Bài 2: Robot thực hiện công việc (ROBOT)

Một chú Robot phải thực hiện N công việc ($N \le 20$). Bình thường công việc thứ i robot phải mất thời gian A_i giây để hoàn thành. Tuy nhiên các công việc có thể liên quan đến nhau, nên dữ liệu đã lưu trữ khi thực hiện một công việc nào đó có thể giúp chú robot thực hiện công việc liền kề sau nó nhanh hơn. Cụ thể, nếu robot vừa hoàn thành công việc thứ j và bây giờ thực hiện công việc t thì thời gian thực hiện công việc thứ t sẽ được giảm đi B_{tj} giây ($B_{tj} \le A_t$).

Yêu cầu : Hãy sắp xếp thứ tự thực hiện N công việc trên để Robot hoàn thành sớm nhất có thể.

Dữ liệu vào: Từ tệp Robot.inp có cấu trúc như sau:

- Dòng 1: Chứa số nguyên dương N ($N \le 20$).
- Dòng 2: Chứa N số nguyên $A_1, A_2,..., A_N$ ($A_i \le 10^9$).
- N dòng sau mỗi dòng chứa N số nguyên là mảng hai chiều B ($B_{ij} \le A_i$, B_{ij} thể hiện thời gian giảm khi thực hiện công việc thứ i liền kề sau công việc j, $B_{ii} = 0$).

Kết quả: Ghi vào tệp Robot.out gồm một số nguyên duy nhất là thời gian nhỏ nhất để hoàn thành N công việc trên.

Ví dụ:

Robot.inp	Robot.out
4 8 4 5 0 1 1 3 0 1 2 4 0	10

Sol:

- Nếu làm công việc t ngay **sau** j thì tiết kiệm được B_{tj} giây.
- Tổng thời gian ban đầu (không dùng giảm trừ) là $S = \sum_{i=1}^{N} A_i$.
- Ta **tối đa hóa tổng tiết kiệm** thu được khi xếp thứ tự công việc.

  Khi đó, **thời gian nhỏ nhất** = $S - (\text{tổng tiết kiệm lớn nhất})$.

Bài toán trở thành: tìm hoán vị P của $\{1..N\}$ để $\sum_{k=2}^{N} B_{P_k, P_{k-1}}$ là **lớn nhất**.

Quy hoạch động trên tập con (bitmask DP)

Trạng thái

- $L[\text{mask}][j]$: **tổng tiết kiệm tối đa** đạt được khi đã làm đúng tập công việc `mask` và **kết thúc** bằng công việc j.
- `mask` là bitmask độ dài N (bit j bật ⇔ công việc j đã làm).

Khởi tạo

- Chọn công việc đầu tiên là i: chưa có tiết kiệm do chưa có "công việc trước".

$$L[1 << (i-1)][i] = 0 \quad \forall i = 1..N.$$

- Các trạng thái khác khởi tạo $-\infty$ (chưa đạt).

Chuyển trạng thái

- Với mọi `mask`, mọi j sao cho $L[\text{mask}][j]$ hữu hạn, thử gắn thêm công việc p **chưa có trong mask**:
  - `nxt = mask | (1 << (p-1))`
  - Khi làm p sau j, cộng tiết kiệm B_{pj}:

$$L[\text{nxt}][p] = \max\Big(L[\text{nxt}][p],\ L[\text{mask}][j] + B_{pj}\Big).$$

6) Ví dụ số cực nhỏ (n = 3) để "thấy chạy"

Giả sử:
B[2][1] = 5, B[3][1] = 2
B[1][2] = 4, B[3][2] = 6
B[1][3] = 1, B[2][3] = 3
Khởi tạo:
L[001][1] = 0
L[010][2] = 0
L[100][3] = 0
Từ L[001][1]=0:

- Thêm 2: nxt=011, new=0+B[2][1]=5 → L[011][2] = 5
- Thêm 3: nxt=101, new=0+B[3][1]=2 → L[101][3] = 2

Từ L[010][2]=0:

- Thêm 1: nxt=011, new=0+B[1][2]=4 → L[011][1] = 4
- Thêm 3: nxt=110, new=0+B[3][2]=6 → L[110][3] = 6

