# Đề thi Olympic - Mảng một chiều

|  |  |
| --- | --- |
| Có thể là hình ảnh về văn bản | Nếu giả sử các giá trị a[i] đề cho không phải nằm trong đoạn từ 1 —> 10^5 mà chỉ nằm trong đoạn từ 1 —> 9 tức là nó chỉ có 1 chữ số thôi. Nếu như thế bài này giải quyết có phải rất đơn giản không? Chỉ là đi sắp xếp mảng giảm dần theo đúng giá trị của nó, giả sử ban đầu cho mảng là: {5, 3, 1, 9, 8, 5, 3} thì sau khi sắp xếp giảm dần mảng sẽ là: {9, 8, 5, 5, 3, 3, 1} và đáp án chính là 9855331.  Nếu áp dụng quy tắc trên với ví dụ mẫu đề cho 3 số là: {12, 907, 91} thì sắp xếp giảm dần xong sẽ có mảng là: {907, 91, 12} trong khi đó đáp án đúng mà đề đưa ra phải là: {91, 907, 12} nghĩa là số 91 sẽ là số lớn nhất lớn hơn 2 số còn lại, rồi 907 sẽ là số lớn nhì lớn hơn 12 còn 12 là số nhỏ nhất. Vậy nếu muốn sắp xếp ra đúng được như vậy ta phải nhìn ra được quy tắc tụi nó so sánh với nhau làm sao để biết được thằng nào lớn hơn thằng nào. Chỉ cần nhìn ra được đúng quy luật tụi nó so sánh với nhau để quyết định thằng nào lớn hơn thằng nào thì đó chính là mấu chốt giúp giải quyết được bài này. |

Ở đây mình sẽ phân tích cho các bạn nhìn ra cách mà tụi nó so sánh quyết định thằng nào lớn hơn thằng nào để biết thằng nào sẽ đứng trước, thằng nào sẽ đứng sau để tạo thành được số lớn nhất. Nó đơn giản làm theo đúng quy luật của bài thôi, nếu ta có 2 số a, b cần so sánh với nhau (hiện tại a đang đứng trước b) thì ta sẽ có 2 trường hợp hoặc kết quả **ab** hoặc kết quả **ba** ta xem trong 2 trường hợp đó trường hợp nào cho kết quả lớn hơn. Nếu trường hợp **ab > ba** thì hiện tại tụi nó đã đứng đúng vị trí (a đứng trước b) nên ta không cần đổi chỗ, nhưng nếu trường hợp **ba > ab** thì ta phải đổi chỗ tụi nó lại vì hiện tại a đang đứng trước b, ta cần đổi chỗ cho b ra trước a. Đơn giản là vậy thôi.

Thử áp dụng quy tắc so sánh trên để đi sắp xếp mảng ví dụ đề bài cho là: {12, 907, 91} giả sử theo thuật toán sắp xếp cơ bản Interchange Sort dùng 2 vòng lặp lồng nhau mà các bạn hay làm hồi học nhập môn lập trình.

Thì đầu tiên so sánh a[0] là 12 sẽ so sánh với a[1] là 907, ta có 2 số là 12907 và 90712 thì ta thấy 90712 sẽ lớn hơn 12907 nên ta sẽ đổi chỗ cho 907 lên, 12 xuống, lúc này mảng được: {907, 12, 91}.

Tiếp tục so sánh a[0] là 907 với a[2] là 91, ta có 2 số là 90791 và 91907 thì ta thấy 91907 sẽ lớn hơn 90791 nên ta sẽ đổi chỗ tụi nó, lúc này mảng được: {91, 12, 907}.

Tiếp tục so sánh a[1] là 12 với a[2] là 907, ta có 2 số là 12907 và 90712 thì ta thấy 90712 sẽ lớn hơn 12907 nên ta sẽ đổi chỗ tụi nó, lúc này mảng được: {91, 907, 12}.

Đến đây đã hết cặp để so sánh. Ta kết luận số lớn nhất được tạo ra sẽ là: 9190712.

#include <iostream>

#include <cstdio>

#include <cstring>

#include <algorithm>

#define MAX 1000

using namespace std;

int n;

string a[MAX];

int main(){

//nhap n

cin >> n;

//nhap tung gia tri cho chuoi s

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

cin >>a[i];

}

//hoan doi neu 2 ky tu ke nhau tao thanh so lon hon

//vi du: 12 3 => 312 thay vi 123

for(int i=0; i<n-1; i++){

for(int j=i+1; j<n; j++){

if(a[i]+a[j] < a[j]+a[i]){

swap(a[i],a[j]);

}

}

}

cout << endl <<"so tim duoc: ";

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

cout << a[i] <<".";

}

//ket thuc chuong trinh

return 0;

}

8

56 23 67 123 789 12 890 456

so tim duoc: 890.789.67.56.456.23.123.12

5

31 607 123 76 221

so tim duoc: 76.607.31.221.123

# Giải thuật tham lam

|  |  |
| --- | --- |
| Có thể là hình ảnh về văn bản | Đọc đề ta thấy rằng cái sau cùng quan tâm chính là chuỗi kết quả sao cho chuỗi này chỉ chứa các ký tự số (không có ký tự chữ cái gì hết nhé, chỉ chứa các ký tự số thôi) và có đủ K ký tự theo đúng thứ tự xuất hiện của nó tức là ký tự nào xuất hiện trước trong chuỗi input thì nó phải nằm trước và hãy cố gắng để chuỗi kết quả số đó là số lớn nhất có thể, đó chính là mật khẩu. Rồi thì in mật khẩu đó ra và kiểm tra tính đối xứng của nó.  Thì việc kiểm tra tính đối xứng quá đơn giản, chỉ cần tạo ra chuỗi mới là chuỗi đảo ngược của chuỗi kết quả, sau đó so sánh 2 chuỗi với nhau nếu bằng nhau thì chuỗi kết quả là chuỗi đối xứng, không bằng nhau thì không đối xứng, đơn giản vậy thôi.  Cái khó là bây giờ làm sao tạo ra được chuỗi kết quả có đúng K ký tự chữ số theo đúng thứ tự nó xuất hiện trong input sao cho chuỗi kết quả số đó là số lớn nhất trong tất cả các trường hợp hợp lệ. |

Thì trước khi đi vào phần xử lý chính đó thì ta phải làm bước đầu tiên gọi là tiền xử lý dữ liệu: Đó là chuỗi đầu vào đang có lộn xộn cả ký tự chữ cái và ký tự chữ số, mà ở đây ta chỉ đang cần xét và làm việc với những ký tự chữ số thôi, còn những ký tự chữ cái thì bỏ qua. Nên ở đây ta có nhiều giải pháp lựa chọn xử lý cho vấn đề này. Ví dụ dựa trên chính chuỗi ban đầu, ta gặp ký tự nào không thỏa là ký tự số thì ta xóa ký tự tại vị trí đó đi, để sau cùng chuỗi chỉ còn lại các ký tự số thôi.

Nếu làm theo cách xóa ở trên thì tính ra độ phức tạp sẽ rất lớn. Vì bản chất khi xóa một phần tử trong mảng (chuỗi bản chất là mảng 1 chiều các ký tự) là các phần tử ở ngay sau vị trí xóa tất cả đồng loạt di dời về phía trước 1 ô. Như thế giả sử chuỗi có độ dài N mà ta xóa ngay vị trí 0 thì nghĩa là N-1 phần tử phía sau sẽ phải di dời lên => thao tác xóa trên mảng sẽ có độ phức tạp trong trường hợp xấu nhất là O(N) với N là số lượng phần tử của mảng. Điều đó nghĩa là nếu ta liên tục lặp đi lặp lại thao tác xóa nhiều ký tự trong mảng, giống như trong trường hợp này ta xét từ đầu đến cuối chuỗi cứ gặp ký tự nào không phải ký tự số thì ta gọi hàm xóa, như thế độ phức tạp lúc này sẽ là O(N \* N) tức là O(N^2) (N lũy thừa 2, phép ^ ở đây mình viết tắt của ký hiệu lũy thừa chứ không phải là phép toán xor bit nha các bạn). Tại sao lại O(N^2) ? Vì mỗi lần xóa 1 ký tự thì độ phức tạp là O(N) và có thể ta sẽ phải xóa cả N ký tự như thế sẽ là N \* N ⇔ N^2.

Trong khi đó nếu muốn tối ưu hơn chỗ này, ta chỉ cần tạo ra 1 chuỗi mới, sau đó duyệt vào từng ký tự trong chuỗi ban đầu, gặp ký tự nào là ký tự số thì ta thêm ký tự đó qua chuỗi mới tạo kia. Như thế ta chỉ cần duyệt hết chuỗi ban đầu thì ta đã có chuỗi mới chỉ chứa các ký tự số trong chuỗi ban đầu theo đúng thứ tự nó xuất hiện. Tính ra độ phức tạp luôn là O(N) thì sẽ tốt hơn việc xóa như ở trên là O(N^2). Thậm chí việc xóa như trên nếu không cẩn thận rất dễ bị sót ký tự, ví dụ nếu xóa 2 hay 3 hay nhiều ký tự liên tiếp nhau nếu mà các bạn quên không giảm vị trí index xuống sau mỗi lần xóa thì sẽ bị sót ký tự chưa được xóa hết đấy nhé.

Qua ví dụ ở trên cũng cho các bạn thấy một kinh nghiệm là nếu chúng ta có thể giải quyết vấn đề mà tránh đụng tới xóa thì ta nên tránh vì việc xóa sẽ có độ phức tạp lớn và dễ sai.

Rồi thì các bước xử lý ở trên rất đơn giản, coi như giờ đây ta đã có chuỗi s là chuỗi chỉ chứa các ký tự số hen, giờ thì ta tìm cách xử lý làm sao để tạo ra chuỗi kết quả cần tìm.

**Bước 1. Tách các số có trong chuỗi ra thành 1 chuỗi mới**

abc abc 123abc2acb1da00 cc → 1232100

0abc abc0 123abc2acb1da cc → 0012321

...

**Bước 2. Xác định độ dài của chuỗi số sau khi tạo ở bước 1**

len = 7

**Bước 3. Chọn ra số lớn nhất trong khoảng (L-k) số, với đề bài ví dụ k=5**

max(123|2100)

max(001|2321)

**Bước 4. Ghi nhận vị trí tìm thấy số lớn nhất → index**

index sẽ chứa vị trí của số lớn nhất

**Bước 5. Tính toán lại chiều dài của chuỗi số và sô ký tự còn lại**

k=k-1

L=len-index

**Bước 6. Quay lại bước 3 cho đến khi k>=L thì in ra những số còn lại**

**Cụ thể:**

|  |  |
| --- | --- |
| 5  abc abc 123abc2acb1da00 cc | k=5  chuỗi số: 1232100, len =7 |
| L=7, k=5, start=0, finish=7-5=2  MAX(1232100) | index = posmax = 2  k=4 |
| L=7-2-1=4  2100 | L=k |

|  |  |
| --- | --- |
| 5  0abc abc0 123abc2acb1da cc | k=5  chuỗi số: 0012321, len =7 |
| L=7, k=5, start=0, finish=7-5=2  MAX(001231) | index = posmax = 2  k=4 |
| L=7-2-1=4  2321 | L=k |

|  |  |
| --- | --- |
| 5  1abc a8c8 ea25 cc20 | k=5  chuỗi số: 1882520, len =7 |
| L=7, k=5, start=0, finish=7-5=2  MAX(1882520) | index = posmax = 1  k=4 |
| L=7-1-1=5, start=2(index+1), finish=5-2=3  MAX(1882520) | index=posmax = 2  k=3 |
| L=7-2-1=4, start=3(index=1), finish=4-3=1  MAX(1882520) | index=posmax=4  k=2 |
| L=7-4-1=2 ==k  20 |  |

**Code:**

#include <bits/stdc++.h>

using namespace std;

// khai bao bien

int k;

FILE \*fp;

string s, num\_string;

string password;

void readdata()

{

getline(cin, s);

k = stoi(s);

getline(cin, s);

// chuyen cac ky tu so vao chuoi num\_string de xu ly

for (int i = 0; i < s.length(); i++)

{

if (s[i] >= '0' && s[i] <= '9')

{

num\_string.push\_back(s[i]);

// chuyen cac ky tu so vao num\_string

}

}

// in ra cac gia tri de kiem tra

cout << k << endl;

for (int i = 0; i < num\_string.length(); i++)

{

cout << num\_string[i];

}

}

void processing()

{

cout << endl << "Chuoi password can tim: " << endl;

int L, len;

// xac dinh chieu dai cua chuoi

L = num\_string.length();

len=L;

//cout << endl << L << endl;

if (k == L)

{

cout << "Password = " << num\_string << endl;

}

else

{

int start = 0;

while(k<L){

//tim ra so lon nhat [start..finish]

int finish = L-k;

char max = num\_string[start];

int index = start;

cout << "L= " << L << " k = "<< k << " start = " << start << " finish = " << finish+start << endl;

for(int i=start; i<=finish+start; i++){

if(max < num\_string[i]){

max = num\_string[i];

index=i;

}

}

start=index+1;

cout << "index = " << index << endl;

cout << "max = " << max -48 << endl;

//luu so lon nhat vao chuoi password

password+=max;

L=len-index-1;//L bot di so luong ky tu

k--;//bot di chieu dai cua password

}

//khi k>=L thi in ra phan con lai

//cout << num\_string.substr(start,k);

//luu phan con lai vao mang password

password+=num\_string.substr(start,k);

cout << password << endl;

}

}

void checkpassword(){

string rever\_res = password;

reverse(rever\_res.begin(), rever\_res.end());

cout << "Chuoi dao nguoc: "<< rever\_res << endl;

if(password == rever\_res){

cout << "YES";

}

else{

cout << "NO";

}

}

int main()

{

// tang toc do doc du lieu

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

cin.tie(0);

cout.tie(0);

// doc du lieu

readdata();

// xu ly

processing();

//check password

checkpassword();

return 0;

}

**Giải thích một số chỗ trong code ở trên:**

1/ 3 dòng đầu tiên của hàm main ( ios::sync\_with\_stdio(0); cin.tie(0); cout.tie(0); ) sẽ giúp đẩy nhanh tốc độ nhập xuất dữ liệu nếu các bạn đang code theo ngôn ngữ C++ (để hiểu vì sao thì các bạn cứ gõ mấy từ này lên Google là sẽ ra rất nhiều bài viết giải thích nhé), nên kinh nghiệm trong các bài mà cần nhập xuất nhiều thì ta nên để thêm 3 dòng này vào đầu để chương trình chạy nhanh hơn. Tất nhiên nếu các bạn code theo ngôn ngữ C thì chẳng cần mấy dòng đó làm gì, tuy nhiên thường khi code các bài lập trình thi đấu thế này ta dùng C++ vì cú pháp ngắn gọn cũng như nhiều hỗ trợ của nó, vậy nên bạn nào code C++ thì nhớ thêm 3 dòng này vào đầu hàm main nhé.

2/ Thường thì mình rất ngán với những bài cần đọc dữ liệu ở dạng chuỗi mà trước đó có đọc số (số nguyên/số thực) rồi sau đó mới đọc chuỗi, vì rất dễ bị rơi vào trường hợp dữ liệu dư thừa bị đưa vào bộ nhớ đệm thì ta phải xóa bộ nhớ đệm đi. Nên giải pháp an toàn mình hay làm đó là đọc hết tất cả dữ liệu theo dạng chuỗi rồi sau đó dữ liệu nào là dạng số thì sẽ convert qua. Như ví dụ bài này dòng đầu tiên là giá trị k thì mình đọc theo dạng chuỗi sau đó mình dùng hàm stoi được hỗ trợ sẵn trong thư viện string của C++ để đổi chuỗi số sang số rồi gán vào biến k. Tất nhiên cái này là mình cẩn thận thôi thực tế các bạn cứ làm bình thường nếu chạy AC là được hihi.

4/ Trong code mình khúc cuối hàm main có dòng: reverse(rever\_res.begin(), rever\_res.end()); thì mình nói luôn sợ bạn nào không biết. Hàm này được hỗ trợ trong thư viện giải thuật (algorithm) nhé các bạn, do là mình dùng thư viện bits/stdc++.h là nó đã bao chứa hết các thư viện nên mình không phải khai báo riêng thư viện giải thuật này. Bạn nào mà không dùng thư viện này thì phải khai báo thư viện giải thuật thì mới dùng hàm này được nhé. Thì hàm này ta sẽ truyền vào 2 con trỏ iterator của cấu trúc dữ liệu (mảng/chuỗi) mà ta chỉ định đảo ngược từ vị trí nào đến vị trí nào. Nên nếu truyền vào begin() và end() thì nghĩa là sẽ đảo ngược toàn bộ từ đầu đến cuối. Hàm không trả về kết quả được đảo ngược mà sẽ đảo ngược trực tiếp trên biến tham số truyền vào thế nên ta mới phải tạo ra một chuỗi string mới là rever\_res lưu giá trị của chuỗi kết quả res sang sau đó đảo lại chuỗi string mới đó rồi thì so sánh kết quả sau khi đảo với kết quả chuỗi res ban đầu và kết luận có đối xứng hay không?

