2 Pointers Method – Phương Pháp 2 Con Trỏ

Nguyễn Quản Bá Hồng*

Ngày 19 tháng 8 năm 2025

Tóm tắt nội dung

This text is a part of the series *Some Topics in Advanced STEM & Beyond*:
URL: https://nqbh.github.io/advanced_STEM/.
Latest version:

- *2 Pointers Method – Phương Pháp 2 Con Trỏ*.
  PDF: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/OLP_ICPC/2_pointers/NQBH_2_pointers.pdf.
  TEX: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/OLP_ICPC/2_pointers/NQBH_2_pointers.tex.
- *Olympiad in Informatics & Association for Computing Machinery–International Collegiate Programming Contest – Olympic Tin Học Sinh Viên OLP & ICPC*.
  PDF: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/OLP_ICPC/NQBH_OLP_ICPC.pdf.
  TEX: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/OLP_ICPC/NQBH_OLP_ICPC.tex.
- Codes:
  - Input: https://github.com/NQBH/advanced_STEM_beyond/tree/main/OLP_ICPC/input.
  - Output: https://github.com/NQBH/advanced_STEM_beyond/tree/main/OLP_ICPC/output.
  - C: https://github.com/NQBH/advanced_STEM_beyond/tree/main/OLP_ICPC/C.
  - C++ implementation: https://github.com/NQBH/advanced_STEM_beyond/tree/main/OLP_ICPC/C++.
  - C#: https://github.com/NQBH/advanced_STEM_beyond/tree/main/OLP_ICPC/C%23.
  - Java: https://github.com/NQBH/advanced_STEM_beyond/tree/main/OLP_ICPC/Java.
  - JavaScript: https://github.com/NQBH/advanced_STEM_beyond/tree/main/OLP_ICPC/JavaScript.
  - Python: https://github.com/NQBH/advanced_STEM_beyond/tree/main/OLP_ICPC/Python.
  - Resources: https://github.com/NQBH/advanced_STEM_beyond/tree/main/OLP_ICPC/resource.

List of classical problems: subarray sum on $\mathbb{N}, \mathbb{Z}, \mathbb{R}$, 2SUM, 3SUM, k-sum, .

Mục lục

1 Introduction to 2 Pointers Method – Giới Thiệu Phương Pháp 2 Con Trỏ

Resources – Tài nguyên.

1. ANTTI LAAKSONEN. *Competitive Programmer's Handbook*. Chap. 8: Amortized Analysis. Sect. 8.1: 2 pointers method, p. 77.
2. DARREN YAO, QI WANG, RYAN CHOU. USACO Guide/2 pointers.

   Abstract. Iterating 2 monotonic pointers across an array to search for a pair of indices satisfying some condition in linear time. – Lặp lại 2 con trỏ đơn điệu trên 1 mảng để tìm kiếm 1 cặp chỉ số thỏa mãn 1 số điều kiện trong thời gian tuyến tính.
3. Geeks for Geeks/2 pointers technique.

*A scientist- & creative artist wannabe, a mathematics & computer science lecturer of Department of Artificial Intelligence & Data Science (AIDS), School of Technology (SOT), UMT Trường Đại học Quản lý & Công nghệ TP.HCM, Hồ Chí Minh City, Việt Nam.
E-mail: nguyenquanbahong@gmail.com & hong.nguyenquanba@umt.edu.vn. Website: https://nqbh.github.io/. GitHub: https://github.com/NQBH.

4. Phan Đình Khôi. VNOI/kỹ thuật 2 con trỏ.

The 2 pointers method iterates 2 pointers across an array, to track the start & end of an interval. It can also be used to track 2 values in an array.

– Phương thức 2 con trỏ lặp lại 2 con trỏ trên 1 mảng để theo dõi điểm bắt đầu và kết thúc của 1 khoảng. Phương thức này cũng có thể được sử dụng để theo dõi 2 giá trị trong 1 mảng.

General ideas of 2 pointers method. In the 2 pointer method, 2 pointers are used to iterate through the array values. Both pointers can move to 1 direction only, which ensures that the algorithm works efficiently.

– *Ý tưởng tổng quan của phương pháp 2 con trỏ.* Trong phương pháp 2 con trỏ, 2 con trỏ được sử dụng để lặp qua các giá trị mảng. Cả hai con trỏ chỉ có thể di chuyển theo 1 hướng, điều này đảm bảo thuật toán hoạt động hiệu quả.

Cho 1 mảng $a = \{a[i]\}_{i=0}^{n-1}$. Gọi l, r lần lượt là con trỏ trái & con trỏ phải (left- & right pointers, respectively). Thay đổi l, r để tìm kiếm trên mảng a, có thể kết hợp với sắp xếp trước đó, yêu cầu của bài toán.

Analysis of Algorithm Complexity . The running time of the 2 pointers algorithm depends on the number of steps the right pointer moves. While there is no useful upper bound on how many steps the pointer can move on a *single* turn, we know that the pointer moves a *total* of $O(n)$ steps during the algorithm, because it only moves to the right. Since both the left & right pointers move $O(n)$ steps during the algorithm, the algorithm works in $O(n)$ time.

– *Phân tích Độ phức tạp Thuật toán.* Thời gian chạy của thuật toán 2 con trỏ phụ thuộc vào số bước mà con trỏ phải di chuyển. Mặc dù không có giới hạn trên hữu ích nào về số bước con trỏ có thể di chuyển trong 1 lượt, chúng ta biết rằng con trỏ di chuyển tổng cộng $O(n)$ bước trong suốt thuật toán, vì nó chỉ di chuyển sang phải. Vì cả con trỏ trái và phải đều di chuyển $O(n)$ bước trong suốt thuật toán, thuật toán hoạt động trong thời gian $O(n)$.

Problem 1 (Codeforces/books)**.** *When Valera has got some free time, he goes to the library to read some books. Today he's got t free minutes to read. That's why Valera took n books in the library & for each book he estimated the time he is going to need to read it. Let's number the books by integers from 1 to n. Valera needs a_i minutes to read the ith book.*

Valera decided to choose an arbitrary book with number i & read the books 1 by 1, starting from this book. I.e., he will 1st read book number i, then book number $i+1$, then book number $i+2$ & so on. He continues the process until he either runs out of the free time or finishes the nth book. Valera reads each book up to the end, i.e., he does not start reading the book if he doesn't have enough free time to finish reading it. Print the maximum number of books Valera can read.

