Fucs và gánh nặng mưu sinh - Lời Giải

Nhóm 3

December 2025

1 Phát biểu bài toán

Server của Fucs nhận được một hàng đợi gồm N tác vụ cần xử lý. Do tài nguyên CPU hữu hạn, tại mỗi thời điểm server chỉ có thể xử lý **một tác vụ duy nhất**. Khi một tác vụ đã bắt đầu chạy, nó phải được thực thi liên tục cho đến khi hoàn thành, không được phép ngắt quãng (nguyên tắc *atomic execution*).

Mỗi tác vụ i được mô tả bởi ba tham số:

- r_i: thời gian chạy của tác vụ.
- t_i: hạn hoàn thành (deadline).
- p_i: lợi nhuận thu được nếu tác vụ hoàn thành không muộn hơn t_i.

Ta có quyền **chọn hoặc không chọn** mỗi tác vụ. Nếu một tác vụ được chọn nhưng hoàn thành sau thời điểm t_i, thì tác vụ đó được xem là thất bại và **không mang lại lợi nhuận**.

Mục tiêu

Hãy chọn một tập con các tác vụ và sắp xếp thứ tự thực hiện chúng sao cho:

- Mọi tác vụ được chọn đều hoàn thành không muộn hơn deadline của chính nó.
- Tổng lợi nhuận thu được là **lớn nhất có thể**.

Input

- Dòng đầu tiên chứa số nguyên N ($1 \leq N \leq 500$).
- N dòng tiếp theo, dòng thứ i chứa ba số nguyên r_i, t_i, p_i.

Output

- Dòng thứ nhất in ra tổng lợi nhuận lớn nhất có thể đạt được.
- Dòng thứ hai in ra dãy chỉ số các tác vụ được chọn (đánh số từ 1 đến N), theo đúng thứ tự thực hiện.

Ràng buộc và yêu cầu chấm điểm

Bài toán gồm hai phần:

- **Subtask 1**: $N \leq 20$, yêu cầu tìm nghiệm tối ưu tuyệt đối.
- **Subtask 2**: $N \leq 500$, chỉ yêu cầu tìm nghiệm gần đúng tốt; điểm số được tính dựa trên độ chênh lệch so với nghiệm của Ban ra đề.

Nhận xét quan trọng Nếu một tập tác vụ S có tồn tại một lịch thực hiện hợp lệ, thì luôn có thể sắp xếp các tác vụ trong S theo thứ tự **deadline tăng dần (Earliest Deadline First – EDD)** mà vẫn đảm bảo hợp lệ. Nhận xét này là nền tảng cho các thuật toán được trình bày ở các phần sau.

2 Phân tích và đánh giá phương pháp

2.1 Phương pháp thiết kế thuật toán

Subtask 1 ($N \leq 20$): Brute Force kết hợp Backtracking

Với kích thước nhỏ của dữ liệu, ta sử dụng phương pháp **duyệt toàn bộ** các tập con tác vụ. Dựa trên nhận xét rằng nếu một tập tác vụ hợp lệ thì luôn tồn tại thứ tự thực hiện theo **Earliest Deadline First (EDD)**, ta sắp xếp các tác vụ theo deadline tăng dần và lần lượt quyết định tại mỗi vị trí việc *chọn* hay *bỏ* tác vụ đó.

Thuật toán được cài đặt bằng **DFS/backtracking**, trong đó:

- Trạng thái gồm vị trí hiện tại, tổng thời gian đã dùng và tổng lợi nhuận.
- Chỉ cho phép chọn tác vụ nếu không vi phạm deadline.
- Sử dụng cận trên đơn giản (tổng lợi nhuận còn lại) để cắt nhánh.

Cách tiếp cận này đảm bảo tìm được nghiệm tối ưu tuyệt đối cho subtask 1.

Subtask 2 ($N \leq 500$): Greedy + Heuristic (Randomized Greedy Repair)

Với dữ liệu lớn hơn, việc duyệt toàn bộ không còn khả thi. Ta sử dụng một thuật toán **heuristic dạng greedy có sửa lỗi (repair)** kết hợp với **random hóa**.

Ý tưởng chính:

- Sắp xếp các tác vụ theo thứ tự deadline tăng dần (EDD).
- Duyệt lần lượt các tác vụ, tạm thời thêm vào tập chọn.
- Nếu tổng thời gian vượt quá deadline hiện tại, loại bỏ một tác vụ "kém hiệu quả" nhất khỏi tập đang chọn.

Tác vụ bị loại được xác định bằng một hàm đánh giá đơn giản dựa trên lợi nhuận và thời gian chạy. Thuật toán này được chạy lặp lại nhiều lần trong một **time limit cố định (4.8 giây)** với các tham số khác nhau để tăng khả năng tìm được nghiệm tốt.

Phương pháp này thuộc nhóm **Randomized Greedy Heuristic**.

