Báo cáo bài tập về nhà - Cut The Rope

Nhóm thực hiện: Nhóm 7
Danh sách thành viên:

- Võ Lê Ngọc Thịnh
- Vũ Minh Phương

1. Mô tả bài toán

Trong trò chơi Cut the Rope, mỗi hộp chứa một tập các cấp độ khác nhau, và người chơi phải hoàn thành các cấp độ để thu thập sao nhằm mở khóa những hộp tiếp theo. Ở bài toán này, bạn đang chơi hộp đầu tiên gồm n cấp độ, mỗi cấp độ i có hai lựa chọn hoàn thành: lấy một sao với thời gian a_i hoặc lấy hai sao với thời gian b_i, trong đó $a_i < b_i$. Mỗi cấp độ chỉ được vượt qua đúng một lần: hoặc không chơi, hoặc lấy một sao, hoặc lấy hai sao. Mục tiêu của bạn là thu thập ít nhất w sao với tổng thời gian nhỏ nhất để có thể mở hộp tiếp theo. Yêu cầu của bài toán là xác định thời gian tối thiểu đó và mô tả cách lựa chọn số sao cho từng cấp độ trong phương án tối ưu.

2. Ý tưởng & Phương pháp thiết kế

Quan sát đầu tiên là nếu ta quyết định chơi một cấp độ k để lấy **hai sao** với thời gian b_k, thì mọi cấp độ đứng **trước** nó (sau khi sắp xếp theo b tăng dần) đều có $b_i \leq b_k$; khi đó, ta luôn có thể thay thế việc lấy hai sao ở cấp k bằng việc lấy ít nhất một sao ở các cấp trước đó mà không làm thời gian tệ hơn. Từ nhận xét này, ta suy ra rằng trong một nghiệm tối ưu luôn tồn tại một **tiền tố** L (sau khi sort theo b) sao cho: tất cả các cấp độ được chơi hai sao đều nằm trong L cấp đầu tiên, và toàn bộ L cấp này chắc chắn được chơi ít nhất một sao.

Khi đã cố định một giá trị L, ta coi như đã chơi L cấp đầu tiên với **một sao** (tốn tổng thời gian $\sum_{i=1}^{L} a_i$), do đó ta đã có L sao và còn thiếu $w - L$ sao nữa. Số sao thiếu này chỉ có thể được bù bằng hai cách:

- Chơi **một sao** tại các cấp độ nằm ngoài tiền tố ($i > L$) với chi phí a_i cho mỗi sao.
- **Nâng cấp** một số cấp trong L cấp đầu tiên từ một sao lên hai sao, mỗi lần nâng cấp thêm đúng một sao với chi phí tăng thêm $b_i - a_i$.

Như vậy, với một L cố định, bài toán rút gọn thành: "Chọn $w - L$ phần tử có chi phí nhỏ nhất trong tập các giá trị $\{ b_i - a_i \mid i \leq L \} \cup \{ a_i \mid i > L \}$". Tổng thời gian ứng với L sẽ bằng tổng thời gian chơi một sao cho L cấp đầu tiên cộng với tổng chi phí của $w - L$ lựa chọn rẻ nhất này. Ta duyệt L từ 0 đến n, tính giá trị tốt nhất có thể đạt được cho mỗi L và lấy kết quả nhỏ nhất.

Về **phương pháp thiết kế thuật toán là Greedy**: sau khi chứng minh được cấu trúc của nghiệm tối ưu (tồn tại một tiền tố L và tất cả các cấp chơi hai sao nằm trong tiền tố đó), ta luôn chọn tham lam những "sao bổ sung" có chi phí nhỏ nhất trong tập ứng viên (các $b_i - a_i$ và a_i như trên). Việc sử dụng CTDL cây Fenwick (BIT) hay cấu trúc tương tự chỉ nhằm hiện thực hoá hiệu quả ý tưởng tham lam này (tìm nhanh tổng của k phần tử nhỏ nhất), chứ không thay đổi bản chất là một chiến lược **tham lam dựa trên trao đổi (exchange argument)**.

