Sum-constrained convex optimization

Sum-constrained convex optimization

Người viết:

- Đặng Đoàn Đức Trung - UT Austin

Reviewer:

- Nguyễn Minh Nhật - Trường THPT chuyên Khoa học Tự nhiên, ĐHQGHN

Giới thiệu

Mục đích của bài viết này là để giới thiệu cho mọi người về một mô típ đã xuất hiện trong nhiều bài tập trung bình khó tới khó -- tối ưu tổng của một số hàm lồi cho trước. Ý tưởng giải chung của các bài toán này thường là như nhau: tham lam trên chi phí tăng của các hàm lồi này (ta sẽ định nghĩa chi phí tăng ở phần tiếp theo).

Bài viết này là phần 1 của một chuỗi 2 bài viết; phần 2 của chuỗi bài viết này mình sẽ giới thiệu thuật toán tìm tổng Minkowski của các bao lồi.

Lưu ý trước khi đọc

Mình sẽ sử dụng các khái niệm sau xuyên suốt bài viết.

- Hàm f định nghĩa trên tập số nguyên được gọi là **hàm lồi** nếu với mọi $i \in \mathbb{Z}$, ta có $f(i) - f(i-1) \leq f(i+1) - f(i)$.

Bài toán tổng quát

Ta xét bài toán tổng quát sau.

Đề bài

Cho n hàm lồi $f_1, f_2, \ldots, f_n$ được định nghĩa trên tập số nguyên không âm và một giá trị k cho trước. Tìm n số nguyên không âm $x_1, x_2, \ldots, x_n$ sao cho:

- $x_1 + x_2 + \cdots + x_n = k$.
- $f(x_1) + f(x_2) + \cdots + f(x_n)$ là nhỏ nhất có thể.

In ra giá trị nhỏ nhất của $f(x_1) + f(x_2) + \cdots + f(x_n)$.

Giới hạn: $1 \leq n, k \leq 2 \cdot 10^5$.

Phân tích

Để việc trình bày được thuận tiện hơn, đầu tiên ta xét bài toán với $n = 2$.

Xét thao tác sau: khởi tạo $x_1 = 0$ và $x_2 = 0$, ta lặp bước sau k lần:

- Nếu $f(x_1 + 1) - f(x_1) \leq f(x_2 + 1) - f(x_2)$, tăng x_1 lên 1; nếu không thì tăng x_2 lên 1.

Ý tưởng cơ bản của cách làm trên là ở mỗi bước, ta tăng tổng $x_1 + x_2$ lên 1; ta chọn cách làm mà tăng tổng $f(x_1) + f(x_2)$ lên ít nhất.

Để chứng minh cách làm trên là đúng, giả sử đáp án của bài toán là O. Ta xét ba dãy sau:

- A_1 là một dãy dài vô tận mà mỗi phần tử của dãy có ý nghĩa là "chi phí phải trả vào $f(x_1)$ để tăng x_1 từ t lên $t + 1$". Nói cách khác, mỗi phần tử của dãy A_1 có giá trị là $f_1(t + 1) - f_1(t)$ với mọi t.
- Tương tự, A_2 bao gồm các phần tử có ý nghĩa là "chi phí phải trả vào $f(x_1)$ để tăng x_2 từ t lên $t + 1$".
- A bao gồm các phần tử có ý nghĩa là "chi phí phải trả vào tổng $f(x_1) + f(x_2)$ để tăng x_i từ t lên $t + 1$" với $i = 1$ hoặc $i = 2$. Dễ dàng nhận thấy là A là dãy ghép từ A_1 và A_2 ($A = A_1 \cup A_2$).

Nhận xét là nếu ta sắp xếp dãy A theo thứ tự tăng dần rồi lấy tổng k phần tử nhỏ nhất, ta sẽ thu được một số S mà chắc chắn là $O \geq S$.

Ngoài ra, ta cũng có thể chứng minh được là thao tác trên sẽ thu được tổng $f(x_1) + f(x_2)$ đúng bằng S. Để ý rằng khi sắp xếp dãy A_1 tăng dần, ta sẽ có $A_1 = [f_1(1) - f_1(0), f_1(2) - f_1(1), f_1(3) - f_1(2), \ldots]$ vì f_1 là hàm lồi. Tương tự, với dãy A_2, $A_2 = [f_2(1) - f_2(0), f_2(2) - f_2(1), f_2(3) - f_2(2), \ldots]$ sau khi sắp xếp các giá trị của A_2 tăng dần. Nói cách khác, dãy A_1 và A_2 được sắp xếp theo t. Bởi vì A chẳng qua là dãy ghép của A_1 và A_2, dễ dàng nhận thấy qua thuật toán merge sort rằng khi thao tác trên dừng, ta thu được k giá trị nhỏ nhất của A.

