Lời giải bài: Fucs và gánh nặng mưu sinh

Nhóm 2: Nguyễn Trường Giang – Nguyễn Ngọc Hưng

Ngày 15 tháng 12 năm 2025

1 Mô tả bài toán

Server của Fucs nhận một hàng đợi gồm N tác vụ cần xử lý trên một CPU. Mỗi tác vụ i được mô tả bởi ba số:

- r_i: thời gian chạy liên tục của tác vụ;
- t_i: hạn hoàn thành (deadline);
- p_i: lợi nhuận nếu hoàn thành đúng hạn.

CPU là tài nguyên hữu hạn và **không được phép ngắt quãng**: một khi đã bắt đầu chạy tác vụ thì phải chạy đến khi kết thúc, rồi mới chuyển sang tác vụ tiếp theo.

Quy tắc:

- Mỗi tác vụ có thể được làm hoặc bỏ qua.
- Nếu chạy tác vụ i xong tại thời điểm C_i:
  - nếu $C_i \leq t_i$ thì nhận được lợi nhuận p_i;
  - nếu $C_i > t_i$ thì coi như thất bại, lợi nhuận của tác vụ này là 0.

Mục tiêu: chọn tập tác vụ và sắp xếp thứ tự thực hiện trên CPU để **tổng lợi nhuận nhận được là lớn nhất có thể**.

2 Phương pháp thiết kế thuật toán

2.1 Ý tưởng tổng quát

Nhận xét đầu tiên:

- Giả sử ta có hai tác vụ A và B với $t_A > t_B$ nhưng lịch hiện tại lại chạy A trước rồi mới tới B. Nếu hoán đổi thứ tự, chạy B trước rồi A, thì các deadline không bị xấu đi, vì tác vụ có deadline sớm hơn được chạy sớm hơn.
- Từ đó suy ra: tồn tại một lịch tối ưu trong đó các tác vụ được sắp xếp theo thứ tự **deadline không giảm**.

Vì vậy, bước đầu tiên của thuật toán là **sắp xếp các tác vụ theo t_i tăng dần** (cùng deadline thì ưu tiên r_i nhỏ, rồi p_i lớn). Đoạn `sort` trong code thực hiện đúng điều này.

Sau khi đã sắp xếp, ta lần lượt xét từng tác vụ theo thứ tự mới. Ở mỗi bước i:

- ta đã có một số cách chọn một tập con các tác vụ trong $1..i-1$, mỗi cách cho ra một lịch hợp lệ với tổng thời gian và tổng lợi nhuận khác nhau;
- khi thêm tác vụ thứ i, với mỗi lịch cũ ta có hai lựa chọn: **làm** hoặc **không làm** tác vụ đó.

Thay vì giữ nguyên cả lịch (danh sách tác vụ), ta chỉ giữ **hai thông số tóm tắt** của mỗi lịch:

- R: tổng thời gian CPU đã chạy (tổng các r_j đã chọn);
- P: tổng lợi nhuận thu được.

Như vậy, thuật toán là một dạng **quy hoạch động "chọn / bỏ"** trên các cặp (R, P), nhưng có thêm một bước rất quan trọng: **cắt bỏ các trạng thái thua kém** để không bị tràn số trạng thái.

2.2 Trạng thái và cắt bỏ trạng thái thua kém

Gọi một lịch sau khi xét xong i tác vụ đầu tiên được tóm tắt bằng cặp (R, P):

- R là tổng thời gian chạy hiện tại;
- P là tổng lợi nhuận đã nhận được.

Ta chỉ lưu một danh sách `prev` gồm các trạng thái như vậy, sao cho:

- các trạng thái được sắp theo R tăng dần;
- khi R tăng thì P cũng tăng dần (mỗi trạng thái sau luôn có lợi nhuận lớn hơn trạng thái trước).

Điều này đạt được bằng cách **cắt bỏ các trạng thái thua kém**:

> Nếu có hai trạng thái (R_1, P_1) và (R_2, P_2) với $R_1 \le R_2$ và $P_1 \ge P_2$, thì trạng thái (R_2, P_2) luôn kém hơn về mọi mặt. Dù về sau ta có thêm tác vụ nào, bắt đầu từ (R_1, P_1) luôn cho lịch không tệ hơn bắt đầu từ (R_2, P_2). $\Rightarrow$ có thể xoá (R_2, P_2) khỏi danh sách.