2.2 Tính phù hợp của phương pháp

Subtask 1 ($N \leq 20$): Brute Force/Backtracking

Với $N \leq 20$, số tập con tối đa là $2^{20} \approx 10^6$, hoàn toàn có thể duyệt hết trong giới hạn thời gian. Ngoài ra, nhờ nhận xét "nếu một tập tác vụ khả thi thì có thể sắp theo EDD vẫn khả thi", ta chỉ cần xét quyết định *chọn/bỏ* theo thứ tự EDD thay vì phải xét mọi hoán vị. Do đó phương pháp brute force kết hợp backtracking là phù hợp và đảm bảo nghiệm tối ưu tuyệt đối (đáp ứng yêu cầu Subtask 1).

Subtask 2 ($N \leq 500$): Heuristic – Randomized Greedy Repair (EDD + Heap Eviction)

Với $N \leq 500$, việc duyệt toàn bộ tập con không còn khả thi. Subtask 2 chấm điểm theo chất lượng nghiệm gần đúng, vì vậy ta chọn một heuristic nhanh, luôn sinh nghiệm hợp lệ và có khả năng cải thiện nghiệm nhờ chạy lặp nhiều lần.

Tên heuristic Thuật toán sử dụng là **Randomized Greedy Repair** (tham khảo dạng "greedy + repair"), cụ thể:

- **Greedy theo EDD**: duyệt tác vụ theo deadline tăng dần.
- **Repair bằng heap**: nếu vi phạm deadline tại bước nào thì loại khỏi tập hiện tại một tác vụ "kém" nhất theo điểm số.

- **Randomization**: thay đổi tham số điểm số giữa các lần chạy để tạo nhiều nghiệm ứng viên khác nhau.

Vì sao chọn heuristic này

- **Luôn hợp lệ**: vì sau mỗi lần vi phạm deadline đều "repair" bằng cách loại bớt tác vụ cho đến khi thỏa điều kiện.
- **Rất nhanh**: mỗi lần chạy chỉ $O(N \log N)$, phù hợp để lặp nhiều vòng trong 4.8 giây.
- **Khả năng khám phá nghiệm**: random hóa giúp tránh phụ thuộc vào một tiêu chí cố định, tăng xác suất gặp nghiệm tốt trên các bộ test khác nhau.

Cấu hình randomization và tham số Trong mỗi lần chạy, ta gán điểm cho tác vụ i theo:

$$score(i) = \frac{p_i^A}{r_i^B}$$

và sử dụng **min-heap** theo score để loại bỏ tác vụ có score nhỏ nhất khi cần repair.

Các tham số được cấu hình như sau:

- **Time gate**: chạy lặp đến thời điểm `time.perf_counter() + 4.8s` để đảm bảo không vượt quá giới hạn thời gian.
- **Seed cố định**: `random.Random(1234567)` để kết quả ổn định giữa các lần chạy (giúp tái lập báo cáo/thử nghiệm).
- **Preset list**: các vòng đầu sử dụng một danh sách tham số cố định để đảm bảo có nghiệm nền tốt:

  $(A, B, \text{noise}) \in \{(1,1,0), (1,0,0), (0,1,0), (1.4,1,0.04), (1.8,1.2,0.06), (2.2,1.6,0.08), (1,2,0.08)\}$

  - $(1,1,0)$: tương ứng dạng "profit-density" cơ bản.
  - $(1,0,0)$: ưu tiên lợi nhuận thuần (profit).
  - $(0,1,0)$: ưu tiên tác vụ ngắn.
  - Các bộ còn lại tăng dần mức phạt thời gian chạy (tăng B) và tăng nhấn mạnh lợi nhuận (tăng A), kèm noise nhỏ để phá hòa.
- **Random exploration sau preset**: sau khi dùng hết preset, sinh ngẫu nhiên:

  $$A = 0.5 + 3.5 \cdot U, \quad B = 0.5 + 3.5 \cdot U,$$

  với $U \sim \text{Uniform}(0,1)$.

- **Noise (phá hòa)**: với xác suất khoảng 35%, dùng noise:

$$\text{noise} = 0.30 \cdot U, \quad U \sim \text{Uniform}(0, 1),$$

còn lại noise = 0. Khi noise > 0, điểm số được nhân:

$$score(i) \leftarrow score(i) \cdot (1 + \epsilon), \quad \epsilon \sim \text{Uniform}(-\text{noise}, \text{noise}).$$

Mục đích của noise là tạo khác biệt giữa các tác vụ có score gần nhau, tăng đa dạng nghiệm giữa các vòng lặp.

Số vòng lặp (iterations) Số vòng lặp không cố định mà phụ thuộc vào tốc độ máy, nhưng do mỗi vòng là $O(N \log N)$ và $N \leq 500$, trong 4.8 giây thuật toán có thể chạy rất nhiều lần; nghiệm tốt nhất qua tất cả các vòng được chọn làm kết quả cuối cùng.