Với n lớn hơn, ta có thuật toán tương tự: khởi tạo mọi $x_i = 0$, lặp bước sau k lần:

- Tìm i sao cho $f_i(x_i + 1) - f_i(x_i)$ là nhỏ nhất rồi tăng x_i lên 1.

Ta có thể sử dụng heap để cài đặt thuật toán trên với độ phức tạp là $O((n + k) \log n)$.

Cài đặt

```cpp
int n; // Số lượng hàm f_i
int f(int i, int x){
    // Trả về f_i(x)
}
long long cost_greedy(int k) {
    vector<int> x(n, 0);
    long long ans = 0;
    priority_queue<pair<int, int>> pq;
    // lưu cặp {f_i(x_i) - f_i(x_i + 1), i} vì priority_queue lấy phần tử lớn
    for (int i = 0; i < n; i++) {
        pq.push({f(i, 0) - f(i, 1), i});
    }
    while (k--) {
        int cost = pq.top().first, i = pq.top().second;
        pq.pop();
        ans -= cost; // cost = f_i(x_i) - f_i(x_i + 1)
```

```
        x[i]++;
        pq.push({f(i, x[i]) - f(i, x[i] + 1), i});
    }
    return ans;
}
```

Minh họa tham chi phí tăng

Link bài: CF - 1428E .

Đề bài

Có n củ cà rốt có độ dài $a_1, a_2, \ldots, a_n$. Người chủ muốn cắt các củ cà rốt này các phần *có độ dài nguyên dương* cho k chú thỏ. Một phần cà rốt có độ dài x sẽ mất x^2 giây để ăn. Người chủ này muốn cắt cà rốt thành k phần sao cho tổng thời gian ăn hết k phần này là nhỏ nhất có thể.

Giới hạn:

- $1 \leq n, k \leq 10^5$.
- $1 \leq a_i \leq 10^6$.

Phân tích

Xét hàm $f_i(p)$ là tổng thời gian ăn nhỏ nhất nếu ta chỉ xét củ cà rốt a_i, và củ cà rốt này được chia thành đúng p phần.

Nhận xét rằng ta có thể tính $f_i(x)$ trong độ phức tạp $O(1)$ với mọi x: để ý rằng khi chia củ cà rốt i thành x phần, các phần này cần có độ dài cách nhau tối đa là 1 (ta có thể chứng minh bằng việc nhận xét rằng $f_i(x)$ là tổng $t_1^2 + t_2^2 + \cdots + t_x^2$ với $t_1 + t_2 + \cdots + t_x = a_i$, từ đó ta có thể dùng cách lập luận tham chi phí tăng để chứng minh điều này, chi tiết xin để dành cho bạn đọc).

Quan trọng hơn, ta nhận thấy là mọi f_i là hàm lồi. Để chứng minh điều này với mọi i, ta giả sử củ cà rốt $n+1$ có độ dài là $2a_i$. Nhận xét rằng với mọi p, $2f_i(p) = f_{n+1}(2p)$. Đây là vì nếu củ cà rốt thứ $n+1$ được chia thành các phần có độ dài x và $x+1$, thì số lượng đoạn cà rốt có độ dài x là chẵn, và tương tự với $x+1$; vì thế, ta luôn có thể chia củ cà rốt thứ $n+1$ làm đôi, rồi cắt mỗi phần giống nhau. Ngoài ra, $f_{n+1}(2p) \leq f_i(p-1) + f_i(p+1)$, bởi vì một cách cắt một củ cà rốt có độ dài $2a_i$ thành $2p$ phần là cắt đôi củ cà rốt, rồi cắt nửa đầu thành $p-1$ phần và nửa sau thành $p+1$ phần. Bởi thế, $2f_i(p) \leq f_i(p-1) + f_i(p+1)$, từ đó ta có $f_i(p+1) - f_i(p) \geq f_i(p) - f_i(p-1)$.