Nhờ thường xuyên xoá những trạng thái như vậy, kích thước danh sách trạng thái sau khi xử lý i tác vụ chỉ ở mức khoảng $O(i)$, không bị tràn.

2.3 Chuyển trạng thái khi xét tác vụ thứ i

Sau khi đã xử lý $i-1$ tác vụ, ta có danh sách `prev`:

$$\texttt{prev} = \big\{(R_k, P_k)\big\}_{k=0}^{K-1}$$

trong đó R_k tăng dần và P_k cũng tăng dần.

Giờ xét tác vụ $J = a[i]$ với thời gian chạy r, deadline t, profit p.

Bước 1: Tạo các ứng viên mới

Từ mỗi trạng thái cũ (R_k, P_k) ta có:

- **Không làm tác vụ** J: giữ nguyên lịch cũ

$$(R_k, P_k) \to (R_k, P_k).$$

- **Làm tác vụ** J: đặt J ở *cuối* lịch hiện tại, khi đó thời gian kết thúc mới là $R_k + r$. Chỉ chấp nhận nếu

$$R_k + r \le t,$$

để kịp deadline, và thu được trạng thái mới:

$$(R_k, P_k) \to (R_k + r,\ P_k + p).$$

Trong code:

- ta tìm `limit` — số lượng trạng thái đầu tiên trong `prev` sao cho $R_k + r \le t$. Vì R_k tăng dần, đây chính là một đoạn *prefix* của `prev`.
- các trạng thái "không lấy J" (gọi là dãy A) là toàn bộ `prev`;
- các trạng thái "có lấy J" (gọi là dãy B) tương ứng với `prev[0..limit-1]` sau khi cộng thêm r, p.

Bước 2: Hợp nhất hai dãy đã sắp

Dãy A đã sắp theo R tăng dần. Dãy B cũng đã sắp theo R tăng dần (vì lấy từ một prefix của `prev`).

Ta thực hiện một bước **merge** giống mergesort giữa A và B:

- sử dụng hai con trỏ `ia` và `ib` duyệt trên A và B;
- luôn chọn phần tử có R nhỏ hơn để đẩy vào mảng `merged`;
- nếu R bằng nhau thì chọn phần tử có P lớn hơn (lịch lời hơn), phần còn lại bỏ qua;
- mỗi phần tử trong `merged` còn lưu:
  - `par`: chỉ số trạng thái cha trong `prev` (để truy vết sau này);
  - `tk`: ID của task nếu tại đây ta "làm J", hoặc -1 nếu là trạng thái "không làm J".

Như vậy, `merged` chứa tất cả các trạng thái có thể có sau khi xét xong tác vụ J, đã được sắp theo R tăng dần, nhưng vẫn còn nhiều trạng thái thua kém.

Bước 3: Cắt bỏ trạng thái thua kém

Ta duyệt `merged` từ trái sang phải, dùng một biến `bestP` để ghi nhận lợi nhuận lớn nhất đã gặp.

- Nếu một trạng thái mới có $P >$ `bestP` thì ta giữ lại: thêm nó vào mảng `cur` và cập nhật `bestP`.
- Nếu $P \leq$ `bestP` thì trạng thái này thua kém (cùng hoặc nhiều thời gian hơn mà lời không hơn), ta bỏ qua.

Sau bước này:

- `cur` lại là danh sách trạng thái tốt: R tăng dần và P cũng tăng dần;
- ta tạo mảng `next` chứa chỉ cặp (R, P) để dùng cho vòng lặp sau, đồng thời lưu `parents[i]` và `takes[i]` từ các trạng thái trong `cur`.

Cuối cùng, gán `prev = next`. Như vậy, ở bước $i + 1$ ta lại làm tương tự với tác vụ tiếp theo.

2.4 Truy vết lịch tác vụ

Sau khi xử lý hết N tác vụ, `prev` chứa các trạng thái tốt sau cùng. Do P tăng dần theo R, trạng thái cuối cùng trong `prev` luôn có lợi nhuận lớn nhất. Đó là trạng thái nghiệm tối ưu.

