

## Quy hoạch động chia để trị (Divide and Conquer DP)

# Quy hoạch động chia để trị (Divide and Conquer DP)


### Tác giả:

- Lê Minh Hoàng - Đại học Khoa học Tự nhiên, ĐHQG-HCM

### Reviewer:

- Ngô Nhật Quang - THPT chuyên Khoa học Tự Nhiên, ĐHQGHN
- Nguyễn Minh Nhật - THPT chuyên Khoa học Tự nhiên, ĐHQGHN
- Nguyễn Minh Hiền - Đại học Công nghệ, ĐHQGHN
- Phạm Hoàng Hiệp - University of Georgia


## Giới thiệu

Bài viết "Tối ưu quy hoạch động 1 chiều" của tác giả Nguyễn Tuấn Tài trong [Tập chí VNOI Xuân Quý Mão \(trang 16\)](#)  đã đề cập đến một phương pháp tối ưu quy hoạch động rất đặc biệt, đó là *phương pháp tối ưu quy hoạch động 1 chiều*.

Trong bài viết này, ta sẽ tìm hiểu về một phương pháp tối ưu quy hoạch động khác, dùng cho các bài toán mà công thức truy hồi có dạng:

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

Công thức trên có độ phức tạp  $\mathcal{O}(mn^2)$ . Ta có thể tối ưu độ phức tạp xuống còn  $\mathcal{O}(mn \log n)$  bằng phương pháp *quy hoạch động chia để trị* nếu hàm chi phí  $C(k, j)$  thoả mãn **điều kiện áp dụng** (được đề cập ở phần tiếp theo).

Ngoài ra, ta còn có thể tối ưu độ phức tạp của công thức trên xuống còn  $\mathcal{O}(mn)$  bằng **kĩ thuật bao lồi (Convex Hull Trick)**  nếu hàm chi phí thoả các điều kiện của kĩ thuật.

## Điều kiện áp dụng

Đầu tiên, ta định nghĩa lại bất đẳng thức tứ giác (quadrangle inequality) và điều kiện đơn điệu (monotonicity condition) như sau:

- ▶ Bất đẳng thức tứ giác:  $\forall a < b \leq c < d$ 
  - ▶ Bất đẳng thức tứ giác xuôi:  $f(a, c) + f(b, d) \leq f(a, d) + f(b, c)$
  - ▶ Bất đẳng thức tứ giác ngược:  $f(a, c) + f(b, d) \geq f(a, d) + f(b, c)$
- ▶ Điều kiện đơn điệu:  $\forall i$ 
  - ▶ Đơn điệu tăng:  $f(i) \leq f(i + 1)$
  - ▶ Đơn điệu giảm:  $f(i) \geq f(i + 1)$

Tiếp theo, ta định nghĩa mảng  $opt$  như sau:

$$opt(i, j) = \arg \min_{k \leq j} [dp(i - 1, k) + C(k, j)]$$

Nói cách khác,  $opt(i, j)$  là giá trị  $k$  nhỏ nhất sao cho  $dp(i - 1, k) + C(k, j)$  đạt giá trị cực tiểu.

Cuối cùng, để có thể áp dụng *quy hoạch động chia để trị* cho công thức đã được đề cập ở trên, mảng  $opt(i)$  cần phải thoả mãn **điều kiện đơn điệu** (tùy bài toán mà ta cần **đơn điệu tăng** hay **đơn điệu giảm**, nhưng vì không mất tính tổng quát nên ta sẽ giả sử là **đơn điệu tăng**):

$$opt(i, j) \leq opt(i, j + 1)$$

Nhưng đôi khi, việc chứng minh điều kiện đơn điệu cho một công thức truy hồi sẽ không hề dễ dàng. Trong một số bài toán, ta có thể chứng minh điều kiện đơn điệu thông qua **bất đẳng thức tứ giác tương ứng** (vì ta đang giả sử **điều kiện đơn điệu** là **đơn điệu tăng** nên **bất đẳng thức tứ giác tương ứng** là **bất đẳng thức tứ giác xuôi**) trên hàm chi phí  $C$ :

$$C(a, c) + C(b, d) \leq C(a, d) + C(b, c), \text{ với } a < b \leq c < d$$

Ta có thể chứng minh rằng, nếu hàm chi phí thoả mãn **bất đẳng thức tứ giác**, thì  $opt(i)$  sẽ thoả mãn **điều kiện đơn điệu tương ứng** (phần chứng minh này sẽ được đặt ở cuối bài viết), tức là:

- ▶ Bất đẳng thức tứ giác xuôi  $\implies$  Đơn điệu tăng
- ▶ Bất đẳng thức tứ giác ngược  $\implies$  Đơn điệu giảm

Lưu ý rằng, ta không thể có được **bất đẳng thức tứ giác** từ **điều kiện đơn điệu tương ứng**, tức là mối quan hệ chỉ xảy ra một chiều.

## Thuật toán

### Ý tưởng

Ý tưởng chính của kỹ thuật này là dựa trên điều kiện  $opt(i, j) \leq opt(i, j + 1)$ .

Giả sử, ta vừa tính được  $opt(i, j)$  trong  $\mathcal{O}(n)$  bằng cách xét tất cả vị trí  $k$  trong đoạn  $[1, n]$ .

Xét  $j' < j$ , ta biết rằng  $opt(i, j') \leq opt(i, j)$ . Do đó, ta có thể tính  $opt(i, j')$  trong  $\mathcal{O}(opt(i, j))$ , thay vì  $\mathcal{O}(n)$ , bằng cách xét tất cả vị trí  $k$  trong đoạn  $[1, opt(i, j)]$ .

## Thuật toán chia để trị

Dựa trên ý tưởng đó, ta có thuật toán chia để trị như sau:

- Đầu tiên, ta tính  $opt(i, n/2)$  trong  $\mathcal{O}(n)$ .
- Tiếp theo, ta tính  $opt(i, n/4)$  (biết rằng  $opt(i, n/4) \leq opt(i, n/2)$ ) và  $opt(i, 3n/4)$  (biết rằng  $opt(i, 3n/4) \geq opt(i, n/2)$ ), tổng độ phức tạp của "tầng" này là  $\mathcal{O}(n)$ .
- Tiếp tục đệ quy để tính  $opt(i, n/8), opt(i, 3n/8), opt(i, 5n/8), opt(i, 7n/8)$ , trong quá trình đệ quy, ta duy trì cận trên và dưới của  $opt$ .

### Độ phức tạp

► Phiên bản cũ

Xét  $j \in [l, r]$ , giả sử ta cần tính tất cả  $opt(i, j)$ , biết rằng  $optl \leq opt(i, j) \leq optr$ .

Đặt  $mid = \lfloor \frac{l+r}{2} \rfloor$ . Nếu độ phức tạp để tính chi phí  $C(k, j)$  là  $\mathcal{O}(1)$  thì ta có thể tính  $opt(i, mid)$  trong  $\mathcal{O}(optr - optl)$ .

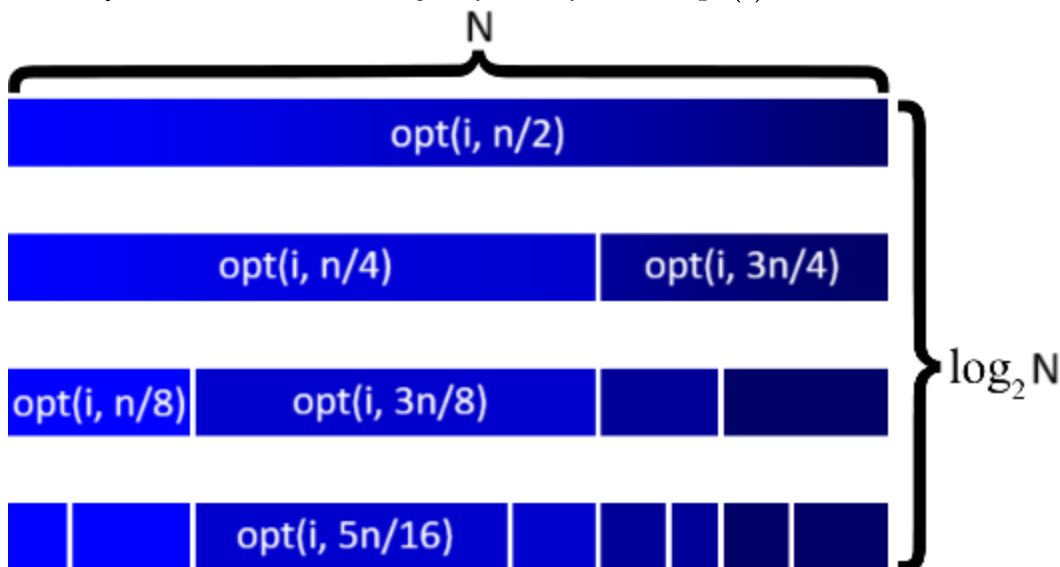
Tiếp theo, ta gọi đệ quy để tính  $opt(i, j)$  của 2 đoạn:

- $j \in [l, mid - 1]$  với  $optl \leq opt(i, j) \leq opt(i, mid)$
- $j \in [mid + 1, r]$  với  $opt(i, mid) \leq opt(i, j) \leq optr$

Nếu bỏ qua  $opt(i, mid)$  thì rõ ràng là  $[optl, opt(i, mid)]$  và  $[opt(i, mid), optr]$  không giao nhau, nên tổng chi phí để tính tất cả  $opt(i, mid)$  thuộc mỗi "tầng" là  $\mathcal{O}(n)$ . Và vì ta luôn chia đôi đoạn cần tính  $[l, r]$  ở mỗi lần đệ quy nên số "tầng" đệ quy là  $\log n$ . Do đó:

- Tổng độ phức tạp để tính  $opt(i)$  là  $\mathcal{O}(n \log n)$ .
- Tổng độ phức tạp của thuật toán sau  $m$  lần tính  $dp$  là  $\mathcal{O}(mn \log n)$ .

Dưới đây là hình minh họa về tổng độ phức tạp để tính  $opt(i)$ :



## Cài đặt

Mặc dù việc triển khai có thể khác nhau tùy theo từng bài toán nhưng chúng đều có một cấu trúc chung.

Với mỗi lần tính  $dp$ , hàm `solve()` gọi `compute(0, n, 0, n)`.

Mỗi khi được gọi, hàm `compute` tính  $opt(i)$  và  $dp(i)$  (`dp_cur`) dựa vào  $dp(i - 1)$  (`dp_before`) và hàm chi phí `C`.

```
1  int m, n;
2  vector<int> dp_before(n + 1), dp_cur(n + 1);
3
4  // hàm chi phí
5  int C(int l, int r);
6
7  // tính dp_cur[l], ..., dp_cur[r]
8  void compute(int l, int r, int optl, int opttr) {
9      if (l > r) return;
10
11     int mid          = (l + r) >> 1;
12     pair<int, int> best = { INT_MAX, -1 };
13
14     // tính dp_cur[mid] và opt[i][mid] dựa vào dp_before và hàm chi
15     for (int k = optl; k <= min(mid, opttr); ++k) {
16         best = min(best, { dp_before[k] + C(k, mid), k });
17     }
18     dp_cur[mid] = best.first;
19     int opt      = best.second;
20
21     // đệ quy để tính dp_cur[l..mid-1] và dp_cur[mid+1..r]
22     compute(l, mid - 1, optl, opt);
23     compute(mid + 1, r, opt, opttr);
24 }
25
26 int solve() {
27     for (int i = 0; i <= n; ++i)
28         dp_before[i] = C(0, i);
29
30     for (int i = 1; i < m; ++i) {
31         compute(0, n, 0, n);
32         dp_before = dp_cur;
33     }
34
35     return dp_before[n];
36 }
```

# Ứng dụng

## Codeforces - 1601C (Optimal Insertion)

Cho 2 mảng  $a_1, a_2, \dots, a_n$  và  $b_1, b_2, \dots, b_m$ .

Ta cần chèn các phần tử của mảng  $b$  vào mảng  $a$  một cách tùy ý (ở đầu, ở giữa, hoặc ở cuối).

Kết quả là ta sẽ nhận được mảng  $c_1, c_2, \dots, c_{n+m}$  có kích thước  $n + m$ .

