F

#define MAX 100100

const long long INF = (long long)1e18 + 7LL;

int numValue, values[MAX]; // dãy nhập vào từ input

long long sumNegative; // tổng tọa độ của các vị trí âm

int numPositive, positive[MAX]; // danh sách các điểm có tọa độ dương

long long sum[MAX]; // mảng tổng dồn

long long result; // biến lưu kết quả

void init(void) {  
 scanf(“%d”, &numValue);  
 FOR(i, 1, numValue) scanf(“%d”, &values[i]);

}

void solve(void) {  
 numPositive = 0;  
 sumNegative = 0;  
   
 // duyệt qua dãy n điểm ban đầu, liệt kê ra các điểm dương và các điểm âm  
 FOR(i, 1, numValue) {  
 if (values[i] > 0) positive[++numPositive] = values[i];  
 else sumNegative += values[i];  
 }  
  
 // sắp xếp các điểm dương theo thứ tự từ trái qua phải  
 sort(positive + 1, positive + numPositive + 1);  
 // tính mảng tổng dồn các điểm tọa độ dương  
 FOR(i, 1, numPositive) sum[i] = sum[i - 1] + positive[i];  
  
 // duyệt thử các trường hợp chọn i điểm đi về 0:  
 FOR(i, 0, numPositive) {  
 // các điểm có tọa độ positive[1], positive[2], ..., positive[i] đi về 0  
 // phần còn lại đi về P  
 int tmp = (numPositive - i) / 2;  
   
 // tổng quãng đường di chuyển của các điểm đi tới P (áp dụng bài toán phụ)  
 long long sumToP =   
 (sum[numPositive] - sum[numPositive - tmp]) - (sum[i + tmp] - sum[i]);  
   
 long long sumTo0 = sum[i]; // tổng quãng đường di chuyển của các điểm đi tới 0  
  
 result = min(result, sumToP + sumTo0 - sumNegative);  
 }

}

void process(void) {  
 solve(); // giải bài toán với trường hợp đặt cổng teleport thứ hai ở điểm dương  
 FOR(i, 1, n) values[i] \*= -1; // đổi dấu tất cả các phần tử trong mảng solve  
 solve(); // giải bài toán với trường hợp đặt cổng teleport thứ hai ở điểm âm  
 // do đã đổi dấu các điểm ở trên nên cách xử lý giống như khi chọn  
 // điểm teleport ở miền dương  
 cout << result << endl;  
}

H.

# 0. Biểu diễn đồ thị

// struct biểu diễn cạnh của đồ thị

struct Edge {  
 int u, v; // hai đầu của cạnh  
 int cost; // trọng số cạnh  
 bool inTree; // đánh dấu cạnh có thuộc cây khung hay không  
  
 Edge(int \_u = 0, int \_v = 0, int \_c = 0) {  
 u = \_u; v = \_v; cost = \_c; inTree = false;  
 }  
};

#define MAX 100100

int numNode, numEdge, numQuery; // số đỉnh, số cạnh của đồ thị và số truy vấn

Edge edges[MAX]; // mảng lưu thông tin các cạnh

vector<int> adj[MAX]; // adj[u] danh sách chỉ số các cạnh kề với đỉnh u

void loadGraph(void) { // đọc đồ thị  
 scanf(“%d%d%d”, &numNode, &numEdge, &numQuery);  
 FOR(i, 1, numNode) adj[i].clear(); // khởi tạo xóa danh sách kề  
  
 FOR(i, 1, numEdge) {  
 int u, v, c; scanf(“%d%d%d”, &u, &v, &c);  
  
 // bản chất là gán edges[i].u = u; edges[i].v = v; edges[i].cost = c  
 edges[i] = Edge(u, v, c);  
 adj[u].push\_back(i);  
 adj[v].push\_back(i);  
 }  
}

# 1. Dựng một cây khung bất kỳ của đồ thị

Chú ý: Không cần phải là cây khung nhỏ nhất.

int dsu[MAX]; // mảng disjoint set

bool join(int u, int v) {

} // hàm gộp hai đỉnh u và v trong disjointset, trả về true nếu trước đó u và v không thuộc cùng một thành phần liên thông; trả về false nếu ngược lại.

void buildSpanningTree(void) {  
 FOR(i, 1, numNode) dsu[i] = -1; // khởi tạo mảng disjoint set  
 FOR(i, 1, numEdge) edges[i].inTree = join(edges[i].u, edges[i].v);  
}