# Ký tự đơn

|  |  |
| --- | --- |
| Có thể là hình ảnh về cuống vé và văn bản | Thì trước tiên mình nói luôn bài này nếu không phân tích kỹ đề bài thì rất có khả năng các bạn sẽ bị làm sai nha mặc dù đề trông rất đơn giản. Nếu bạn nào đọc đề và làm theo hướng dùng mảng đánh dấu 26 phần tử tương ứng với 26 chữ cái từ a đến z để thống kê tần suất xuất hiện của từng chữ cái. Bạn nào mà làm như vậy thì coi chừng sai nhé, dù cho bạn có nộp đúng AC trên trang web này nhưng không có nghĩa là bạn đúng, vì các bộ test trên web này chỉ là do admin web tự làm ra chứ nó không phải bộ test chính thức của sở giáo dục. |

Tại sao lại sai nếu làm theo hướng như vậy? Thì các bạn đọc kỹ đề bài nha, đề bài chỉ nói là xâu s sẽ không có ký tự khoảng trắng, còn lại không hề nói là chỉ toàn các ký tự chữ cái từ a đến z nha. Vậy nên nó có thể có bất kỳ ký tự gì miễn khác ký tự khoảng trắng ví dụ có thể sẽ có các ký tự in hoa như A, B, C hay những ký tự lạ như @, !, $, %, &, \* … vì thế nếu bạn nào nhìn vào test ví dụ đề đưa ra mà mặc định chuỗi chỉ toàn các ký tự chữ cái thì coi như xong rồi nha :)))

Rồi thì với đề bài này, với giới hạn độ dài của chuỗi chỉ tối đa 255 ký tự thì có thể thấy rằng dù các bạn dùng thuật toán tệ đến cỡ nào đi nữa mình nghĩ cũng khó lòng mà TLE được, trừ khi phải tệ lắm lắm hihi. Vậy nên có thể nói lúc này chỉ là tư tưởng làm sao để giải quyết ra đáp án đúng thôi.

Thì một tư tưởng đơn giản nhất đó là vét cạn, với mỗi ký tự s[i] ta so sánh nó với toàn bộ các ký tự trong chuỗi (kể cả với chính nó), mỗi lần trùng với ký tự nào thì tăng biến đếm lên, và sau cùng nếu đếm là 1 tức là nó chỉ trùng với chính nó thôi, ngoài ra không trùng với thằng nào khác thì nó được tính vào đáp án. Ta sẽ đếm xem có bao nhiêu ký tự thỏa mãn như vậy thì đó chính là đáp án bài toán.

**Lưu ý:**

Bước nhập chuỗi, do đề đã nói chuỗi không có khoảng trắng vì thế ta có thể dùng cin >> để nhập. Còn nếu đề không nói ràng buộc chuỗi không có khoảng trắng, tức là chuỗi có thể có khoảng trắng thì các bạn không được dùng cin >> để nhập nhé vì dùng nó là nó sẽ dừng lại khi đến cuối chuỗi hoặc khi gặp khoảng trắng nên khả năng là chuỗi sẽ đọc chưa hết đó, lúc này phải dùng **getline(cin, s);**

Thì một tư tưởng đơn giản nhất đó là vét cạn, với mỗi ký tự s[i] ta so sánh nó với toàn bộ các ký tự trong chuỗi (kể cả với chính nó), mỗi lần trùng với ký tự nào thì tăng biến đếm lên, và sau cùng nếu đếm là 1 tức là nó chỉ trùng với chính nó thôi, ngoài ra không trùng với thằng nào khác thì nó được tính vào đáp án. Ta sẽ đếm xem có bao nhiêu ký tự thỏa mãn như vậy thì đó chính là đáp án bài toán.

#include<bits/stdc++.h>

//khai bao bien

string s;

int main(){

//nhap chuoi ky tu tu ban phim

getline(cin,s);

//in ra chuoi vua nha

cout<<s<<endl;

//dem co bao nhieu ky tu chi xuat hien 1 lan duy nhat

int n\_word=0;

for(int i=0; i<s.length(); i++){

int count=0;

for(int j=0; j<s.length(); j++){

if(s[i]==s[j]){

count++;

}

}

if(count==1){

n\_word++;

}

}

//in ra so ky tu chi xuat hien 1 lan trong chuoi tren

cout<<n\_word<<endl;

return 0;

}

**Đánh giá độ phức tạp của cách làm này:**

+ Độ phức tạp không gian (Space Complexity): O(N) với N là độ dài chuỗi s.

+ Độ phức tạp thời gian (Time Complexity): O(N) của bước nhập chuỗi + O(N^2) của bước xử lý chính. Với N là độ dài của chuỗi s. Phép ^ ở đây là viết tắt ý nghĩa là lũy thừa, cụ thể là N lũy thừa 2, chứ không phải là phép toán xor nha các bạn. Theo quy tắc cộng BigO thì ta sẽ lấy theo thành phần lớn nhất nên tổng quát ta có thể kết luận là O(N^2).

Thì với độ phức tạp O(N^2) như vậy, ta thấy đề cho N tối đa 255, như thế 255^2 = 65,025. Trong khi đó với giới hạn thời gian chạy 1 giây thì ngưỡng quy định là (3 đến 5)\*10^7 tức là 30 triệu đến 50 triệu hay cao lắm là 10^8 tức là 100 triệu. Nên với 65,025 thì nó quá nhỏ và ta yên tâm dù là cách tệ nhất này thì cũng không thể nào bị TLE được.

Thực ra đến đây là xong rồi đó các bạn. Nếu giả sử đang trong phòng thi mà gặp đúng bài này thì là mình thì mình cũng làm đúng cách ở trên thôi, vì giới hạn quá nhỏ nên chủ yếu ra đáp án đúng là được chứ cũng chẳng cần chạy nhanh làm gì vì gì rồi cũng AC thôi, chọn cách đơn giản nhất nhẹ đầu và nhanh gọn để code ra, dành thời gian làm bài khác hehe.

Tuy nhiên giờ trong ngữ cảnh bài viết này, mình thử nâng cấp độ khó thêm nhé, giờ giả sử độ dài chuỗi trong bài này không chỉ tối đa là 255 ký tự, mà tối đa lên đến 10^6 chẳng hạn. Nếu như thế thì cách làm ở trên phá sản, bởi vì (10^6)^2 = 10^12 thì nó đã vượt quá xa ngưỡng giới hạn (3 đến 5)\*10^7 hay cao lắm là 10^8 rồi, nên cách làm ở trên chắc chắn sẽ bị TLE. Vậy cần phải tìm một cách làm khác tốt hơn.

#include <bits/stdc++.h>

// khai bao bien

string s;

int main()

{

// nhap chuoi ky tu tu ban phim

getline(cin, s);

// in ra chuoi vua nhap

// cout<<s<<endl;

// sap xep chuoi theo thu tu

sort(s.begin(), s.end());

// in ra chuoi sau khi sort

cout << s << endl;

// kiem tra cac ky tu canh nhau co == nhau

int dem = 0;

int j = 0;

for (int i = 0; i < s.length();)

{

cout << "(i,j)=" << i << "," << j << endl;

if (s[i] != s[j])

{

dem++;

cout << "dem= " << dem << endl;

i++;

j++;

// kiem tra ben phai

if (s[j] == s[j+1])

{

dem--;

}

else

{

// khong lam gi ca

}

}

else//s[i]==s[j]

{

i = j + 1;

j = i + 1;

}

}

return 0;

}

**Độ phức tạp O(n^2)**

#include <bits/stdc++.h>

// khai bao bien

string s;

int main()

{

// nhap chuoi ky tu tu ban phim

getline(cin, s);

// sap xep chuoi theo thu tu

sort(s.begin(), s.end());

// in ra chuoi sau khi sort

cout << s << endl;

// kiem tra cac ky tu canh nhau co == nhau

int dem = 0;

for(int i = 0; i < s.length(); i++){

bool check = true;

//khong phai la ky tu dau tien

if(i -1 >= 0 && s[i]==s[i-1] ){

check=false;

}

//khong phai la ky tu ket thuc

if(i+1 <= s.length() && s[i]==s[i+1]){

check = false;

}

//kiem tra de tang bien dem

if(check){

dem++;

}

}

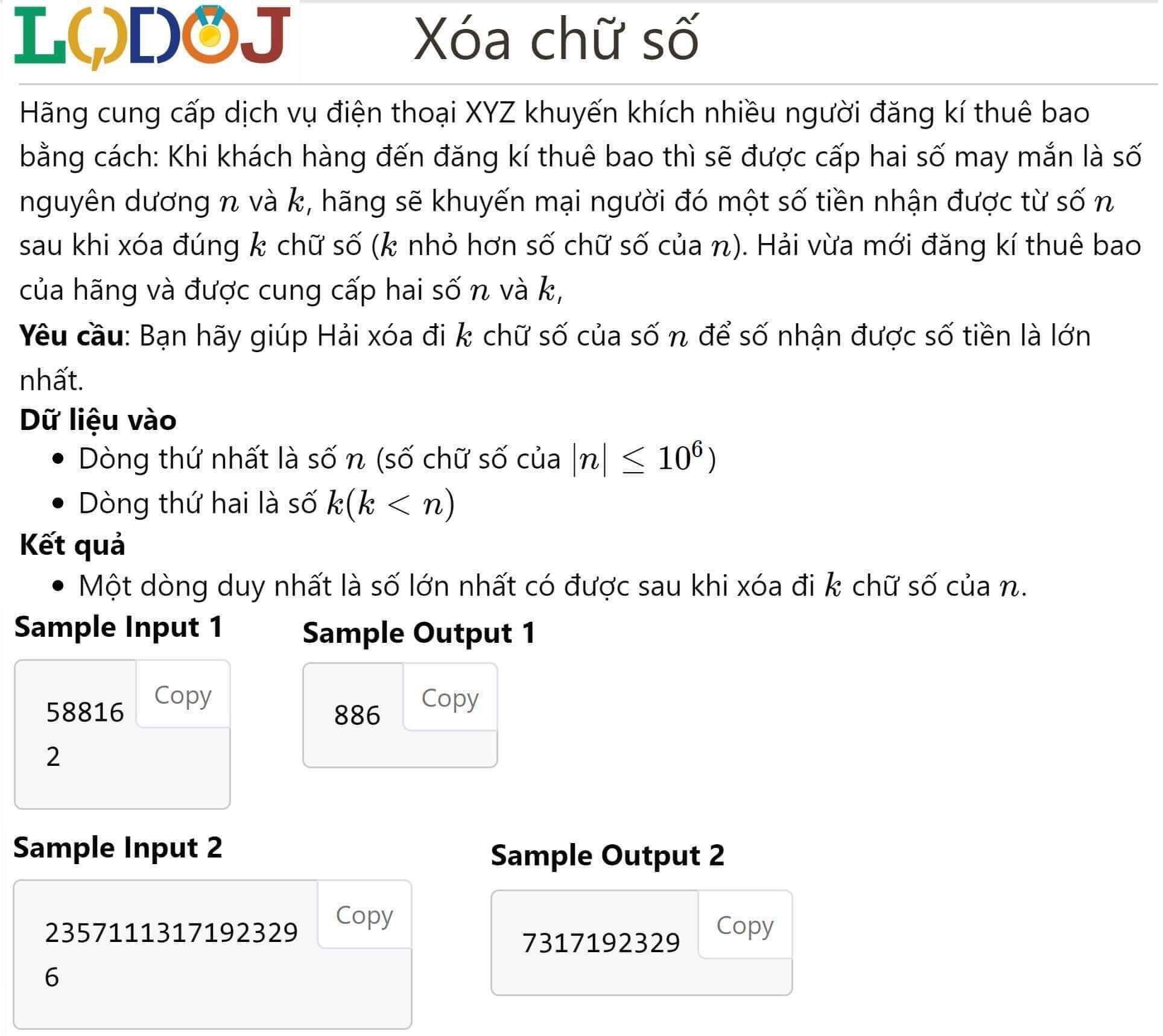
cout << dem;

return 0;

}

**Độ phức tạp O(nlogn)**

# Số lớn nhất



Ý tưởng bài này cho số nguyên N và xoá đi K chữ số để tìm được số lớn nhất.

Sử dụng giải thuật tham lam như sau:

#include<bits/stdc++.h>

//khai bao bien

string number, bignumber;

int k;

//chuong trinh chinh

int main(){

// tang toc do doc du lieu

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

cin.tie(0);

cout.tie(0);

//nhap du lieu: so nguyen va khong co khoang cach

cin>>k;

cin>>number;

int len=number.length();

int L=len-k;

if(L==0){

cout<<number;

}

else{

int start = 0;

while(k>0){

//tim ra so lon nhat [start..finish]

int finish = L-k;

char max = number[start];

int index = start;

//cout << "L= " << L << " k = "<< k << " start = " << start << " finish = " << finish+start << endl;

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

if(max < number[i]){

max = number[i];

index=i;

}

}

start=index+1;

//cout << "index = " << index << endl;

//cout << "max = " << max -48 << endl;

//luu so lon nhat vao chuoi password

bignumber+=max;

L=len-index-1;//L bot di so luong ky tu

k-=index;

}

//khi k>=L thi in ra phan con lai

//cout << num\_string.substr(start,k);

//luu phan con lai vao mang password

bignumber+=number.substr(start,k);

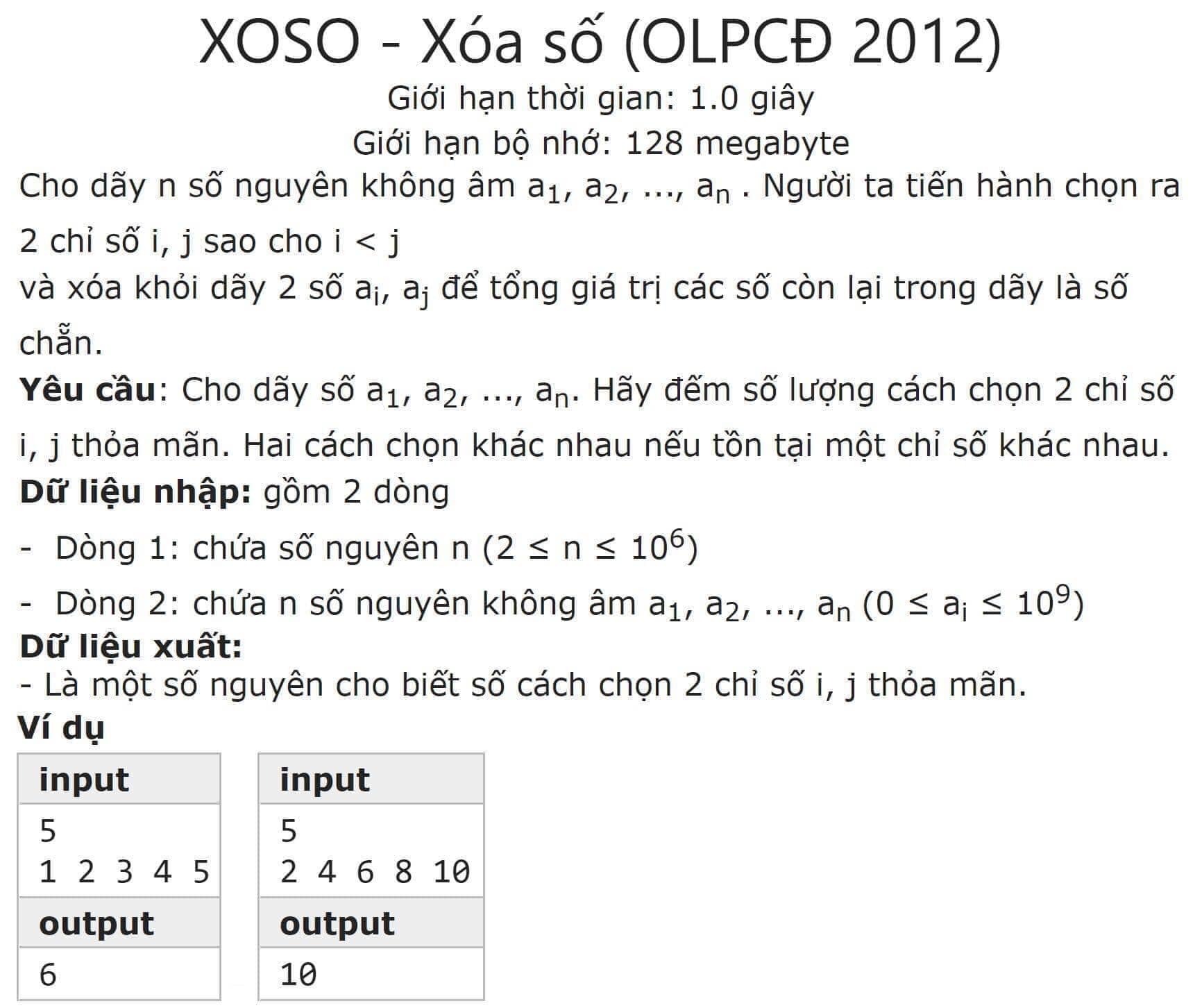
cout << bignumber << endl;

}

return 0;

}

# Xoá số trong chuỗi



#include<bits/stdc++.h>

//chuong trinh chinh

int main(){

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

cin.tie(0);

cout.tie(0);

//readdata

int k;

cin >> k;

int s[k];

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

cin>>s[i];

}

//dem so luong cac ky tu so chan va so le

int c\_chan=0;

int c\_le=0;

long long tong=0;

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

if(s[i]%2==0){

c\_chan++;

}

else{

c\_le++;

}

tong+=s[i];

}

cout<<"so pt chan = " <<c\_chan<<", so pt le = "<<c\_le<<", tong = "<<tong<<endl;

//processing.....

