick

`Index k is better than j ⇔` **`m(j, k) > -a[i]`**

Property 1: If `m(j, k) < m(k, l)`, then there is no need to consider k

Case 1: If `m(j, k) > -a[i]`

- then surely `m(k, l) > -a[i]`
- k is better than j, but **l is better than k**

Convex Hull Trick

`Index k is better than j` ⇔ **`m(j, k) > -a[i]`**

Property 1: If `m(j, k) < m(k, l)`, then there is no need to consider k

Case 2: If `m(j, k) <= -a[i]`

- **j is not worse than k**

There is no case k must be chosen

Convex Hull Trick

`Index k is better than j ⇔` **`m(j, k) > -a[i]`**

Property 2: If `m(j, k) > -a[i]`, there is no need to consider j in subsequent steps (steps i+1, ..., N)

`a[ ]` is strictly increasing

`m(j, k) > -a[i] > -a[i']`

`k is` **`always better`** `than j in subsequent steps`

Convex Hull Trick

Property 1: If `m(j, k) < m(k, l)`, then there is no need to consider k

we only need to maintain a monotone queue `Q[L..R]`

- such that `m(Q[i], Q[i+1]) >= m(Q[i+1], Q[i+2])`
- **monotone on slope function** instead of values itself

Convex Hull Trick

Property 2: If `m(j, k) > -a[i]`, there is no need to consider j in subsequent steps (steps i+1, ..., N)

`m(Q[i], Q[i+1]) >= m(Q[i+1], Q[i+2])`

we can pop `Q[L]` (front) from the monotone queue

- until `m(Q[L], Q[L+1]) <= -a[i]`

After that, `Q[L]` will be the **best** index,

- `Q[L]` is not worse than `Q[L+1]`, `Q[L+1]` is not worse than `Q[L+2]`, ...

Convex Hull Trick: step by step

Step 1: Pop elements in the front that we will never use again [Property 2]

```
while (R-L >= 1) and (m(Q[L], Q[L+1]) > -a[i])
    L++;
```

Convex Hull Trick: step by step

Step 2: Update answer using Q[L]

```
if (L <= R)
    dp[i] = dp[Q[L]] + a[i] * b[Q[L]];
```

Convex Hull Trick: step by step

Step 3: Pop elements at the back that will never be considered [Property 1]

```
while (R-L >= 1) and (m(Q[R-1], Q[R]) < m(Q[R], i))
    R--;
```

Convex Hull Trick: step by step

Step 4: Insert i at the back

```
R++;
Q[R] = i;
```

Convex Hull Trick: step by step

```
1. while (R-L >= 1) and (m(Q[L], Q[L+1]) > -a[i])
       L++;
2. if (L <= R)
       dp[i] = dp[Q[L]] + a[i] * b[Q[L]];
3. while (R-L >= 1) and (m(Q[R-1], Q[R]) < m(Q[R], i))
       R--;
4. R++;
   Q[R] = i;
```

Convex Hull Trick

CHT (at least in this problem) is variant of monotone queue optimization

The monotonicity does not lie in the values itself, but in the "slope function"

Each transition takes O(1)

Time complexity: O(N)

Convex Hull Trick

Tips for implementing CHT:

1. Write down the condition for “k better than j” and do the algebra correctly

2. When g is not strictly monotone (i.e. may have same values), direct computation of slope formula will give division by 0, special handle it

3. Also, using double for slope calculation may sometimes result in precision error. Use integer multiplication to compare when possible.

Convex Hull Trick

`dp[i] = max`$_{j<i}$`(dp[j] + f[i] * g[j])`

`f/g = i/i, i/d, d/i, d/d, n/i, n/d`

`i`: increasing, `d`: decreasing, `n`: neither

`i/i` and `d/d` are not interesting

For `n/i` and `n/d`, property 2 does not hold; need binary search, std::set

`n/n` can be solved by CDQ D&C

D&C Optimization

D&C Optimization

[Recursion, Divide & Conquer]

Divide the problem into smaller and independent sub-problems that are the same as the original problem

Due to **monotonicity** in problem, D&C can be used to speed up the DP

D&C Optimization

The basic form of DP formula:

`dp[i][j] = min`$_{k<j}$`(dp[i-1][k] + f(k, j))`

Let `C[i][j]` be the smallest index k’ such that

- `dp[i][j] = dp[i-1][k’] + f(k’, j)`
- i.e. the transition from `(i-1, k’)` to `(i, j)` is optimal among all choices of k
- i.e. `dp[i-1][k’] + f(k’, j) <= dp[i-1][k] + f(k, j) for all k`

D&C Optimization

When can we apply D&C Optimization?

```
C[i][j] <= C[i][j+1] for all j
```

Another form of monotonicity!