Bởi thế, ta có thể dùng ý tưởng tham lam được đề cập ở phần bài toán tổng quát để cài đặt bài này với độ phức tạp là $O((n+k)\log n)$.

Cài đặt

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

struct carrot {
    int a, b;
```

```
    carrot(int _a, int _b) : a(_a), b(_b) {}

    long long value() const {
        // tính f_i(b). mỗi phần có độ dài là a / b hoặc là a / b + 1, và có a
        int cnt = a / b, ov = a % b;
        return 1LL * cnt * cnt * (b - ov) + 1LL * (cnt + 1) * (cnt + 1) * ov;
    }

    long long next() const {
        return carrot(a, b).value() - carrot(a, b + 1).value();
    }

    long long operator<(const carrot& oth) const {
        return next() < oth.next();
    }
};

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, k; cin >> n >> k;
    long long ans = 0;
    priority_queue<carrot> pq;
    for (int i = 0; i < n; i++) {
        int a; cin >> a;
        carrot cur(a, 1);
        ans += cur.value();
        pq.push(cur);
    }
    // để ý rằng x_1 = x_2 = ... = x_n = 1; ta tiếp tục từ đây
    // ta cũng không cần phải quan tâm trường hợp ta cắt một củ cà rốt thành nl
    for (int i = n; i < k; i++) {
        carrot u = pq.top(); pq.pop();
        ans -= u.next();
        pq.push({u.a, u.b + 1});
    }
    cout << ans << "\n";
}
```

Tham chi phí tăng + chặt nhị phân

Link bài: CF - 1344D .

Đề bài

Cho mảng a gồm n số nguyên dương và một số k, bạn cần tìm mảng b gồm n số nguyên thỏa mãn:

- $0 \leq b_i \leq a_i$ với mọi i.

- $\sum_{i=1}^{n} b_i = k$.
- $\sum_{i=1}^{n} b_i(a_i - b_i^2)$ là lớn nhất có thể.

In ra mảng b thỏa mãn 3 điều kiện này.

Giới hạn:

- $1 \le n \le 10^5$.
- $1 \le a_i \le 10^9$.
- $1 \le k \le \sum_{i=1}^{n} a_i$.

Phân tích

Nếu ta đặt $f_i(x) = x(a_i - x^2)$, thì ta nhận thấy là $f_i(x)$ là *hàm lõm* (hay nói cách khác, $-f_i(x)$ là hàm lồi). Để chứng minh điều này, với mọi x ta có

$$f_i(x+1) - f_i(x) = -3x^2 - 3x - 1 + a_i$$

và vế phải giảm dần khi x tăng dần. Vì thế, ta có thể dùng ý tưởng tổng quát như trên. Tuy nhiên, với $k \sim 10^{14}$ ở bài toán này, ta không thể trực tiếp sử dụng thuật toán trên. Để tối ưu, ta có thể chặt nhị phân chi phí tăng lớn thứ k. Khi đang xét chặt nhị phân với giá trị m, ta cần tìm với mọi i giá trị b_i lớn nhất sao cho $f_i(b_i + 1) - f_i(b_i) \ge m$; thao tác này có thể được thực hiện với độ phức tạp $O(1)$ nếu sử dụng công thức phương trình bậc hai, hoặc với độ phức tạp $O(\log a_i)$ nếu sử dụng thêm 1 vòng chặt nhị phân. Vì thế, ta có thể giải bài trên với độ phức tạp là $O(n \log^2 A)$ hoặc $O(n \log A)$ với $A = \sum_{i=1}^{n} a_i$.

Cài đặt

Một số lưu ý khi cài đặt thuật toán trên: sẽ có thể tồn tại giá trị m mà số lượng chi phí tăng $< m$ bé hơn k, nhưng số lượng chi phí tăng $\le m$ thì lại lớn hơn k (vì có thể có nhiều chi phí tăng đúng bằng m). Ta cần phải cẩn thận khi cài đặt trường hợp này.

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

const long long INF = 4E18;

// chi phí tăng từ b lên b + 1 với a
long long cost(long long a, long long b) {
    return a - 3 * b * b - 3 * b - 1;
}

// chặt nhị phân tìm b lớn nhất sao cho cost(a, b) >= t
int get_last(long long a, long long t) {
    int l = 0, r = a + 1;
    while (l + 1 < r) {
        int m = (l + r) / 2;
        (cost(a, m) >= t ? l : r) = m;
    }
    return l;
};
```

```cpp
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    int n; cin >> n;
    long long k; cin >> k;
    vector<int> a(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    // chặt nhị phân tìm giá trị ri nhỏ nhất sao cho số lượng chi phí tăng >=
    long long le = -INF, ri = INF;
    while (le + 1 < ri) {
        long long mi = (le + ri) / 2;
        long long cnt = 0;
        for (int i = 0; i < n; i++) {
            cnt += get_last(a[i], mi);
        }
        (cnt <= k ? ri : le) = mi;
    }
    // có thể xảy ra trường hợp số lượng chi phí tăng >= ri bé hơn k, nhưng số
    // ở đây sau khi dựng đáp án với chi phí tăng >= ri, ta chạy thêm 1 lần để
    // sao cho tổng b đúng bằng k
    long long tot = 0;
    vector<int> b(n);
    vector<pair<long long, int>> nxt;
    for (int i = 0; i < n; i++) {
        b[i] = get_last(a[i], ri);
        tot += b[i];
        if (b[i] < a[i]) {
            nxt.push_back({cost(a[i], b[i]), i});
        }
    }
    sort(nxt.begin(), nxt.end(), greater<pair<long long, int>>());
    // tăng từ tot lên k
    for (int i = 0; i < k - tot; i++) {
        b[nxt[i].second]++;
    }
    for (int i = 0; i < n; i++) {
        cout << b[i] << " ";
    }
}
```