if(c\_le == k && k%2!=0){

//neu toan bo day so la so le cac chu so le: 5 (1 3 5 7 9)

cout<<0;

}

else{

if(c\_chan==k && k%2!=0){

//neu toan bo day so la so le cac chu so chan: 5 (2 4 6 8 10)

cout<<k\*(k-1)/2;

}

else{

if(c\_le==k && k%2==0){

//neu toan bo day so la so chan cac chu so le: 4 (1 3 5 7)

cout<<k\*(k-1)/2;

}

else{

if(c\_chan==k && k%2==0){

//neu toan bo day so la so chan cac chu so chan: 4 (2 4 6 8)

cout<<k\*(k-1)/2;

}

else{

if(c\_le%2!=0 && c\_chan%2==0){

//day so co le so le va chan so chan: 5 (1 2 3 4 5)

cout<<c\_chan\*c\_le;

}

else{

if(c\_le%2!=0 && c\_chan%2!=0){

//day so co le so le va le so chan: 6 (1 2 3 4 5 6)

cout<<c\_chan\*c\_le;

}

else{

if(c\_le%2==0 && c\_chan%2!=0){

//day so co chan so le va le so chan: 5 (1 2 3 4 6)

cout<<c\_chan+c\_le;

}

else{

if(c\_le%2==0 && c\_chan%2==0){

//day so co chan so le va chan so chan: 8 (1 2 3 4 5 6 7 8)

cout<<c\_chan\*(c\_chan-1)/2 + c\_le\*(c\_le-1)/2;

}

}

}

}

}

}

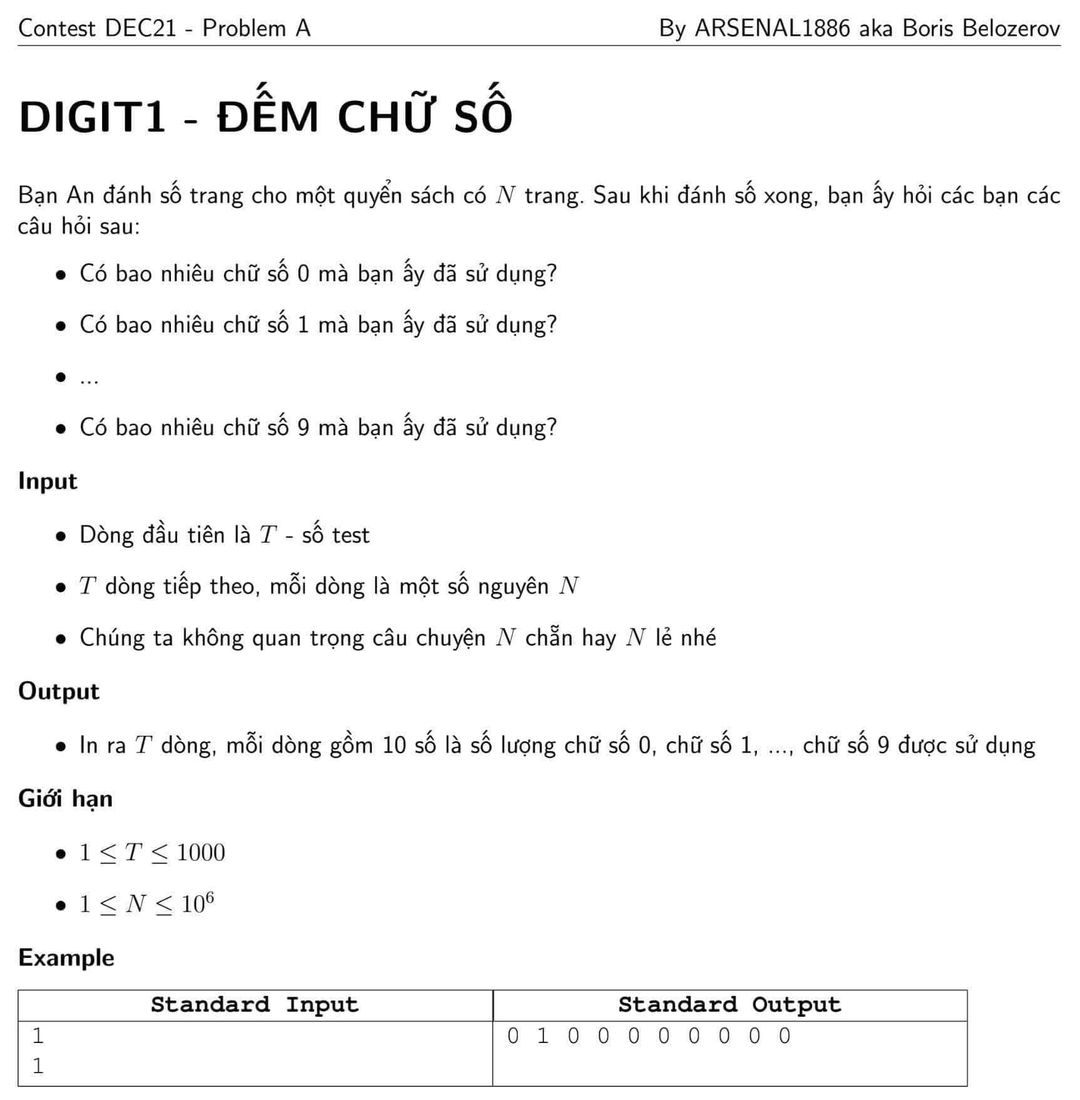
}

}

return 0;

}

# Đếm số



**Cách làm:**

Với chỉ 1 bộ test, ví dụ 1 10

1

10

1 2 1 1 1 1 1 1 1 1

nghĩa là: 0 (1), 1(2), 2(1), 3(1), 4(1), 5(1), 6(1), 7(1), 8(1), 9(1)

#include <bits/stdc++.h>

// khai bao bien

int test;

int page;

int digit[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

// 0 1 2 3 4 5 6 7 8 9

#include <bits/stdc++.h>

// khai bao bien

int test;

int page;

int digit[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

// 0 1 2 3 4 5 6 7 8 9

int main()

{

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

cin.tie(0);

cout.tie(0);

// nhap du lieu

cin >> test;

cin >> page;

// kiem tra cac chu so

for (int i = 1; i <= page; ++i)

{

int x = i;

while (x != 0)

{

int du=x%10;

digit[du]++;

x /= 10;

}

}

// in ra mang digit

for (int i = 0; i < 10; i++)

{

cout << digit[i] << " ";

}

return 0;

}

Đề bài yêu cầu là nhiều bộ test nên ta có thể mở rộng bài toán trên ra thành nhiều mảng một chiều.

|  |  |
| --- | --- |
| bộ test 1 | [0,1,2,3,4,5,6,7,8,9] |
| bộ test 2 | [0,1,2,3,4,5,6,7,8,9] |
| bộ test 3 | [0,1,2,3,4,5,6,7,8,9] |
| ... | ... |
| bộ test n | [0,1,2,3,4,5,6,7,8,9] |

#include <bits/stdc++.h>

// khai bao bien

int test;

int page[1000];

int digit[100000][10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; // 0 1 2 3 4 5 6 7 8 9

int main()

{

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

cin.tie(0);

cout.tie(0);

// nhap du lieu

cin >> test;

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

cin >> page[i];

}

for (int j = 0; j < test; j++)

{

// kiem tra cac chu so

for (int i = 1; i <= page[j]; ++i)

{

int x = i;

while (x != 0)

{

int du = x % 10;

digit[j][du]++;

x /= 10;

}

}

}

// in ra mang digit

for (int i = 0; i < test; i++)

{

for (int j = 0; j < 10; j++)

{

cout << digit[i][j] << " ";

}

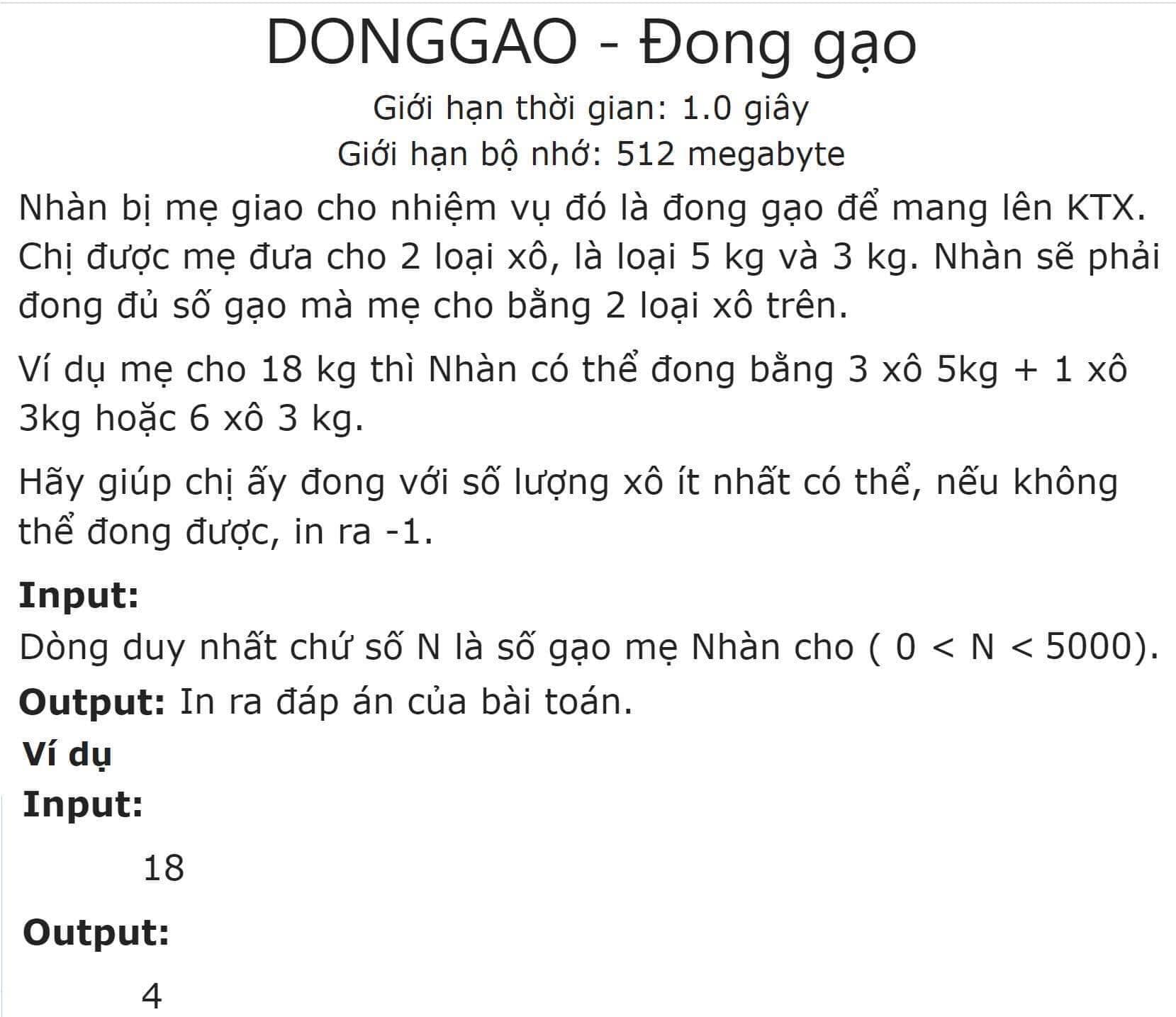
cout<<endl;

}

return 0;

}

# Đong gạo



N là số gạo cần lấy, ta có 2 loại xô là xô 3 kg và xô 5 kg. Thì nguyên tắc mỗi lần chọn xô nào để múc là phải múc đầy xô đó chứ không có chuyện múc thiếu nha các bạn, hỏi xem phải múc với số lần ít nhất là bao nhiêu lần để múc đủ hết N số gạo cần lấy ? In ra đáp án số lần múc ít nhất đó, còn nếu không thể múc được đủ N số gạo với quy tắc múc ở trên thì in ra -1

Ví dụ:

1/ Nếu N = 10 thì ta có đáp án là 2 => 5 + 5 = 10

2/ Nếu N = 11 thì ta có đáp án là 3 => 5 + 3 + 3 = 11

3/ Nếu N = 12 thì ta có đáp án là 4 => 3 + 3 + 3 + 3 = 12

4/ Nếu N = 7 thì đáp án là -1

Bài này cũng nhiều dạng tương tự nên mình sẽ nói theo cách tổng quát như sau: Có 2 số A, B với nguyên tắc A < B (mình cố tình chọn A, B để các bạn hiểu A đứng trước, B đứng sau trong bảng chữ cái, để hình tượng cho dễ là A < B nhé). Như trong bài này thì A = 3, B = 5. Thì hãy xem cần tính tổng bao nhiêu số A cộng nhau cũng như bao nhiêu số B cộng nhau (không nhất thiết phải có đủ cả 2 giá trị A B, có thể chỉ có 1 trong 2) sao cho được tổng bằng N ? Và hãy đưa ra đáp án sao cho ít số cộng lại với nhau nhất. Như ví dụ khi N = 18 thì ta có thể là: 5 + 5 + 5 + 3 = 18 (4 lần cộng) nhưng cũng có thể là 3 + 3 + 3 + 3 + 3 + 3 = 18 (6 lần cộng) và đáp án sẽ tính theo số lần cộng ít nhất tức là 4 lần. Còn nếu không có cách nào để cộng lại với nhau ra được tổng bằng N thì ta sẽ in -1.

Thì bài này ta có thể áp dụng phương pháp tham lam (greedy) để giải quyết. Kể cả bạn nào có không biết về phương pháp tham lam này thì thực ra đôi khi các bạn đang giải quyết vấn đề theo chính tư tưởng của tham lam mà các bạn không biết. Cho bạn nào không biết thì mình có thể nói nhanh như thế này: Tham lam ở đây nghĩa là ta sẽ cố gắng chọn ra cách giải quyết tốt nhất trong tất cả các cách ở hiện tại rồi đi theo cách giải quyết tốt nhất đó nếu lại đứng trước nhiều lựa chọn cách giải quyết khác nhau nữa thì ta lại cố gắng chọn theo cách giải quyết tốt nhất trong số đó tóm lại cứ tốt nhất mà triển vì thế tên gọi nó là tham lam. Quy luật nó là như vậy thôi chứ còn lại không có công thức từng bước cụ thể giải quyết như mấy thuật toán khác. Mà ta có thể giải quyết theo cách bất kỳ miễn cái tư tưởng mà ta giải quyết luôn là chọn theo hướng tốt nhất trong các lựa chọn thì đó chính là tư tưởng của tham lam. Thì tại sao bài này lại có thể nhìn ra tư tưởng của nó là tham lam? Đọc đề bài là thấy rõ mùi rồi hehe muốn xử lý theo quy luật gì gì đó để sau cùng đáp án phải là nhỏ nhất/lớn nhất các kiểu, đọc kiểu này phát thấy mùi tham lam ngay Thì thực ra tham lam chỉ là dạng tư tưởng thôi, còn lại để hiện thực được nó thì có thể cần đến những kỹ thuật bổ trợ khác nữa. Mình lấy ví dụ ngay trong chính bài tập này, các bạn cứ tiếp tục xem nhé.

Thì ta có thể nhìn ra một số vấn đề như sau: Ví dụ nếu N mà chia hết cho B (là số lớn nhất trong 2 số) thì chắc chắn đáp án sẽ là N / B đúng không các bạn? Ví dụ A = 3, B = 5 và N = 15 thì chắc chắn đáp án sẽ là 15 / 5 = 3 tức là 5 + 5 + 5 = 15 đúng không các bạn? Mình tin tư duy này nhiều bạn sẽ nhìn ra ngay vì nó quá rõ, vì nếu có thể chia hết cho số lớn nhất thì kết quả sau khi chia sẽ là kết quả nhỏ nhất (tỷ lệ nghịch).

Vậy nếu N không chia hết cho B (số lớn nhất) thì ta phải xử lý thế nào? Lúc này đây tư tưởng tham lam hiểu là ta sẽ tính X = N / B là kết quả phần nguyên của phép chia, và Y = N % B là kết quả phần dư của phép chia. Trong trường hợp hiện tại nghĩa là Y sẽ khác 0 (vì không chia hết). Thì lúc này đây ta kiểm tra xem phần dư Y này có chia hết cho A luôn không? Nếu có thì đáp án sẽ là X + Y / A. Ví dụ: A = 3, B = 5, N = 13 thì ta tính ra được X = 13 / 5 = 2, Y = 13 % 5 = 3, và ta thấy Y % A = 0 (Y chia hết cho A) => đáp án sẽ là: X + Y / A = 2 + 3 / 3 = 2 + 1 = 3.

Còn nếu trong tình huống phần dư Y không chia hết cho A, ví dụ khi A = 3, B = 5, N = 16 thì X = 16 / 5 = 3, Y = 16 % 5 = 1 thì 1 không chia hết cho 3. Lúc này đây ta sẽ giảm bớt 1 giá trị của X điều đó nghĩa là Y sẽ được cộng với B, sau đó kiểm tra nếu Y chia hết cho A thì kết quả sẽ là X + Y / A. Nếu Y vẫn không chia hết cho A thì lại tiếp tục giảm bớt 1 giá trị của X và tiếp tục Y được cộng thêm với giá trị B rồi lại kiểm tra xem Y có chia hết cho A không? Cứ thế lặp đến khi nào Y chia hết A hoặc khi nào X đã về 0 thì không thể giảm được nữa, nếu vẫn không tìm ra trường hợp nào Y chia hết cho A thì kết luận -1. Như ví dụ trên thì giảm X còn 2, Y được cộng thêm 5 thành 6, thì 6 chia hết cho 3 => đáp án: 2 + 6/3 = 2 + 2 = 4.

