Overview

Problems	#tried	#AC	Fastest solution
A	516	99	5 minutes by **BK.Juniors**
B	281	54	32 minutes by **BK.Tornado**
C	152	26	16 minutes by **DOS**
D	132	15	42 minutes by **ACGTeam**
E	25	1	262 minutes by **ACGTeam**
F	19	0	N/A
G	121	43	26 minutes by **WINDOWS**
H	60	11	56 minutes by **Mjolnir**
I	44	2	85 minutes by **ACGTeam**
J	44	10	101 minutes by **BK.Juniors**
K	294	104	5 minutes by **UBUNTU**
L	44	4	150 minutes by **LINUX**

Lời giải được chuẩn bị bởi:

- RR
- Lê Đôn Khuê
- Lê Yên Thanh
- Lăng Trung Hiếu
- Nguyễn Duy Khương

Lời giải các bài giả sử bạn đã quen thuộc với những khái niệm cơ bản như:

- Độ phức tạp tính toán

Nếu có thắc mắc về lời giải, bạn có thể hỏi ở: group VNOI.

Tất cả code của các bài có thể tìm được ở Github. Đây là những lời giải chính thức của BTC.

A

Lời giải nhanh nhất: ở **phút thứ 5** của đội **BK.Juniors**

Gọi đoạn cần tìm là [L, R]. Như vậy đoạn [L, R] cần thỏa mãn:

- Có 1 số duy nhất bằng m.
- Không có số nào nhỏ hơn m.

Lời giải 1:

Tư tưởng:

Ta xét lần lượt từng vị trí i trên mảng A. Giả sử i chính là vị trí nhỏ nhất của đoạn [L, R] cần tìm. Rõ ràng ta phải có A[i] = m.

Khi đã cố định i, ta cần tìm 2 điểm L và R. Vì việc tìm L và R là đối xứng nhau, nên ta chỉ xét việc tìm L. Việc tìm R sẽ làm hoàn toàn tương tự.

Để tìm L, ta làm như sau:

- Đầu tiên, gán L = i
- Chừng nào ta có thể mở rộng L về phía bên trái, mà vẫn thoả mãn i là số nhỏ nhất duy nhất, ta giảm L. Nói cách khác, chừng nào A[L-1] > m, ta giảm L.
    - Chú ý, việc "mở rộng" L về phía bên trái luôn làm tổng của đoạn [L, R] tăng lên, do

Ví dụ:

- Với dãy A = [1, 3, 2, 6, 2, 4] và m = 2. Đánh số các số trong mảng A từ 1.
- Khi i = 3, A[i] = 2, ta cần tìm L.
    - Đầu tiên gán L = 3
    - Vì A[L-1] = A[2] = 3 > m, nên ta giảm L → L = 2
    - Vì A[L-1] = A[1] = 1 < m, nên ta không thể tiếp tục giảm L.
    - Ta tìm được L = 2
- Khi i = 5, A[i] = 2, ta cần tìm L.
    - Đầu tiên gán L = 5
    - Vì A[L-1] = A[4] = 6 > m, nên ta giảm L → L = 4
    - Vì A[L-1] = A[3] = 2 = m, nên ta không thể giảm L nữa.
    - Do đó, ta tìm được L = 4.

Chứng minh:

Ta có thể chứng minh được thuật toán trên có độ phức tạp O(N):

- Ở mỗi vị trí A[i] = m, để tính L, số bước giảm L không thể lớn hơn khoảng cách từ i đến số đầu tiên bên trái i mà bằng m.
- Do đó, tổng tất cả các thao tác tính L không thể lớn hơn độ dài dãy. Nói cách khác độ phức tạp là O(N).

Để tính R, ta cũng làm tương tự với độ phức tạp O(N).

Cài đặt: RR

Lời giải 2:

Cũng tương tự như cách 1, ta xét phần tử thứ i và giả sử i là phần tử nhỏ nhất trong đoạn [L,R].

Với mỗi phần tử i của mảng A, ta cần tìm đoạn [L, R] dài nhất, sao cho i thuộc đoạn [L, R] và A[i] = min(A[L], A[L+1], ..., A[R]).

Dễ thấy đây chính là định nghĩa của 2 mảng L và R trong kỹ thuật tìm Min/Max trên đoạn tịnh tiến.

Do vậy, bạn chỉ cần tính 2 mảng L và R như trong link trên, rồi từ đó với mỗi i, tìm được L, R trong O(1).

Cài đặt: RR

Fun fact

3 đội trong top 5 của kỳ thi ban đầu giải sai 1 lần bài này: ACGTeam, LINUX, PECaveros. Trong đó đội vô địch ACGTeam bị sai 1 phát do đọc thiếu 1 chữ "minimum" trong đề.

B

Lời giải nhanh nhất 1 lần nữa thuộc về một đội đến từ ĐH Bách Khoa: **phút 32** của đội **BK.Tornado**.

Tư tưởng

Ở bài toán này, bạn được yêu cầu giải bài toán xuôi: cho lượng nước W, tính cột lớn nhất mà nước chảy đến.

Ta xét bài toán ngược lại: Nếu nước chỉ chảy đến cột lớn nhất là i, lượng nước tối đa W là bao nhiêu?

Ta ký hiệu bài toán xuôi là f: cho W, tính f(W) = cột lớn nhất.
Ký hiệu bài toán ngược là g: cho i, tính g(i) = W là lượng nước lớn nhất.

