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 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. 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.

Một cách làm cơ bản nhất dễ dàng nhìn ra đó là vét cạn xét hết tất cả các cặp (i, j) thỏa mãn i < j rồi tính hiệu a[j] - a[i] và liên tục so sánh để sau cùng tìm ra hiệu lớn nhất. Tức là xem index bắt đầu từ 0, thì ta sẽ lần lượt xét j = 1 và i là tất cả những thằng < j. Lý do tại sao j = 1 vì nếu chọn j = 0 thì trước đó chẳng có thằng nào cả nên j sẽ bắt đầu từ 1. Và cứ thế j++ đến khi nào hết thì thôi, với mỗi giá trị j thì vòng lặp for i sẽ duyệt các giá trị i từ j - 1 về 0. Như ví dụ với N = 5 thì ta sẽ có các cặp sau:

j = 1, i = 0 => {1, 0}

j = 2, i = 1, 0 => {2, 1}, {2, 0}

j = 3, i = 2, 1, 0 => {3, 2}, {3, 1}, {3, 0}

j = 4, i = 3, 2, 1, 0 => {4, 3}, {4, 2}, {4, 1}, {4, 0}

Cứ thế ta xét hết tất cả các cặp, với mỗi cặp đi tính hiệu a[j] - a[i] rồi so sánh qua tất cả các cặp để tìm ra hiệu lớn nhất và đó chính là đáp án.

Source code để các bạn tham khảo tư tưởng xử lý ở trên:

#include <bits/stdc++.h>

using namespace std;

int main()

{

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

cin.tie(0);

cout.tie(0);

int n;

cin >> n;

int a[n];

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

{

cin >> a[i];

}

int Max = INT\_MIN;

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

{

for(int i = j - 1; i >= 0; --i)

{

Max = max(Max, a[j] - a[i]);

}

}

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(false); cin.tie(0); cout.tie(0); ) hiểu cơ bản là nó đồng bộ luồng nhập xuất của C++ với C, nghĩa là tuy ta dùng cú pháp nhập xuất của C++ nhưng tương đương là ta dùng cú pháp nhập xuất của C. Tại sao lại như vậy? Vì tốc độ nhập xuất của C nhanh hơn C++ rất nhiều, nên 1 kinh nghiệm trong các bài mà cần nhập xuất nhiều, như trường hợp bài này ta thấy N tối đa là 10^5 số cần nhập 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, Biến Max ta khởi tạo ban đầu là INT\_MIN tức là giá trị nhỏ nhất của kiểu int (-2,147,483,648) để đảm bảo mọi giá trị lần đầu tiên so sánh với nó sẽ luôn lớn hơn nó mà cập nhật lại Max.

3, Câu lệnh: Max = max(Max, a[j] - a[i]); là cách viết gọn của: if(a[j] - a[i] > Max) { Max = a[j] - a[i]; }. Hàm max(a, b) sẽ trả về giá trị lớn nhất trong 2 giá trị a, b truyền vào, đây là hàm được hỗ trợ sẵn trong C++. Tương tự với hàm max thì ta cũng có hàm min, nếu các bạn về sau có gặp thì cũng hiểu hen.

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 đọc mảng + O((N\*(N - 1))/2) của bước xử lý so sánh các cặp (i, j). Thì phần O((N\*(N - 1))/2) tính ra sẽ là (N^2 - N)/2 <=> (N^2)/2 - N/2. Thì trừ cho N/2 cũng tương tự như cộng cho -N/2 nên theo quy tắc cộng BigO ta bỏ qua chỉ lấy thành phần lớn nhất, vì thế chỉ lấy theo (N^2)/2. Mà (N^2)/2 <=> 0.5 \* N^2 thì theo quy tắc nhân BigO nhân cho hằng số thì ta có thể bỏ qua nên chỗ này ta có thể kết luận là O(N^2). Vậy sẽ là O(N) + O(N^2) thì theo quy tắc cộng BigO sẽ lấy theo thành phần lớn nhất nên tổng quát lại độ phức tạp thời gian của đoạn code ở trên ta có thể kết luận là O(N^2).