Còn nếu A = 3, B = 5, N = 7 thì ta thấy X = 7 / 5 = 1, Y = 7 % 5 = 2, thấy Y không chia hết cho A, giảm X xuống còn 0, Y += 5 => Y = 7, 7 không chia hết cho 3, lúc này không giảm X được nữa vì X đã là 0 => kết quả là -1

Vậy tổng quát lại có thể phát biểu ý tưởng giải quyết bài này như sau:

Hàm xử lý nhận đầu vào 3 tham số kiểu số nguyên là N, A, B (ràng buộc A < B ). Hàm trả về kiểu số nguyên là kết quả. Quy trình xử lý trong hàm như sau:

Vòng lặp for khởi tạo X = N / B, Y = N % B. Điều kiện lặp X >= 0 và bước lặp X--; Bên trong vòng lặp xét nếu Y chia hết cho A thì return X + Y / A. Nếu không thì Y += B. Trước khi kết thúc hàm thì return -1; tức là nếu nó còn chạy được xuống đó mà không kết thúc trước đó (kết thúc trong vòng lặp) nghĩa là không có giá trị Y thỏa mãn chia hết cho A thì kết quả sẽ là -1.

#include <iostream>

using namespace std;

int solve(int n, int a, int b){

for(int x = n / b, y = n % b; x >= 0; --x){

if(y % a == 0){

return x + y / a;

}

y += b;

}

return -1;

}

int main(){

int n;

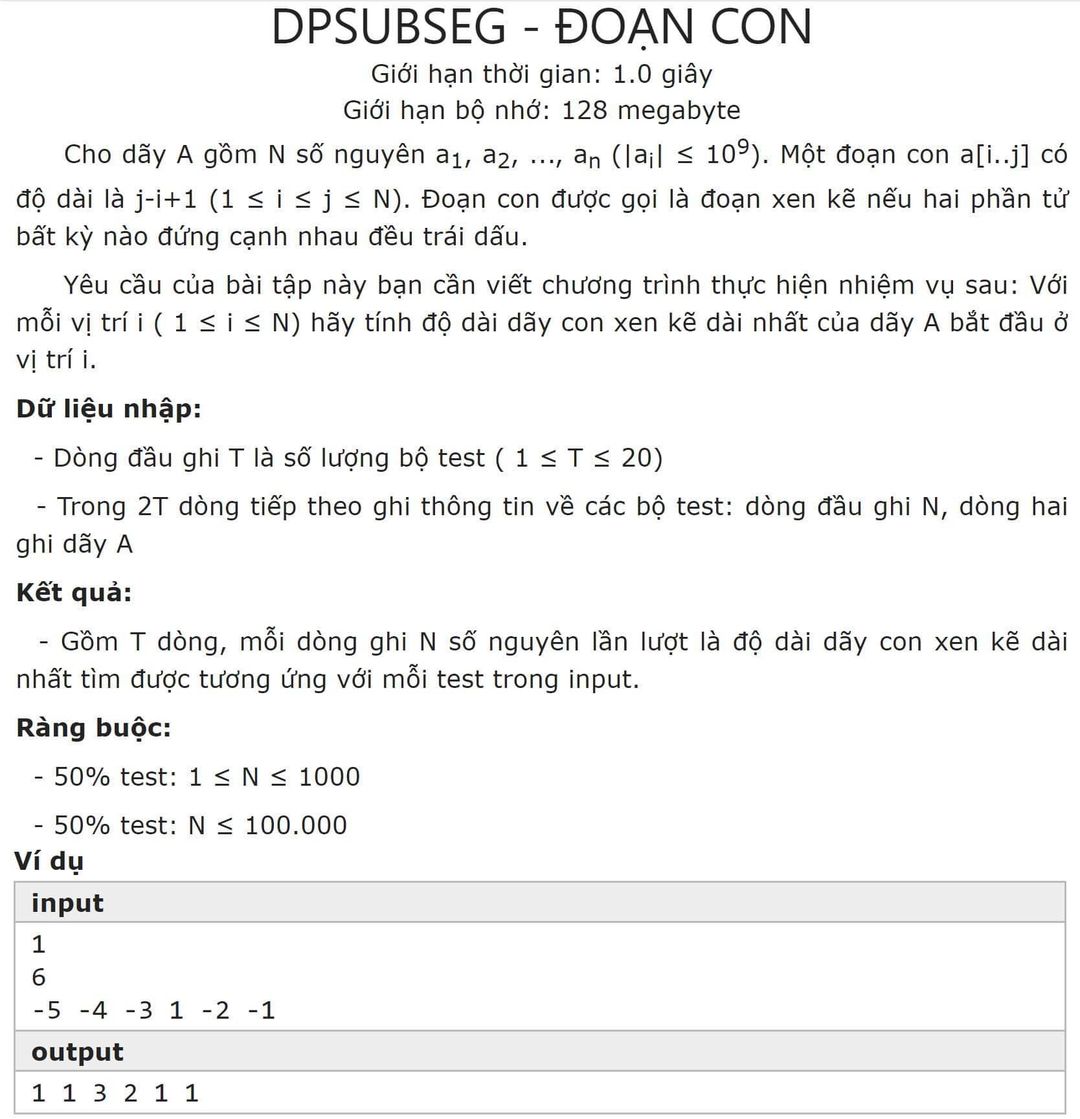
cin >> n;

cout << solve(n, 3, 5);

return 0;

}

# Đoạn con



Nếu theo mô tả của đề bài thì một cách xử lý hết sức đơn giản đó là cứ xét lần lượt từng vị trí i của mảng rồi xét từ vị trí đó trở lên đếm xem có bao nhiêu phần tử trái dấu, chỉ dừng lại khi gặp phần tử cùng dấu rồi thì in kết quả đếm đó ra và lặp lại quy trình trên khi xét qua vị trí i + 1 tiếp theo và cứ thế xét hết mảng.

#include <bits/stdc++.h>

// khai bao bien

int t, n;

int m[10000];

int main()

{

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

cin.tie(0);

cout.tie(0);

// read data

cin >> t;

for (int k = 0; k < t; k++)

{

cin >> n;

//nhap day con

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

{

cin >> m[i];

}

// tim day con

cout<<endl;

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

{

int cnt = 1;

for (int j = i + 1; j < n; j++)

{

if ((m[j] >= 0 && m[j - 1] < 0) || (m[j] < 0 && m[j - 1] >= 0))

{

cnt++;

}

else

{

break;

}

}

cout << cnt << " ";

}

}

return 0;

}

**Giải thích 1 số chỗ trong cách làm ở trên:**

1/ 3 dòng đầu tiên của hàm main ( ios::sync\_with\_stdio(0); cin.tie(0); cout.tie(0); ) sẽ giúp đẩy nhanh tốc độ nhập xuất dữ liệu nếu các bạn đang code theo ngôn ngữ C++ (để hiểu vì sao thì các bạn cứ gõ mấy từ này lên Google là sẽ ra rất nhiều bài viết giải thích nhé), nên kinh nghiệm trong các bài mà cần nhập xuất nhiều ví dụ như bài này ta thấy có tối đa 20 bộ test, mỗi bộ test có tối đa 10^5 số cần nhập, như thế cả thảy sẽ có thể nhập lên đến 20 \* 10^5 = 2 \* 10^6 tức là 2 triệu số là một con số lớn thì ta hãy nên để 3 dòng này vào để chạy nhanh hơn. Tất nhiên nếu các bạn code theo cú pháp của C thì chẳng cần mấy dòng đó làm gì, tuy nhiên thường khi code các bài lập trình thi đấu thế này ta dùng C++ vì cú pháp ngắn gọn cũng như nhiều hỗ trợ của nó, vậy nên bạn nào code C++ thì nhớ thêm 3 dòng này vào đầu hàm main nhé. Như bài này mình thử nếu với cùng 1 đoạn code xử lý thuật toán tối ưu nếu để thêm 3 dòng đó vào thì chạy AC với thời gian < 200 ms nhưng nếu bỏ đi 3 dòng đó thì vẫn AC nhưng thời gian > 700 ms. Để các bạn thấy 3 dòng này giúp tối ưu chạy nhanh hơn thế nào.

2/ Vì cần xét cặp 2 phần tử kế cận nhau nên cách làm ở trên khi vòng lặp for j xét từ i + 1 thì bản chất ta sẽ so sánh 2 phần tử là a[j] và a[j - 1]. Ví dụ nếu mảng có 10 phần tử với index từ 0 đến 9 thì như thế khi i = 0 thì j bắt đầu từ 1, so sánh a[j] và a[j - 1] lúc này là so sánh a[1] và a[0] sau đó j++; tăng lên 2 thì so sánh a[j] và a[j - 1] lúc này là so sánh a[2] và a[1] và cứ thế so sánh a[3] và a[2], a[4] và a[3] …

Đánh giá độ phức tạp của cách làm này:

+ Độ phức tạp không gian (Space Complexity): O(N) lưu mảng a.

+ Độ phức tạp thời gian (Time Complexity): Vòng lặp chạy T lần để xét đủ T bộ test, với mỗi bộ test chạy N lần để đọc dữ liệu vào mảng, sau đó bước xử lý 2 vòng lặp lồng nhau nếu trong tình huống toàn bộ các cặp phần tử đều trái dấu nhau thì bước xử lý này sẽ chạy đủ (N\*(N-1))/2 lần. Vậy số lần lặp của toàn bài theo cách làm này trong trường hợp xấu nhất sẽ là T \* (N + (N\*(N-1))/2). Về mặt lý thuyết độ phức tạp thì (N\*(N-1))/2 ⇔ (N^2 - N)/2 ⇔ (N^2 - N) \* 0.5 thì theo quy tắc nhân BigO khi nhân với hằng số ta có thể bỏ qua vì thế chỉ còn N^2 - N ⇔ N^2 + (-N) thì theo quy tắc cộng BigO ta sẽ lấy theo thành phần lớn nhất nên chỉ lấy theo N^2, cũng như + N của bước đọc input thì theo quy tắc cộng BigO như đã nói ta có thể bỏ qua. Vậy tổng quát lý thuyết độ phức tạp có thể kết luận là O(T \* N^2) (N^2 tức là N lũy thừa 2 nha các bạn, phép ^ ở đây mình viết tắt của ký hiệu lũy thừa chứ không phải là phép xor bit đâu nha).

Thì với độ phức tạp ở trên kể cả nếu tính chính xác số lần lặp theo T \* (N + (N\*(N-1))/2) thì thử tính xem trong trường hợp xấu nhất tức là mọi dữ liệu đều lớn nhất, T = 20 và N = 10^5 thì như thế số lần lặp cả thảy là: 100,001,000,000 (100 tỷ + 1 triệu lần lặp) và với số lần lặp thế này đã vượt quá xa ngưỡng giới hạn thời gian chạy 1 giây chỉ được phép ở mức (3 đến 5) \* 10^7 tức là 30 triệu đến 50 triệu hay cao lắm là 10^8 tức là 100 triệu thì số lần lặp này đã gấp hơn cả 1000 lần giới hạn được phép vì thế chắc chắn sẽ bị TLE với cách làm này. Ta cần phải tìm ra một cách tốt hơn.

Thì cách làm ở trên có chỗ không tốt ta dễ dàng nhìn ra được, các bạn hãy thử giả sử khi xét tại vị trí i = 3 và khi xét đến vị trí i = 8 và thấy tại vị trí 8 này là vị trí cuối cùng thỏa mãn (do a[8] và a[9] cùng dấu) => từ vị trí 3 đến vị trí 8 sẽ là các phần tử thỏa mãn 2 phần tử đứng cạnh nhau trái dấu nhau, vậy lúc này tại vị trí 3 ta sẽ in ra 6 (vì từ 3 đến 8 có 6 phần tử, công thức: 8 - 3 + 1 = 6). Rồi thì các bạn xem nhé vậy chắc chắn tại các vị trí 4, 5, 6, 7, 8 sẽ lần lượt in ra: 5, 4, 3, 2, 1 đúng không các bạn? Bởi ta đã biết từ vị trí 3 đến vị trí 8 là các phần tử thỏa mãn nên từ 3 đến 8 có 6 phần tử thì từ 4 đến 8 sẽ có 5 phần tử, từ 5 đến 8 sẽ có 4 phần tử, từ 6 đến 8 sẽ có 3 phần tử và cứ thế … trong khi đó với cách làm ở trên thì sau khi xét xong vị trí i = 3 in ra được kết quả 6 rồi thì xét tiếp lên vị trí i + 1 tức là vị trí 4 rồi thì vẫn tiếp tục lặp lại quy trình xét như cũ lại xét so sánh 4 với 5, 5 với 6, 6 với 7 … mà không hề tận dụng gì lại kết quả từ lần xét vị trí 3 trước đó, đó là cái dở ở cách làm trên.

Vậy nếu đã nhìn ra cái dở của cách làm trên rồi thì ta sẽ xử lý làm sao để tối ưu lại? Đó là tại vị trí i ta lưu start = i rồi ta xét tiếp lên các vị trí i ở trên, tìm ra vị trí i mà tại đó không thỏa mãn (khi so với i + 1), rồi thì ta sẽ in ra các kết quả từ i - start + 1 về 1, như ví dụ trên start = 3 và gặp i = 8 không thỏa mãn, thì lúc này ta sẽ in ra các kết quả i - start + 1 về 1 tức là in từ 8 - 3 + 1 = 6 về 1 là độ dài các đoạn con từ vị trí start, start + 1, start + 2, … , i rồi thì start sẽ được cập nhật lại bằng i + 1 là vị trí tiếp tục xét sau vị trí i không thỏa mãn (tức là lúc này start = 8 + 1 = 9) và cứ thế xét tiếp i++ đến cuối.

Tuy nhiên cách làm trên phải cẩn thận đến tình huống nếu giả sử không tìm ra phần tử tại vị trí i nào mà tại đó trái dấu với i + 1 thì sao? Lúc này sẽ chẳng thể có kết luận in ra được vì cách làm trên chỉ kết luận khi đã tìm ra vị trí phần tử không thỏa mãn. Vậy nên từ đầu ta nên tạo mảng có N + 1 phần tử và tại vị trí index N ta sẽ lấy ngay chính phần tử a[N - 1] gán sang và như thế khi xét vị trí i cuối cùng là N - 1 thì so sánh với a[N] sẽ cùng dấu và khi đó ta có thể kết luận được. Ví dụ mảng 10 phần tử tức là index hợp lệ từ 0 đến 9, giả sử khi xét vị trí start = 3, cho rằng mảng này tất cả các phần tử đều hợp lệ như thế khi xét đến a[9] sẽ cùng dấu với a[10] nên vị trí 9 không thỏa, thì lúc này 9 - 3 + 1 = 7 và ta sẽ in từ 7 về 1 ứng với các vị trí start = 3 đến i = 9 rồi thì lúc này vòng lặp for cũng kết thúc vì i++ tiếp tục sẽ là 10 thì không thỏa < N nữa.

**Cải tiến**

#include <bits/stdc++.h>

using namespace std;

int main() {

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

cin.tie(0);

cout.tie(0);

int t;

cin >> t;

while(t--){

int n;

cin >> n;

int a[n + 1];

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

cin >> a[i];

}

a[n] = a[n - 1];

int start = 0;

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

if((a[i] < 0 && a[i + 1] < 0) || (a[i] >= 0 && a[i + 1] >= 0)){

for(int j = i - start + 1; j >= 1; --j){

cout << j << " ";

}

start = i + 1;

}

}

cout << endl;

}

return 0;

}

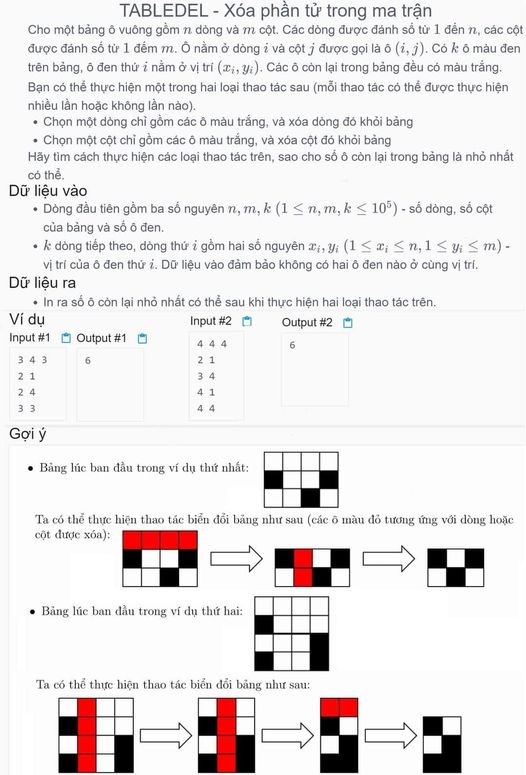
P/s: Có thể có bạn sẽ suy nghĩ vì sao xét điều kiện a[i] và a[i + 1] có cùng dấu không thì đón giản cứ nhân tụi nó lại nếu là số dương thì chắc chắn 2 thằng cùng dấu, tức là if(a[i] \* a[i + 1] >= 0). Nhưng mình không làm vậy vì mình sợ có tình huống như kiểu 1 thằng là 0, 1 thằng là số âm thì nhân nhau vẫn sẽ ra 0, như thế vẫn thoả điều kiện cùng dấu trong khi đó thực tế là trái dấu nhau. Nên thôi mình cứ ghi rõ ra như ở trên cho chắc hihi.