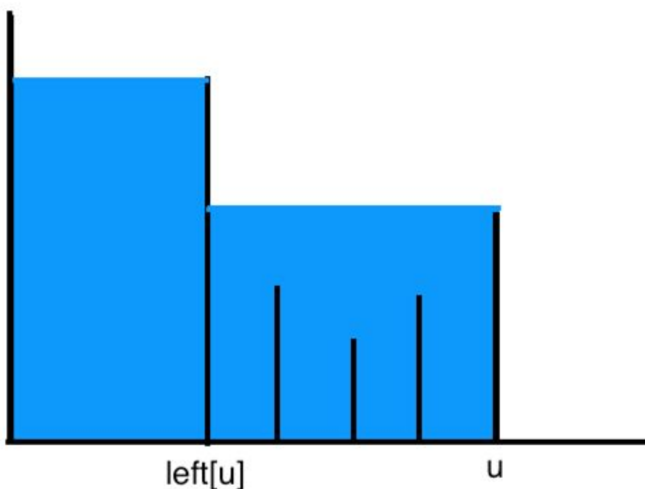
Việc tính bài toán xuôi (tính f) là khó, do miền giá trị của W có thể lên đến 10^15. Nhưng việc giải bài toán ngược (tính g) lại đơn giản hơn, do miền giá trị của i chỉ đến 10^5. Để tính g ta có thể quy hoạch động và tính trước toàn bộ hàm g với tất cả các giá trị của i.

Khi có mảng g, ta có thể dùng phương pháp chặt nhị phân để tính f.

Tính mảng g

Việc tính g gồm 2 bước:

1. Tính trước mảng left[u] là cột đứng trước và gần cột u nhất mà height[i] > height[u]. (Khởi tạo cột 0 có chiều cao rất lớn)
2. Tính g[u] là lượng nước tối đa nếu nước không tràn qua cột u.



Bước 1 đã được giải thích rất cụ thể trong bài viết: [Tìm min-max trên đoạn tịnh tiến].
Với bước 2, từ hình vẽ, có thể thấy g(u) là tổng của:

- Lượng nước từ đầu đến cột left(u), chính là g(left(u))
- Thể tích nước từ sau cột left[u] đến cột u. Chú ý ta cần trừ đi thể tích các cột từ left(u) đến u: (position[i] - position[u]) * h[u] - tổng(các cột trong khoảng từ sau cột i đến u)

Chặt nhị phân

Để tính f(W), ban đầu ta biết f(W) nằm trong khoảng [0, N].

Xét cột mid = N/2.
Nếu g(mid) >= W, rõ ràng đến cột mid, ta có thể chứa được nhiều nước hơn, nên kết quả <= mid.
Ngược lại, kết quả > mid.

Như vậy, với 1 phép so sánh, ta biết được f(W) nằm trong [0, mid] hay [mid+1, N]: ta đã giảm đi được 2 lần khoảng cần tìm. Tiếp tục làm như vậy, ta sẽ tính được hàm f với độ phức tạp O(logN):

Pseudo code:

```
L = 0
R = N
res = N
while L <= R:
    mid = (L + R) / 2
    if g(mid) >= W:
        res = mid
        R = mid - 1
    else:
        L = mid + 1
Kết quả cuối cùng là res
```

Cài đặt: RR

C

Lời giải nhanh nhất thuộc về đội **DOS** đến từ ĐH Công Nghệ ở **phút thứ 16**.

Quy hoạch động

Đây là một bài toán Quy hoạch động khá cơ bản. Nếu chưa nắm chắc về thuật toán quy hoạch động, các bạn có thể đọc thêm trong những bài viết sau:

- Nhập môn Quy hoạch động
- Một vài bài QHĐ về Palindrome
- Một số bài QHĐ điển hình

Thông thường, với các bài toán quy hoạch động có dạng đếm số lượng số nhỏ hơn hoặc bằng một số X cho trước, ta làm như sau:

f(i, is_lower) = số lượng cách:

- chọn ra i chữ số đầu tiên
- sao cho với i chữ số đầu tiên này, số ta đang xây dựng vẫn nhỏ hơn hoặc bằng số X.
- Nếu số đang xây dựng đã chắc chắn nhỏ hơn X, is_lower = 1. Ngược lại, is_lower = 0.

Ví dụ:

X = 32

i	0	1	2
is_lower = 0	1 (số gồm 0 chữ số)	3 (các số 0?, 1?, 2?)	32 (gồm 0?, 1?, 2?, 30, 31)
is_lower = 1	0	1 (số 3?)	1 (số 32)

Để tính mảng f, ta làm như sau:

- Giả sử trạng thái QHĐ hiện tại là (i, is_lower)
- Ta thử thêm lần lượt các chữ số 0 đến 9

- Khi thêm chữ số c vào trạng thái hiện tại là (i, is_lower) gồm i chữ số, ta sẽ thu được trạng thái mới (i+1, is_lower2) gồm (i+1) chữ số.
    - Nếu is_lower = 1, is_lower2 chắc chắn bằng 1, do nếu phần gồm i chữ số đã chắc chắn nhỏ hơn số X, thì phần gồm i+1 chữ số cũng chắc chắn nhỏ hơn X.
    - Nếu is_lower = 0, và chữ số c ta thêm vào nhỏ hơn chữ số tương ứng của X, thì is_lower2 = 1. Ngược lại, is_lower2 = 0.
- Khi cài đặt, ta xét lần lượt các trạng thái (i, is_lower) theo thứ tự tăng dần của i. Với mỗi trạng thái, ta xét tất cả các chữ số c, tính is_lower2, rồi cập nhật:
    - f(i+1, is_lower2) += f(i, is_lower).