3. Mô tả chi tiết thuật toán

1. Sắp xếp toàn bộ n cấp độ theo thời gian hai sao b_i tăng dần.
   Tách thành các mảng $A[i] = a_i$, $D[i] = b_i - a_i$ và tính mảng tổng tiền tố `prefixA`.

2. Thực hiện nén giá trị cho toàn bộ các chi phí trong tập $A \cup D$, sau đó khởi tạo hai Fenwick Tree để lưu:
   - số lượng phần tử theo từng giá trị nén,
   - tổng giá trị tương ứng.

3. Ban đầu (ứng với $L = 0$), đưa toàn bộ các chi phí $A[i]$ vào Fenwick Tree (vì tất cả cấp độ đều ở ngoài tiền tố).

4. Duyệt L từ 0 đến n.
   Với mỗi L, ta đã chắc chắn chơi một sao cho L cấp đầu tiên, và còn thiếu $w - L$ sao.
   - Nếu $w - L \leq 0$, chi phí chỉ là `prefixA[L]`.
   - Nếu $w - L > 0$, dùng Fenwick Tree để tìm tổng nhỏ nhất của $w - L$ phần tử trong tập chi phí bổ sung, rồi cộng với `prefixA[L]`.

5. Sau khi xét xong L, chuẩn bị cho $L + 1$ bằng cách:
   - xoá chi phí $A[L]$ khỏi Fenwick Tree,
   - thêm chi phí $D[L] = b_L - a_L$ vào Fenwick Tree
     (tức chuyển cấp độ này từ "ngoài L" sang "trong L").

6. Ghi nhận giá trị L cho chi phí nhỏ nhất.
   Để khôi phục phương án, sinh tập ứng viên chi phí gồm:
   - $D[i]$ với $i < L$ (nâng cấp 1→2 sao),
   - $A[i]$ với $i \geq L$ (lấy 1 sao).
     Chọn ra $w - L$ phần tử nhỏ nhất và gán số sao tương ứng.

7. Sắp xếp lại trạng thái các cấp độ theo chỉ số gốc và in ra thời gian tối ưu cùng kịch bản lựa chọn sao.

4. Lý do lựa chọn phương pháp

Bài toán yêu cầu tìm phương án thu thập **ít nhất** w **sao** với **tổng thời gian nhỏ nhất**, trong đó mỗi cấp độ chỉ được chơi đúng một lần và có hai lựa chọn chi phí. Việc duyệt toàn bộ các tổ hợp là bất khả thi do số lượng cấp độ lên đến $3 \cdot 10^5$. Đồng thời, cấu trúc của bài toán không thỏa mãn điều kiện chồng lấp con (overlapping subproblems), nên không thể áp dụng Dynamic Programming hiệu quả.

Điểm then chốt giúp định hướng lời giải là nhận xét về **cấu trúc của nghiệm tối ưu**: sau khi sắp xếp theo b_i, luôn tồn tại một **tiền tố** L sao cho mọi cấp độ được chơi hai sao đều thuộc về tiền tố này. Đây là một dạng **thuộc tính trao đổi (exchange**

property), đặc trưng của các bài toán có thể giải bằng **Greedy approach**.
Khi đã cố định được L, bài toán con trở nên rất đơn giản: ta chỉ cần chọn ra $w - L$ chi phí nhỏ nhất trong hai tập ứng viên $\{b_i - a_i\}$ và $\{a_i\}$ - một nhiệm vụ mà tư duy tham lam (luôn lấy giá trị rẻ nhất còn lại) xử lý chính xác và hiệu quả.

Cuối cùng, để hiện thực hóa chiến lược tham lam này trên dữ liệu lớn, ta kết hợp thêm một cấu trúc dữ liệu Fenwick Tree giúp truy vấn nhanh tổng của k phần tử nhỏ nhất. Như vậy, phương pháp **Greedy approach kết hợp với cấu trúc dữ liệu hỗ trợ** là phù hợp nhất, vừa đảm bảo đúng bản chất thuật toán, vừa đáp ứng giới hạn thời gian của đề bài.