Với cách làm này độ phức tạp về không gian vẫn là O(N) do ta sử dụng mảng a. Nhưng độ phức tạp về thời gian được cải thiện rõ ràng ở bước xử lý, lúc này tuy cũng là 2 vòng lặp lồng nhau nhưng các bạn tính kỹ ra sẽ thấy nó chỉ chạy hết cả thảy là 2 \* N lần thôi chứ không phải như bước xử lý trước đó là hết (N \* (N - 1)) / 2 lần. Tại sao lại như vậy? Thì các bạn cứ giả sử mảng có 100 phần tử, giả sử khi xét từ vị trí phần tử thứ 1 và thấy tại phần tử thứ 30 không thỏa mãn, tức là lúc này nó đã lặp đủ 30 lần để xét thấy phần tử 30 không thỏa, rồi thì lúc này nó lại lần ngược lại in ra bấy nhiêu phần tử đã xét qua đó giảm về 1, tức là coi như xét 2 lần quãng đường đã xét từ phần tử 1 đến 30 đầu tiên đi theo chiều xuôi sau đó đi ngược về => tổng cộng 2 lần. Rồi xét từ phần tử 31 giả sử xét đến phần tử thứ 80 lại thấy không thỏa mãn thì nó lại xét lùi về để in ra giá trị, rồi xét từ phần tử 81 đến 100 lại thấy không thỏa mãn thì lại xét lùi về để in ra giá trị => cả thảy số lần lặp chỉ là 2 \* N thôi.

Vậy theo cách này tổng số lần lặp lúc này sẽ là T \* (N + 2 \* N) ⇔ T \* 3 \* N. Với T = 20, N = 10^5 thì 20 \* 3 \* 10^5 = 6,000,000 (6 triệu) lần lặp thì với số lần lặp này ít hơn ngưỡng giới hạn 1 giây (30 triệu đến 100 triệu) ít hơn cả từ 5 đến 16 lần vì thế yên tâm sẽ không bị TLE.

Ở trên là tính chính xác thôi còn nếu kết luận về mặt tổng quát thì độ phức tạp thời gian sẽ là O(T \* N) các bạn nhé vì nhân với hằng số ta có thể bỏ qua theo quy tắc nhân BigO. Đến đây các bạn đã thấy cái hay của giải thuật chưa? Ta tối ưu độ phức tạp từ O(T \* N^2) xuống còn O(T \* N). Từ việc số lần lặp hơn 100 tỷ rút xuống chỉ còn 6 triệu để thấy tốt hơn thế nào.

# Xoá hàng - cột trong ma trận (mảng đánh dấu)



Thì bài này nếu bạn nào mà đi làm theo đúng cách đề bài mô tả tức là đi tạo ra ma trận có N dòng M cột, ví dụ cho là ma trận có kiểu bool (true/false) rồi các bạn đánh dấu tọa độ những ô đen trên ma trận là false rồi sau đó duyệt ma trận xét dòng nào toàn là true (ô trắng) thì xóa dòng đó đi sau khi xóa hết các dòng thì bắt đầu xét qua các cột cũng làm tương tự như trên cột nào toàn là true (ô trắng) thì xóa cột đó đi rồi sau cùng đếm số ô còn lại là kết quả, đúng như đề bài đã mô tả.

Thì nếu bạn nào mà làm như vậy thì các bạn đã “toang” ngay từ bước khởi đầu rồi, tức là bước tạo ra ma trận có N dòng, M cột. Giả sử cho rằng ma trận kiểu bool tức là mỗi phần tử chỉ chiếm 1 byte, thì việc tạo ma trận N dòng M cột sẽ có số lượng phần tử là N \* M tức là chiếm N \* M byte. Hãy xem giới hạn độ lớn đề bài đưa ra, N và M tối đa có thể lên đến 10^5, như thế N \* M tối đa có thể là 10^10 (10 tỷ) phần tử tức là cần 10 tỷ byte. Thử làm phép tính nhé các bạn, nếu trường hợp xấu nhất này xảy ra thì bộ nhớ sẽ cần:

10,000,000,000 / 1024 = 9,765,625 KB

9,765,625 / 1024 = 9,536.743 MB

trong khi đó giới hạn bộ nhớ đề bài chỉ cho có 256 MB thôi, tức là ta đã gấp cả 37 lần giới hạn bộ nhớ nên chắc chắn cách làm này là không ổn rồi sẽ bị lỗi MLE (Memory Limit Exceeded: Quá giới hạn bộ nhớ cho phép) hoặc thậm chí nếu cho rằng bộ nhớ bài này cho phép đáp ứng được như vậy luôn (mặc dù thực tế không bao giờ được) thì việc duyệt xử lý có thể xem là duyệt hết toàn bộ ma trận nghĩa là duyệt hết 10^10 phần tử thì cũng sẽ bị lỗi TLE (Time Limit Exceeded: Quá giới hạn thời gian cho phép) vì bài này cho giới hạn 1 giây thì tối đa 10^8 thôi, 10^10 là vượt quá cả 100 lần rồi.

Nên tóm lại cách làm ở trên là không ổn nhé ta có thể nhìn thấy rõ ngay từ đầu để khỏi mất công đi cài đặt nha các bạn.

Giờ ta sẽ nói về cách làm đúng đắn cho bài này. Cũng vẫn dựa theo một phần tư tưởng cách thức xử lý như đề bài mô tả nhưng ta không nhất thiết phải đánh dấu từng ô trong ma trận mà ta sẽ đi đánh dấu cả dòng, cả cột xem dòng nào, cột nào có sự xuất hiện của ô đen. Tức là dùng kỹ thuật “mảng đánh dấu”, bạn nào chưa biết thì xem dưới đây sẽ rõ nha:

Hãy nhìn ví dụ đầu tiên, ma trận có 3 dòng x 4 cột. Đầu tiên ta tạo mảng đánh dấu a có 3 + 1 = 4 phần tử (để index từ 0 đến 3, coi như ta bỏ qua không dùng index 0 chỉ dùng từ 1 đến 3 ứng với vị trí thứ tự dòng) và mảng này khởi tạo tất cả là false. Tiếp đó ta tạo mảng đánh dấu b có 4 + 1 = 5 phần tử (để index từ 0 đến 4, coi như ta bỏ qua không dùng index 0 chỉ dùng từ 1 đến 4 ứng với vị trí thứ tự cột) và mảng này khởi tạo tất cả là false.

Rồi thì với K vị trí ô đen (i, j) đọc vào, ta sẽ đánh dấu a[i] = true; b[j] = true; Như ví dụ 1 thì sau quá trình đọc ta sẽ có thống kê như sau:

a[1]: false

a[2]: true

a[3]: true

b[1]: true

b[2]: false

b[3]: true

b[4]: true

Sau khi đã đánh dấu xong K vị trí ô đen, giờ đây ta duyệt lại mảng a (là mảng dòng) để xem có bao nhiêu dòng trắng? Lúc này nếu dòng nào là false tức là nó vẫn chưa bị biến thành đen tức là dòng trắng, thì như ta thấy chỉ có mỗi duy nhất dòng a[1] là false tức là chỉ có 1 dòng trắng duy nhất. Vậy nghĩa là số ô trên ma trận ta sẽ bỏ đi = 1 \* M (tại vì 1 dòng là có M ô trên dòng đó và ta đang bỏ đi 1 dòng nên số ô bỏ đi = 1 \* M). Như trường hợp này sẽ là 1 \* 4 = 4.

Tiếp tục xét qua mảng b để coi có cột nào toàn là màu trắng không? Thì ta thấy có duy nhất cột 2, nghĩa là ta có thể bỏ đi toàn bộ phần tử trên cột 2. Thì theo lẽ số lượng phần tử trên cột sẽ chính là số dòng của ma trận, nhưng ta phải nhớ là trước đó khi xét dòng là ta đã bỏ đi 1 dòng rồi nên số dòng của ma trận lúc này chỉ còn 2 dòng, vì thế số ô bỏ đi trên cột 2 lúc này sẽ là: 1 \* 2 = 2.

Vậy cả thảy ta đã bỏ đi 4 + 2 = 6 ô. Như thế số ô còn lại sẽ là N \* M - 6 tức là 3 \* 4 - 6 = 12 - 6 = 6 ô. Đáp án in ra 6.

Thử với ví dụ số 2 đề đưa ra, ta sẽ có mảng a và b là:

a[1]: false

a[2]: true

a[3]: true

a[4]: true

b[1]: true

b[2]: false

b[3]: false

b[4]: true

Xóa 1 dòng (dòng 1) => số ô bị xóa đi = 1 \* 4 = 4

Xóa 2 cột (cột 2, 3) => số ô bị xóa đi = 2 \* 3 = 6

Vậy tổng số ô bị xóa đi = 4 + 6 = 10

=> Số ô còn lại = 4 \* 4 - 10 = 16 - 10 = 6. Đáp án in ra 6.

Source code để các bạn tham khảo nếu làm theo hướng này:

#include <bits/stdc++.h>

using namespace std;

int main(){

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

cin.tie(0);

cout.tie(0);

long long n, m, k;

cin >> n >> m >> k;

bool a[n + 1] = {false}, b[n + 1] = {false};

while(k--){

int x, y;

cin >> x >> y;

a[x] = true;

b[y] = true;

}

long long temp\_n = n;

for(int i = 1; i <= n; ++i){

if(a[i] == false){

temp\_n--;

}

}

long long cnt = (n - temp\_n) \* m;

for(int i = 1; i <= m; ++i){

if(b[i] == false){

cnt += temp\_n;

}

}

cout << (n \* m) - cnt;

return 0;

}

Giải thích 1 số chỗ trong cách làm ở trên:

1/ 3 dòng đầu tiên của hàm main ( ios::sync\_with\_stdio(0); cin.tie(0); cout.tie(0); ) sẽ giúp đẩy nhanh tốc độ nhập xuất dữ liệu nếu các bạn đang code theo ngôn ngữ C++ (để hiểu vì sao thì các bạn cứ gõ mấy từ này lên Google là sẽ ra rất nhiều bài viết giải thích nhé), nên 1 kinh nghiệm trong các bài mà cần nhập xuất nhiều, như trường hợp bài này ta thấy có tối đa K = 10^5 tức là có thể sẽ có đến 2 \* 10^5 số cần nhập thì đây là một con số lớn thì ta hãy nên để 3 dòng này vào để chạy nhanh hơn. Tất nhiên nếu các bạn code theo cú pháp của C thì chẳng cần mấy dòng đó làm gì, tuy nhiên thường khi code các bài lập trình thi đấu thế này ta dùng C++ vì cú pháp ngắn gọn cũng như nhiều hỗ trợ của nó, vậy nên bạn nào code C++ thì nhớ thêm 3 dòng này vào đầu hàm main nhé.

2/ bool a[n + 1] = {false}; nghĩa là khởi tạo mảng a có n + 1 phần tử và tất cả đều là giá trị false (cũng có thể hiểu là 0, trong lập trình thì 0 ⇔ false, 1 ⇔ true). Nói rõ hơn thì chỗ này nó sẽ chỉ gán giá trị ô đầu tiên (a[0]) là giá trị 0 thôi, nhưng nguyên tắc khi khởi tạo mảng chỉ cần ô đầu tiên có giá trị (giá trị bất kỳ không nhất thiết phải là 0) thì toàn bộ các ô còn lại của mảng sẽ mặc định có giá trị là 0. Vậy nên nếu ngay ô đầu tiên mà cũng là 0 thì toàn bộ các ô trong mảng đều mang giá trị 0 hết.

3/ Biến temp\_n để lưu lại số dòng còn lại của ma trận sau khi đã xóa dòng. Biến cnt để thống kê số lượng các ô đã bị xóa. Sau quá trình xóa dòng thì số ô bị xóa = số dòng bị xóa nhân cho số cột, số dòng bị xóa được tính bằng n - temp\_n tức là số dòng ban đầu trừ cho số dòng còn lại thì sẽ ra số dòng bị xóa. Vậy nên mới có dòng lệnh: long long cnt = (n - temp\_n) \* m; Rồi sau đó duyệt xem cứ thấy cột nào bị xóa thì số ô bị xóa sẽ cộng thêm số dòng còn lại tức là cnt += temp\_n; Sau cùng đáp án sẽ là (n \* m) - cnt; tức là số ô của toàn bộ ma trận trừ cho số ô bị xóa thì sẽ ra số ô còn lại.

4/ Các bạn phải khai báo kiểu dữ liệu cho các biến n, m, temp\_n, cnt tất cả là kiểu long long nhé (kiểu số nguyên 8 byte) nhé để tránh bị tràn số. Ví dụ nếu bạn nào mà khai báo biến n, m kiểu int (kiểu số nguyên 4 byte) thì ở bước tính n \* m - cnt; thì việc tính n \* m nó tạo ra vùng nhớ tạm để chứa kết quả và vùng nhớ tạm đó sẽ căn cứ theo kiểu dữ liệu của các biến thành phần nên nếu n, m kiểu int thì vùng nhớ tạm cũng kiểu int, mà nếu thế thì int làm sao có thể chứa được 10^10 (nếu khi n = m = 10^5 thì n \* m = 10^10) từ đó bị tràn số và sẽ sai nên tốt nhất cứ để hết kiểu long long cho chắc nha các bạn.

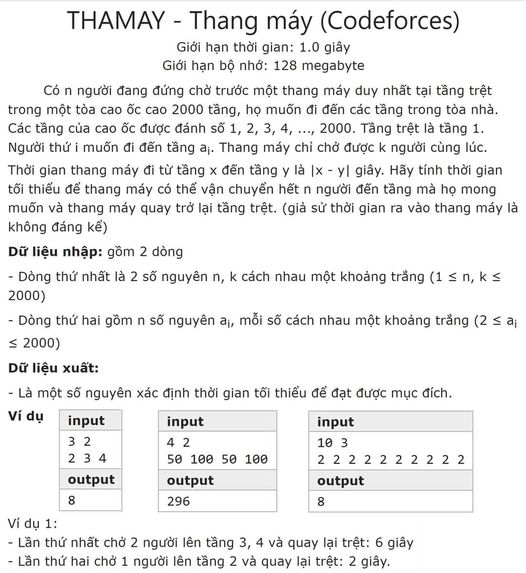
Đánh giá độ phức tạp của cách làm này:

+ Độ phức tạp không gian (Space Complexity): O(N) của mảng a + O(M) của mảng b. Tổng quát thì là O(N + M) nếu theo quy tắc cộng BigO sẽ lấy theo thành phần lớn nhất vì thế có thể kết luận là O(max(N, M)). Còn nếu tính chính xác thì khi trường hợp xấu nhất N = M = 10^5 thì tổng bộ nhớ sẽ là 2 \* 10^5 byte (do mảng bool mỗi phần tử chỉ chiếm 1 byte). 200,000 byte / 1024 = 195.3125 KB / 1024 = 0.2 MB thì như thế yên tâm sẽ không bị lỗi MLE vì giới hạn cho phép đến 256 MB ở đây ta chỉ dùng có 0.2 thôi hehe.

+ Độ phức tạp thời gian (Time Complexity): O(N) của việc khởi tạo mảng a tất cả là false + O(M) của việc khởi tạo mảng b tất cả là false + O(K) đọc dữ liệu vị trí tọa độ các ô đen + O(N) duyệt mảng a đếm các dòng bị xóa + O(M) duyệt mảng b đếm các cột bị xóa. Tổng quát theo quy tắc cộng BigO sẽ lấy theo thành phần lớn nhất vì thế có thể kết luận là O(max(N, M, K)).

Thử xem trong trường hợp xấu nhất khi mọi dữ liệu đều lớn nhất tức là N = M = K = 10^5 như thế số lần lặp cả thảy của toàn bài sẽ là: 5 \* 10^5. Với giới hạn này thì yên tâm sẽ không bị TLE vì nó ít hơn cả 100 đến 200 lần so với ngưỡng giới hạn 1 giây là (3 đến 5) \* 10^7 hay cao lắm là 10^8.

# Thang máy (tham lam)



Mình giải thích nhanh qua đề chỗ này kẻo bạn nào không biết thì sẽ rõ nha, trong đề có đoạn: “Thời gian thang máy đi từ tầng x đến tầng y là |x - y| giây”. Thì |x - y| cái | | ở đây là ký hiệu của “trị tuyệt đối” trong toán học các bạn nhé. Bạn nào không nhớ thì mình nhắc lại, ví dụ |-2| tức là trị tuyệt đối của -2 thì kết quả sẽ là 2, |2| tức là trị tuyệt đối của 2 thì kết quả sẽ là 2. Tóm lại trị tuyệt đối ở đây sẽ giúp biến kết quả bên trong nó số dương thì vẫn giữ nguyên còn nếu là số âm thì nó sẽ biến đổi thành giá trị dương tương ứng, như ví dụ |-2| thì biến đổi -2 thành 2 (biến dấu - thành dấu +). Vậy ta có thể hiểu |x - y| với x là tầng hiện tại, y là tầng đích đến, ý ở đây là cứ lấy 2 tầng trừ nhau rồi nếu kết quả âm thì biến thành dương, còn nếu kết quả dương thì giữ nguyên, hay có thể hiểu là cứ lấy tầng lớn trừ tầng nhỏ.