Như vậy, ta thu được thuật toán với độ phức tạp O(L), trong đó L là số chữ số của X.

Trở lại bài toán

Trong bài toán này, vì ta cần quan tâm đến các số chia hết cho 8, và cần quan tâm đến số lượng chữ số 6 và 8, nên ta thêm 2 chiều vào mảng quy hoạch động:

f(i, is_lower, mod8, good_digit), trong đó:

- i và is_lower có ý nghĩa giống như trên
- mod8: số dư của số đang xây dựng khi chia 8
- good_digit: số lượng chữ số 6 và 8.

Để tính mảng f, từ trạng thái hiện tại là f(i, is_lower, mod8, good_digit), ta thêm 1 chữ số c và chuyển đến trạng thái mới (i+1, is_lower2, mod8_2, good_digit2) như sau:

- is_lower2 tính như trên
- mod8_2 = (mod8 * 10 + c) % 8
- Nếu c = 6 hoặc 8, thì good_digit2 = good_digit + 1, ngược lại good_digit2 = good_digit.

Cài đặt:

- RR
- Hieu

D

Đội giải nhanh nhất bài này chính là đội vô địch - **ACGTeam** của Hàn Quốc ở **phút 42**.

Đây cũng là một bài quy hoạch động kinh điển kết hợp với kĩ thuật nhân ma trận.

O(N*L + L^4)

Đầu tiên, ta giải quyết bài toán với n nhỏ:

Đặt f(i, x) = số cách xây dựng xâu gồm i ký tự, sao cho đoạn cuối của xâu đang xây dựng đã khớp với x ký tự đầu của xâu p.

Để đơn giản, khi x = |p|, ta luôn luôn giữ nguyên x = |p| khi chuyển trạng thái. Khi đó kết quả cần tìm chính là f(n, |p|).

Để chuyển trạng thái, với mỗi trạng thái (i, x), ta thử thêm 1 ký tự c bất kỳ. Khi đó ta sẽ chuyển đến trạng thái mới (i+1, x2). Chú ý x2 phải là lớn nhất có thể.

Ví dụ:
p = ABCABCABC

Giả sử ta đang ở trạng thái (i, 5) với i bất kỳ. x = 5 nghĩa là ta đã khớp được 5 ký tự (ABCABC). Giờ giả sử ta thêm ký tự C. Vì phần cuối của xâu mới là ABCABC, nên rõ ràng xâu mới khớp với 3 ký tự đầu hoặc 6 ký tự đầu của xâu p. Tuy nhiên ta phải chọn x2 = 6.

Từ đó ta thu được cách làm như sau:

- Xét lần lượt các trạng thái (i, x).
- Lần lượt xét các ký tự c. Khi thêm c, ta tính x2.
- f(i+1, x2) += f(i, x).

Để tăng tốc, ta thấy rằng x2 chỉ phụ thuộc vào x và c. Do đó ta khởi tạo trước mảng go(x, c) = x2 với ý nghĩa nếu ta đang khớp với x ký tự đầu của xâu p, và thêm ký tự mới là c, thì ta sẽ được khớp với x2 ký tự đầu của p. Do độ dài xâu p nhỏ, nên ta có thể xét tất cả các giá trị của x2 rồi kiểm tra xâu bằng nhau. Độ phức tạp cho việc khởi tạo này sẽ là O(L^4).

Sau khi có mảng go, ta có thể cài đặt thuật QHĐ như trên với độ phức tạp O(L*26).

O(L^3 * logN)

Để tăng tốc, ta sử dụng kỹ thuật nhân ma trận. Các bạn có thể đọc cụ thể ở link này. Khi đó ta sẽ thu được thuật toán O(L^3 * logN), với L^3 là thời gian để nhân ma trận.

Ngoài ra bài này có 1 điểm chú ý là các bạn cần nhân 2 số không quá 10^12. Nếu các bạn dùng phép nhân thông thường, kết quả sẽ lên đến 10^24, vượt quá giới hạn kiểu số 64 bit. Trong kỳ thi có 1 số bạn dùng cách nhân Ấn độ tương tự như tính luỹ thừa của 1 số. Cách này sẽ có độ phức tạp là O(L^3 * logN * log(MAX_VALUE)) và rất khó để chạy trong thời gian cho phép.

Để giải quyết, các bạn có thể làm như sau:

Giả sử cần nhân A và B, mỗi số có 12 chữ số. Tách B thành 2 nửa, mỗi nửa gồm 6 chữ số:

Ví dụ:

B = 123456789012
B1 = 123456 = B / 10^6
B2 = 789012 = B % 10^6

A * B = (A * B1 * 10^6) + A * B2

(A * B) % MOD = ((A * B1 % MOD * 10^6) + A * B2) % MOD

Khi nhân như vậy, tất cả các kết quả đều không quá 10^18.

Cài đặt:

RR

Ngoài ra, phần tính mảng go trong bài này có thể cài đặt rất hiệu quả với độ phức tạp O(L*26), bằng thuật toán Suffix Automata. Các bạn có thể tham khảo thêm ở cài đặt của Lê Yên Thanh.