5. Phân tích độ phức tạp

Độ phức tạp thời gian:

- Việc sắp xếp n cấp độ theo b_i tốn $O(n \log n)$.
- Mỗi cấp độ được thêm vào Fenwick Tree đúng một lần và bị chuyển đổi từ $A[i]$ sang $D[i]$ đúng một lần, mỗi lần cập nhật là $O(\log n)$.
- Với mỗi giá trị tiền tố L (từ 0 đến n), ta thực hiện tối đa một truy vấn tìm tổng của k phần tử nhỏ nhất bằng Fenwick Tree, cũng có độ phức tạp $O(\log n)$.

Vậy tổng độ phức tạp thời gian của thuật toán là: $O(n \log n)$,

Độ phức tạp không gian:

- Các mảng lưu trữ A, D, `prefixA` , `idxs` , và bảng nén giá trị đều có kích thước $O(n)$.
- Hai Fenwick Tree (`bitc` và `bits`) có kích thước $O(n)$ sau khi nén giá trị.
- Các cấu trúc phụ (mảng ứng viên khi khôi phục đáp án) có kích thước $O(n)$.

Vậy tổng độ phức tạp không gian của thuật toán là: $O(n)$

6. Code tham khảo

```
import sys
import heapq

input = sys.stdin.readline

n, w = map(int, input().split())
levels = []
for i in range(n):
    a, b = map(int, input().split())
    levels.append((a, b, i))

levels.sort(key=lambda x: x[1])

A = [0] * n
D = [0] * n
idxs = [0] * n
prefixA = [0] * (n + 1)

for i, (a, b, idx) in enumerate(levels):
    A[i] = a
    D[i] = b - a
    idxs[i] = idx
    prefixA[i + 1] = prefixA[i] + a

vals = sorted(set(A + D))
m = len(vals)
comp = {v: i + 1 for i, v in enumerate(vals)}

idA = [comp[a] for a in A]
idD = [comp[d] for d in D]

bitc = [0] * (m + 2)
bits = [0] * (m + 2)

def add(i, dc, ds):
    while i <= m:
        bitc[i] += dc
        bits[i] += ds
        i += i & -i

def sumc(i):
    s = 0
    while i > 0:
        s += bitc[i]
        i -= i & -i
    return s
```

```
def sums(i):
    s = 0
    while i > 0:
        s += bits[i]
        i -= i & -i
    return s

for i in range(n):
    add(idA[i], 1, A[i])

maxlog = m.bit_length()

def kth_sum(k):
    idx = 0
    tmp_k = k
    bitmask = 1 << (maxlog - 1)
    while bitmask:
        t = idx + bitmask
        if t <= m and bitc[t] < tmp_k:
            idx = t
            tmp_k -= bitc[t]
        bitmask >>= 1
    pos = idx + 1
    cnt_before = sumc(pos - 1)
    s_before = sums(pos - 1)
    v = vals[pos - 1]
    return s_before + v * (k - cnt_before)

best_cost = 10**30
bestL = 0
total_count = sumc(m)

for L in range(n + 1):
    base_time = prefixA[L]
    need = w - L
    if need <= 0:
        cost = base_time
        if cost < best_cost:
            best_cost = cost
            bestL = L
    else:
        if need <= total_count:
            cost = base_time + kth_sum(need)
            if cost < best_cost:
                best_cost = cost
                bestL = L
    if L < n:
        add(idA[L], -1, -A[L])
```

```
        add(idD[L], 1, D[L])

res_state = [0] * n
for i in range(bestL):
    res_state[i] = 1

need = w - bestL
if need > 0:
    cand = []
    for i in range(bestL):
        cand.append((D[i], i, 2))
    for i in range(bestL, n):
        cand.append((A[i], i, 1))
    cand.sort()
    for j in range(need):
        _, i, t = cand[j]
        res_state[i] = t

answer = [0] * n
for i in range(n):
    answer[idxs[i]] = res_state[i]

print(best_cost)
print("".join(map(str, answer)))
```