Tham đạo hàm

Link bài: SEERC 2022 - M .

Đề bài

Jerry có một cái cây vô hướng có n đỉnh, trong đó mỗi đỉnh được đặt c_i miếng phô mai.

Có một chú chuột hiện đang đứng ở đỉnh 1 và muốn đi tới đỉnh n để thoát khỏi cái cây này. Ở mỗi bước, chú chuột này sẽ dùng mũi đánh hơi số lượng phô mai ở những đỉnh kề đỉnh chú chuột đang đứng, rồi di chuyển tới một đỉnh kề ngẫu nhiên với xác suất tỉ lệ thuận với số lượng phô mai hiện có ở đỉnh này. Lưu ý rằng chú chuột này sẽ không bao giờ quay trở lại những đỉnh chú chuột đã đi trước đó. Nếu không còn đỉnh kề nào hợp lệ để di chuyển, chú chuột sẽ bị kẹt ở đỉnh này mãi mãi.

Jerry muốn giúp chú chuột này thoát khỏi mê cung (tức là đi tới đỉnh n) với xác suất lớn nhất có thể. Để làm được điều này, Jerry có thể đặt thêm một vài miếng phô mai vào các đỉnh ở trong cây. Tuy nhiên, Jerry chỉ có thể sử dụng tối đa x miếng phô mai, và Jerry phải đặt một số nguyên các miếng phô mai vào mọi đỉnh.

In ra một cách đặt phô mai để Jerry tối đa hóa xác suất chú chuột kia có thể tẩu thoát.

Giới hạn:

- $1 \leq n \leq 2 \cdot 10^5$.
- $1 \leq x, c_i \leq 10^9$.

Phân tích

Đầu tiên ta nhận thấy là ta chỉ nên đặt phô mai ở những đỉnh trên đường đi từ 1 tới n không bao gồm 1. Ngoài ra, với mỗi đỉnh u trên đường đi từ 1 tới n, ta có thể tính được b_u là tổng số lượng phô mai ở những đỉnh là con của cha của u --- nói cách khác, $\frac{c_u}{b_u}$ là xác suất đi tới đỉnh u từ đỉnh cha của u trước khi thao tác. Sau khi thêm x_u miếng phô mai vào đỉnh u, xác suất này trở thành $\frac{c_u+x_u}{b_u+x_u}$. Vì thế, xác suất đi tới n sẽ là là $\prod_{u\in\{1\to n\}} \frac{c_u+x_u}{b_u+x_u}$. Để thuận tiện hơn, ta xét logarit của hàm này: $\sum_{u\in\{1\to n\}} \log(c_u+x_u) - \log(b_u+x_u)$.

Nhận xét rằng nếu ta đặt $f_u(t) = \log(c_u+t) - \log(b_u+t)$, thì f_u là hàm lõm. Vì thế, ta có thể đưa về bài toán quen thuộc sau: chọn x_u sao cho:

- x_u là số nguyên không âm với mọi $u \in \{1 \to n\}$.
- $\sum_{u\in\{1\to n\}} x_u = x$.
- $\sum_{u\in\{1\to n\}} f_u(x_u)$ là lớn nhất có thể.