E

Bài E là một bài cài đặt rất phức tạp, kết hợp với thao tác TREE rất khó để cài đặt hiệu quả. Vì vậy chỉ có duy nhất team vô địch - **ACGTeam** - làm được ở **phút 262**.

Cách 1

Đầu tiên, mình sẽ nói cách dễ nhận ra nhất. Đây là cách mà thông thường các bạn sẽ nghĩ ra, tuy nhiên cài đặt cực kỳ phức tạp.

Vì số thao tác MKDIR và RM không quá 5000, nên số thư mục tại bất kỳ thời điểm nào không quá 5000.

Để cài đặt được bài này, đầu tiên ở mỗi thư mục, ta cần lưu các thư mục con của nó một cách hiệu quả. Có thể dùng map của C++:

```
struct Dir {
    string name; // tên thư mục hiện tại
    map<string, Dir*> children; // các thư mục con
    Dir* parent;
} *root;
```

Ta xét lần lượt các thao tác:

- MKDIR: ta chỉ đơn giản là tạo ra một Dir mới, thêm vào map children của nút cha.
    - Cài đặt:
      Dir* create_dir(string name, Dir* parent) { // tạo **Dir** với tên **name**, là con của **parent**.
      ```
          Dir* res = new Dir();
          res->parent = parent;
          res->children.clear();
          res->name = name;

          parent->children[name] = res;
      }
      ```
- RM: ta chỉ cần xoá thư mục:
  ```
  parent->children.erase(name)
  ```
- CD: nhờ map children và con trỏ trỏ đến parent, ta dễ dàng di chuyển giữa các Dir.

- SZ: để cài đặt thao tác này, tại mỗi Dir, ta lưu thêm kích thước của nó. Như vậy, khi thực hiện thao tác MKDIR và RM, ta cập nhật biến kích thước ở nút hiện tại và tất cả tổ tiên của nó:

```
// thay đổi kích thước tất cả tổ tiên của cur, cộng thêm update (chú ý update có thể âm).
void update_ancestors(Dir* cur, int update) {
	while (cur != NULL) {
		cur->size += update;
		cur = cur->parent;
	}
}
```

- Để thực hiện thao tác LS, ta đơn giản chỉ cần in ra 10 phần tử của children. Bạn chỉ cần quen với map là có thể làm được.
- Để thực hiện thao tác TREE, cách thông thường là DFS để in ra 5 phần tử đầu và 5 phần tử cuối. Tuy nhiên để in ra 5 phần tử cuối, ta có thể phải mất độ phức tạp O(N). Cách này không thể qua được Time limit. Cả 2 đội ACGTeam và PECaveros lúc đầu đều làm cách DFS thông thường và đều bị TLE. Cách giải quyết trực quan nhất là: ở mỗi Dir, ta lưu thêm con cuối cùng của nó:

```
struct Dir {
	int size;
	string name;
	map<string, Dir*> children;
	Dir* parent;
	Dir* last_child;
}
```

Để in ra 5 phần tử cuối, đầu tiên ta in ra last_child, sau đấy từ last_child đi ngược lên cha nó, rồi lại đi xuống phần tử cuối của cha nó (last_child->parent->last_child). Cách cài đặt này có độ phức tạp là O(số phần tử cần in ra). Chi tiết cài đặt có thể xem ở cài đặt ở dưới.

Chú ý rằng ta cần cập nhật last_child với mỗi thao tác MKDIR và RM.

- Để thực hiện thao tác UNDO, ta lưu lại các thao tác có thể bị undo vào stack:
  - UNDO MKDIR: chỉ cần xoá Dir tương ứng.
  - UNDO RM: Ta cần lưu lại Dir đã bị xoá, từ đó lấy lại được Dir đó và tất cả con cháu của nó.
  - UNDO CD: Ta cần lưu lại Dir trước khi CD.
  - Chú ý sau khi UNDO MKDIR hoặc UNDO RM, ta cần cập nhật tất cả size và last_child của tất cả tổ tiên.

Cài đặt: RR

Có thể thấy cài đặt này rất phức tạp, do:

- Thao tác TREE rất khó cài đặt đúng

- Ta cần duy trì rất nhiều thông tin: size và last_child, khi làm các thao tác MKDIR, RM, UNDO.

Cách 2:

Ở mỗi Dir, ta lưu các giá trị cache của:
- SZ
- LS
- TREE

Cấu trúc của ta sẽ trông như sau:

```
struct Dir {
	string name;
	Dir* parent;
	map<string, Dir*> children;

	// caching data
	int sz;
	vector<string> ls;
	vector<string> tree;

	bool cached_sz, cached_ls, cached_tree;
};
```

Ta cần duy trì 3 biến boolean cached_sz, cached_ls, cached_tree thể hiện ta đã có giá trị cache của 3 thao tác chưa.
- Ban đầu, cả 3 biến này bằng false
- Khi ta thực hiện thao tác SZ, LS, TREE:
  - Nếu chưa có cache, ta tính kết quả các thao tác này. Với thao tác TREE hoặc SZ, ta mất O(N). Sau khi tính, ta lưu cache, và gán giá trị boolean tương ứng thành true.
  - Nếu đã có cache, ta chỉ cần in ra cache.
  - Khi ta thực hiện RM, MKDIR, UNDO RM hoặc UNDO MKDIR, ta cần gán lại cả 3 biến bằng false, cho tất cả tổ tiên của Dir hiện tại.