Input. *The 1st line contains 2 integers $n \in [10^5], t \in [10^9]$ – the number of books & the number of free minutes Valera's got. The 2nd line contains a sequence of n integers $a_1, a_2, \ldots, a_n$, $a_i \in [10^4]$, $\forall i \in [n]$, where number a_i shows the number of minutes that the boy needs to read the ith book.*

Output. *Print a single integer – the maximum number of books Valera can read.*

Sample.

book.inp	book.out
4 5 3 1 2 1	3
3 3 2 2 3	1

Bài toán 1 (Sách)**.** *Khi Valera có chút thời gian rảnh, anh ấy đến thư viện để đọc sách. Hôm nay anh ấy có t phút rảnh để đọc. Đó là lý do tại sao Valera đã mượn n cuốn sách trong thư viện & với mỗi cuốn sách, anh ấy ước tính thời gian cần thiết để đọc hết. Hãy đánh số các cuốn sách theo số nguyên từ 1 đến n. Valera cần a_i phút để đọc cuốn sách thứ i.*

Valera quyết định chọn 1 cuốn sách bất kỳ có số i & đọc từng cuốn sách một, bắt đầu từ cuốn sách này. Tức là, anh ấy sẽ đọc cuốn sách số i trước, sau đó là cuốn sách số $i+1$, rồi cuốn sách số $i+2$ & cứ tiếp tục như vậy. Anh ấy tiếp tục quá trình này cho đến khi hết thời gian rảnh hoặc đọc hết cuốn sách thứ n. Valera đọc hết mỗi cuốn sách cho đến hết, tức là anh ấy sẽ không bắt đầu đọc cuốn sách nếu không có đủ thời gian rảnh để đọc hết. In ra số lượng sách tối đa mà Valera có thể đọc.

Đầu vào. *Dòng đầu tiên chứa 2 số nguyên $n \in [10^5], t \in [10^9]$ – số lượng sách & số phút miễn phí mà Valera có. Dòng thứ 2 chứa 1 dãy n số nguyên $a_1, a_2, \ldots, a_n$, $a_i \in [10^4]$, $\forall i \in [n]$, trong đó a_i cho biết số phút mà cậu bé cần để đọc cuốn sách thứ i.*

Đầu ra. *In ra 1 số nguyên duy nhất – số lượng sách tối đa mà Valera có thể đọc.*

1st solution: 2 pointers method. We want to find the longest contiguous segment of books that can be read within t minutes. To accomplish this, we can define `left, right` to represent the beginning & end of the segment. Both will start at the beginning of the array. These numbers can be thought of as pointers, hence the name "2 pointers".

– Chúng ta muốn tìm đoạn sách liền kề dài nhất có thể đọc được trong vòng t phút. Để thực hiện điều này, chúng ta có thể định nghĩa `left, right` để biểu diễn điểm đầu & điểm cuối của đoạn sách. Cả hai đều bắt đầu từ đầu mảng. Những con số này có thể được coi là các con trỏ, do đó có tên là "2 con trỏ".

For every value of `left` in increasing order, let's increase `right` until the total time for the segment is maximized while being $\leq t$. `ans` will store the maximum value of `right` $-$ `left` $+ 1$ (segment size) that we have encountered so far. After increasing `left` by 1, the time used decreases, hence the right pointer never has to move leftwards. Since both pointers will move at most n times, the overall time complexity is $O(n)$.

– Với mỗi giá trị của `left` theo thứ tự tăng dần, tăng `right` cho đến khi tổng thời gian cho đoạn thẳng đạt giá trị cực đại trong khi vẫn giữ nguyên $\leq t$. `ans` sẽ lưu trữ giá trị tối đa của `right` $-$ `left` $+ 1$ (kích thước đoạn thẳng) mà chúng ta đã gặp

cho đến nay. Sau khi tăng `left` thêm 1, thời gian sử dụng sẽ giảm xuống, do đó con trỏ phải không bao giờ phải di chuyển sang trái. Vì cả hai con trỏ sẽ di chuyển tối đa n lần, nên độ phức tạp thời gian tổng thể là $O(n)$.

C++ implementation:

1. USACO Guide's C++ implementation: book: sliding window idea, time complexity $O(n)$.

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, t;
    cin >> n >> t;
    vector<int> a(n);
    for (int& v : a) cin >> v;
    int r = -1, sum = 0, ans = 0;
    for (int l = 0; l < n; sum -= a[l++]) {
        while (r + 1 < n && sum + a[r + 1] <= t) sum += a[++r];
        ans = max(ans, r - l + 1);
    }
    cout << ans << '\n';
}
```

2. NQBH's C++ implementation: book: TLE test 9 CodeForces

```
#include <iostream>
using namespace std;