Ta có thể áp dụng kĩ thuật ở trên: chặt nhị phân chi phí tăng cuối cùng t, rồi ở mỗi đỉnh u ta chặt nhị phân x_u nhỏ nhất sao cho $f_u(x_u+1) - f_u(x_u) < t$. Tuy nhiên, độ phức tạp của cách làm này là $O(n \log x \log \epsilon^{-1})$ với ϵ là độ chính xác cần đạt được; với bài toán này, cách làm này sẽ bị vượt quá thời gian cho phép (ϵ có thể xuống tới 10^{-19}).

Một ý tưởng cho bài toán này là *bỏ giới hạn rằng x_u là số nguyên*, rồi từ nghiệm thực $\hat{x}_u$ nhận được, ta chuyển về x_u bằng cách nào đó (kĩ thuật này được gọi là integer program relaxation). Nói cách khác, xét bài toán sau: cho $f_u(t) = \log(c_u+t) - \log(b_u+t)$, chọn $\hat{x}_u$ sao cho

- x_u là *số thực không âm* với mọi $u \in \{1 \to n\}$.
- $\sum_{u\in\{1\to n\}} \hat{x}_u = x$.
- $\sum_{u\in\{1\to n\}} f_u(\hat{x}_u)$ là lớn nhất có thể.

Ở đây, hàm g trên tập số thực là hàm lõm nếu $g'(t) \leq 0$ với mọi t. Tương tự, hàm g trên tập số thực là hàm lồi nếu $g'(t) \geq 0$ với mọi t. Nhận thấy là hàm f_u được định nghĩa ở trên cũng là một hàm lõm trên tập số thực.

Để giải bài toán trên với nghiệm thực, ta xét ý tưởng tham chi phí tăng nhưng với bước tiến nhỏ. Nói cách khác, ta chọn một hằng số nhỏ dx nào đó, rồi thực hiện thao tác sau $\frac{x}{dx}$ lần:

- Chọn u sao cho $f_u(\hat{x}_u + dx) - f_u(\hat{x})$ lớn nhất có thể. Một cách diễn đạt tương đương là ta chọn u sao cho $\frac{f_u(\hat{x}_u+dx)-f_u(\hat{x})}{dx}$ là lớn nhất có thể.
- Tăng $\hat{x}_u$ lên dx.

Ta nhận thấy là với nghiệm nguyên, ta lặp thao tác trên với $dx = 1$. Với nghiệm thực, ta lặp thao tác trên nhiều lần với dx tiến tới 0. Nhận thấy rằng khi $dx \to 0$, $\frac{f_u(\hat{x}_u+dx)-f_u(\hat{x})}{dx} = f'_u(\hat{x}_u)$. Từ nhận xét trên và bởi vì f'_u là hàm liên tục, ta có thể kết luật rằng nghiệm $\hat{x}$ thỏa mãn điều kiện sau với mọi u:

- Với mọi u sao cho $\hat{x}_u > 0$, đạo hàm của f_u tại các điểm này là bằng nhau. Nói cách khác, tồn tại số t sao cho với mọi u mà $\hat{x}_u > 0$, ta có $f'_u(\hat{x}_u) = t$.
- Với mọi u sao cho $\hat{x}_u = 0$, ta có $f'_u(\hat{x}_u) \le t$.

Vì thế, ta có thể tìm nghiệm thực của bài toán trên như sau: ta chặt nhị phân giá trị t; ở mỗi vòng chặt nhị phân và với mỗi u, ta giải $\hat{x}_u$ sao cho $f'_u(\hat{x}_u) = t$ (hoặc gán $\hat{x}_u = 0$ nếu $f'(\hat{x}_u) \le t$). Độ phức tạp của phần này là $O(n \log \epsilon^{-1})$ vì ta có thể trực tiếp giải $\hat{x}_u$.

▶ Chi tiết giải

Ta nhận được nghiệm thực $\hat{x}_u$ của bài toán thực trên; bây giờ ta phải chuyển đổi nghiệm thực này thành nghiệm nguyên x_u của bài toán ban đầu. Ta có nhận xét cuối cùng: nghiệm dương phải thỏa mãn $x_u \ge \lfloor \hat{x}_u \rfloor$ với mọi u, hoặc $x_u \le \lceil \hat{x}_u \rceil$ với mọi u.