Nói cách khác, thứ tự của các phần tử của mảng  $a$  trong mảng  $c$  phải được giữ nguyên.

Ngược lại, các phần tử của mảng  $b$  có thể xuất hiện trong mảng  $c$  theo bất kỳ thứ tự nào.

Hỏi số cặp nghịch thế ít nhất có thể của mảng  $c$  là bao nhiêu? Biết rằng một cặp  $(i, j)$  được gọi là nghịch thế nếu  $i < j$  và  $c_i > c_j$ .

Giới hạn:

- $1 \leq n, m \leq 10^6$
- $1 \leq a_i, b_i \leq 10^9$

### Ý tưởng

Đầu tiên, ta sắp xếp mảng  $b$  tăng dần (để  $b_i \leq b_{i+1}$ ).

Với mỗi  $i$ , gọi  $p_i$  là vị trí nhỏ nhất sao cho khi ta chèn  $b_i$  vào trước  $a_{p_i}$  thì số cặp nghịch thế tăng lên là ít nhất. Nếu chèn  $b_i$  ở cuối mảng  $a$  thì  $p_i = n + 1$ .

Ta có nhận xét: Nếu tồn tại  $i < j$  và  $p_i > p_j$  thì ta có thể đổi chỗ  $b_i$  và  $b_j$  để giảm số cặp nghịch thế (bạn đọc có thể tự chứng minh). Do đó, ta có  $p_1 \leq p_2 \leq \dots \leq p_m$ .

Từ đây, ta có thể áp dụng *quy hoạch động chia để trị* để tìm giá trị của mảng  $p$ , sau đó dựng mảng  $c$  và tính số cặp nghịch thế.

Bài toán bây giờ là làm thế nào để tính nhanh giá trị  $p_{mid}$  khi có  $optl$  và  $optr$ .

Để chứng minh, số cặp nghịch thế tăng thêm được xác định bởi biểu thức sau:

$$\underbrace{|a_i > b_{mid}|}_{\text{với } i < optl} + \underbrace{|a_i > b_{mid}|}_{\text{với } optl \leq i < p_{mid}} + \underbrace{|b_{mid} > a_i|}_{\text{với } p_{mid} \leq i \leq optr} + \underbrace{|b_{mid} > a_i|}_{\text{với } optr < i}$$

Nếu biết trước  $optl$  và  $optr$ , rõ ràng (1) và (4) là như nhau với mọi vị trí  $p_{mid} \in [optl, optr]$ , vậy nên ta có thể bỏ qua. Còn lại (2) và (3), ta dựng 2 mảng  $prf$  và  $suf$  để tính nhanh  $p_{mid}$  trong  $\mathcal{O}(optr - optl)$ .

### Cài đặt

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

const int N = 1e6 + 9;
```

```

int n, m;
int a[N], b[N], c[N * 2];

int p[N];
int prf[N], suf[N];

int bit[N * 2];

int calcPos(int optl, int optr, int bi) {
    prf[optl - 1] = suf[optr + 1] = 0;
    for (int i = optl; i <= optr; ++i) prf[i] = prf[i - 1] + (a[i]
    for (int i = optr; i >= optl; --i) suf[i] = suf[i + 1] + (bi >

    int pos = optl;
    for (int i = optl + 1; i <= optr; ++i)
        if (prf[pos - 1] + suf[pos] > prf[i - 1] + suf[i]) pos = i
    return pos;
}

void compute(int l, int r, int optl, int optr) {
    if (l > r) return;

    int mid = (l + r) >> 1;
    p[mid] = calcPos(optl, optr, b[mid]);

    compute(l, mid - 1, optl, p[mid]);
    compute(mid + 1, r, p[mid], optr);
}

void compress_c() {
    vector<int> tmp(c + 1, c + n + 1);
    sort(tmp.begin(), tmp.end());
    tmp.erase(unique(tmp.begin(), tmp.end()), tmp.end());
    for (int i = 1; i <= n; ++i)
        c[i] = lower_bound(tmp.begin(), tmp.end(), c[i]) - tmp.beg
}

void upd(int p, int v) {
    for (; p; p ^= p & -p) bit[p] += v;
}

int get(int p) {
    int res = 0;
    for (; p <= n; p += p & -p) res += bit[p];
    return res;
}

long long solve() {
    cin >> n >> m;
    for (int i = 1; i <= n; ++i) cin >> a[i];
    for (int i = 1; i <= m; ++i) cin >> b[i];
    a[n + 1] = INT_MAX;

```