int main() {
    int n, t, count, sum, l, r, ans = 0;
    cin >> n >> t;
    int a[n];
    for (int i = 0; i < n; ++i) cin >> a[i];
    for (l = 0; l < n; ++l) { // count max number of books that can be read from l-th book
        count = 0; // reset count
        sum = 0; // reset sum from point l to pointer r
        for (r = l; r < n; ++r) {
            sum += a[r];
            if (sum > t) break;
            else ++count;
        }
        ans = max(ans, count);
    }
    cout << ans;
}
```

□

Remark 1 (Cf. 2 pointers method vs. sliding method method)**.** *You can visualize these pointers as maintaining a sliding window of books for this problem. – Có thể hình dung các con trỏ này như đang duy trì 1 cửa sổ sách trượt cho vấn đề này.*

Phương pháp 2 con trỏ rất liên quan với phương pháp cửa sổ trượt vì 2 con trỏ này đánh dấu biên trái & biên phải của cửa sổ trượt, còn cửa sổ trượt thì nắm tất cả thông tin về mảng hoặc chuỗi nằm giữa 2 con trỏ & bao gồm cả ở vị trí 2 con trỏ. 1 cách nom na, có thể hiểu phương pháp 2 con trỏ là phần biên của phần nội dung của phương pháp cửa sổ trượt.

Bài toán 2 (Monotone sequence concatenation – Ghép dãy đơn điệu)**.** *Cho 2 dãy số nguyên đã được sắp xếp không giản $a = \{a_i\}_{i=1}^{m}, b = \{b_i\}_{i=1}^{n}$ lần lượt có $n, m \in \mathbb{N}$ phần tử. Ghép chúng thành dãy $c = \{c_i\}_{i=1}^{m+n}$ được bố trí theo thứ tự không giảm.*

Constraint. $m, n \in [10^5], a_i, b_j \in \overline{0, 10^9}$, $\forall i \in [m], j \in [n]$.

Sample.

monotone_sequence_concatenation.inp	monotone_sequence_concatenation.out
5 6 1 3 6 8 10 2 6 7 12 14 15	1 2 3 6 6 7 8 10 12 14 15

Solution.

Phân tích. Vì 3 dãy a, b, c không giảm, i.e., $a_1 \leq a_2 \leq \cdots a_m, b_1 \leq b_2 \leq \cdots \leq b_n, c_1 \leq c_2 \leq \cdots \leq c_{m+n}$, nên $c_1 = \min\{a_1, b_1\}$. Tại mỗi thời điểm, phần tử tiếp theo của dãy c sẽ là phần tử nhỏ nhất trong các phần tử chưa được đưa vào dãy c, nên bằng cách so sánh phần tử nhỏ nhất chưa được chọn ở dãy a & phần tử nhỏ nhất chưa được chọn ở dãy b, phần tử nhỏ hơn sẽ được chọn vào dãy c. Ban đầu, lúc dãy c chưa có phần tử nào, i.e., $c = \emptyset$ thì a_1, b_1 lần lượt là phần tử nhỏ nhất chưa được chọn của 2 dãy a, b. Tại mỗi thời điểm, nếu đưa phần tử a_i vào dãy c thì phần tử nhỏ nhất chưa được chọn của dãy a sẽ là a_{i+1}, còn nếu đưa phần tử b_j vào dãy c thì phần tử nhỏ nhất chưa được chọn của dãy b sẽ là b_{j+1}.

Giải pháp. Dựa vào các phân tích trên ta có giải pháp sử dụng 2 con trỏ như sau:

- Dãy a có con trỏ i, con trỏ này bắt đầu ở vị trí đầu dãy a & được thể hiện như phần tử nhỏ nhất chưa được chọn trong dãy a.
- Dãy b có con trỏ j, con trỏ này bắt đầu ở vị trí đầu dãy b & được thể hiện như phần tử nhỏ nhất chưa được chọn trong dãy b.
- Ta sẽ lặp lại công việc này, cho đến khi đưa hết các phần tử trong 2 dãy a, b vào dãy c. Nếu tất cả các phần tử trong 1 trong 2 dãy a, b đều đã được đưa vào dãy c thì đưa lần lượt các phần tử trong dãy còn lại vào dãy c. Ngược lại, so sánh 2 phần tử ở 2 con trỏ, đưa phần tử có giá trị nhỏ hơn vào dãy c, nếu 2 phần tử có giá trị như nhau thì chọn 1 trong 2, rồi tăng vị trí ở phần tử được đưa vào lên 1 đơn vị.

Độ phức tạp. Vị trí con trỏ i luôn tăng & tăng không quá n lần, vị trí con trỏ j cũng luôn tăng & tăng không quá m lần, nên độ phức tạp của thuật toán là $O(m + n)$.

C++ implementation:

1. NQBH's C++ implementation: monotone sequence concatenation:

```
#include <iostream>
using namespace std;

int main() {
    int m, n;
    cin >> m >> n;
    int a[m], b[n], c[m + n];
    for (int i = 0; i < m; ++i) cin >> a[i];
    for (int i = 0; i < n; ++i) cin >> b[i];
    int idx_a = 0, idx_b = 0; // 2 pointers
    for (int i = 0; i < m + n; ++i) {
        if (a[idx_a] <= b[idx_b]) {
            c[i] = a[idx_a];
            ++idx_a;
        }
        else {
            c[i] = b[idx_b];
            ++idx_b;
        }
    }
    for (int i = 0; i < m + n; ++i) cout << c[i] << ' ';
}
```

or shorter:

```
#include <iostream>
using namespace std;

int main() {
    int m, n;
    cin >> m >> n;
    int a[m], b[n], c[m + n];
    for (int i = 0; i < m; ++i) cin >> a[i];
    for (int i = 0; i < n; ++i) cin >> b[i];
    int idx_a = 0, idx_b = 0; // 2 pointers
    for (int i = 0; i < m + n; ++i)
        if (a[idx_a] <= b[idx_b]) c[i] = a[idx_a++];
        else c[i] = b[idx_b++];
    for (int i = 0; i < m + n; ++i) cout << c[i] << ' ';
}
```

2. VNOI's C++ implementation: monotone sequence concatenation:

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n, m;
    cin >> n >> m;
    vector<int> a(n), b(m), c;
    for (int& v : a) cin >> v;
    for (int& v : b) cin >> v;
    int i = 0, j = 0;
    while (i < n || j < m)
        if (j == m || (i < n && a[i] <= b[j])) c.push_back(a[i++]);
        else c.push_back(b[j++]);
    for (auto it : c) cout << it << " ";
}
```

□

Remark 2. *Bài toán này giống như 1 đứa trẻ nhảy lò cò giữa 2 dãy ô* $a = a_1, a_2, \ldots, a_n$ *&* $b = b_1, b_2, \ldots, b_m$ *được xếp như hình vẽ: nếu* $m > n$

a_1	a_2	$\cdots$	a_n	$\cdots$	a_m
b_1	b_2	$\cdots$	b_n		

nếu $n > m$*:*

a_1	a_2	$\cdots$	a_n		
b_1	b_2	$\cdots$	b_n	$\cdots$	b_m

Nhiệm vụ của đứa trẻ khá đơn giản: chỉ cần nhảy vào ô nhỏ nhất giữa ô trước mặt & ô đầu tiên của cột bên kia là được & nếu 2 ô này có giá trị bằng nhau thì nhảy ô nào trước cũng được. Khi đó vết nhảy của đứa trẻ chính là dãy c được ghép từ 2 dãy a, b *theo thứ tự không giảm.*

1.1 Subarray sum – Tổng các phần tử của mảng con

Resources – Tài nguyên.

1. Geeks for Geeks/subarray with given sum.

Problem 2 (Subarray sum on $\mathbb{N}$ – Tổng mảng con trên $\mathbb{N}$)**.** *Given an array* $\{a_i\}_{i=1}^n \subset \mathbb{N}^\star$ *(1-based indexing) of* n *positive integers & a target sum* $x \in \mathbb{N}^\star$*. Find a subarray whose sum is* x *or report that there is no such subarray.*

– Cho 1 mảng $\{a_i\}_{i=1}^n \subset \mathbb{N}^\star$ *(đánh chỉ số dựa trên 1) gồm* n *số nguyên dương & 1 tổng đích* x*. Tìm 1 mảng con có tổng là* x *hoặc báo cáo rằng không có mảng con nào như vậy.*

We can reformulate the problem mathematically as follows:

$$\text{Find } l, r \in [n],\ l \leq r \text{ s.t. } \sum_{i=l}^{r} a_i = x.$$

Solution. This problem can be solved in $O(n)$ time (i.e., linear time) by using the 2 pointers method. The idea is to maintain pointers that point to the 1st & last value of a subarray. On each turn, the left pointer moves 1 step to the right, & the right pointer moves to the right as long as the resulting subarray sum is at most x. If the sum becomes exactly x, a solution has been found.