Ví dụ với SZ:

```
int calculate_sz(Dir* cur) {
	if (cached_sz) return sz;
	return dfs(cur);
}
```

```
int dfs(Dir* cur) {
	cur->cached_sz = true;
	int res = 1;
	for (auto p : cur)
		sz += dfs(p.second);
	return cur->sz = res;
}
```

Nhờ giá trị cache, số lần tối đa ta cần tính SZ, LS, TREE là O(N), do chỉ có những lần ta MKDIR, RM, UNDO MKDIR, UNDO RM mới cần tính lại các giá trị cache.

Cài đặt đơn giản hơn hẳn cách trên: [RR]

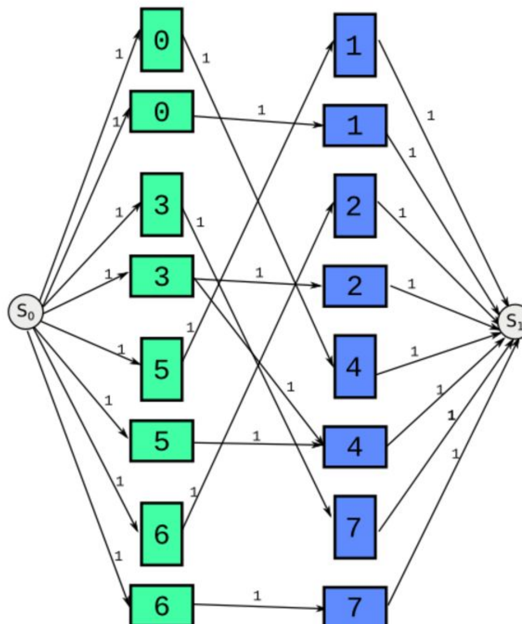
F

Đây là bài duy nhất trong kì thi không có đội nào AC mặc dù nếu đánh giá về độ khó thì dễ nghĩ ra hơn so với bài **L** và dễ cài đặt hơn so với bài **E**. Trong 1 tiếng cuối 2 đội ACGTeam và LINUX đã cố gắng giải bài này nhưng do đi sai hướng là sử dụng DP nên không thể làm được.

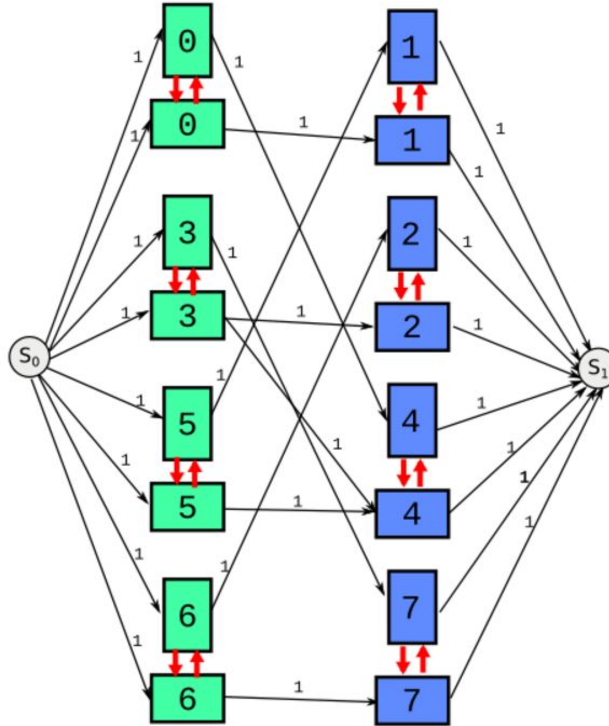
Về cơ bản đây là một bài luồng min cost. Cách làm là dựa trên tính chất ô đen/trắng của bàn cờ vua mà chia thành đồ thị 2 phía. Có nhiều cách để xây dựng đồ thị, dưới đây là cách xây dựng của tác giả:

		0	1
2	3	4	5
6	7		

Với mỗi ô ta sẽ chia làm đỉnh trên đồ thị, một đỉnh đại diện cho liên kết ống ngang, 1 đỉnh đại diện cho liên kết ống dọc. Mỗi ống ngang sẽ kết nối với ống ngang của ô kề với nó, tương tự cho ống dọc. Như vậy nếu ta xây dựng đồ thị như vậy và tìm luồng thì bài toán sẽ tương đương với yêu cầu CHỈ SỬ DỤNG ỐNG CONG MÀ KHÔNG DÙNG ỐNG NGANG HAY DỌC.



Giờ giữa mỗi đỉnh ngang và dọc của mỗi ô ta sẽ tạo 2 cung thông lượng là 1, một cung đi từ dọc sang ngang, một cung đi từ ngang sang dọc, xét các ô nằm bên trái của đồ thị 2 phía, ta thấy rằng cung đi từ ô dọc sang ô ngang đại diện cho việc đặt 1 ống ngang ở ô đó, cung đi từ ô ngang sang ô dọc đại diện cho việc đặt một ống dọc. Như vậy ta sẽ để giá trị của cung đi từ dọc sang ngang là **h**, cung đi từ ngang sang dọc là **v**. Chú ý là đối với các ô nằm bên phải đồ thị 2 phía sẽ ngược giá trị ngang/dọc về ý nghĩa nên cần xây dựng đồ thị cho đúng.



