# I. Машинки

Read three integers n, k, and p from the input, indicating that there are n positions, k ground positions, and p orders.

Traverse the orders in order in reverse order, and store the position index of each order in the vector of the corresponding position in positions.

Iterate over each order in order:

Get the position index of the current order from positions and pop it.

If the ground position collection ground already contains the combination of the current position and the order number, it means that the position is already occupied and needs to be removed, and put the last position index of the order into the corresponding position vector in positions.

Increment the counter for the answer.

If the size of the set of ground positions reaches k, remove the last element.

Add the combination of position index and order number of the current order to the ground position collection ground.

Code：

#include <bits/stdc++.h>

using namespace std;

int n, k, p;

int answer = 0;

auto solve() {

set< pair<int, int> > ground;

cin >> n >> k >> p;

vector< vector<int> > positions(n + 1, vector<int>());

vector<int> order(p);

int i = 0;

do {

cin >> order[i];

++i;

} while (i < p);

i = 0;

do {

positions[i].emplace\_back(p);

++i;

} while (i <= n);

i = p - 1;

do {

positions[order[i]].emplace\_back(i);

--i;

} while (i >= 0);

i = 0;

do {

positions[order[i]].pop\_back();

if (ground.count({ i, order[i] })) {

ground.erase({ i, order[i] });

ground.emplace(positions[order[i]].back(), order[i]);

++i;

continue;

}

++answer;

if (ground.size() == k) ground.erase(--ground.end());

ground.emplace(positions[order[i]].back(), order[i]);

++i;

} while (i < p);

return answer;

}

int main(void) {

ios::sync\_with\_stdio(0);

cin.tie(0);

cout.tie(0);

cout << solve();

return 0;

}

# J. Гоблины и очереди

Import the required header files and namespaces.

Declare and initialize three queues: leftLine, rightLine and resultSet.

Define an enumeration type Corblin, which contains three values of Normal, Special and Samen, used to represent the type of the input string.

In the main function, read the input n value, indicating that n rows of data will be input next.

Use a do-while loop to loop n times, reading each line of the input string.

If the length of the left line is less than that of the right line and the right line is not empty, move the first element of the right line to the end of the left line.

If it is of type Normal, convert the input string to an integer starting from the second character and add it to the end of the right line.

If it is a Special type, convert the input string to an integer starting from the second character and add it to the head of the left line.

If it is Samen type, judge whether the left line is empty, if it is not empty, add the first element of the left line to the end of the result set, remove it from the left line, and add one to the output counter; if the left line is empty and The right line is not empty, add the first element of the right line to the end of the result set

Code：

#include <iostream>

#include <deque>

#include <vector>

using namespace std;

deque<int> leftLine(0);

deque<int> rightLine(0);

deque<int> resultSet;

enum Corblin {

Normal = 0,

Special,

Samen

};

enum Corblin judge(string s){

if(s[0] == '+'){

return Normal;

} else if(s[0] == '-'){

return Samen;

} else {

return Special;

}

}

int main() {

int n;

cin >> n;

cin.get();

int i = 0;

int output = 0;

string input;

do {

getline(cin, input);

enum Corblin result = judge(input);

if(leftLine.size() < rightLine.size() && !rightLine.empty()){

leftLine.push\_back(rightLine.front());

rightLine.pop\_front();

}

if(result == Normal){

rightLine.push\_back(stoi(input.substr(2)));

} else if(result == Special){

rightLine.push\_front(stoi(input.substr(2)));

} else if(result == Samen){

if(!leftLine.empty()){

resultSet.emplace\_back(leftLine.front());

leftLine.pop\_front();

output++;

} else if(!rightLine.empty()){

resultSet.emplace\_back(rightLine.front());

rightLine.pop\_front();

output++;

}

}

i++;

} while(i < n);

for(int k = 0 ; k < output; k++){

cout << resultSet[k] << endl;

}

return 0;

}

# K. Менеджер памяти-1

Memory blocks are managed through insertion and deletion operations. Initially, the entire memory block is represented as a block with a starting index of 1 and a size of n, and the block is inserted into blocks and blocks\_by\_size.

The main loop of the algorithm reads the operation type k from the input, and then performs the corresponding operation according to the value of k.

If k is less than or equal to 0, it means to release a memory block. First get the index and size of the previously allocated memory block from the history array. Then find the corresponding block according to the index, and delete it from blocks and blocks\_by\_size. Depending on the location of the freed block and the conditions of adjacent blocks, a merge operation may be required, that is, the adjacent blocks are merged into a larger block and reinserted into the data structure.