```

54     sort(b + 1, b + m + 1);
55     compute(1, m, 1, n + 1);
56
57     // dựng mảng c từ mảng a, b và p
58     for (int i = 1, j = 1, sz = 0; i <= n + 1; ++i) {
59         while (j <= m && p[j] == i) c[++sz] = b[j++];
60         if (i <= n) c[++sz] = a[i];
61     }
62     n += m;
63     compress_c();
64
65     fill(bit + 1, bit + n + 1, 0);
66     long long res = 0;
67     for (int i = 1; i <= n; ++i) {
68         res += get(c[i] + 1);
69         upd(c[i], 1);
70     }
71     return res;
72 }
73
74 int main() {
75     int test;
76     cin >> test;
77     while (test--) {
78         cout << solve() << '\n';
79     }
80 }

```

Độ phức tạp thời gian:  $\mathcal{O}(n \log n)$

Độ phức tạp không gian:  $\mathcal{O}(n)$

### VNOJ - nkleaves (Leaves)

Ta có thể tóm tắt bài toán như sau.

- ▶ Cho mảng  $w$  độ dài  $n$  là trọng lượng của  $n$  chiếc lá (đánh số từ 1 đến  $n$ ) và một số  $k$ .
- ▶ Ta cần chia  $n$  chiếc lá thành đúng  $k$  đoạn liên tiếp (độ dài đoạn phải khác 0), mỗi đoạn sẽ gom thành một đồng lá ở vị trí trái cùng. Chi phí để di chuyển một chiếc lá bằng tích trọng lượng của chiếc lá và khoảng cách di chuyển. Nói cách khác, chi phí của một đồng là 
$$\sum_{i=l}^r w_i \times (i - l).$$
- ▶ Yêu cầu: tìm chi phí nhỏ nhất để gom  $n$  chiếc lá thành đúng  $k$  đồng lá.

Giới hạn:

- ▶  $0 < n \leq 10^5$

- $0 < k \leq 10, k < n$
- $w_i \leq 1000$

## Ý tưởng

Đầu tiên, gọi  $C(l, r)$  là chi phí để gom những chiếc lá liên tiếp trong đoạn  $[l, r]$  thành một đồng lá ở vị trí trái cùng  $l$ . Nói cách khác,  $C(l, r) = \sum_{i=l}^r [w_i \times (i - l)]$ .

Để tính nhanh  $C(l, r)$ , ta viết lại công thức như sau:

$$\begin{aligned} C(l, r) &= \sum_{i=l}^r [w_i \times (i - l)] \\ &= \sum_{i=l}^r [w_i \times i - w_i \times l] \\ &= \sum_{i=l}^r (w_i \times i) - l \times \sum_{i=l}^r w_i \end{aligned}$$

Ta tính trước 2 mảng cộng dồn  $prf_1[i] = \sum_{j=1}^i w_j$  và  $prf_2[i] = \sum_{j=1}^i (w_j \times j)$  để có thể tính  $C(l, r)$  trong  $\mathcal{O}(1)$ .

Tiếp theo, đặt  $dp(i, j)$  là chi phí tối thiểu để gom  $j$  chiếc lá đầu tiên thành  $i$  đồng lá. Ta có công thức truy hồi với độ phức tạp thời gian  $\mathcal{O}(kn^2)$  cho bài toán này là

$$dp(i, j) = \min_{i-1 \leq k < j} [dp(i-1, k) + C(k+1, j)], \forall j \geq i$$

Vì hàm chi phí  $C$  thoả bất đẳng thức tứ giác xuôi  $C(a, c) + C(b, d) \leq C(a, d) + C(b, c)$  (bạn đọc có thể tự chứng minh) nên ta có thể áp dụng *quy hoạch động chia để trị*, giảm độ phức tạp thời gian xuống còn  $\mathcal{O}(kn \log n)$ .

## Cài đặt

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

const int N = 1e5 + 9;
int n, k;
int w[N], prf1[N];
long long prf2[N];
vector<long long> dp_before, dp_cur;

long long C(int l, int r) {
    return (prf2[r] - prf2[l - 1]) - 1LL * l * (prf1[r] - prf1[l - 1]);
}

void compute(int l, int r, int optl, int optr) {
    if (l > r) return;
```



```

16
17     int mid                = (l + r) >> 1;
18     pair<long long, int> best = { LONG_LONG_MAX, -1 };
19
20     for (int i = optl; i <= min(mid, optr); ++i) {
21         best = min(best, { dp_before[i] + C(i + 1, mid), i });
22     }
23     dp_cur[mid] = best.first;
24     int opt     = best.second;
25
26     compute(l, mid - 1, optl, opt);
27     compute(mid + 1, r, opt, optr);
28 }
29
30 int main() {
31     cin >> n >> k;
32     for (int i = 1; i <= n; ++i) {
33         cin >> w[i];
34         prf1[i] = prf1[i - 1] + w[i];
35         prf2[i] = prf2[i - 1] + w[i] * i;
36     }
37
38     dp_before.assign(n + 1, LONG_LONG_MAX / 2);
39     dp_before[0] = 0;
40     dp_cur.resize(n + 1);
41     for (int i = 1; i <= k; ++i) {
42         compute(i, n, i - 1, n - 1);
43         dp_before.swap(dp_cur);
44     }
45
46     cout << dp_before[n];
47     return 0;
}