– Bài toán này có thể được giải quyết trong thời gian $O(n)$ (tức là thời gian tuyến tính) bằng cách sử dụng phương pháp 2 con trỏ. Ý tưởng là duy trì các con trỏ trỏ đến giá trị đầu tiên & cuối cùng của 1 mảng con. Mỗi lần dịch chuyển, con trỏ trái di chuyển 1 bước sang phải, & con trỏ phải di chuyển sang phải miễn là tổng mảng con thu được không quá x. Nếu tổng bằng đúng x, thì đã tìm được nghiệm.

C++ implementation:

1. NQBH's C++ implementation: subarray sum:

```cpp
#include <iostream>
using namespace std;

int main() {
```

```
    int n, x;
    cin >> n >> x;
    int a[n];
    for (int i = 0; i < n; ++i) {
        cin >> a[i];
        if (a[i] == x) {
            cout << i << ' ' << i;
            return 0;
        }
    }
    int l = 0, r = 1; // left- & right pointers
    int sum = a[0] + a[1];
    while (sum != x) {
        if (sum < x && r == n - 1) {
            cout << "Impossible";
            return 0;
        }
        if (sum < x && r < n - 1) sum += a[++r];
        if (sum > x && l < n - 1) sum -= a[l++];
    }
    cout << l << ' ' << r;
}
```

□

A little bit different version of the above problem:

Problem 3 (Subarray sum – Tổng các phần tử của mảng con)**.** *Given a* 1*-based indexing array* `arr[]` *of nonnegative integers & an integer* x*. Return the left & right indexes (*1*-based indexing) of a subarray whose sum of its elements is equal to* x*. In case of multiple satisfying subarrays, return the subarray indexes which come 1st on moving from left to right. If no such subarray exists, return* -1*.*

– Cho 1 mảng lập chỉ mục dựa trên 1 `arr[]` *gồm các số nguyên không âm & 1 số nguyên* x*. Trả về các chỉ mục trái & phải (lập chỉ mục dựa trên* 1*) của 1 mảng con có tổng các phần tử bằng* x*. Trong trường hợp có nhiều mảng con thỏa mãn, trả về các chỉ mục mảng con đứng đầu tiên khi di chuyển từ trái sang phải. Nếu không tồn tại mảng con nào như vậy, trả về* -1*.*

Sample.

subarray_sum.inp	subarray_sum.out
8 23 15 2 4 8 9 5 10 23	2 5
6 7 1 10 4 0 3 5	3 5
2 0 1 4	−1

1st solution: Naive approach: using nested loop $O(n^2)$ *time &* $O(1)$ *space – Cách tiếp cận ngây thơ: sử dụng vòng lặp lồng nhau* $O(n^2)$: Use a nested loop where the outer loop picks a starting element, & the inner loop calculates the cumulative sum of elements starting from this element. For each starting element, the inner loop iterates through subsequent elements & adding each element to the cumulative sum until the given sum is found or the end of the array is reached. If at any point the cumulative sum equals the given sum, then return starting & ending indices (1-based). If no such sub-array is found after all iterations, then return -1.

– Sử dụng vòng lặp lồng nhau, trong đó vòng lặp ngoài chọn 1 phần tử bắt đầu, & vòng lặp trong tính tổng tích lũy của các phần tử bắt đầu từ phần tử này. Với mỗi phần tử bắt đầu, vòng lặp trong sẽ lặp qua các phần tử tiếp theo & cộng từng phần tử vào tổng tích lũy cho đến khi tìm được tổng cho trước hoặc đến cuối mảng. Nếu tại bất kỳ điểm nào, tổng tích lũy bằng tổng cho trước, thì trả về chỉ số bắt đầu & kết thúc (dựa trên 1). Nếu không tìm thấy mảng con nào như vậy sau tất cả các lần lặp, thì trả về -1.

C++ implementation:

1. G4G's C++ implementation: https://www.geeksforgeeks.org/dsa/find-subarray-with-given-sum/:

```
#include <iostream>
#include <vector>
using namespace std;

// function to find a continuous subarray whose sum = a given number
vector<int> subarray_sum_naive(vector<int> arr, int target) {
    vector<int> res;
```

```
    int n = arr.size();
    for (int s = 0; s < n; ++s) { // pick a staarting point for a subarray
        int curr = 0; // current sum from starting & ending indices
        // consider all ending points for the picked starting point
        for (int e = s; e < n; ++e) {
            curr += arr[e];
            if (curr == target) {
                res.push_back(s + 1);
                res.push_back(e + 1);
                return res;
            }
        }
    }
    return {-1}; // if no subarray is found
}

int main() {
    int n, x;
    cin >> n >> x;
    vector<int> a(n);
    for (int& v : a) cin >> v;
    vector<int> res = subarray_sum_naive(a, x);
    for (int ele : res) cout << ele << " ";
}
```

C#: G4G's C#: subarray sum: https://www.geeksforgeeks.org/dsa/find-subarray-with-given-sum/:

```
using System;
using System.Collections.Generic;

class GfG {
    // function to find a continuous subarray which adds up to a given number
    static List<int> subarray_sum(int[] arr, int target) {
        List<int> res = new List<int>();
        int n = arr.Length;
        for (int s = 0; s < n; ++s) { // pick a starting point for a subarray
            int curr = 0;
            // consider all ending points for the picked starting point
            for (int e = s; e < n; ++e) {
                curr += arr[e];
                if (curr == target) {
                    res.Add(s + 1);
                    res.Add(e + 1);
                    return res;
                }
            }
        }
        res.Add(-1); // if no satisfying subarray is found
        return res;
    }

    static void Main() {
        int[] arr = {15, 2, 4, 8, 9, 5, 10, 23};
        int target = 23;
        List<int> res = subarray_sum(arr, target);
        foreach (var ele in res)
        Console.Write(ele + " ");
    }
}
```