Rồi thì bài này dùng phương pháp tham lam để xử lý thôi. Kể cả bạn nào có không biết về phương pháp tham lam này thì thực ra đôi khi các bạn đang giải quyết vấn đề theo chính tư tưởng của tham lam mà các bạn không biết. Cho bạn nào không biết thì mình có thể nói nhanh như thế này: Tham lam ở đây nghĩa là ta sẽ cố gắng chọn ra cách giải quyết tốt nhất trong tất cả các cách ở hiện tại rồi đi theo cách giải quyết tốt nhất đó nếu lại đứng trước nhiều lựa chọn cách giải quyết khác nhau nữa thì ta lại cố gắng chọn theo cách giải quyết tốt nhất trong số đó tóm lại cứ tốt nhất mà triển vì thế tên gọi nó là tham lam. Quy luật nó là như vậy thôi chứ còn lại không có công thức từng bước cụ thể giải quyết như mấy thuật toán khác. Mà ta có thể giải quyết theo cách bất kỳ miễn cái tư tưởng mà ta giải quyết luôn là chọn theo hướng tốt nhất trong các lựa chọn thì đó chính là tư tưởng của tham lam. Thì tại sao bài này lại có thể nhìn ra tư tưởng của nó là tham lam? Đọc đề bài là thấy rõ mùi rồi hehe muốn xử lý theo quy luật gì gì đó để sau cùng đáp án phải là nhỏ nhất/lớn nhất các kiểu, đọc kiểu này phát thấy mùi tham lam ngay Thì thực ra tham lam chỉ là dạng tư tưởng thôi, còn lại để hiện thực được nó thì có thể cần đến những kỹ thuật bổ trợ khác nữa. Mình lấy ví dụ ngay trong chính bài tập này, các bạn cứ tiếp tục xem nhé.

Cái đánh lừa của bài này ở chỗ câu nói: “Thời gian đi từ tầng x đến tầng y là |x - y| giây” làm ta có cảm giác kiểu như ta phải đi tính thời gian cụ thể trong mỗi lần di chuyển giữa mỗi tầng. Trong khi đó nếu nhìn nhận kỹ hơn vấn đề thì ta sẽ thấy như sau, mình giả sử theo chiều ngang là chiều của tòa nhà từ dưới lên trên nhé (từ tầng 1 lên các tầng bên trên):

|---------|-------|-------|--------->

1 2 3 4

Các bạn sẽ thấy vấn đề như sau: Nếu giả sử ta cần di chuyển một nhóm người lên đủ các tầng 2, 3, 4 thì từ tầng 1 đến tầng 2 sẽ mất |1 - 2| = 1 giây, từ tầng 2 đến tầng 3 sẽ mất |2 - 3| = 1 giây, từ tầng 3 đến tầng 4 sẽ mất |3 - 4| = 1 giây, rồi từ tầng 4 xuống tầng 1 sẽ mất: |4 - 1| = 3 giây. Tổng cộng: 1 + 1 + 1 + 3 = 6 giây.

Nếu nhìn kỹ các bạn sẽ thấy, từ tầng 1 đến tầng 2, từ tầng 2 đến tầng 3, từ tầng 3 đến tầng 4 thì tổng thời gian đó cũng bằng với thời gian di chuyển từ tầng 1 đến tầng 4, ta có thể nhìn thấy rõ trên trục số điều này.

|---------|-------|-------|---->

1 2 3 4

1--------|

2-------|

3------|

1------------------------|

Lần lượt từng đoạn thẳng bên dưới là từng đoạn đi từ 1 -> 2, 2 -> 3, 3 -> 4 và 1 -> 4. Các bạn sẽ thấy rõ đoạn 1 -> 4 dưới cùng nó bao chứa hoàn toàn 3 đoạn ở trên.

Vậy nếu trường hợp như trên khi ta thấy trong nhóm di chuyển có người cần đến tầng 4 (tầng cao nhất hiện tại trong danh sách các tầng cần di chuyển đến) thì ta cứ tính thẳng |1 - 4| và nhân cho 2 lần vì 1 lần đi, 1 lần về => |1 - 4| \* 2 = 3 \* 2 = 6 là ra nhanh chóng đáp án thay vì phải tính 1 + 1 + 1 + 3 = 6.

Vậy ta thấy một điều thời gian lâu nhất đến từ việc càng phải đi lên tầng càng cao thì thời gian càng lớn. Vậy giải pháp tham lam ở đây là khi trong danh sách nếu có người chọn đi lên những tầng cao thì ta hãy cố gắng gom nhóm càng gom nhiều người đi lên những tầng cao đó trong một lần đi thì các lần sau sẽ ít những lần phải đi lên cao như vậy, chứ nếu không các lần sau lại có những người cần đi lên tầng cao thì cứ đi lên như vậy lại phải tốn thời gian rất nhiều.

Ví dụ sau các bạn sẽ thấy rõ, giả sử có 3 người cần đi lên các tầng lần lượt là: 2, 3, 4. Thang máy mỗi lần chỉ chở tối đa 2 người.

Nếu giả sử ta chọn chở người cần lên tầng 2 và người cần lên tầng 3, khi đó thời gian sẽ tốn = |1 - 3| \* 2 = 2 \* 2 = 4. Sau đó chở nốt người cuối cùng lên tầng 4 thì thời gian sẽ tốn = |1 - 4| \* 2 = 3 \* 2 = 6. Vậy cả thảy sẽ tốn 4 + 6 = 10 giây.

Trong khi đó nếu ưu tiên chở người cần lên tầng 3 và người cần lên tầng 4 trước, khi đó thời gian sẽ tốn = |1 - 4| \* 2 = 3 \* 2 = 6. Sau đó chở người cần lên tầng 2 thì thời gian sẽ tốn = |1 - 2| \* 2 = 1 \* 2 = 2. Vậy cả thảy sẽ chỉ tốn 6 + 2 = 8 giây.

Ta thấy rõ ràng theo cách thứ 2 sẽ chỉ cần 8 giây thay vì cách đầu là 10 giây, vậy cách 2 sẽ tối ưu hơn.

Vậy mấu chốt ở đây ta có thể nhìn thấy giải pháp tham lam ở đây đó là ta hãy gom nhóm k người trong danh sách những người đang cần đi thang máy sao cho nhóm k người này sẽ là những người cần đi lên những tầng cao nhất, và cứ thế theo đúng quy luật này áp dụng với các nhóm k người còn lại cho đến khi hết danh sách cần đi, mỗi lần chở nhóm k người như vậy, ta sẽ tính thời gian căn cứ theo tầng cao nhất trong lần đi đó như đã nói ở trên. Và cộng tổng thời gian trong tất cả các lần đi đó, đó chính là đáp án.

Hãy thử áp dụng quy luật ở trên cho các ví dụ mẫu để nhìn ra quy luật, như ví dụ đầu tiên là có 3 người cần đi lên các tầng 2, 3, 4 và ta có k = 2 tức là thang máy mỗi lần chỉ chở tối đa được 2 người. Thì ta sẽ ưu tiên nhóm 2 người cần đi lên tầng 3, 4 trước và tính ra thời gian = |1 - 4| \* 2 = 3 \* 2 = 6. Sau đó chở người còn lại cần lên tầng 2, thời gian = |1 - 2| \* 2 = 1 \* 2 = 2. Vậy tổng thời gian là: 6 + 2 = 8 giây.

Như ví dụ tiếp theo là có 4 người cần đi lên các tầng 50, 100, 50, 100 và ta có k = 2 tức là thang máy mỗi lần chỉ chở tối đa được 2 người. Thì ta sẽ ưu tiên nhóm 2 người cần đi lên tầng 100 trước và tính ra thời gian = |1 - 100| \* 2 = 99 \* 2 = 198. Sau đó chở 2 người còn lại cần lên tầng 50, thời gian = |1 - 50| \* 2 = 49 \* 2 = 98. Vậy tổng thời gian là: 198 + 98 = 296 giây.

Còn như ví dụ 3 thì toàn bộ những người đều cần đi lên tầng giống nhau nên cứ theo quy luật ở trên mà tính thôi, lúc này sẽ cần 4 lần vận chuyển lên tầng 2, thời gian sẽ là: |1 - 2| \* 2 \* 4 = 1 \* 2 \* 4 = 8 giây.

Vậy một giải pháp ta có thể thấy là ta đi sắp xếp danh sách các tầng cần đi lên của N người sắp theo thứ tự giảm dần, tức là tầng cần đi lên cao nhất sẽ nằm ở đầu và cứ thế giảm dần. Rồi ta cứ thế xét từng đoạn K phần tử từ đầu đến cuối, với mỗi đoạn sẽ chọn phần tử đầu tiên chính là tầng cao nhất trong K người cần đi đó và tính thời gian theo đó. Ví dụ nếu ta có danh sách 10 tầng sau khi sắp giảm dần, tức là index từ 0 đến 9, giả sử K = 3, như thế các nhóm K người sẽ đi trong từng lần là các index: {0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {9}. Thì với nhóm {0, 1, 2} ta sẽ tính thời gian theo tầng đầu tiên tức là index 0, nhóm {3, 4, 5} ta sẽ tính thời gian theo tầng đầu tiên tức là index 3, nhóm {6, 7, 8} sẽ tính thời gian theo tầng index 6, nhóm cuối cùng thì là index 9. Vậy các bạn có nhìn ra quy luật gì không? Index tính từ 0 và cứ thế bước nhảy + K sẽ tính index tại chính đó: 0, 0 + 3 = 3, 3 + 3 = 6, 6 + 3 = 9.

Thử một ví dụ danh sách 5 tầng tức là index từ 0 đến 4, K = 2, như thế các lần đi sẽ là: {0, 1}, {2, 3}, {4}. Ta sẽ thấy thời gian tính đầu tiên là với tầng tại index 0, sau là với tầng tại index 0 + 2 = 2, sau là với tầng tại index 2 + 2 = 4.

Thử một ví dụ danh sách 3 tầng tức là index từ 0 đến 2, K = 5. Như thế ta thấy toàn bộ mọi người đều sẽ được đi trong 1 lần, tức là: {0, 1, 2}. Vậy thì sẽ tính thời gian theo tầng tại index 0.

Đến đây các bạn đã thấy rõ quy luật xử lý rồi hen.

Source code để các bạn tham khảo với tư tưởng xử lý ở trên mình đã nói:

#include <bits/stdc++.h>

using namespace std;

int main(){

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

cin.tie(0);

cout.tie(0);

int n, k;

cin >> n >> k;

int a[n];

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

cin >> a[i];

}

sort(a, a + n, greater<int>());

long long res = 0;

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

res += (a[i] - 1) \* 2;

}

cout << res;

return 0;

}

Giải thích một số chỗ trong code ở trên:

1/ 3 dòng đầu tiên của hàm main ( ios::sync\_with\_stdio(false); cin.tie(0); cout.tie(0); ) sẽ giúp đẩy nhanh tốc độ nhập xuất dữ liệu nếu các bạn đang code theo ngôn ngữ C++ (để hiểu vì sao thì các bạn cứ gõ mấy từ này lên Google là sẽ ra rất nhiều bài viết giải thích nhé), nên 1 kinh nghiệm trong các bài mà cần nhập xuất nhiều, như trường hợp bài này ta thấy N tối đa là 2000, vậy trong trường hợp xấu nhất ta sẽ cần nhập cả 2000 số nên đây cũng là số lượng lớn thì ta nên để thêm 3 dòng này vào đầu để chương trình chạy nhanh hơn. Tất nhiên nếu các bạn code theo cú pháp của C thì chẳng cần mấy dòng đó làm gì, tuy nhiên thường khi code các bài lập trình thi đấu thế này ta dùng C++ vì cú pháp ngắn gọn cũng như nhiều hỗ trợ của nó, vậy nên bạn nào code C++ thì nhớ thêm 3 dòng này vào đầu hàm main nhé.

2/ sort(a, a + n, greater<int>()); thì đây là hàm sắp xếp được hỗ trợ sẵn trong thư viện giải thuật của C++ với độ phức tạp tối ưu là O(N \* logN). Hàm sort trong code này của mình có ý nghĩa là sắp xếp danh sách bắt đầu từ phần tử đầu tiên (a) và sắp xếp đủ n phần tử (a + n), và đi sắp xếp theo tiêu chí giảm dần (greater<int>()). Nếu giả sử bỏ phần tham số thứ 3: greater<int>() đi chỉ còn 2 tham số đầu tiên tức là: sort(a, a + n); thì hiểu là sắp xếp theo tiêu chí tăng dần.

3/ Thì vòng lặp đơn giản sau khi danh sách đã sắp giảm dần tức là đầu danh sách sẽ là những phần tử lớn nhất, thì ta duyệt từ đầu về cuối, với bước lặp là i += k và với mỗi giá trị index i hợp lệ thì ta tính thời gian để đi từ tầng 1 đến tầng a[i] đó, đó là tầng cao nhất trong nhóm k tầng cần đi nên căn cứ thời gian theo đó, tính cả chiều đi và về nên công thức sẽ là: |1 - a[i]| \* 2, do ta biết tầng 1 là tầng thấp nhất nên ở đây ta không cần phải gọi hàm abs(1 - a[i]) mà ta cứ lấy a[i] - 1 thôi vì bảo đảm kết quả luôn ra số dương, rồi nhân 2 và cộng dồn vào biến res kết quả. Cứ thế sau khi kết thúc vòng lặp thì in ra biến res chính là kết quả tổng thời gian ít nhất để di chuyển hết.

Đánh giá độ phức tạp của cách làm này:

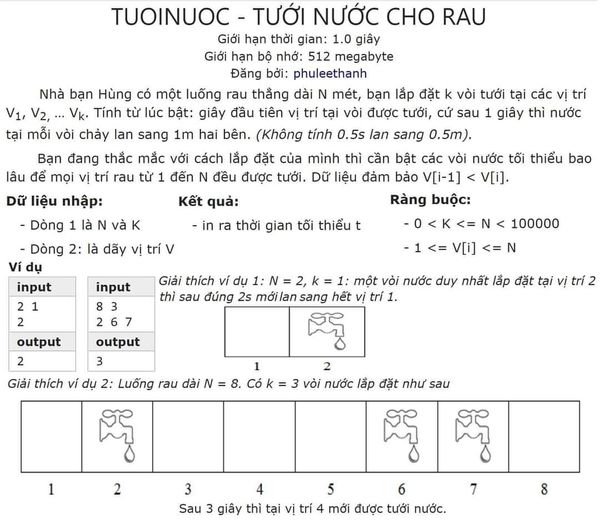
+ Độ phức tạp không gian (Space Complexity): O(N) với N là số lượng phần tử của mảng.

+ Độ phức tạp thời gian (Time Complexity): O(N) của bước nhập mảng + O(N \* logN) của bước sắp xếp mảng giảm dần + O(N/K) của bước duyệt tính tổng thời gian. Tổng quát theo quy tắc cộng BigO sẽ lấy theo thành phần lớn nhất nên ta có thể kết luận độ phức tạp là O(N \* logN).

Vậy với độ phức tạp ở trên, trong trường hợp xấu nhất khi N = 2000, K = 1 thì số lần lặp cả thảy sẽ là: 2000 + 2000 \* log(2000) + 20000 = 2000 + 2000 \* 11 + 2000 tức là = 13 \* 2000 = 26,000 lần lặp. Trong khi đó với giới hạn thời gian 1 giây thì ngưỡng giới hạn số lần lặp là (3 đến 5)\*10^7 tức là 30 triệu đến 50 triệu hay cao lắm là 10^8 tức là 100 triệu nên với ngưỡng 26,000 lần lặp này coi như là 3\*10^4 thì ít hơn ngưỡng giới hạn cả 1000 đến 2000 lần vì thế yên tâm không sợ bị TLE.

Thậm chí với giới hạn N chỉ tối đa 2000 kể cả bạn nào không dùng hàm sort ở bước sắp xếp mà dùng những thuật toán không tối ưu như Interchange Sort, Bubble Sort, Selection Sort … với độ phức tạp O(N^2) thì cũng vẫn không sợ bị TLE nhé vì 2000^2 = 4\*10^6 thì vẫn chưa bị vượt ngưỡng. Tuy nhiên nếu đề bài này nâng giới hạn N có thể lên tối đa là 20,000 chẳng hạn tức là gấp 10 lần nữa thì như thế dùng những thuật toán sắp xếp O(N^2) chắc chắn sẽ bị TLE, lúc này các bạn phải dùng hàm sort như ở trên. Và mình khuyên các bạn cứ dùng hàm sort này luôn vì nó đảm bảo độ phức tạp luôn ở mức O(N \* logN) có thể xem là tốt nhất trong các thuật toán sắp xếp trong tình huống dữ liệu lộn xộn không quy tắc rồi, và cái nữa là nó nhanh gọn để gọi ra hehe, bạn nào chưa biết thì nên tìm hiểu về hàm này đi nhé.