Với độ phức tạp O(N^2) thì khi trong trường hợp xấu nhất N = 10^5 thì (10^5)^2 = 10^10 thì đã vượt quá ngưỡng giới hạn 1 giây là (3 đến 5)\*10^7 hay cao lắm là 10^8, cụ thể là vượt gấp cả 100 đến 200 lần vì thế code này chắc chắn sẽ bị TLE. Thực ra ngay từ đầu khi ta nghĩ ra ý tưởng này thì ta có thể nhẩm tính độ phức tạp trước để biết rằng sẽ không ổn mà không cần phải mất công cài đặt, dành thời gian nghĩ ra hướng làm tốt hơn. Tuy nhiên nếu các bạn chưa nghĩ ra được hướng làm nào tốt hơn và chắc chắn đúng thì hãy cứ làm theo cách vét cạn này bởi qua đó sẽ giúp các bạn nhìn ra những cái dở làm cơ sở để nhìn ra cách làm tối ưu hơn khắc phục những cái dở của cách hiện tại.

Giờ ta đi bàn luận cách làm tối ưu cho bài này. Ta sẽ thấy rằng cách làm vét cạn ở trên có cái dở là nó luôn phải xét đi xét lại những vị trí mà trước đó nó đã xét. Như ví dụ với N = 5 thì ta sẽ có các cặp sau:

j = 1, i = 0 => {1, 0}

j = 2, i = 1, 0 => {2, 1}, {2, 0}

j = 3, i = 2, 1, 0 => {3, 2}, {3, 1}, {3, 0}

j = 4, i = 3, 2, 1, 0 => {4, 3}, {4, 2}, {4, 1}, {4, 0}

Các bạn sẽ thấy vị trí i = 0 bị xét lặp lại đủ 4 lần, i = 1 bị xét lặp lại 3 lần, i = 2 bị xét lặp lại 2 lần. Và đó chính là chỗ chưa được tối ưu của cách làm này. Nếu mỗi vị trí i ta chỉ xét qua đúng 1 lần thôi chứ không bị xét lặp lại nhiều lần như thế này thì sẽ tối ưu hơn các bạn đồng ý chứ?

Vậy làm sao để mỗi vị trí i chỉ xét 1 lần duy nhất? Trước tiên ta phải hiểu được vì sao cách làm ở trên lại cứ phải xét lặp lại nhiều lần vị trí i? Nói chính xác là khi xét qua vị trí j tăng lên tiếp theo thì cứ phải xét lại các vị trí i đứng trước nó từ đó vị trí i liên tục bị xét lặp lại qua các vị trí j mới. Nó làm vậy để đảm bảo từ vị trí j mới đó sẽ đảm bảo xét qua tất cả các cặp có thể để hiệu giữa phần tử tại vị trí j đó với tất cả các phần tử i trước nó là lớn nhất có thể.

Vậy ta sẽ nhìn ra mấu chốt: Để hiệu giữa a[j] và các phần tử a[i] trước đó là lớn nhất có thể, thì nghĩa là a[j] sẽ cần trừ cho thằng a[i] nhỏ nhất trong các vị trí i. Tại vì trừ cho thằng nhỏ nhất thì kết quả sẽ được lớn nhất, đây là quy luật toán học cơ bản. Vậy khi xét vị trí j mới, nếu ta có thể biết được từ đầu (index 0) đến trước vị trí j hiện tại (j - 1) thì phần tử nhỏ nhất là bao nhiêu? Gọi đó là Min, thì chỉ cần lấy a[j] - Min là ta sẽ ra được hiệu lớn nhất của mảng tính đến vị trí j. Và cứ thế khi xét hết toàn bộ các vị trí j thì ta sẽ có được hiệu lớn nhất là đáp án.

Như thế ta có thể nhìn ra một cách làm tối ưu như sau: Ta có biến Min lưu giá trị nhỏ nhất của các phần tử mảng tính từ đầu tiên, và biến Max lưu giá trị hiệu lớn nhất cần tìm. Dữ liệu cho đảm bảo tối thiểu mảng sẽ luôn có 2 phần tử nên ta khởi tạo ban đầu Min = min(a[0], a[1]) tức là giá trị nhỏ nhất giữa a[0] và a[1] và Max là a[1] - a[0]. Ta cứ thế duyệt tuần tự từng phần tử của mảng tiếp tục từ index 2, lúc này ta hiểu từng phần tử xét đến chính là a[j] như ta nói ở trên, thì với mỗi a[j] ta sẽ so sánh nếu a[j] - Min mà lớn hơn Max thì ta cập nhật lại hiệu lớn nhất chính là Max = a[j] - Min. Nếu a[j] mà nhỏ hơn cả Min thì cập nhật lại Min = a[j]. Cứ thế khi xét hết toàn bộ index j của mảng thì đáp án chính là Max.