Từ L[100][3]=0:

- Thêm 1: nxt=101, new=0+B[1][3]=1 → L[101][1] = 1
- Thêm 2: nxt=110, new=0+B[2][3]=3 → L[110][2] = 3

Giờ từ L[011][2]=5 (đã làm {1,2}, cuối 2):

- Thêm 3: nxt=111, new=5+B[3][2]=5+6=11 → L[111][3] = 11

Từ L[011][1]=4:

- Thêm 3: new=4+B[3][1]=4+2=6 → L[111][3] = max(11,6)=11

Từ L[110][3]=6:

- Thêm 1: new=6+B[1][3]=6+1=7 → L[111][1] = 7

Từ L[110][2]=3:

- Thêm 1: new=3+B[1][2]=3+4=7 → L[111][1] = max(7,7)=7

Từ L[101][3]=2:

- Thêm 2: new=2+B[2][3]=2+3=5 → L[111][2] = 5

Từ L[101][1]=1:

- Thêm 2: new=1+B[2][1]=1+5=6 → L[111][2] = max(5,6)=6

Kết thúc mask=111:

L[111][1] = 7
L[111][2] = 6
L[111][3] = 11 ← lớn nhất
bestSave = 11, vậy thời gian nhỏ nhất = sum(A) - 11.
Bạn có thể "nhìn thấy" chuỗi tối ưu:

- 1 → 2 (+5), rồi 2 → 3 (+6) ⇒ tổng giảm 11.

2) Từng dòng một

for (int mask = 0; mask < total; ++mask) {

- Duyệt **mọi tập công việc** từ rỗng đến đầy đủ (total = 1<<n).
- Mỗi mask biểu diễn một tập con của {1..n}.

```
  for (int j = 1; j <= n; ++j) {
    long long cur = L[mask][j];
    if (cur == NEG_INF) continue;
```

- Xét **công việc cuối cùng** là j.
- Lấy giá trị hiện tại cur. Nếu là NEG_INF (−∞) thì **trạng thái chưa từng đạt** → bỏ qua.

```
    for (int p = 1; p <= n; ++p) {
      if (((mask >> (p - 1)) & 1) == 0) { // p chưa làm
```

- Thử chọn **công việc kế tiếp** p.
- Điều kiện ((mask >> (p-1)) & 1) == 0 nghĩa là **bit của p trong mask đang bằng 0** → p **chưa ở trong tập** → có thể thêm.

```
        int nxt = mask | (1 << (p - 1));
```

- Tạo **mask mới** nxt bằng cách **bật bit p** vào mask hiện tại:
  - 1 << (p-1) tạo mặt nạ có đúng 1 bit của p.
  - mask | ... thêm p vào tập (hợp bit).

```
        long long val = cur + B[p][j]; // làm j xong làm p
```

- Nếu làm p **ngay sau** j, ta **được giảm** thêm B[p][j].
- val = tổng tiết kiệm mới = tiết kiệm cũ cur + phần tăng mới.

```
        if (val > L[nxt][p]) L[nxt][p] = val;
```

- Cập nhật **trạng thái đích**: đã làm nxt và **kết thúc** ở p.
- Lấy **max** vì có nhiều đường đi khác nhau cùng dẫn đến (nxt, p); ta giữ đường có **tổng tiết kiệm lớn nhất**.

```
      }
    }
  }
}
```

---

3) Invariant (bất biến logic)

- Trước khi cập nhật, L[mask][j] đã là **tối ưu** (do các bước trước đã cập nhật).
- Sau vòng p, mọi trạng thái (nxt, p) nhận được ứng viên từ (mask, j) với phần thưởng B[p][j].
- Duyệt mask từ nhỏ → lớn (ít bit → nhiều bit) phù hợp với DP trên tập con.

---

4) Hình dung nhanh bằng ví dụ (3 việc)

Giả sử đang ở mask = 0b011 (đã làm {1,2}), j = 2, cur = L[011][2].

- Xét p = 3 (chưa làm vì bit 3 là 0):

- nxt = 0b011 | 0b100 = 0b111
- val = cur + B[3][2]
- Cập nhật L[111][3] = max(L[111][3], val)