# Tưới nước



Các bạn đọc kỹ đề hen, ngay tại giây đầu tiên (giây thứ 1) thì nước sẽ chảy ra tại vị trí đặt các vòi. Sau đó tại mỗi vị trí vòi thì nước sẽ lan ra vị trí lân cận 2 bên (đề bài chỗ này nói 1m của 2 bên thì hơi khó hiểu, nên nói rõ là mỗi vị trí lân cận 2 bên), cứ mỗi giây sẽ lan ra đồng thời vị trí lân cận ở cả 2 phía. Như thế này cho các bạn dễ hình dung, ví dụ ta có chiều dài N = 5 nghĩa là có 5 vị trí (1, 2, 3, 4, 5) và có 1 vòi nước duy nhất tại vị trí 3. Như vậy giây đầu tiên nước sẽ chảy ra tại vị trí 3 (là vị trí đặt vòi nước). Rồi 1 giây tiếp theo tức là tại giây thứ 2 thì nước sẽ lan ra 2 vị trí lân cận là 2, 4. Rồi 1 giây tiếp theo tức là tại giây thứ 3 thì nước sẽ lan ra 2 vị trí lân cận tiếp theo là 1, 5. Như thế sau 3 giây là nước sẽ lan ra hết N = 5 vị trí.

Thì các bạn nhìn vào test ví dụ 1 khi ta có N = 2 tức là có 2 vị trí 1, 2 và vòi nước đặt tại vị trí 2. Như thế thì giây đầu tiên nước chảy ra tại vòi, giây thứ 2 nước sẽ chảy lan sang vị trí 1 (ở bài này do chỉ có 2 vị trí nên ta không tính việc nó sẽ chảy sang vị trí 3). Như thế chỉ cần 2 giây là toàn bộ đều sẽ được tưới nước nên output là 2.

Tiếp tục ở test ví dụ 2, có N = 8 vị trí (1, 2, 3, 4, 5, 6, 7, 8 ) và vòi nước đặt tại các vị trí 2, 6, 7. Thì giây thứ 1 nước sẽ chảy tại các vị trí {2}, {6}, {7}. Mình cố tình dùng ký hiệu { } để bọc lại vị trí nước sẽ lan của từng vòi cho các bạn dễ nhìn. Sang giây thứ 2 thì vị trí nước tại các vòi sẽ lan như sau: {1, 3}, {5, 7}, {6, 8}. Lúc này vẫn còn vị trí 4 nước chưa lan tới. Nên tiếp tục sang giây thứ 3 thì vị trí nước tại các vòi sẽ lan như sau: {4}, {4, 8}, {5}. Lúc này đây vị trí 4 đã được lan tới, nghĩa là toàn bộ N = 8 vị trí đã được tưới nước, kết quả là sau 3 giây nên output là 3.

Vậy bây giờ đề bài đưa ra có tối đa N = 100,000 vị trí cần tưới nước và có K <= N vị trí đặt vòi nước với từng vị trí đặt vòi đảm bảo luôn nằm trong đoạn từ 1 đến N. Hãy tính toán xem cần tối thiểu bao nhiêu giây để toàn bộ các vị trí đều được tưới nước?

Trước khi đi vào cách giải quyết của bài thì mình muốn các bạn phải đảm bảo biết qua điều này để về sau các bạn xem mới hiểu:

Với các hệ thống web chấm bài giới hạn thời gian chạy của bài tối đa chỉ được 1 giây thì các bạn sẽ rất dễ bị rơi vào lỗi Time Limit Exceeded (TLE: Chạy quá giới hạn thời gian cho phép). Thì như mình đã nói rất nhiều lần là với các hệ thống web chấm bài hiện nay nếu không muốn bị TLE thì tối đa số phép toán trên toàn bài của các bạn chỉ nên ở ngưỡng (3 đến 5)\*10^7 là ngưỡng an toàn với phần lớn hệ thống chấm bài, hoặc với một số hệ thống chấm server mạnh hơn thì có thể nâng ngưỡng lên 10^8 nhưng cũng hên xui nhé. Và thực ra ta không phải đếm chi tiết từng số phép toán trên toàn bài mà ta dựa theo số lần vòng lặp chạy bởi bên trong vòng lặp là nó xử lý lặp đi lặp lại nhiều phép toán nên cứ canh theo đó, các bạn cứ hiểu là tối đa toàn bài chỉ nên lặp (3 đến 5)\*10^7 lần là an toàn hoặc tệ nhất là 10^8 (lúc này thì hên xui nhé) còn vượt ngưỡng này là khả năng TLE rất cao nhe. Lưu ý là ngưỡng mình vừa nói ở trên là với ngôn ngữ C/C++ thôi nhé chứ nếu với những ngôn ngữ khác như Python, C# hay Java nó chạy chậm hơn C/C++ thì ngưỡng đó phải thấp hơn, hoặc người ra đề sẽ chủ động đưa ra giới hạn nếu nộp với C/C++ thì thời gian chạy quy định tối đa 1 giây, các ngôn ngữ khác thì 2 giây. Và ngưỡng mình đưa ra ở trên nếu với đề cho thời gian 2 giây thì các bạn cứ tỷ lệ thuận nhân 2 lên với ngưỡng, 3 giây thì nhân 3, nếu 100 ms tức là 1/10 giây thì các bạn chia 10 cho ngưỡng đó nhé, cứ thế thôi.

Rồi thì như bài này đưa ra thời gian chạy giới hạn là 1 giây nên cứ như ở trên mình đã nói nhé. Giờ ta đi phân tích cách giải quyết cho bài này.

Đề bài có nói câu: Sau 1 giây thì nước tại mỗi vòi sẽ lan sang đều 1m (hiểu là 1 vị trí, đề nói làm khó hiểu chỗ này) của 2 bên qua đó có thể thấy dữ kiện là tốc độ nước chảy ra tại mỗi vòi đều có công suất như nhau. Nếu là như vậy thì các bạn có thể hình dung điều sau đây, mình ví dụ nhé, các bạn xem hình minh họa dưới đây:

S---|-----|----|----|---F

A B C D

Giả sử A B C D là vị trí các vòi, S hiểu là Start tức là vị trí đầu tiên cần tưới và F hiểu là Finish tức là vị trí cuối cùng cần tưới. Nếu xem công suất chảy nước tại mỗi vòi đều bằng nhau, vậy có phải nếu nước chảy từ vòi A lan sang bên trái sẽ có thể tưới được hết đến vị trí S chắc chắn sẽ nhanh hơn nước chảy từ vòi B để có thể tưới được hết đến vị trí S đúng không các bạn? Bởi lúc này SA < SB mà công suất chảy đều xem như là như nhau nên thằng nào ngắn hơn thì thằng đó sẽ nhanh hơn. Các bạn cũng có thể tưởng tượng 2 vận động viên chạy xe đạp đều có tốc độ chạy như nhau, 1 người xuất phát từ A, 1 người xuất phát từ B. Và cả 2 cùng mục tiêu chạy đến đích S. Thì chắc chắn rằng người xuất phát từ A sẽ đến đích sớm hơn người xuất phát từ B bởi vì nhìn hình ta thấy rõ SA < SB. Vậy nên trong trường hợp này chỉ cần quan tâm mất bao lâu để nước từ vòi A chạm đến S chứ không cần quan tâm đến vòi B, tương tự là C, D bởi rõ ràng SA < SB < SC < SD thì nhanh nhất chỉ cần SA.

Tiếp tục ta muốn lấp đầy đoạn AB thì sẽ có thể hoặc là nước chảy từ vòi A sẽ lan sang phải đến B, hoặc nước chảy từ vòi B sẽ lan sang trái đến A. Và rõ ràng đồng thời cả 2 vòi cùng chảy thì tốc độ lấp đoạn AB sẽ càng được rút ngắn lại. Tưởng tượng A = 2, B = 5. Nếu nước chảy từ A muốn lấp đến B thì phải cần 4 giây (giây thứ 1 tại A = 2, giây thứ 2 là vị trí 3, giây thứ 3 là vị trí 4, giây thứ 4 là vị trí B = 5). Tương tự ngược lại nếu nước chảy từ B cũng sẽ cần 4 giây để có thể lan hết đến A. Nhưng đó là trong ngữ cảnh từng vòi chảy riêng lẻ. Còn nếu đồng thời cả 2 vòi cùng chảy thì giây thứ 1 nước chảy tại vị trí 2 5. Giây thứ 2 nước chảy tại vị trí 3 4. Như vậy chỉ cần 2 giây là toàn bộ đoạn AB đã được tưới hết.

Để lấp được đoạn AB tất nhiên ta cũng có thể thấy nước chảy từ vòi C cũng sẽ lan qua trái đến B rồi lan qua trái đến A là như thế cũng lấp được đoạn AB. Nhưng rõ ràng như thế sẽ mất rất nhiều thời gian bởi vì vị trí C đã đứng sau vị trí B, rõ ràng muốn lấp đoạn AB thì cần phối hợp vị trí vòi A và vòi B chứ cần gì phải đến vòi C đúng không các bạn?

Vậy nên ta có thể thấy: Để tưới hết toàn bộ các vị trí từ S đến F ta sẽ tính thời gian của từng bước như sau:

Bước 1: Từ vòi A lan sang trái đến S để lấp đoạn SA

Bước 2: Từ vòi A lan sang phải, từ vòi B lan sang trái để lấp đoạn AB

Bước 3: Từ vòi B lan sang phải, từ vòi C lan sang trái để lấp đoạn BC

Bước 4: Từ vòi C lan sang phải, từ vòi D lan sang trái để lấp đoạn CD

Bước 5: Từ vòi D lan sang phải đến F để lấp đoạn DF

Với mỗi quy trình trong 5 bước ở trên ghi nhận lại thời gian của từng bước, thì kết quả sẽ là thời gian lớn nhất trong 5 bước, nó chính là đáp án của bài toán.

Ví dụ bước 1 cần 2 giây, bước 2 cần 3 giây, bước 3 cần 1 giây, bước 4 cần 5 giây, bước 5 cần 4 giây. Vậy rõ ràng khi tất cả cùng làm đồng thời thì thằng lớn nhất sẽ là thằng hoàn tất sau cùng, và khi đó tất cả đều được tưới hết => sẽ cần 5 giây.

Vậy đến đây các bạn chắc đã phần nào nhìn ra được cách giải quyết rồi đúng không? Đó là ta đi tính thời gian để nước được lấp đầy giữa 2 vòi, riêng vòi đầu thì tính so với điểm đầu, vòi cuối thì tính so với điểm cuối.

Ví dụ giờ nói riêng tính từ vòi đầu đến điểm đầu thì sao? Như trong ví dụ ở trên ta sẽ thấy là từ A đến S. Thì rõ ràng giây đầu tiên nước chảy tại A, sau đó mỗi giây lan ra 1 vị trí và cứ thế đến S. Vậy giả sử A = 4, như thế giây đầu tiên chảy tại vị trí 4, và vị trí 4 cách vị trí đầu tiên (là vị trí 1) là 3 khoảng cách, như thế sẽ cần 3 giây => tổng lại: 1 + 3 = 4 giây. Hay nếu A = 7 thì sẽ cần 7 giây, A = 10 sẽ cần 10 giây, A = 1 (là ngay vị trí đầu tiên) thì cũng cần 1 giây, vẫn đúng. Vậy ta thấy rõ vị trí vòi đầu tiên để lấp hết đến vị trí đầu tiên của ruộng sẽ cần thời gian đúng bằng vị trí vòi đầu tiên luôn.

Vậy vị trí vòi cuối cùng đến điểm cuối cùng thì sao? Ví dụ giả sử N = 10, vòi cuối cùng tại vị trí 6. Như thế giây đầu tiên chảy tại vị trí 6, như thế khoảng cách còn lại sẽ là các vị trí 7, 8, 9, 10 tức là 4 vị trí => cần 4 giây nữa, cộng với 1 giây đầu tiên => cần 5 giây để lấp đến vị trí N = 10. Ta nhận thấy số giây sẽ bằng 1 + khoảng cách tính từ vị trí vòi cuối cùng đến vị trí cuối cùng của dãy. Vậy công thức chính là: 1 + N - Vị trí vòi nước cuối. Cụ thể 1 + 10 - 6 = 5. Thử kiểm tra nếu vị trí vòi cuối đặt tại 8 thì sao => 1 + 10 - 8 = 3 giây. Nếu vị trí vòi cuối đặt tại N = 10 luôn thì sao => 1 + 10 - 10 = 1 giây tức là nước chảy ra ngay tại vòi, vẫn đúng hen các bạn.

Giờ thì đến trường hợp cần lấp đầy từ vòi này đến vòi kia thì sao? Như ví dụ cần lấp từ A đến B. Hãy thử qua vài ví dụ và nháp tay ra kết quả để chúng ta nhìn ra quy luật tổng quát. Ví dụ A = 2, B = 3 => cần 1 giây. A = 2, B = 4 => cần 2 giây. A = 2, B = 5 => cần 2 giây. A = 2, B = 6 => cần 3 giây. A = 2, B = 7 => cần 3 giây. A = 2, B = 8 => cần 4 giây. Đó sơ sơ nhiêu đây. Rồi giờ các bạn có nhìn ra vấn đề không? Nếu như ở vòi đầu tiên và vòi cuối cùng thì chỉ có 1 mình nó chảy, nhưng giờ đây trong tình huống này thì cả 2 vòi cùng chảy ngược hướng và sẽ chạm nhau. Nên có phải nếu từ A = 2 muốn đi đến B = 7 thì quãng đường cần chảy sẽ là 2, 3, 4, 5, 6, 7 => 6 vị trí (tính nhanh bằng cách B - A + 1 tức là 7 - 2 + 1 = 6). Thì bình thường nếu chảy từ A đến B sẽ cần 6 giây. Nhưng do 2 vòi chảy từ 2 phía ngược nhau với tốc độ bằng nhau vì thế mỗi vòi chỉ cần chảy 1 nửa quãng đường là sẽ chạm tới nhau, tức là 6/2 = 3 giây là được. Tuy nhiên hãy thử với ví dụ A = 2, B = 6 => quãng đường chảy sẽ là: 6 - 2 + 1 = 5. Nếu lấy 5/2 = 2.5 tức là mỗi vòi chỉ chảy 2.5 giây là sẽ lấp đầy hết quãng đường AB. Tuy nhiên lúc này đề bài có nói là không tính 0.5 giây mà chỉ tính chẵn 1 giây nên ta hiểu 2.5 cũng làm tròn lên thành 3 giây nhé các bạn. Vậy ta sẽ thấy công thức tổng quát số giây để lấp đầy đoạn AB sẽ là quãng đường AB chia 2 sau đó làm tròn lên số nguyên gần nhất. Thử kiểm chứng công thức: A = 2, B = 3 => (3 - 2 + 1) / 2 = 1 giây. A = 2, B = 4 => (4 - 2 + 1) / 2 = 1.5 làm tròn lên 2 giây.

Rồi vậy thì công thức giải quyết cho bài này giờ đây rất đơn giản: Ta lưu trữ các vị trí các vòi vào mảng a và các vị trí này đề đã đảm bảo được sắp theo thứ tự tăng dần. Khởi tạo Max = giá trị lớn nhất giữa a[0] là vị trí vòi đầu tiên cũng chính là số giây để vòi đầu tiên chảy đến điểm bắt đầu của ruộng và 1 + n - a[k - 1] là quãng thời gian vòi cuối cùng chảy đến vị trí cuối cùng của ruộng. Rồi vòng lặp bắt đầu xét từ phần tử thứ 2 trong mảng, gọi đó là a[i] thì ta sẽ tính toán thời gian của a[i] với a[i - 1] bằng cách làm tròn((a[i] - a[i - 1] + 1) / 2) rồi so sánh cập nhật lại Max và cứ thế lặp đến hết vòi cuối cùng. Sau cùng in ra giá trị Max chính là thời gian ít nhất để toàn bộ ruộng được tưới.

Source code để các bạn tham khảo nếu làm theo hướng này:

#include <bits/stdc++.h>

using namespace std;

int main(){

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

cin.tie(0);

cout.tie(0);

int n, k;

cin >> n >> k;

int a[k];

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

cin >> a[i];

}

int Max = max(a[0], 1 + n - a[k - 1]);

for(int i = 1; i < k; ++i){

Max = max(Max, (int)ceil((a[i] - a[i - 1] + 1) / 2.0));

}

cout << Max;

return 0;

}

Giải thích một số chỗ trong code ở trên:

1/ 3 dòng đầu tiên của hàm main ( ios::sync\_with\_stdio(0); cin.tie(0); cout.tie(0); ) sẽ giúp đẩy nhanh tốc độ nhập xuất dữ liệu nếu các bạn đang code theo ngôn ngữ C++ (để hiểu vì sao thì các bạn cứ gõ mấy từ này lên Google là sẽ ra rất nhiều bài viết giải thích nhé), nên kinh nghiệm trong các bài mà cần nhập xuất nhiều, như trường hợp bài này ta thấy phần input có thể lên đến tối đa 100,000 số cần nhập thì đây là số lượng lớn thì ta nên để thêm 3 dòng này vào đầu để chương trình chạy nhanh hơn. Tất nhiên nếu các bạn code theo ngôn ngữ C thì chẳng cần mấy dòng đó làm gì, tuy nhiên thường khi code các bài lập trình thi đấu thế này ta dùng C++ vì cú pháp ngắn gọn cũng như nhiều hỗ trợ của nó, vậy nên bạn nào code C++ thì nhớ thêm 3 dòng này vào đầu hàm main nhé.