Kết luận về tính phù hợp Heuristic này phù hợp với Subtask 2 vì: (i) luôn tạo nghiệm hợp lệ, (ii) rất nhanh nên chạy được nhiều vòng trong time limit, (iii) random hóa tham số giúp tăng cơ hội đạt nghiệm tốt trên các bộ test khác nhau, đáp ứng đúng yêu cầu chấm điểm theo chất lượng nghiệm.

3 Phân tích độ phức tạp

Subtask 1

Thời gian: $O(2^n)$ (với pruning thường chạy nhanh hơn đáng kể).
Không gian: $O(n)$ cho stack/biến trạng thái.

Subtask 2

Mỗi lần chạy greedy:

- Duyệt n tác vụ theo EDD.
- Mỗi push/pop heap là $O(\log n)$.

Thời gian mỗi lần: $O(n \log n)$.
Tổng thời gian: bị chặn bởi **time gate 4.8s**, nên số vòng lặp chạy được càng nhiều thì cơ hội gặp nghiệm tốt càng cao.
Không gian: $O(n)$.

4 Giải thích code (Python)

Code gồm 2 nhánh:

- Nếu $n \leq 20$: gọi `dfs` để tìm nghiệm tối ưu.

- Nếu $n > 20$: chạy heuristic trong 4.8s:
  1. Sort EDD.
  2. Lặp: chọn tham số (A, B, noise).
  3. Scan EDD: push tác vụ vào heap, nếu vi phạm deadline thì pop score nhỏ nhất.
  4. Nếu tổng profit tốt hơn best thì lưu lại tập chọn (lấy từ heap).
  5. In kết quả best theo EDD.

5 Cài đặt (Python)

```python
import sys, time, heapq, random

def solve():
    data = sys.stdin.buffer.read().split()
    if not data:
        return
    it = iter(data)
    n = int(next(it))

    r = [0]*n
    d = [0]*n
    p = [0]*n
    for i in range(n):
        r[i] = int(next(it))
        d[i] = int(next(it))
        p[i] = int(next(it))

    edd = sorted(range(n), key=lambda i: (d[i], r[i]))

    if n <= 20:
        rem = [0]*(n+1)
        s = 0
        for i in range(n-1, -1, -1):
            s += p[edd[i]]
            rem[i] = s

        bestP = 0
        bestM = 0

        def dfs(i, t, pr, m):
            nonlocal bestP, bestM
            if pr + rem[i] <= bestP:
```

```
                return
            if pr > bestP:
                bestP, bestM = pr, m
            if i == n:
                return
            j = edd[i]
            if t + r[j] <= d[j]:
                dfs(i+1, t + r[j], pr + p[j], m | (1 << i))
            dfs(i+1, t, pr, m)

        dfs(0, 0, 0, 0)
        ans = [str(edd[i] + 1) for i in range(n) if (bestM >> i) & 1]
        sys.stdout.write(str(bestP) + "\n" \
            + (" ".join(ans) if ans else "") + "\n")
        return

    ok = [r[i] <= d[i] for i in range(n)]
    rng = random.Random(1234567)

    presets = [
        (1.0, 1.0, 0.00),
        (1.0, 0.0, 0.00),
        (0.0, 1.0, 0.00),
        (1.4, 1.0, 0.04),
        (1.8, 1.2, 0.06),
        (2.2, 1.6, 0.08),
        (1.0, 2.0, 0.08),
    ]

    def run_once(A, B, nz):
        tt = 0
        pr = 0
        h = []
        for j in edd:
            if not ok[j]:
                continue
            sc = (p[j] ** A) / (r[j] ** B)
            if nz:
                sc *= 1.0 + (rng.random() * 2.0 - 1.0) * nz
            heapq.heappush(h, (sc, j))
            tt += r[j]
            pr += p[j]
            while tt > d[j]:
                _, k = heapq.heappop(h)
                tt -= r[k]
```

```
                pr -= p[k]
        return pr, h

    bestP = 0
    best_set = []

    end_time = time.perf_counter() + 4.8
    i = 0
    while time.perf_counter() < end_time:
        if i < len(presets):
            A, B, nz = presets[i]
        else:
            A = 0.5 + 3.5 * rng.random()
            B = 0.5 + 3.5 * rng.random()
            nz = 0.0 if rng.random() < 0.65 else 0.30 * rng.random()
        i += 1

        pr, h = run_once(A, B, nz)
        if pr > bestP:
            bestP = pr
            best_set = [idx for _, idx in h]

    chosen = [0]*n
    for j in best_set:
        chosen[j] = 1

    ans = [str(i + 1) for i in edd if chosen[i]]
    sys.stdout.write(str(bestP) + "\n" + " ".join(ans) + "\n")

if __name__ == "__main__":
    solve()
```