Để khôi phục dãy tác vụ:

- đặt `at` = chỉ số trạng thái cuối cùng;
- duyệt ngược i từ $N - 1$ xuống 0:
  - nếu `takes[i][at] != -1` nghĩa là ở bước i ta đã chọn tác vụ có ID đó, thêm ID này vào danh sách;
  - cập nhật `at = parents[i][at]` để quay lại trạng thái cha.
- sau khi duyệt xong, đảo ngược danh sách ID để có thứ tự từ trái sang phải.

Danh sách này chính là dãy tác vụ cần in ở dòng thứ hai.

3 Tính phù hợp của phương pháp

Subtask 1: $N \leq 20$

Với N rất nhỏ, ta có thể:

- duyệt mọi tập con tác vụ (2^N), sắp xếp theo nhiều cách và chọn nghiệm tốt nhất;
- hoặc làm quy hoạch động kiểu knapsack theo tổng thời gian chạy.

Tuy nhiên, nhóm chọn dùng luôn **quy hoạch động theo cặp (thời gian, lợi nhuận) và cắt bỏ trạng thái thua kém** như trên cho cả hai subtask, vì:

- thuật toán vẫn cho **nghiệm tối ưu tuyệt đối** (không phải heuristic hay xấp xỉ);
- độ phức tạp với $N \leq 20$ rất nhỏ, dễ debug và so khớp với test mẫu;
- không cần viết hai lời giải khác nhau cho hai subtask, report gọn gàng hơn.

Subtask 2: $N \leq 500$

Trong subtask 2, các giá trị r_i, t_i, p_i có thể lớn đến 10^{12}. Những hướng "DP theo thời gian" kiểu:

$$\texttt{dp[time]} = \text{lợi nhuận tốt nhất nếu dùng time}$$

là không khả thi, vì `time` có thể quá lớn, không thể dựng bảng.
Thuật toán mà nhóm sử dụng phù hợp vì:

- Trạng thái chỉ dùng **các cặp** (R, P) **thực sự xuất hiện**, không quét theo mọi giá trị thời gian từ 0 đến $\max t_i$.
- Sau mỗi bước, ta **cắt bỏ** các trạng thái thua kém, nên số trạng thái còn lại chỉ cỡ $O(i)$, với i là số tác vụ đã xét.
- Việc thêm một tác vụ mới chỉ cần gộp hai dãy trạng thái đã sắp và duyệt tuyến tính để cắt bỏ, không cần heuristic phức tạp, không cần tuning tham số.
- Toàn bộ quá trình vẫn là **quy hoạch động chính xác**: mọi cách chọn/bỏ đều được xét, mọi trạng thái bị loại đều chắc chắn không thể dẫn đến nghiệm tốt hơn về sau.

Như vậy, phương pháp *"DP theo cặp (thời gian, lợi nhuận) + cắt bỏ trạng thái thua kém"* vừa đảm bảo nghiệm tối ưu, vừa có độ phức tạp đủ nhỏ cho $N \leq 500$, nên phù hợp cho cả hai subtask.

4 Phân tích độ phức tạp thời gian và không gian

Độ phức tạp thời gian

Gọi N là số tác vụ.

- **Bước sắp xếp ban đầu:** Sắp xếp mảng `a` theo $(t_i, r_i, -p_i)$ tốn

$$O(N \log N).$$

- **Vòng lặp quy hoạch động:** Giả sử sau khi xét i tác vụ đầu tiên, số trạng thái hiện có là K_i. Dựa trên cách cắt bỏ trạng thái kém, ta có thể xem $K_i = O(i)$ (vì khi R tăng thì P cũng tăng, nên không thể có quá nhiều trạng thái).

  Ở bước i:

- ta chia thành hai dãy A (không lấy tác vụ i) và B (lấy tác vụ i từ một prefix), tổng kích thước $\leq K_i + K_i = 2K_i$;
- merge hai dãy A và B tốn $O(K_i)$ vì chúng đều đã sắp theo R;
- cắt bỏ trạng thái kém bằng cách duyệt `merged` một lần nữa, cũng chỉ tốn $O(K_i)$.

Vậy chi phí cho bước i là $O(K_i) = O(i)$ và tổng chi phí DP là:

$$\sum_{i=1}^{N} O(i) = O(N^2).$$

Kết hợp lại, độ phức tạp thời gian tổng thể:

$$T(N) = O(N \log N + N^2) = O(N^2).$$

Với $N \leq 500$, $N^2 = 2{,}5 \cdot 10^5$, hoàn toàn thoải mái trong giới hạn 5 giây của đề bài.