2/ Chỗ tính toán trong vòng lặp phép chia các bạn phải để chia cho 2.0 để nó hiểu phép chia đó là phép chia ra kết quả thực nhé, do mảng a là mảng số nguyên nên a[i] - a[i - 1] + 1 ra kết quả số nguyên, nên nếu chia cho 2 thì 2 cũng là số nguyên từ đó phép chia đó là dạng: số nguyên / số nguyên thì kết quả sẽ ra số nguyên từ đó sẽ bị sai nha các bạn. Ví dụ lúc này sẽ là: 5/2 thì kết quả chỉ ra 2 từ đó hàm ceil làm tròn lên số nguyên gần nhất cũng chỉ là 2 thì rõ ràng là sai vì nếu đúng phải ra 5/2 = 2.5 từ đó hàm ceil làm tròn lên 3. Nên ta để chia cho 2.0 ý cho compiler biết là số 2 đó ở định dạng số thực. Thì chỉ cần phép chia mà thành phần tử số hay mẫu số có kiểu số thực thì compiler sẽ cho ra kết quả chia ra số thực chứ không phải chia ra số nguyên nha các bạn. Hàm ceil nằm trong thư viện toán học, do ta đã dùng thư viện bits/stdc++.h đã bao chứa các thư viện khác nên không cần include riêng, nhưng nếu không dùng thư viện bits/stdc++.h này thì các bạn nhớ phải include thư viện toán học cmath vào thì mới sử dụng hàm ceil được nhé.

Đánh giá độ phức tạp của cách làm này:

+ Độ phức tạp không gian (Space Complexity): O(K) do ta dùng mảng a có K phần tử để lưu trữ vị trí các vòi nước.

+ Độ phức tạp thời gian (Time Complexity): O(K) của bước đọc vị trí các vòi nước từ input vào mảng a + O(K) của bước xử lý duyệt mảng a để tính thời gian chảy giữa 2 vòi. Nên tổng sẽ là O(2 \* K) thì theo quy tắc nhân BigO khi nhân với hằng số ta có thể bỏ qua vì thế có thể kết luận tổng quát là O(K).

Thì với độ phức tạp thời gian O(2 \* K) trong trường hợp K tối đa = N = 10^5 thì yên tâm sẽ không bị TLE nha các bạn vì như hồi đầu bài mình có nói nếu tổng số lần lặp <= (3 đến 5)\*10^7 là chắc ăn không thể nào TLE được, ở đây 2 \* 10^5 là ít hơn cả 150 đến 250 lần. Nộp lên web thì Accepted với time = 15 ms nhé các bạn.

Vậy đến đây nhìn lại ta sẽ thấy bài này chỉ là bài tập cơ bản về mảng 1 chiều thôi. Bản chất trên trang web cũng liệt kê bài tập này thuộc về thể loại mảng 1 chiều.

Tuy nhiên mình muốn đem đến một góc nhìn khác cho các bạn tối ưu hơn cho bài này. Bài này vẫn theo hướng giải quyết như đã trình bày nhưng ta không cần sử dụng mảng tức là độ phức tạp bộ nhớ chỉ là O(1) hằng số nha các bạn, và ta có thể kết hợp cả quá trình đọc dữ liệu từ input và quá trình xử lý lại trong duy nhất 1 vòng lặp, tức là độ phức tạp về mặt lý thuyết vẫn sẽ là O(K) nhưng thực tế thì tốt hơn code hiện tại vì code hiện tại tính đúng ra sẽ là O(2 \* K) tức là 2 lần vòng lặp còn chúng ta có thể tối ưu chỉ cần O(K) tức là 1 lần lặp duy nhất thôi. Để được như vậy thì các bạn xem mình phân tích dưới đây nhé.

Ta sẽ đại diện biến x đọc dữ liệu từng số từ input vào, đồng thời có biến temp\_x để trước khi biến x nhận giá trị mới từ input thì temp\_x sẽ nhận giá trị x hiện tại lưu sang sau đó x nhận giá trị mới, và cứ thế khi x nhận giá trị mới xong thì đi tính thời gian giữa temp\_x và x.

Source code để các bạn tham khảo nếu làm theo hướng này:

#include <bits/stdc++.h>

using namespace std;

int main(){

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

cin.tie(0);

cout.tie(0);

int n, k, x, temp\_x;

cin >> n >> k;

cin >> x;

int Max = x;

while(--k){

temp\_x = x;

cin >> x;

Max = max(Max, (int)ceil((x - temp\_x + 1) / 2.0));

}

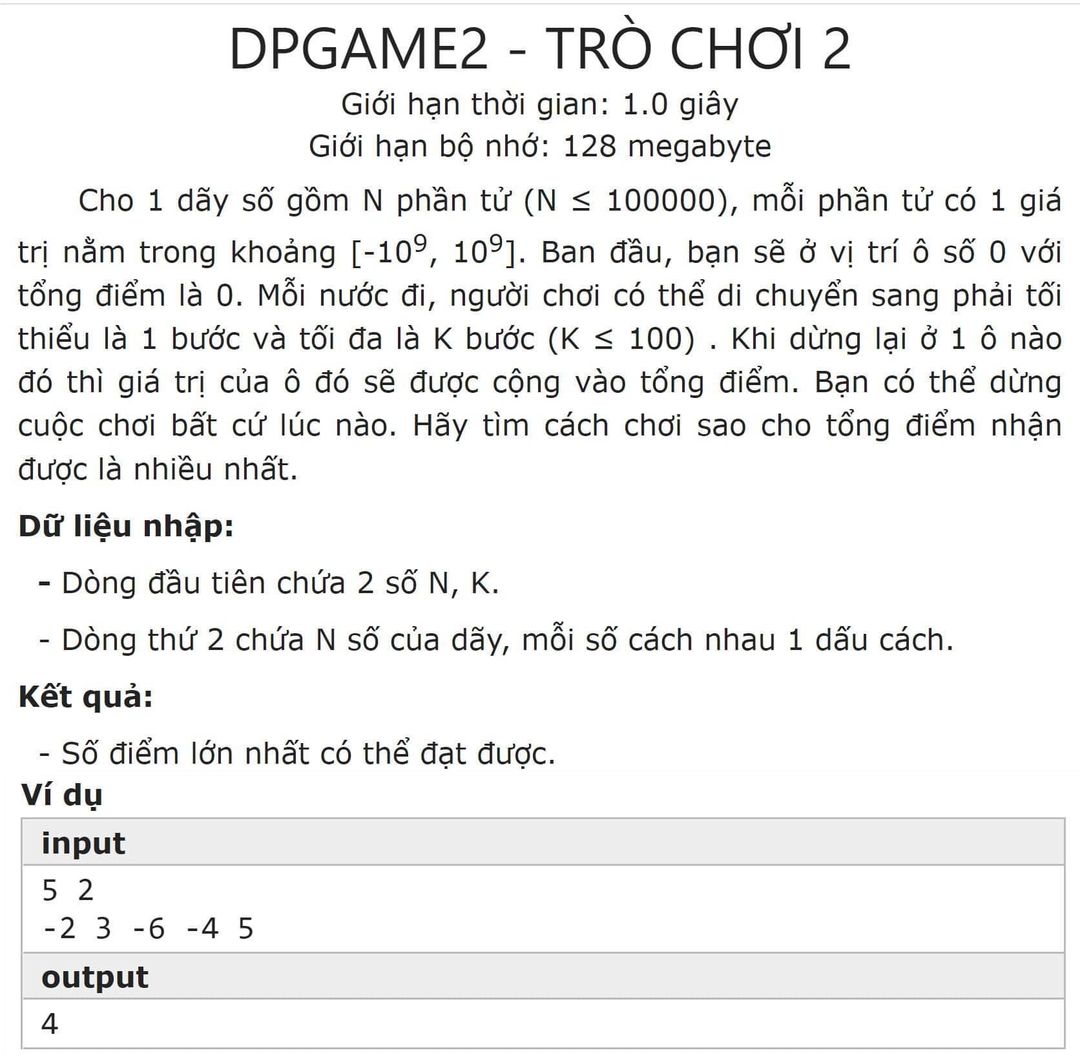
cout << max(Max, 1 + n - x);

return 0;

}

Trước khi vào vòng lặp mình chủ động cin >> x; để đọc giá trị x đầu tiên từ input, từ đó gán Max = x vì đó cũng là thời gian nước chảy từ vòi đầu tiên đến vị trí đầu tiên của ruộng. Và khi đã đọc trước 1 giá trị thì xem như số lần lặp tiếp theo chỉ còn k-1 lần, vì thế mình để while(--k) nghĩa là giảm k đi 1 đơn vị trước rồi sau đó kiểm tra nếu k khác 0 thì vòng lặp sẽ thỏa mãn, nếu k bằng 0 thì vòng lặp kết thúc. Rồi cứ thế gán temp\_x = x rồi tiến hành nhập giá trị x mới từ input rồi tính toán cập nhật lại Max. Sau cùng thì sẽ xuất ra kết quả số lớn nhất giữa Max và thời gian từ vòi cuối cùng đến vị trí cuối cùng của ruộng.

# Trò chơi 2



Bài này áp dụng phương pháp Quy Hoạch Động (Dynamic Programming) để giải quyết nhé các bạn. Cho bạn nào đã có biết cơ bản qua về quy hoạch động rồi thì **4 bước giải một bài toán quy hoạch động** như sau:

Bước 1: Xác định bài toán con: Gọi F[i] là tổng điểm lớn nhất có được khi xét qua i ô số (tức là tính từ ô đầu tiên gọi là thứ 1 và xét đến hết ô thứ i)

Bước 2: Xác định bài toán cơ sở: F[0] = 0 bởi nếu không đi ô nào hết thì tổng điểm lớn nhất sẽ vẫn là 0 (điểm ban đầu của người chơi)

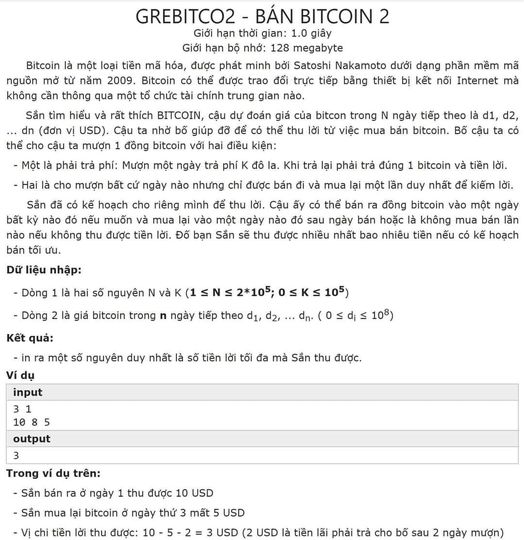
Bước 3: Xác định đáp án bài toán: Cả thảy ta có N ô số, và đề có nói là người chơi có thể dừng bất kỳ lúc nào vậy nên đáp án bài toán cần trả về điểm số lớn nhất có thể của người chơi, không phải lúc nào tại ô F[N] cũng có điểm số lớn nhất, điều đó chỉ đúng nếu các ô số đều là giá trị dương, nhưng ở đây các ô số có thể có giá trị âm, các bạn tưởng tượng nếu giả sử có N = 10 ô, 5 ô đầu toàn số dương, 5 ô cuối toàn số âm. Vậy có phải điểm số lớn nhất là khi xét đến hết ô số 5 và lấy được tổng 5 ô đầu, ta có thể dừng sớm bởi nếu tiếp tục xét tiếp 5 ô sau thì điểm sẽ bị giảm đi thì đâu còn là lớn nhất nữa. Vậy nên điểm số lớn nhất ở đây ta sẽ xét tất cả các F[i] với i từ 0 đến N thì giá trị F[i] nào lớn nhất thì sẽ lấy theo nó. Lưu ý là phải tính cả trường hợp i từ 0, tức là xét cả F[0] nhé các bạn vì nếu các bạn chỉ tìm F[i] lớn nhất với i từ 1, lỡ kết quả lớn nhất đó ra số âm, như thế thì sẽ sai bởi người chơi thậm chí ngay từ đầu tiên cũng không cần bước đi bước nào, vì đề đã có nói “người chơi có thể dừng chơi bất kỳ lúc nào” mà, còn nếu đã chọn bước đi thì tối thiểu 1 ô, tối đa K ô. Như thế ngay khi người chơi ở vạch xuất phát (ô 0 với 0 điểm) thì người chơi cũng được quyền kết thúc luôn, như thế bảo toàn điểm số 0. Vậy nên chỗ này các bạn lưu ý kẻo sai.

Bước 4: Xác định công thức truy hồi: Ta thấy rằng nếu muốn đi đến vị trí i thì trước đó người chơi phải ở một trong các vị trí i - 1 hoặc i - 2 hoặc i - 3 hoặc … hoặc i - k. Tối thiểu phải là vị trí i - k để với tối đa k bước nhảy tức là có thể là 1 hay 2 hay 3 hay … k thì người chơi sẽ đến được vị trí i. Vậy để tổng điểm thu được là lớn nhất khi xét đến vị trí i thì ta sẽ thử đi tính tổng điểm mà người chơi khi đứng ở từng vị trí trước đó (tức là tổng điểm của người chơi cũng đã lớn nhất khi tính từ đầu đến ngay vị trí hiện tại đó) và từ đó nhảy đến vị trí i và tiếp nhận thêm điểm số tại vị trí i để xem trong tối đa k vị trí đó thì F[i] sẽ lấy theo tổng điểm lớn nhất thu được. Tức là: F[i] = max(F[i - 1] + A[i], F[i - 2] + A[i], F[i - 3] + A[i], … , F[i - k] + A[i]). Lưu ý chỗ này để cho đơn giản bớt thao tác tính toán thay vì ta cứ lấy giá trị A[i] (là điểm số của ô thứ i) cộng vào từng F[i - 1], F[i - 2], F[i - 3], …, F[i - k] rồi so sánh xem kết quả lớn nhất là thằng nào từ đó gán vào F[i] thì ta chỉ cần tìm giá trị lớn nhất của các F[i - 1], F[i - 2], F[i - 3], … F[i - k] sau đó lấy giá trị lớn nhất đó cộng với A[i] rồi gán vào F[i] là được, ta sẽ bớt đi phép toán cộng cứ liên tục cộng K lần, giờ đây chỉ cộng 1 lần duy nhất là được.

Lưu ý: Tối đa chỉ lấy K vị trí trước đó, nếu không đủ K thì vị trí lấy cuối cùng là 0 (là vị trí xuất phát ban đầu của người chơi). Ví dụ nếu K = 4 và đang xét i = 7 thì từ vị trí i = 7 ta sẽ xét 4 vị trí người chơi đứng trước đó là: 3, 4, 5, 6. Nhưng nếu giả sử i đang xét là 3 thì nghĩa là ta chỉ xét các vị trí người chơi đứng trước đó là 0, 1, 2 thôi. Chứ không có chuyện xét vị trí: -1 nha các bạn, các bạn mà không ràng buộc chỗ này nó gọi vào F[-1] là lỗi ngay đấy vì mảng bắt đầu từ index 0 mà, làm gì có chuyện index -1.

COde???

# BITCOIN2



Cái đề thì dài dòng lắm nhưng tóm gọn lại đề bài có thể phát biểu đơn giản thế này thôi, bạn nào phải nhìn ra được cái đề rút gọn lại thế này thì mới làm được bài này nha: Cho dãy N phần tử các giá trị a1, a2, a3 … an. Gọi index i, j là chỉ số từng phần tử trong mảng thỏa mãn i < j. Tìm giá trị lớn nhất của max(0, a[i] - a[j] - (j - i) \* K). Đơn giản là vậy thôi.

Tại sao có thể phát biểu đề bài đơn giản như vậy? Vì các bạn xem nhé, các giá trị mảng a là giá dự đoán bitcoin trong các ngày tiếp theo. Và theo ràng buộc đề đưa ra là chỉ bán được 1 lần duy nhất và sau khi bán thì mua được 1 lần duy nhất rồi thôi. Chính vì vậy ta xem giá trị a[i] chính là giá thời điểm bán đi và giá trị a[j] chính là giá thời điểm mua lại. Chắc chắn i phải < j vì nếu trường hợp bằng nhau tức là bán đi và mua lại ngay trong ngày thì coi như tiền lãi vẫn là 0 thì ta đã có ràng buộc là có thể không bán và không mua thì tiền lãi vẫn là 0. Nên nếu việc ta ràng buộc i <= j thì vẫn đúng thôi nhưng cứ thế sẽ bị xét trùng rất nhiều phần tử không cần thiết. Ta chỉ cần ràng buộc i < j rồi thì tìm giá trị của 2 cặp i, j bất kỳ thỏa mãn a[i] - a[j] - (j - i) \* K là lớn nhất sau đó so sánh nó với 0 và chọn theo thằng lớn hơn. Tại sao lại phải trừ cho (j - i) \* K ? vì a[i] là giá thời điểm bán, a[j] là giá thời điểm mua lại, 2 thằng này trừ nhau thì ra tiền lời, thì còn phải trừ cho tiền lãi khoản vay là K dollar/ngày, vì thế khoảng cách giữa 2 ngày chính là j - i và nhân cho K sẽ ra tổng số tiền phải trả, thì tiền lời có được từ việc mua bán phải trả lãi cho khoản vay (tức là trừ đi) và phần còn lại chính là tiền lời thực sự giữ lại được.

#include <bits/stdc++.h>

// khai bao bien

int n;//so ngay theo doi bitcoin

int k;//so tien lai hang ngay

int main()

{

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

cin.tie(0);

cout.tie(0);

cin>>n>>k;

int a[n];//gia tri cua bit coin theo ngay

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

cin>>a[i];

}

//di tim so tien loi toi da

long long Max = 0;

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

for(int j = i + 1; j < n; ++j){

if(a[i] - a[j] - (j - i) \* k) {

Max = a[i] - a[j] - (j - i) \* k;

}

}

}

cout << Max;

return 0;

}