Ciel and Gondolas

CF321E Ciel and Gondolas

Given `N`, `G`, and an `NxN` symmetric matrix `s[][]` containing values from `0` to `9`

`s[i][i] = 0` for all `i`

Divide `[1, N]` into `G` disjoint groups

- $[1,a_1],[a_1+1,a2], \ldots, [a_{G-1}+1,a_G]$ $(a_G = N)$

Find the minimal total cost

Ciel and Gondolas

For each group [L, R], calculate `sum(s[i][j] | L <= i, j <= R)`

- pairwise sum within group [L, R]
- group [1, 3] → `s[1][1] + s[1][2] + s[1][3] + s[2][1] + … + s[3][3]`

Add them up to get the total cost of this partition

Find the minimal cost

Ciel and Gondolas

N = 5, G = 2

```
0 0 1 1 1
0 0 1 1 1
1 1 0 0 0
1 1 0 0 0
1 1 0 0 0
```

```
Answer = 0 (group = [1, 2], [3, 5])
```

Ciel and Gondolas

`dp[i][j]` = minimal cost of partitioning `[1, j]` into `i` groups

`answer = dp[G][N]`

`Let f(L, R) = sum(s[i][j] | L <= i, j <= R)`

- pairwise sum within `[L, R]`

$$dp[i][j] = \min_{k<j}(dp[i-1][k] + f(k+1, j))$$

Ciel and Gondolas

`dp[i][j] = min`$_{k<j}$`(dp[i-1][k] + f(k+1, j))`

`f(k+1, j)` can be calculated in `O(1)` by 2D partial sum

- Optimization

`O(GN)` state

Naïve implementation: $O(GN^2)$

D&C Optimization → **`O(GN log N)`**

Ciel and Gondolas

When can we apply D&C Optimization?

Let `C[i][j]` be the smallest index k' such that

- `dp[i][j] = dp[i-1][k'] + f(k', j)`

`C[i][j] <= C[i][j+1] for all j`

For this problem, it is true! (see the proof by Alex Tung)

- or you can verify by program

D&C Optimization

Instead of calculating dp iteratively, use recursion instead

The key idea is to write a recursive function to perform the DP transition

```
void solve(int i, int L, int R, int optL, int optR);
```

The above function calculates `dp[i][L..R]`, knowing that `C[i][j]` is between `optL` and `optR` for `L <= j <= R`

D&C Optimization

```
void solve(int i, int L, int R, int optL, int optR);
```

The above function calculates `dp[i][L..R]`, knowing that `C[i][j]` is between `optL` and `optR` for `L <= j <= R`

```
Let M = (L+R)/2
Find dp[i][M] and C[i][M] (opt)
Then call solve() for the left and the right parts
 -  solve(i, L, M-1, optL, opt), solve(i, M+1, R, opt, optR)
```

D&C Optimization: step by step

Step 1: Base case

```
if (L > R) return;
```

D&C Optimization: step by step

Step 2: Find dp[i][M] and C[i][M]

```
int opt = optL;                          //opt represents C[i][M]
for(p = optL + 1; p <= optR; p++)
    if(dp[i-1][p] + f(p+1, M) < dp[i-1][opt] + f(opt+1, M))
    opt = p;
```

(For maximization problems, change “<" to “>")

D&C Optimization: step by step

Step 3: Update dp[i][M]

```
dp[i][M] = dp[i-1][opt] + f(opt+1, M);
```

D&C Optimization: step by step

Step 4: Recursively solve the left and right parts

```
solve(i, L, M - 1, optL, opt);
solve(i, M + 1, R, opt, optR);
```

Here, The condition `C[i][j] <= C[i][j + 1]` is used to narrow the range of candidate transitions from `[optL, optR]` to `[optL, opt]` and `[opt, optR]` respectively.

D&C Optimization: step by step

```
void solve(int i, int L, int R, int optL, int optR){
1. if (L > R) return;

2. int opt = optL;                    //opt represents C[i][M]
   for(p = optL + 1; p <= optR; p++)
     if(dp[i-1][p] + f(p+1, M) < dp[i-1][opt] + f(opt+1, M))
       opt = p;

3. dp[i][M] = dp[i-1][opt] + f(opt+1, M);

4. solve(i, L, M − 1, optL, opt); solve(i, M + 1, R, opt, optR);
 }
```

Ciel and Gondolas

Set `dp[0][0] = 0` and `dp[0][i] = INF for i > 0`

Call `solve(i, 1, N, 1, N) for i = 1, ..., G`

It can be shown that each solve() runs in time complexity O(N log N)

- `log N` layer, each layer iterate `O(N)` elements

Overall time complexity: `O(GN log N)`

Ciel and Gondolas

Minor details:

- Use rolling array for DP calculation
- Huge input (40002 numbers), need fast I/O methods to get AC

Practice Problems for DP optimization

- CF311B Cats Transport
- CF660F Bear and Bowling 4
- Hackerrank Guardians of the Lunatics
- APIO 2010 Commando
- APIO 2014 Splitting the Sequence
- … and more in the CF blog mentioned in reference

Other Interesting DP Problems

- CF 590D Top Secret Task
- CF 489E Hiking
- HKOJ M1331 Resources Conflict
- HKOJ M1724 Guess the Number
- HKOJ M1741 Fill in the Bag

Other DP optimizations

Knuth optimization

Optimization using CDQ D&C

- Will learn CDQ in D&C (II)

"Alien Trick"

- IOI 2016 Alien
- Only 1 contestant got full...
- HKOI 2018 Training Camp slide

References

- Tasks from HKOJ, Codeforces, CCC, NOI
- A summary of different types of DP Optimization
  http://codeforces.com/blog/entry/8219
- HKOI 2021 DP(III)
  https://assets.hkoi.org/training2021/dp-iii.pdf

Model Solutions (by Alex Tung)

- Bowling for Numbers ++
  https://ideone.com/D2LQmj

- Kalila and Dimna in the Logging Industry
  https://ideone.com/Y65oHV

- Ciel and Gondolas
  https://ideone.com/ZQ7pmY

Q & A