Độ phức tạp bộ nhớ

Các cấu trúc dữ liệu chính:

- Mảng tác vụ `a` kích thước N: $O(N)$.
- Các vector `prev`, `merged`, `cur` chứa trạng thái: mỗi cái tối đa khoảng $O(N)$ phần tử (vì sau bước i có $O(i)$ trạng thái).
- Các mảng `parents` và `takes`:
  - ở bước i, có khoảng $K_i = O(i)$ trạng thái;
  - tổng số phần tử lưu trong toàn bộ `parents` và `takes` là

$$\sum_{i=1}^{N} K_i = O(N^2).$$

Do đó, độ phức tạp bộ nhớ là:

$$S(N) = O(N^2),$$

rất nhỏ so với giới hạn 1024MB của đề, kể cả khi mỗi trạng thái dùng kiểu `__int128` để lưu tổng lợi nhuận.

5 Code demo

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

using ll = long long;

```

```cpp
struct Job { int id; ll r, t, p; };

struct Node { ll R; __int128 P; };
struct Out  { ll R; __int128 P; int par, tk; };

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int N; cin >> N;
    vector<Job> a(N);
    for (int i = 0; i < N; ++i) {
        cin >> a[i].r >> a[i].t >> a[i].p;
        a[i].id = i + 1;
    }

    sort(a.begin(), a.end(), [](const Job& A, const Job& B){
        if (A.t != B.t) return A.t < B.t;
        if (A.r != B.r) return A.r < B.r;
        return A.p > B.p;
    });

    vector<Node> prev;
    prev.push_back({0, (__int128)0});

    vector<vector<int>> parents, takes;
    parents.reserve(N);
    takes.reserve(N);

    vector<Out> merged; merged.reserve(1<<12);
    vector<Out> cur;    cur.reserve(1<<12);

    for (int i = 0; i < N; ++i) {
        const Job &J = a[i];

        int limit = (int)prev.size();
        while (limit > 0 && prev[limit-1].R + J.r > J.t) --limit;

        merged.clear();
        merged.reserve(prev.size() + limit);

        int ia = 0, ib = 0;
        auto atA = [&](int k)->Out { return { prev[k].R, prev[k].
            P, k, -1 }; };
        auto atB = [&](int k)->Out { return { prev[k].R + J.r,
            prev[k].P + (__int128)J.p, k, a[i].id }; };

        while (ia < (int)prev.size() || ib < limit) {
            bool takeA = false;
            if (ib == limit) takeA = true;
            else if (ia == (int)prev.size()) takeA = false;
```

```
            else {
                ll Ra = prev[ia].R, Rb = prev[ib].R + J.r;
                if (Ra < Rb) takeA = true;
                else if (Ra > Rb) takeA = false;
                else { // Ra == Rb
                    __int128 Pa = prev[ia].P, Pb = prev[ib].P + (
                       __int128)J.p;
                    takeA = (Pa >= Pb);
                }
            }
            merged.push_back(takeA ? atA(ia++) : atB(ib++));
        }

        cur.clear(); cur.reserve(merged.size());
        __int128 bestP = -1;
        for (const auto &o : merged) {
            if (o.P > bestP) {
                cur.push_back(o);
                bestP = o.P;
            }
        }

        vector<int> par(cur.size()), tk(cur.size());
        vector<Node> next; next.reserve(cur.size());
        for (size_t k = 0; k < cur.size(); ++k) {
            par[k] = cur[k].par;
            tk[k]  = cur[k].tk;
            next.push_back({cur[k].R, cur[k].P});
        }
        parents.push_back(move(par));
        takes.push_back(move(tk));
        prev.swap(next);
    }

    long long bestProfit = prev.empty() ? 0LL : (long long)prev.
       back().P;

    // backtrack
    vector<int> ids;
    ids.reserve(N);
    int at = (int)prev.size() - 1;
    for (int i = N - 1; i >= 0 && at != -1; --i) {
        int tk = takes[i][at];
        if (tk != -1) ids.push_back(tk);
        at = parents[i][at];
    }
    reverse(ids.begin(), ids.end());

    cout << bestProfit << "\n";
    for (size_t i = 0; i < ids.size(); ++i) {
        if (i) cout << ' ';
```

```
104             cout << ids[i];
105         }
106         cout << "\n";
107         return 0;
108 }
```