Java: G4G's Java: subarray sum: https://www.geeksforgeeks.org/dsa/find-subarray-with-given-sum/:

```
import java.util.ArrayList;
import java.util.List;

```

```
class GfG {
    // function to find a continuous subarray whose sum = a given number
    static ArrayList<Integer> subarray_sum(int[] arr, int target) {
        ArrayList<Integer> res = new ArrayList<>();
        int n = arr.length;
        for (int s = 0; s < n; ++s) { // pick a starting point for a subarray
            int curr = 0;
            // consider all ending points for the picked starting point
            for (int e = s; e < n; ++e) {
                curr += arr[e];
                if (curr == target) {
                    res.add(s + 1);
                    res.add(e + 1);
                    return res;
                }
            }
        }
        res.add(-1); // if no satisfying subarray is found
        return res;
    }

    public static void main(String[] args) {
        int[] arr = {15, 2, 4, 8, 9, 5, 10, 23};
        int target = 23;
        ArrayList<Integer> res = subarray_sum(arr, target);
        for (int ele : res)
        System.out.print(ele + " ");
    }
}
```

Java Script: G4G's Java Script: subarray sum: https://www.geeksforgeeks.org/dsa/find-subarray-with-given-sum/:

```
// function to find a continuous subarray whose sum = a given number
function subarraySum(arr, target) {
    let res = [];
    let n = arr.length;

    // Pick a starting point for a subarray
    for (let s = 0; s < n; s++) {
        let curr = 0;

        // Consider all ending points
        // for the picked starting point
        for (let e = s; e < n; e++) {
            curr += arr[e];
            if (curr === target) {
                res.push(s + 1);
                res.push(e + 1);
                return res;
            }
        }
    }
    // If no subarray is found
    return [-1];
}

// Driver Code
let arr = [15, 2, 4, 8, 9, 5, 10, 23];
let target = 23;
let res = subarraySum(arr, target);

console.log(res.join(' '));
```

Python:

1. G4G's Python: subarray sum: https://www.geeksforgeeks.org/dsa/find-subarray-with-given-sum/:

```
# function to find a continuous subarray which adds up to a given number
def subarray_sum(arr, target):
    res = []
    n = len(arr)
    for s in range(n): # pick a starting point for a subarray
        curr = 0 # initialize current calculated sum
        for e in range(s, n): # consider all ending points for the picked starting point
            curr += arr[e]
            if curr == target:
                res.append(s + 1)
                res.append(e + 1)
                return res
    return [-1] # if no subarray is found

if __name__ == "__main__":
    n, x = map(int, input().split())
    arr = list(map(int, input().split()))
    res = subarray_sum(arr, x)
    for ele in res:
        print(ele, end = " ")
```

□

2nd solution: Sliding Window: $O(n)$ time & $O(1)$ space. As we know that all the elements in the given subarray are positive, if a subarray has sum greater than the given sum then there is no possibility that adding elements to the current subarray will be equal to the given sum. Hence the idea is to use a similar approach to a sliding window.

- Start with an empty window.
- Add elements to the window while the current sum is less than the target sum.
- If the sum is greater than the target sum, remove elements from the start of the current window.
- If the current sum is equal to the target sum, return the result.

– *Cửa sổ trượt: $O(n)$ thời gian & $O(1)$ không gian.* Như chúng ta đã biết, tất cả các phần tử trong mảng con đã cho đều dương, nếu 1 mảng con có tổng lớn hơn tổng đã cho thì không có khả năng việc thêm các phần tử vào mảng con hiện tại sẽ bằng tổng đã cho. Do đó, ý tưởng là sử dụng 1 cách tiếp cận tương tự như cửa sổ trượt.

- Bắt đầu với 1 cửa sổ trống.
- Thêm các phần tử vào cửa sổ khi tổng hiện tại nhỏ hơn tổng mục tiêu.
- Nếu tổng lớn hơn tổng mục tiêu, xóa các phần tử khỏi đầu cửa sổ hiện tại.
- Nếu tổng hiện tại bằng tổng mục tiêu, trả về kết quả.

C++ implementation:

1. G4G's C++ implementation: subarray sum by sliding window:

```
#include <iostream>
#include <vector>
using namespace std;

vector<int> subarray_sum_sliding_window(vector<int>& arr, int target) {
    int s = 0, e = 0; // initialize window
    vector<int> res;
    int curr = 0; // current calculated sum
    for (size_t i = 0; i < arr.size(); ++i) {
        curr += arr[i];
        // if current sum becomes >=, set end & try adjusting start
        if (curr >= target) {
            e = i;
            // while current is greater, remove starting elements of current window
            while (curr > target && s < e)
            curr -= arr[s++];
            if (curr == target) { // if found a satisfying subarray
                res.push_back(s + 1);
```

```
                res.push_back(e + 1);
                return res;
            }
        }
    }
    return {-1}; // if no satisfying subarray is found
}

int main() {
    int n, x;
    cin >> n >> x;
    vector<int> a(n);
    for (int& v : a) cin >> v;
    vector<int> res = subarray_sum_sliding_window(a, x);
    for (int ele : res) cout << ele << " ";
}
```

□

Problem 4 (Subarray sum on $\mathbb{Z}$)**.** *Given an unsorted array of integers $\{a_i\}_{i=1}^n \subset \mathbb{Z}$, find a subarray that adds to a given number $x \in \mathbb{Z}$. (a) If there is more than 1 satisfying subarray, print any of them. (b) Print all satisfying subarrays.*

Sample.

subarray_sum_negative.inp	subarray_sum_negative.out
6 33 1 4 20 3 10 5	2 4
5 −10 2 12 −2 −20 10	1 3
6 20 −10 0 2 −2 −20 10	−1

Resources – Tài nguyên.

- Geeks for Geeks/subarray with given sum – handles negative numbers.

1st solution: Naive approach: using nested loop: $O(n^2)$ time & $O(1)$ space. □

2nd solution: Prefix Sum & Hash Map: $O(n)$ time & $O(n)$ space. □

3rd solution: Hashing + prefix sum: $O(n)$ time & $O(n)$ space – Băm + tiền tố tổng: $O(n)$ thời gian & $O(n)$ không gian. The above sliding-window solution does not work for arrays with negative numbers. To handle all cases, we use hashing & prefix sum. The idea is to store the sum of elements of every prefix of the array in a hashmap, i.e., every index stores the sum of elements up to that index hashmap. So to check if there is a subarray with sum equal to the target sum, check for every index $i \in [n]$, & sum up to that index as `curr_sum`. If there is a prefix with a sum equal to `curr_sum - target`, then the subarray with the given sum is found.