Source code để các bạn tham khảo tư tưởng xử lý ở trên:

#include <bits/stdc++.h>

using namespace std;

int main()

{

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

cin.tie(0);

cout.tie(0);

int n;

cin >> n;

int a[n];

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

{

cin >> a[i];

}

int Min = min(a[0], a[1]), Max = a[1] - a[0];

for(int j = 2; j < n; ++j)

{

Max = max(Max, a[j] - Min);

Min = min(Min, a[j]);

}

cout << Max;

return 0;

}

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 đọc mảng từ input + O(N) của bước duyệt mảng xử lý cập nhật Min, Max. Vậy tổng cộng là O(2 \* N) thì theo quy tắc nhân BigO khi nhân với hằng số ta có thể bỏ qua, vậy nên độ phức tạp tổng quát sẽ là O(N).

Với độ phức tạp O(N) này thì yên tâm sẽ không bị TLE rồi vì N lớn nhất là 10^5, thì 10^5 vẫn đảm bảo thấp hơn ngưỡng giới hạn (3 đến 5)\*10^7 hay cao lắm là 10^8 vì thế ta yên tâm.

Thực ra ta có thể tối ưu hơn được nữa cả về độ phức tạp không gian và thời gian. Bởi vì các bạn nhìn xem nhé, vòng lặp for đầu tiên đọc dữ liệu từ input cũng là duyệt từ đầu đến cuối đọc dữ liệu vào mảng, rồi vòng lặp for thứ 2 cũng là duyệt lại các giá trị của mảng từ đầu đến cuối. Vậy ta có thể chỉ cần dùng 1 vòng for duy nhất vừa kết hợp quá trình đọc dữ liệu vào mảng rồi lấy giá trị đó đi xử lý luôn không cần phải làm thêm vòng lặp nữa làm gì, thêm cái nữa vì lấy luôn giá trị đó đi xử lý và ta thấy khi xét tiếp tục lên các giá trị a[j] tiếp theo thì a[j] đó chỉ so sánh với 2 biến Min, Max mà cập nhật lại, hoàn toàn không cần so sánh với các a[j] trước đó, vậy nên ta cũng chẳng cần thiết phải lưu trữ các giá trị vào mảng, ta cứ đại diện biến x đọc dữ liệu rồi lấy biến x đó đi xử lý so sánh với Min, Max và cập nhật lại Min, Max thôi.

Source code để các bạn tham khảo tư tưởng xử lý ở trên:

#include <bits/stdc++.h>

using namespace std;

int main()

{

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

cin.tie(0);

cout.tie(0);

int n, x, Min, Max;

cin >> n >> Min >> x;

Max = x - Min;

Min = min(Min, x);

for(int i = 2; i < n; ++i)

{

cin >> x;

Max = max(Max, x - Min);

Min = min(Min, x);

}

cout << Max;

return 0;

}

Giải thích một chút để tránh các bạn khó hiểu, như code trước đó ta khởi tạo Min = min(a[0], a[1]) và Max = a[1] - a[0] đúng không? Thì ở đây cũng tương tự như vậy thôi sau khi cin >> n xong thì ta cin >> Min tương ứng là giá trị của phần tử a[0] sẽ lưu vào Min, sau đó cin >> x tức là giá trị a[1] lưu vào x. Rồi Max = x - Min tương ứng x lúc này đang là a[1] và Min đang là a[0] nên nghĩa là Max = a[1] - a[0] tương tự như code trước đó, rồi Min = min(Min, x) lúc này Min đang là giá trị a[0], x đang là a[1] nên nghĩa là so sánh giữa a[0] và a[1] tìm ra giá trị nhỏ nhất cập nhật lại cho Min. Rồi thì vòng for bắt đầu từ phần tử thứ 3 của mảng và đại diện biến x đọc dữ liệu vào, biến x lúc này đây tương tự code trước đó chính là giá trị a[j]. Mọi thứ đều tương tự hết các bạn thấy không?

Thì code ở trên sẽ tối ưu hơn code trước đó về mặt không gian khi ta không cần dùng mảng lưu trữ, tức là độ phức tạp không gian lúc này chỉ là O(1). Và tuy độ phức tạp thời gian vẫn là O(N) về mặt lý thuyết nhưng về mặt thực tế nó được tối ưu hơn khi chỉ cần 1 vòng lặp duyệt qua N phần tử còn code trước đó cần đến 2 vòng lặp như vậy, tức là code trước đó xét đúng thực tế là O(2 \* N) còn code này chỉ là O(N) thôi. Nhưng về mặt tổng quát thì cả 2 code vẫn đều kết luận chung là O(N).