# 2. Cài đặt thuật toán LCA để tìm khoảng cách giữa hai đỉnh trên cây khung

#define LOG 17

int par[MAX][LOG + 1]; // mảng cha trong thuật toán LCA

int high[MAX]; // high[u] = số cạnh trên đường đi từ u đến gốc  
long long distRoot[MAX]; // distRoot[u] = tổng trọng số các cạnh trên đường đi từ u đến gốc

// DFS trên cây khung

void dfs(int u) {  
 // duyệt qua tất cả các cạnh kề với đỉnh u, nhưng chỉ tính các cạnh được chọn vào cây khung  
 for (int i : adj[u]) if (edges[i].inTree) { // chú ý i là chỉ số của cạnh  
 int v = u == edges[i].u ? edges[i].v : edges[i].u;  
 if (v == par[u][0]) continue;  
  
 par[v][0] = u;  
 high[v] = high[u] + 1;  
 distRoot[v] = distRoot[u] + edges[i].cost;  
 dfs(v);  
 }  
}

int lca(int u, int v) = tìm LCA của nút u và nút v trên cây khung

long long distTree(int u, int v) { // khoảng cách giữa u và v trên cây khung  
 return distRoot[u] + distRoot[v] - 2 \* distRoot[lca(u, v)];

}

# 3. Với mỗi cạnh không thuộc cây khung, chọn ra một trong hai đầu, cho vào danh sách đỉnh quan trọng

Chú ý: Đồ thị trong bài có không quá 51 cạnh không thuộc cây khung. Vi vậy,

vector<int> nodes; // danh sách các đỉnh quan trọng

nodes.clear();

// duyệt qua các cạnh của đồ thị không thuộc cây khung

for (int i = 1; i <= numEdge; i++) if (!edges[i].inTree) nodes.push\_back(edges[i].u);

// bỏ đi các đỉnh xuất hiện nhiều lần trong danh sách đỉnh quan trọng

#define ALL(v) (v).begin(), (v).end()

sort(ALL(nodes));  
nodes.resize(unique(ALL(nodes)) - nodes.begin());

# 4. Với mỗi đỉnh quan trọng, dùng thuật toán Dijkstra để tính độ dài đường đi ngắn nhất từ đỉnh quan trọng tới mọi đỉnh trên đồ thị

Chú ý: Ở bước này ta không quan tâm đến cây khung nữa, và khi dijkstra ta xét tất cả các cạnh của đồ thị, bất kể là chúng thuộc cây khung hay không

vector<long long> dist[MAX];

void dijkstra(int s) { // Dijkstra từ đỉnh s, tìm đường đi ngắn nhất tới mọi đỉnh khác

dist[s].assign(numNode + 1, INF); // khởi tạo đường đi ngắn nhất tới mọi đỉnh là INF  
  
 priority\_queue<...> q;  
 dist[s][s] = 0; q.push(make\_pair(0, s));  
  
 while (!q.empty()) { ... }  
}

for (int s : nodes) dijkstra(s);

# 5. Trả lời truy vấn độ dài đường đi ngắn nhất từ s đến e

long long query(int s, int e) {  
 // trường hợp 1: Đường đi chỉ sử dụng các cạnh thuộc cây khung  
 long long result = distTree(s, e);

// trường hợp 2: Đường đi sử dụng cả các cạnh không thuộc cây khung  
 // duyệt qua tất cả các đỉnh trong danh sách đỉnh quan trọng  
 for (int u : nodes) result = min(result, dist[u][s] + dist[u][e]);

return result  
}

J.

# Subtask 3

Gọi f(mask, i) là số cách xếp một tập con những người được quy định bởi dãy mask (mask: lưu lại trạng thái những người nào đã dùng, những người nào chưa dùng), i là số người đã dùng bị che bởi một người khác.

f(0, 0) = 1: Trạng thái khởi đầu, chưa có ai được xếp vào hàng, số người bị che bằng 0.

Chuyển trạng thái: bản chất là cho thêm một người vào vị trí tiếp theo của hàng:

#define MASK(i) (1 << (i))

#define BIT(x, i) (((x) >> (i)) & 1) // kiểm tra bit thứ i của số x

int h[]; // mảng chiều cao của n người: h[0], h[1], ... h[n-1]

int f[MASK(20)][22];

f[0][0] = 1;

for (int mask = 0; mask < MASK(n); mask++) {  
 // tính chiều cao của người cao nhất trong số những người đã vào hàng   
 int maxHigh = 0;  
 for (int i = 0; i < n; i++) if (BIT(mask, i)) maxHigh = max(maxHigh, h[i]);  
  