– Giải pháp cửa sổ trượt ở trên không hoạt động với các mảng có số âm. Để xử lý mọi trường hợp, chúng tôi sử dụng băm & tiền tố tổng. Ý tưởng là lưu trữ tổng các phần tử của mọi tiền tố của mảng trong 1 hashmap, tức là, mỗi chỉ mục lưu trữ tổng các phần tử lên đến hashmap chỉ mục đó. Vì vậy, để kiểm tra xem có mảng con nào có tổng bằng tổng mục tiêu hay không, kiểm tra mọi chỉ mục $i \in [n]$, & tổng đến chỉ mục đó là `curr_sum`. Nếu có 1 tiền tố có tổng bằng `curr_sum - target`, thì mảng con có tổng đã cho sẽ được tìm thấy. □

Remark 3 (Tương quan của bài toán subarray sum với các thuật toán tìm đường đi ngắn nhất trên đồ thị (algorithms for shortest path problems on graphs))**.** *Thuật toán Dijkstra cũng chỉ hoạt động được cho các đồ thị có trọng số không âm, không hoạt động được với các đồ thị có lẫn trọng số âm & trọng số không âm. Chỉ có thuật toán Bellman–Ford mới hoạt động cho đồ thị với trọng số thực tùy ý.*

1.2 Longest subarray with given maximum sum – Mảng con dài nhất với tổng phần tử tối đa cho trước

Bài toán 3 (Longest subarray with given maximum sum – Mảng con dài nhất với tổng phần tử tối đa cho trước)**.** *Cho dãy số nguyên dương $a = \{a_i\}_{i=1}^n \subset \mathbb{N}^\star$ có $n \in \mathbb{N}^\star$ phần tử. Tìm độ dài đoạn con dài nhất trong dãy a sao cho tổng các phần tử trong đoạn này không quá s. Dữ liệu đảm bảo các phần tử trong dãy a đều có giá trị $\leq s$.*

Giới hạn. $n \in [10^6], a_i \in [10^9]$, $\forall i \in [n]$, $s \leq 10^{18}$.

1st solution: 2-pointers method. Gọi $\text{sum}(l,r)$ là tổng các phần tử có chỉ số nằm trong đoạn $[l,r]$, i.e., hàm $\text{sum} : [n]^2 \to \mathbb{N}^\star$, $(l,r) \mapsto \text{sum}(l,r) \coloneqq \sum_{i=l}^{r} a_i = a_l + a_{l+1} + \cdots + a_r$. 1 đoạn con $[l,r]$ được gọi là đoạn con "tốt" nếu $\text{sum}(l,r) \le s$. Khi đó bài toán yêu cầu tìm độ dài của đoạn con "tốt" dài nhất.

Ta có 2 nhận xét quan trọng:

1. Vì dãy a gồm các số nguyên dương nên $\text{sum}(1,r) > \text{sum}(2,r) > \cdots > \text{sum}(r-1,r) > \text{sum}(r,r)$ (tính đơn điệu giảm theo biến thứ nhất). Nếu đoạn con $[l,r]$ là đoạn con "tốt" thì đoạn $[x,r]$ cũng là đoạn con "tốt", $\forall x \ge l$. Nếu đoạn con $[l,r]$ không là đoạn con "tốt" thì với mọi $x \le l$, đoạn $[x,r]$ không là đoạn con "tốt".

2. Với r là 1 vị trí bất kỳ, nếu như l là vị trí nhỏ nhất sao cho đoạn $[l,r]$ là 1 đoạn "tốt" thì đoạn con $[x,r]$ là 1 đoạn con "tốt", $\forall x \ge l$; đoạn con $[x,r]$ không là 1 đoạn "tốt", $\forall x < l$; & đoạn con $[l,r]$ là 1 đoạn con "tốt" dài nhất trong các đoạn con "tốt" có vị trí kết thúc tại r.

Từ đó, với mỗi $r \in [n]$, nếu ta xác định được vị trí l được định nghĩa thông qua hàm $f : [n] \to [n]$ được đặt bởi công thức

$$l \coloneqq f(r) \coloneqq \min\{x \in [r]; [x,r] \text{ là 1 đoạn con "tốt"}\} = \min\{x \in [r]; \text{sum}(x,r) \le s\}. \tag{1}$$

thì ta có thể biết được độ dài của đoạn con "tốt" dài nhất của dãy a. Cụ thể hơn, đoạn con dài nhất có vị trí kết thúc ở $r \in [n]$ có dạng $[f(r), f(r)+1, \ldots, r-1, r]$ & có độ dài bằng $r - f(r) + 1$. Cuối cùng, độ dài của đoạn con "tốt" dài nhất của dãy a là giá trị lớn nhất của độ dài các đoạn con "tốt" dài nhất với vị trí kết thúc từ 1 đến n:

$$\max_{r\in[n]} r - f(r) + 1 = \max_{r\in[n]} r - \min\{x \in [r]; [x,r] \text{ là 1 đoạn con "tốt"}\} + 1 = \max_{r\in[n]} r - \min\{x \in [r]; \text{sum}(x,r) \le s\} + 1. \tag{2}$$

Lemma 1. *Hàm $f : [n] \to [n]$ xác định bởi công thức (1) là 1 hàm không giảm.*

Chứng minh. Chứng minh này có thể trình bày ngắn gọn hơn nhưng ở đây tác giả quyết định viết chi tiết để thể hiện cấu trúc định nghĩa của hàm f & ý nghĩa tại sao phải định nghĩa hàm f như vậy. Vì hàm $\text{sum}(\cdot,\cdot)$ là 1 hàm đơn điệu giảm theo biến thứ nhất nên $\text{sum}(x,r) > \text{sum}(f(r),r)$, $\forall x \in [n], x < f(r)$ mà theo định nghĩa của hàm f thì $f(r)$ là chỉ số l nhỏ nhất để $\text{sum}(l,r) \le s$ nên suy ra

$$\text{sum}(f(r),r) \le s < \text{sum}(x,r), \ \forall x \in [n], x < f(r).$$

Kết hợp điều này với việc f là hàm đơn điệu tăng theo biến thứ 2, thu được:

$$\text{sum}(f(r),r) \le s < \text{sum}(x,r) < \text{sum}(x,r+1), \ \forall x \in [n], x < f(r).$$

Theo định nghĩa của hàm f, $f(r+1)$ là chỉ số x nhỏ nhất thỏa $\text{sum}(x,r+1) \le s$, mà đánh giá cuối cho ta $\text{sum}(x,r+1)$, $\forall x \in [n], x < f(r)$, suy ra $f(r+1) \ge f(r)$, $\forall r \in [n]$, i.e., f là 1 hàm khôn giảm. □

Áp dụng bổ này này cho dãy a & vì các phần tử trong dãy a đều có giá trị $\le s$ nên luôn tồn tại vị trí $l \in [n], l \le r$ sao cho đoạn $[l,r]$ là 1 đoạn tốt $\forall r \in [n]$, i.e., hàm f định nghĩa bởi công thức (1) có giá trị xác định $\forall r \in [n]$ nên hàm f được đặt tốt (wellposedness).