Sau khi xây dựng đồ thị xong thì chỉ việc sử dụng thuật toán tìm luồng min cost và kiểm tra xem có thể xây dựng được đầy đủ các đường ống nước không là ổn. Bài này BTC đã cố tình set time khá thoải mái là 5s (trong khi code lời giải chạy chỉ mất 0.2s) nên chỉ cần các thuật toán tìm luồng min cost cơ bản là có thể làm được

Cài đặt:

- RR
- Lăng Trung Hiếu

G

Đây bài được đánh giá là dễ thứ 3 trong đề thi (dễ hơn bài B) nhưng số đội làm được lại ít hơn kì vọng của BTC. Đội giải nhanh nhất bài này là đội **WINDOWS** của ĐH Công nghệ ở **phút 26**.

Fun fact: Mặc dù đây là bài rất dễ, nhưng ngay cả đội vô địch ACGTeam cũng phạm sai lầm trong contest và nộp sai đến 3 lần.

Để giải bài này, các bạn cần nắm được thuật toán tìm cây khung của đồ thị như Kruskal hoặc Prim. Nếu các bạn chưa biết, có thể đọc trong sách của thầy Lê Minh Hoàng, download ở link này.

Ta có S trạm đã có dữ liệu và cần chọn một số cạnh nối để nối N-S trạm còn lại với S trạm này.

Tạo thêm một đỉnh giả (N+1), nối đỉnh này với S đỉnh bắt đầu với trọng số bằng 0. Điều này đảm bảo sau khi áp dụng thuật toán Kruskal, tổng chi phí không thay đổi và N-S đỉnh còn lại sẽ liên thông với 1 trong S trạm ban đầu.

Cài đặt: RR Thanh

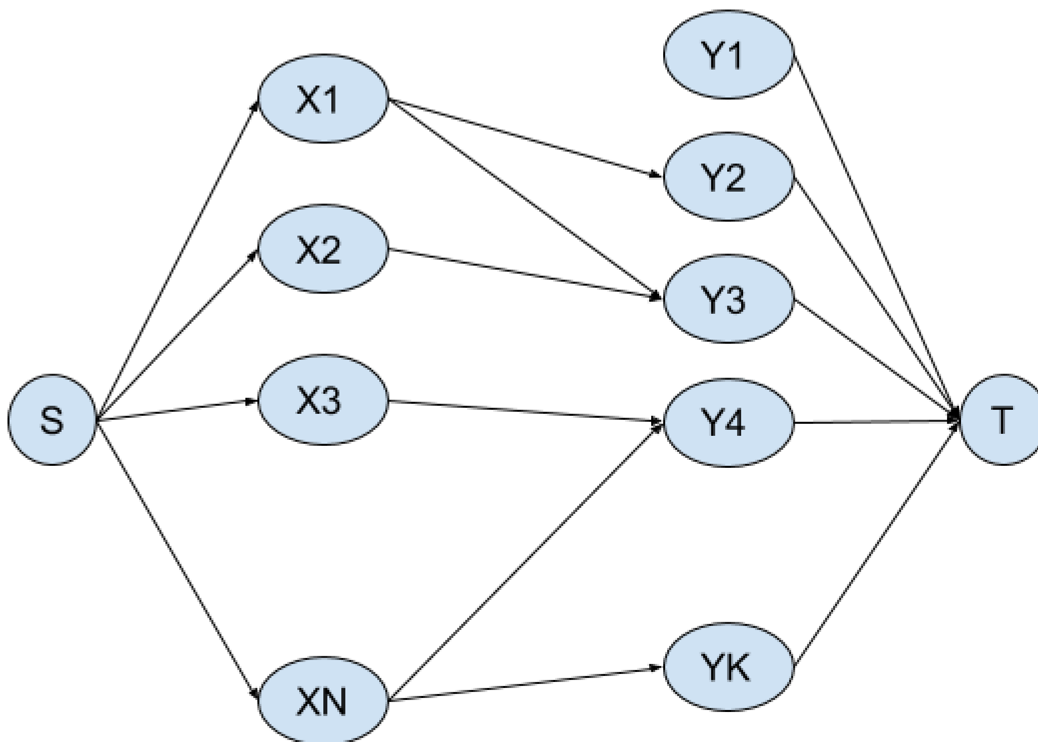
H

Đây là một bài luồng cổ điển. Đáng tiếc trong contest có một số bạn cài đặt tham lam cũng có thể AC. Đội đầu tiên AC là đội **Mjolnir** ở **phút 56**. Sau đó ở **phút 112**, đội **ACGTeam** là đội AC thứ 2, và sử dụng thuật luồng.

Nếu chưa biết về luồng các bạn có thể đọc trong sách của thầy Lê Minh Hoàng.

Ta xây dựng đồ thị luồng như sau:

- 1 đỉnh phát S
- 1 đỉnh thu T
- N đỉnh thể hiện N công việc: X1, ..., XN
- K đỉnh thể hiện K đơn vị thời gian: Y1, ..., YK



Các cạnh:

- S - Xi có khả năng thông qua là thời gian để hoàn thành công việc Xi.
- Nếu công việc Xi có thể làm từ thời điểm Ai đến Bi, thì nối đỉnh Xi với tất cả các đỉnh trong khoảng [Ai, Bi], khả năng thông qua bằng 1.
- Cạnh Yi - T có khả năng thông qua bằng M - số máy.

Khi tìm được luồng cực đại, nếu luồng bằng tổng khối lượng công việc, ta biết luồng thoả mãn:

- Mỗi công việc phải được làm hết thời gian yêu cầu.
- Mỗi công việc, tại mỗi thời gian hợp lệ, chỉ làm tối đa 1 phần việc của nó.
- Với mỗi đơn vị thời gian, không quá M công việc khác nhau được thực hiện.