If k is greater than 0, it means that a memory block of size k is to be allocated. First find blocks with a size not smaller than k from blocks\_by\_size, and select one of the blocks for allocation. Removes the block from the data structure and returns its starting index as the result of the assignment. If the remaining size after allocation is greater than 0, insert the remaining part as a new block into the data structure, sorted by size.

After each operation, save the operation type k and the result index index into the history array for subsequent operations.

Code：

#include <iostream>

#include <map>

#include <unordered\_map>

using namespace std;

map<int, int> blocks;

multimap<int, int> blocks\_by\_size;

void insert(const pair<int, int>& pair) {

blocks.insert(pair);

blocks\_by\_size.insert({pair.second, pair.first});

}

void remove(const multimap<int, int>::iterator& it) {

blocks.erase(it->second);

blocks\_by\_size.erase(it);

}

void remove\_by\_size(const multimap<int, int>::iterator& it) {

auto it\_d = blocks\_by\_size.find(it->second);

while (it\_d->second != it->first) it\_d++;

blocks\_by\_size.erase(it\_d);

blocks.erase(it);

}

int main() {

int n, m, k, index, size;

cin >> n >> m;

pair<int, int> history[m];

insert({1, n});

int a = 0;

do{

cin >> k;

if (k <= 0) {

int index\_x = history[abs(k) - 1].second;

int size\_x = history[abs(k) - 1].first;

if (index\_x != -1) {

auto it\_r = blocks.lower\_bound(index\_x);

auto it\_l = (it\_r != blocks.begin()) ? prev(it\_r) : blocks.end();

if (it\_r != blocks.end() && it\_r->first == index\_x + size\_x) {

if (it\_l != blocks.end() && it\_l->first + it\_l->second == index\_x) {

index = it\_l->first;

size = it\_l->second + it\_r->second;

remove\_by\_size(it\_l);

remove\_by\_size(it\_r);

insert({index, size + size\_x});

} else {

size = it\_r->second;

remove\_by\_size(it\_r);

insert({index\_x, size + size\_x});

}

} else {

if (it\_l != blocks.end() && it\_l->first + it\_l->second == index\_x) {

index = it\_l->first;

size = it\_l->second;

remove\_by\_size(it\_l);

insert({index, size + size\_x});

} else insert({index\_x, size\_x});

}

index = 0;

}

} else {

auto it = blocks\_by\_size.lower\_bound(k);

if (it == blocks\_by\_size.end()) index = -1;

else {

index = it->second;

size = it->first - k;

remove(it);

if (size > 0) insert({index + k, size});

}

cout << index << endl;

}

history[a] = {k, index};

a++;

}while(a < m);

return 0;

}

# L. Минимум на отрезке

Creates a double-ended queue (deque) dq for storing element indices in the sliding window.

Use a loop to iterate over the first subarray of the sliding window from 0 to k-1 and do the following:

When dq is not empty and the current element a[i] is smaller than the tail element a[dq.back()], delete the tail element until dq is empty or a[i] is greater than or equal to a[dq.back()].

Add the current element index i to the tail of dq.

Output a[dq.front()], which is the minimum value of the current sliding window.

Use another loop to iterate over the remainder of the array (i.e. the subsequent subarrays of the sliding window) starting at k and do the following:

When dq is not empty and the index of the first element dq.front() is less than or equal to the current index i-k, it means that the first element of the queue is no longer in the current sliding window, so it is deleted from dq.

When dq is not empty and the current element a[i] is smaller than the tail element a[dq.back()], delete the tail element until dq is empty or a[i] is greater than or equal to a[dq.back()].

Add the current element index i to the tail of dq.

Output a[dq.front()], which is the minimum value of the current sliding window.

Code：

#include <iostream>

#include <deque>

using namespace std;

int main() {

int n, k;

cin >> n >> k;

int a[n];

for (int i = 0; i < n; i++) {

cin >> a[i];

}

deque<int> dq;

for (int i = 0; i < k; i++) {

while (!dq.empty() && a[i] < a[dq.back()]) {

dq.pop\_back();

}

dq.push\_back(i);

}

cout << a[dq.front()] << " ";

for (int i = k; i < n; i++) {

while (!dq.empty() && dq.front() <= i - k) {

dq.pop\_front();

}

while (!dq.empty() && a[i] < a[dq.back()]) {

dq.pop\_back();

}

dq.push\_back(i);

cout << a[dq.front()] << " ";

}

return 0;

}