▶ Chứng minh

Từ hai nhận xét này, ta nhận thấy nghiệm nguyên có thể được tạo nên từ nghiệm thực bằng một trong hai cách sau:

$x_u = \lfloor \hat{x}_u \rfloor$

- Khởi tạo $x_u = \lfloor \hat{x}_u \rfloor$ rồi chạy tham chi phí tăng không quá n bước (vì $\sum \lfloor \hat{x}_u \rfloor \ge x - n$).
- Khởi tạo $x_u = \lceil \hat{x}_u \rceil$ rồi chạy thuật toán đảo của tham chi phí tăng (xoá chi phí giảm nhỏ nhất) không quá n bước (vì $\sum \lceil \hat{x}_u \rceil \le x + n$).

Trên thực tế, với riêng bài toán này, ta chỉ cần sử dụng cách thứ nhất. Mình không biết cách chứng minh điều này, tuy nhiên kể cả khi ta thực hiện cả 2 cách thì độ phức tạp của phần này là $O(n \log n)$.

Vì thế, bài toán có thể được giải với độ phức tạp $O(n(\log n + \log \epsilon^{-1}))$.

Cài đặt

Ở đây mình không sử dụng ϵ; thay vào đó, điều kiện thoát của mình là khi tổng của $\hat{x}_u$ cách x tối đa là n.

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

```

```cpp

long double log_prob(long double c, long double b) {
    return log(c) - log(b);
}

long double cost(long double c, long double b, long double add) {
    return log_prob(c + add + 1, b + add + 1) - log_prob(c + add, b + add);
}

// giải phương trình d/dx(log(c + x) - log(b + x)) = t
long double solve(long double c, long double b, long double t) {
    if (t * b >= (b - c) / c) {
        return 0.0;
    } else {
        return 0.5 * (sqrt(4 * (b - c) / t + (c - b) * (c - b)) - c - b);
    }
}

int main() {
    int n, x; cin >> n >> x;
    vector<int> c(n);
    vector<vector<int>> adj(n);
    for (int i = 0; i < n; i++) {
        cin >> c[i];
    }
    for (int i = 1; i < n; i++) {
        int u, v; cin >> u >> v; u--; v--;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vector<tuple<int, int, int>> pat;
    {
        // khởi tạo đường đi từ 1 tới n
        function<bool(int, int)> DFS;
        DFS = [&](int u, int p) {
            if (u == n - 1) {
                return true;
            } else {
                int nod = -1, tot = 0;
                for (int v : adj[u]) {
                    if (v != p) {
                        tot += c[v];
                        if (DFS(v, u)) {
                            nod = v;
                        }
                    }
                }
                // ta chỉ thêm đỉnh vào pat nếu đỉnh này thỏa mãn b_u != c_u
                if (nod != -1 && c[nod] != tot) {
                    pat.push_back({c[nod], tot, nod});
                }
```

```
                return nod != -1;
            }
        };
        DFS(0, -1);
    }
    // chặt nhị phân đạo hàm
    long double le = 0, ri = 1;
    while (true) {
        long double mi = (le + ri) / 2, cur = 0;
        for (auto [c, b, i] : pat) {
            cur += solve(c, b, mi);
        }
        (cur <= x ? ri : le) = mi;
        // điều kiện thoát: x - n <= tổng \hat{x}_i <= x
        if (cur <= x && cur + n >= x) {
            break;
        }
    }
    vector<int> ans(n);
    int tot = 0;
    // khởi tạo nghiệm nguyên là phần nguyên của nghiệm thực
    priority_queue<tuple<long double, int, int, int>> pq;
    for (auto [c, b, i] : pat) {
        int add = solve(c, b, ri);
        ans[i] = add;
        tot += ans[i];
        pq.push({cost(c, b, ans[i]), c, b, i});
    }
    // chạy thêm bài bước tham chi phí tăng
    while (tot < x && !pq.empty()) {
        auto [ignore, c, b, i] = pq.top(); pq.pop();
        ans[i]++;
        tot++;
        pq.push({cost(c, b, ans[i]), c, b, i});
    }
    for (int i = 0; i < n; i++) {
        cout << ans[i] << " ";
    }
}
```

Bài tập áp dụng

- CodeSprint 2023 - L
- ICPC Danang 2019 - F