Giải pháp. Với các phân tích trên, ta giải quyết bài toán với phương pháp 2 con trỏ như sau:

- Initialization – bước khởi tạo: 2 con trỏ l, r sẽ đặt ở vị trí 1.
- Di chuyển con trỏ r lần lượt từ 1 đến n. Sau mỗi lần di chuyển con trỏ r nếu $\text{sum}(l,r) \le s$ thì giữ nguyên vị trí con trỏ l, còn nếu $\text{sum}(l,r) > s$ thì tăng vị trí con trỏ l cho đến khi $\text{sum}(l,r) \le s$ để đạt được giá trị $l = f(r)$. Khi đó với vị trí $l = f(r)$ & r hiện tại, ta biết đoạn "tốt" dài nhất với vị trí kết thúc tại r là đoạn $[l,r]$.
- Độ dài đoạn con "tốt" dài nhất theo (2) chính là giá trị độ dài lớn nhất của các đoạn "tốt" dài nhất với vị trí kết thúc tại r, với $r \in [n]$.

C++ implementation. Để có thể tính được tổng các phần tử từ l đến r trong khi l, r đang chạy, ta sử dụng biến `sum` để lưu lại tổng của đoạn $[l,r]$ hiện tại – kỹ thuật chính của phương pháp cửa sổ trượt (main technique of sliding window method). Sau khi di chuyển r sang phải, biến `sum` sẽ cộng thêm giá trị $a_r = \texttt{a[r]}$. Trước khi di chuyển l sang phải, biến `sum` sẽ trừ đi giá trị $a_l = \texttt{a[l]}$.

1. VNOI's C++: Longest subarray with given maximum sum https://wiki.vnoi.info/algo/basic/two-pointers:

```
#include <iostream>
using namespace std;

int main() {
    int n, s, ans = 0, sum = 0;
    cin >> n >> s;
    int a[n];
    for (int i = 0; i < n; ++i) cin >> a[i];
    for (int l = 1, r = 1; r <= n; ++r) {
```

```
10          sum += a[r];
11          while (sum > s) sum -= a[l++];
12          ans = max(ans, r - l + 1);
13      }
14      cout << ans;
15  }
```

Độ phức tạp. Vị trí con trỏ r luôn tăng, vị trí con trỏ l luôn tăng & $l \leq r$. Mỗi vị trí l, r tăng không quá n lần, nên độ phức tạp của giải pháp là $O(n)$. □

Question 1. *Liệu phương pháp 2 con trỏ có thể quản lý nổi dãy con của 1 dãy số cho trước – loại dữ liệu không liền kề nhau (non-contiguous data structure types)?*

Bài toán 4 (Longest subsequence with sum $\leq x$ – Dãy con dài nhất với tổng phần tử $\leq x$)**.** *Cho dãy số nguyên dương $a = \{a_i\}_{i=1}^n \subset \mathbb{N}^\star$ có $n \in \mathbb{N}^\star$ phần tử. Tìm độ dài dãy con dài nhất trong dãy a sao cho tổng các phần tử trong dãy con này không quá s.*

1.3 2SUM & 3SUM problem – Bài toán tổng 2 & 3 phần tử

Another problem that can be solved using the 2 pointers method is the following 2SUM problem.

2 pointers is really an easy & effective technique that is typically used for 2 sum in sorted arrays, closest 2 sum, 3 sum, 4 sum, trapping rain water & many other popular interview questions. E.g., given a non-sorted or sorted array `arr` (w.l.o.g., sorted in ascending order by default) & a `target`, find if there exists any pair of elements $(\texttt{arr}[i], \texttt{arr}[j])$ such that their sum is equal to the target.

– 2 con trỏ thực sự là 1 kỹ thuật dễ dàng & hiệu quả, thường được sử dụng cho tổng 2 trong các mảng đã sắp xếp, tổng 2 gần nhất, tổng 3, tổng 4, bẫy nước mưa & nhiều câu hỏi phỏng vấn phổ biến khác. Ví dụ: cho 1 mảng `arr` đã sắp xếp hoặc chưa sắp xếp (không mất tính tổng quát, mặc định được sắp xếp theo thứ tự tăng dần) & 1 `target`, hãy tìm xem có tồn tại cặp phần tử nào $(\texttt{arr}[i], \texttt{arr}[j])$ sao cho tổng của chúng bằng với mục tiêu hay không.

Question 2 (Input of 2SUM problem)**.** *What are all possible inputs of* 2SUM *problem? – Tất cả các đầu vào có thể có của bài toán* 2SUM *là gì?*

Answer. Input của bài toán 2SUM phải là 1 mảng số, về giá trị của các phần tử của mảng này, có thể là số nguyên dương, số nguyên, hoặc số thực. Chú ý bài toán 2SUM trên tập số tự nhiên/nguyên không âm $\mathbb{N}$ không quá khác biệt với bài toán 2SUM trên tập số nguyên dương $\mathbb{N}^\star$ vì nếu dãy $\{a_i\}_{i=1}^n$ có chứa các phần tử bằng 0, ta chỉ cần xét thêm, so với bài toán 2SUM trên tập số nguyên dương $\mathbb{N}^\star$, xem có phần tử a_i nào đúng bằng x không ngay từ lúc nhập mảng a vào để $a_i + 0 = x$.