 // chuyển trạng thái quy hoạch động  
 for (int i = 0; i < n; i++) if (f[mask][i] > 0) {  
 for (int j = 0; j < n; j++) if (!BIT(mask, j)) {  
 f[mask | MASK(j)][i + (h[j] < maxHigh ? 1 : 0)] += f[mask][i];  
 }  
 }  
}  
  
cout << tổng (f[MASK(n) - 1][0..k]);

# Subtask 6

Chia n người thành các nhóm có cùng chiều cao. Giả sử n người được chia vào m nhóm. Các nhóm này được sắp xếp theo thứ tự nhóm 1 gồm những người cao nhất, nhóm 2 gồm những người thấp hơn, ..., nhóm thứ m là những người thấp nhất. Gọi số người trong mỗi nhóm lần lượt là c1 c2 ... cm (c1 người cao nhất, c2 người thấp hơn,... cm người thấp nhất).

Cách xây dựng cấu hình (lời giải, thứ tự sắp xếp...) ở subtask này như sau: Ta xây dựng từng nhóm người vào trong dãy, từ nhóm người cao nhất tới nhóm người thấp nhất. Ở mỗi bước chuyển trạng thái, ta thêm một nhóm người vào dãy hiện tại: Nhóm người này có thể trà trộn tùy ý vào trong những người đã được xếp ở bước trước.

Giả sử có 3 nhóm người, nhóm 1 có 5 người, nhóm 2 có 2 người, nhóm 3 có 4 người.

() -> (1, 1, 1, 1, 1) -> (1, 2, 1, 1, 2, 1, 1) -> (3, 1, 2, 1, 3, 1, 2, 1, 3, 1, 3)  
 -> (2, 2, 1, 1, 1, 1, 1) -> (2, 2, 1, 3, 1, 3, 1, 3, 1, 3, 1)

Gọi f(i, j) là số cách xếp i nhóm người đầu tiên (nhóm có c1 c2 ... ci người) vào dãy sao cho có j người bị che tầm nhìn. Bước chuyển trạng thái có dạng f(i + 1, j + delta) += f(i, j).

Giả sử sau khi xếp c(I+1) người thuộc nhóm i+1 vào hàng, có thêm t người bị che tầm nhìn: trong hàng bây giờ có tổng cộng j+t người bị che tầm nhìn trong đó j người thuộc các nhóm phía trước (các nhóm từ 1 đến i) và t người thuộc nhóm i+1. Chú ý, mọi người thuộc nhóm i+1 đều thấp hơn hẳn những người đã có trong hàng trước đó (những người thuộc nhóm c1 c2 ... ci), do đó những người mới vào hàng không thể gây che mắt những người cũ. Để một người thuộc nhóm i+1 không bị che mắt, họ phải đứng đầu hàng.

Công thức quy hoạch động: F(i + 1, j + t) += F(i, j) \* C(k, si - 1 + t) \* c(i+1)!

m: số nhóm

c[1], c[2], ..., c[m]: số người thuộc các nhóm

s[1], s[2], ..., s[m]: mảng tổng tiền tố của mảng c

f[0][0] = 1;

for (int i = 0; i < m; i++) for (int j = 0; j <= k; j++)  
 for (int t = 0; j + t <= k && t <= c[i + 1]; t++)  
 f[i + 1][j + t] =   
 (f[i + 1][j + t] + 1LL \* comb(t, s[i] + t - 1) \* frac[c[i + 1]) % MOD;  
cout << (f[m][0] + f[m][1] + ... + f[m][k]) << endl;

## Tính tổ hợp chập k của n phần tử theo modulo

Trường hợp đặc biệt: k, n <= 1e6, mod ~1e9 (modulo nguyên tố và lớn hơn n). Áp dụng định lý Fermat bé (a/b mod p = a \* b^(p-2) mod p): nCk = n! / k! / (n-k)! = n! \* (k! ^ (mod-2)) \* ((n-k)! ^ (mod-2)). Tính nCk trong O(1)bằng cách chuẩn bị trước các giá trị n! và n!^(mod - 2)

const int MOD = 998244353;  
int pw(int x, int k) {  
 return x lũy thừa k theo modulo MOD;  
}  
  
int frac[N]; // n! % MOD  
int finv[N]; // n! lũy thừa (MOD - 2) % MOD  
  
void prepare(void) {  
 frac[0] = finv[0] = 1;  
 for(int i = 1; i < N; i++) {  
 frac[i] = 1LL \* frac[i - 1] \* i % MOD;  
 finv[i] = pw(frac[i], MOD - 2);  
 }  
}  
  
int comb(int k, int n) { // tính tổ hợp chập k của n phần tử   
 if (k > n) return 0;  
 return 1LL \* frac[n] \* finv[k] % MOD \* finv[n - k] % MOD;  
}