```

Độ phức tạp thời gian:  $\mathcal{O}(kn \log n)$

Độ phức tạp không gian:  $\mathcal{O}(n)$

## Bài tập áp dụng

- ▶ [Codeforces - 321E \(Ciel and Gondolas\)](#) 
- ▶ [Codeforces - 834D \(The Bakery\)](#) 
- ▶ [Codeforces - 868F \(Yet Another Minimization Problem\)](#) 
- ▶ [Atcoder - ARC067D \(Yakiniku Restaurants\)](#) 
- ▶ [Codechef - CHEFAOR \(Chef and Bitwise OR Operation\)](#) 

## Liên hệ giữa bất đẳng thức tứ giác và điều kiện đơn điệu

Vì không mất tính tổng quát, trong phần này, ta sẽ chứng minh rằng, nếu hàm chi phí  $C$  thoả mãn bất đẳng thức tứ giác xuôi  $C(a, c) + C(b, d) \leq C(a, d) + C(b, c)$ , thì  $opt(i)$  sẽ thoả mãn đơn điệu tăng  $opt(i, j) \leq opt(i, j + 1)$ .

Ta sẽ chứng minh điều này bằng phép phản chứng: giả sử tồn tại vị trí  $j$  thoả  $opt(i, j) > opt(i, j + 1)$ .

Để thuận tiện cho việc chứng minh, ta đặt  $p = opt(i, j)$ ,  $q = opt(i, j + 1) \Rightarrow p > q$ , và  $dp_x(i, j) = dp(i - 1, x) + C(x, j)$ . Ta có:

$$\begin{cases} dp_p(i, j) < dp_q(i, j) & (\text{vì } p = opt(i, j)) \\ dp_p(i, j + 1) > dp_q(i, j + 1) & (\text{vì } q = opt(i, j + 1)) \end{cases}$$

$$\Leftrightarrow \begin{cases} dp(i - 1, p) + C(p, j) < dp(i - 1, q) + C(q, j) & (1) \\ dp(i - 1, p) + C(p, j + 1) > dp(i - 1, q) + C(q, j + 1) & (2) \end{cases}$$

Lấy (1) trừ (2), ta được:

$$C(p, j) - C(p, j + 1) < C(q, j) - C(q, j + 1)$$

$$\Leftrightarrow C(p, j) + C(q, j + 1) < C(q, j) + C(p, j + 1)$$

Áp dụng bất đẳng thức tứ giác cho hàm chi phí  $C$  với bộ số  $q < p \leq j < j + 1$ , ta có:

$$C(p, j) + C(q, j + 1) \geq C(q, j) + C(p, j + 1)$$

Điều này là vô lý. Do đó, ta có được điều phải chứng minh.