Đồ thị này có O(N+K) đỉnh. Nếu cài đặt luồng hiệu quả vẫn có thể qua được test.
Cài đặt: RR

Để hiệu quả hơn, ta có thể giảm số đỉnh xuống O(N): Thay vì mỗi đỉnh thể hiện 1 đơn vị thời gian, ta cho mỗi đỉnh thể hiện 1 khoảng thời gian: những đơn vị thời gian cạnh nhau và vai trò giống hệt nhau, ta gộp lại thành 1.

Khi đó, mỗi đỉnh Yi thay vì thể hiện đơn vị thời gian i, sẽ thể hiện 1 khoảng thời gian [Li, Ri]. Ta cần gán khả năng thông qua một cách phù hợp.

Cài đặt: Lê Yên Thanh

I

Bài này là kết hợp của các thuật toán:

- Lý thuyết trò chơi - Grundy
- Misere Nim
- Sàng nguyên tố 1 đoạn [L, R] có $R - L$ không quá 10^5 và $R \le 10^{12}$ để đếm số ước.
- Quy hoạch động

Bài này tuy từng bước đơn giản, nhưng để kết hợp lại là điều không hề dễ. Chỉ có đội vô địch **ACGTeam** làm được bài này ở **phút 85** và đội **LINUX** của ĐH Công nghệ ở **phút 236**.

Bài này gồm 3 bước:

- Bước 1: Ta cần tính hàm Grundy của các số x bất kỳ. Với $x = p1^{k1} * p2^{k2} * ...$, thì $Grundy(x) = k1 + k2 + ...$ Nói cách khác, Grundy(x) là số số nguyên tố trong biểu diễn thừa số nguyên tố của x.
- Bước 2: Với mỗi số, ta chuyển nó thành số ước nguyên tố. Như vậy bài toán trở thành Misere Nim (Nim ngược). Điều kiện thắng là:
  - Nếu dãy gồm toàn số 1, phải có chẵn số 1.
  - Nếu dãy có ít nhất 1 số lớn hơn 1, tổng xor > 0.
- Bước 3: Quy hoạch động để đếm số tập hợp: đặt f(i, x, all_one) = số cách chọn ra tập hợp từ i số đầu tiên, nếu tổng xor = x. Nếu toàn bộ các số trong tập được chọn là 1, thì all_one = true.

Cài đặt:

- Lăng Trung Hiếu
- RR

J

Lưu ý từ điều kiện các quả bóng không tương tác với nhau ta có khi ta cắt một mặt cắt nằm ngang bất kỳ thì diện tích bóng cắt qua sẽ nhỏ hơn diện tích bể nên lượng nước trong bể (trừ bóng) sẽ tỉ lệ thuận với chiều cao của bể hay khi chiều cao nước cao hơn thì thể tích nước trong bể sẽ lớn hơn. Từ đây có thể dùng chặt nhị phân và điều kiện khi so sánh là tương quan giữa lượng nước trong bể ban đầu với lượng nước trong bể khi độ cao của nước là giá trị cần tính. Vậy giả sử ta có độ cao cuối cùng sau khi đổ nước vào bể là h, thì ta có thể tính được:

- Thể tích của bể so với mực nước: W*D*h
- Thể tích nước bị chiếm của từng quả bóng. Gọi tổng thể tính là Vb
- Thể tích của nước đổ vào V (là thể tích đề bài cho).

Nếu V > W*D*h - Vb thì mực nước cuối cùng sẽ phải lớn hơn h. Trường hợp còn lại mức nước cuối sẽ không lớn hơn h.

Vậy vấn đề mốt chốt của bài toán chỉ là tìm thể tích bị chiếm của từng quả bóng. Cần chia thành hai trường hợp:

- Trường hợp thứ nhất là khối lượng riêng của quả bóng lớn hơn khối lượng riêng của nước (> 1), trong trường hợp này thì quả bóng luôn chạm đáy bể.
- Trường hợp thứ hai là khối lượng riêng của quá bóng nhỏ hơn hoặc bằng khối lượng riêng của nước, dùng công thức Acsimet ta có thể tính được lượng nước quả bóng chiếm. Khi đó sẽ có hai trường hợp:
    - Một là quả bóng sẽ nổi, khi đó trọng lượng nước tương ứng với thể tích quả bóng chìm trong nước bằng với trọng lượng của quả bóng từ đây có thể dễ dàng tính được thể tích của bóng chiếm.
    - Hai là quả bóng sẽ chạm đáy vì độ cao của nước không đủ để trọng lượng nước mà quả bóng chiếm bằng với trọng lượng quả bóng.

Để tính thể tính quả bóng trong nước khi tính toán ta cần dùng công thức thể tích của mặt cắt hình cầu theo link sau: https://en.wikipedia.org/wiki/Spherical_cap

K

Bài này là một bài biếu và rất nhiều đội đã làm được. Nhanh nhất là đội UBUNTU ở phút thứ 5.

Cách 1: Tham lam
Sắp xếp lại dãy 3N tăng dần.
Sắp xếp các sinh viên vào các đội như sau:

SV	1	2	3	...	N	...								3N
Đội	1	2	3	...	N	1	1	2	2	3	3	...	N	N

Màu xanh là trung vị của các đội

Chứng minh:

- Gọi 3 thành viên trong team là #1, #2, #3 (#1 là mạnh nhất, #2 là các thành viên ta cần tìm).
- Xét #2 của team mạnh nhất:
  - Rõ ràng #2 phải đứng vị trí >= 2 từ phải sang (do còn phải chọn #1 của team đó).
- Xét #2 của team mạnh nhì:
  - Rõ ràng #2 phải đứng ở vị trí >= 4 từ phải sang (do còn có #1, #2 của team mạnh nhất và #1 của team đó).
- ...
- #2 của team mạnh thứ i sẽ đứng ở vị trí >= 2*i.
- Do đó, cách chọn trên là tối ưu.

Cài đặt: RR

Cách 2: quy hoạch động:

Sắp xếp lại dãy 3N tăng dần.
Gọi F[i][C3][C2] là: Xét i sinh viên đầu tiên, đã chọn C3 bạn vào vị trí yếu nhất của C3 đội, C2 bạn vào vị trí trung vị của C2 đội, i - C3 - C2 bạn vào vị trí mạnh nhất của i - C3 - C2 đội.

Dễ thấy để F[i][C1][C2] có nghĩa ta cần: C3 >= C2 >= i - C3 - C2.

Công thức quy hoạch động cụ thể các bạn có thể theo dõi ở đây.

L

Bài này chỉ có 4 đội giải được trong suốt kì thi, trong đó chỉ có **LINUX** là 1 trong 2 đội VN duy nhất giải được và là đội giải được nhanh nhất ở **phút thứ 150**.

Tư tưởng

Tư tưởng giải bài này là với mỗi cột, tức chủ đề mà ta gặp phải thì nếu **S** là tập các sinh viên mà ta KHÔNG chọn thì sẽ tốn bao nhiêu thời gian, từ đó suy ra được nếu chọn tập **T** các sinh viên thì sẽ tốn thời gian bao nhiêu cho từng chủ đề.

Với mỗi chủ đề, giả sử ta sắp xếp thời gian các sinh viên giải được chủ đề này theo thứ tự giảm dần trên mảng t[1..n]:

- Nếu ta chọn tập tất cả các sinh viên thì sẽ tốn thời gian là **t[1]**;
- Nếu chọn tập tât cả các sinh viên NGOẠI TRỪ sinh viên làm lâu NHẤT thì tốn thời gian là **t[2] = t[1] - (t[1]-t[2])**
- Nếu chọn tập tât cả các sinh viên NGOẠI TRỪ sinh viên làm lâu NHẤT và lâu NHÌ thì tốn thời gian là **t[3] = t[1] - (t[1]-t[2]) - (t[2] - t[3])**;
- ...

Như vậy giả sử với mỗi chủ đề ta sẽ có biến **Sum** là tổng các thời gian t[1], tức thời gian làm lâu nhất cho từng chủ đề, đồng thời ta tính được mảng **DP[0 .. 2^n-1], DP[bitMask]** là thời gian có thể giảm đi nếu ta KHÔNG chọn các sinh viên trong tập **bitMask**. Cuối cùng dựa trên mảng **DP** và **Sum** ta có thể tính được nếu chọn tập các sinh viên là **T** thì sẽ tốn thời gian là **Sum - DP[(2^n-1) ^ T]**

Tính tổng tập con

Trong quá trình tính mảng **DP** ta sẽ cần phải giải 1 bài toán con:

- Cho mảng **A[0 .. 2^n -1]**
- Cần tính mảng **F[0 .. 2^n -1]** với **F[S] = sum(A[T] | T là 1 tập con của S)**.

Cách làm là tính mảng F như sau:

- Khởi tạo F = A
- Với mỗi bit **i** từ 0 đến n-1:
- Với mỗi tập S chứa **2^i**, **F[S] += F[S - 2^i]**

Chứng minh:

- Để chứng minh cách làm trên đúng, ta cần chứng minh sau mỗi bước tại bit thứ i, giá trị của F[S] = tổng tất cả các A[S'] với S' thoả mãn:
    - S' là tập con của S
    - Các bit từ i+1 đến n-1 giống S.
- Ta dùng quy nạp để chứng minh.

  - Hiển nhiên trước vòng lặp i = 0, điều kiện trên đúng, do ta chưa làm gì cả.
  - Giả sử đã đúng đước bước k-1. Tại bước k:
    - nếu S không chứa bit k thì F[S] không thay đổi → đúng
    - nếu S chứa bit k, thì F[S] = F[S] + F[S - 2^k]. Vì sau bước k-1:
      - F[S] = tổng tất cả A[S'] với S' là tập con của S và S' có bit k đến n-1 giống S. Vì bit k của S là 1, nên nói cách khác:
        - S' là tập con của S, có bit k+1 đến n-1 giống S và bit k = 1.
      - F[S - 2^k] = tổng tất cả A[T'] với T' là tập con của (S - 2^k) và T' có bit k đến n-1 giống (S - 2^k). Do bit k của S là 1, nên nói cách khác:
        - T' là tập con của S, có bit k+1 đến n-1 giống S và bit k = 0.
      - Do đó, gộp {S'} với {T'} lại sẽ được tập G' chứa tất cả các tập con của S có bit từ k+1 đến n-1 giống S.
    - → đúng

Độ phức tạp cho tổng cộng là **O(m * n * log(n) + n * 2^n)**

Các bạn có thể xem lời giải bằng Code để hiểu rõ hơn:

- Lê Yên Thanh
- Lăng Trung Hiếu