Về tính sắp xếp, mảng a đầu vào có thể đã được sắp xếp theo thứ tự tăng/giảm dần (không khác biệt lắm cho phương pháp tìm kiếm nhị phân trên các mảng đã sắp xếp – binary search on sorted arrays). Nếu output của bài toán chỉ quan tâm đến cặp giá trị của 2 phần tử của mảng a, thì tính sắp xếp hay không sắp xếp của mảng a không quan trọng: Với mảng a chưa sắp xếp, ta có thể sắp xếp mảng a để tiện hoặc tăng tốc độ tìm kiếm. Nhưng nếu output của bài toán quan tâm đến cặp chỉ số thỏa mãn thì việc mảng a đã sắp xếp đơn điệu hay chưa được sắp xếp sẽ tạo ra sự khác biệt lớn trong thuật giải. □

Question 3 (Output of 2SUM problem)**.** *How deeply do we want to know or obtain when we solve* 2SUM *problem? – Chúng ta muốn biết hoặc thu thập thông tin sâu sắc đến mức nào khi giải bài toán* 2SUM*?*

Answer. Ta có các trường hợp output của bài toán 2 sum như sau:

1. *Chỉ yêu cầu xuất ra 1 cặp giá trị thỏa mãn, không quan tâm 2 chỉ số của chúng*: Nếu ta chỉ cần muốn biết có tồn tại 2 *giá trị* a_i, a_j của mảng mà a_i có thể bằng a_j sao cho $a_i + a_j = x$ thì ta có thể sắp xếp mảng trước mà không cần quan tâm đến việc các chỉ số gốc đã bị xáo trộn trong quá trình sắp xếp.

2. *Yêu cầu xuất ra tất cả cặp giá trị thỏa mãn, không quan tâm các chỉ số của chúng*:

3. *Yêu cầu xuất ra 1 hay tất cả cặp chỉ số (i, j) mà 2 phần tử tại đó thỏa mãn $a_i + a_j = x$*: Nếu sắp xếp lại mảng để việc tìm kiếm được nhanh hơn thì sẽ làm xáo trộn chỉ số của mãn, khiến yêu cầu xuất 2 chỉ số thỏa mãn sẽ gặp khó khăn hoặc không thực hiện được. Nếu bắt buộc phải sắp xếp, cần 1 phương tiện để có thể truy về cặp chỉ số gốc nếu tìm thấy (các) cặp chỉ số (i, j) thỏa mãn.

Nhìn chung, yêu cầu chỉ xuất ra cặp giá trị (a_i, a_j) thỏa mãn $a_i + a_j = x$ sẽ dễ hơn yêu cầu xuất ra cặp chỉ số (i, j) thỏa mãn $a_i + a_j = x$ vì ta có thể sắp xếp để tăng tốc độ tìm kiếm cho yêu cầu trước, nhưng không thể sử dụng các thuật toán sắp xếp 1 cách hiển nhiên cho yêu cầu sau vì sẽ làm xáo trộn bộ chỉ số gốc của mảng chưa được sắp xếp. □

Bài toán 2SUM có nhiều cách phát biểu. Sau đây là 2 cách: 1 kiểu thuật toán & 1 kiểu toán học.

Problem 5 (2SUM)**.** *Given an array of $n \in \mathbb{N}^\star$ numbers & a target sum $x \in \mathbb{N}$, find 2 array values such that their sum is x, or report that no such values exist.*

– Cho 1 mảng gồm $n \in \mathbb{N}^\star$ số & tổng mục tiêu $x \in \mathbb{N}$, tìm 2 giá trị mảng sao cho tổng của chúng là x hoặc báo cáo rằng không tồn tại giá trị nào như vậy.

Given $x \in \mathbb{N}$ & an array $\{a_i\}_{i=1}^n \subset \mathbb{N}$, find $i, j \in [n]$, $i \neq j$ s.t. $a_i + a_j = x$.

Bài toán 5 (2SUM for monotonic arrays/ – Bài toán 2SUM cho mảng/dãy đơn điệu)**.** *Cho 1 mảng số nguyên a có $n \in \mathbb{N}^\star$ phần tử khác nhau đã được sắp xếp tăng dần. Tìm 2 vị trí khác nhau bất kỳ sao cho tổng của 2 phần tử ở 2 vị trí đó có giá trị là x.*

Giới hạn. $n \in \overline{2, 10^6}, a_i, x \in \overline{0, 10^9}$.

Solution.

Phân tích. Tận dụng tính đơn điệu của mảng: Mảng $a = [a_1, a_2, \ldots, a_n]$ gồm n phần tử đã được sắp xếp tăng dần nghĩa là $a_1 < a_2 < \cdots < a_{n-1} < a_n$. Giả sử ta tìm được 1 cặp chỉ số $i, j \in [n]$, $i < j$ thỏa $a_i + a_j > x$ thì $a_k + a_j > a_i + a_j > x$, $\forall k \in [n]$, $k > i$, & $a_i + a_k > a_i + a_j > x$, $\forall k \in [n]$, $k > j$, i.e., nếu ta tìm được 1 cặp chỉ số $(i, j) \in [n]^2$ mà tổng $a_i + a_j > x$ thì không cần quan tâm đến việc tăng 1 trong 2 chỉ số hoặc tăng cả 2 chỉ số i, j nữa. Tương tự, nếu ta tìm được 1 cặp chỉ số $i, j \in [n]$, $i < j$ thỏa $a_i + a_j < x$ thì $a_k + a_j < a_i + a_j < x$, $\forall k \in [n]$, $k < i$, & $a_i + a_k < a_i + a_j < x$, $\forall k \in [n]$, $k < j$, i.e., nếu ta tìm được 1 cặp chỉ số $(i, j) \in [n]^2$ mà tổng $a_i + a_j < x$ thì không cần quan tâm đến việc giảm 1 trong 2 chỉ số hoặc giảm cả 2 chỉ số i, j nữa.

Như vậy, tại 1 thời điểm bất kỳ, các phần tử ta cần quan tâm đến sẽ là các phần tử có chỉ số nằm trong đoạn $[i, j]$ nào đó & đoạn $[i, j]$ này được quản lý bởi 2 con trỏ $i, j \in [n]$. Vài nhận xét:

- Nếu $i = j$, trong mảng a không tồn tại 2 vị trí khác nhau mà tổng của 2 phần tử ở đó bằng x.
- Ngược lại: Nếu $a_i + a_j = x$, ta đã tìm được 2 vị trí cần tìm là i, j. Nếu $a_i + a_j < x$, ta không quan tâm đến a_i nữa & các phần tử ta cần quan tâm đó là các phần tử có chỉ số nằm trong đoạn $[i + 1, j]$. Nếu $a_i + a_j > x$, ta không quan tâm đến a_j nữa & các phần tử ta cần quan tâm đó là các phần tử có chỉ số nằm trong đoạn $[i, j - 1]$.

Giải pháp. Từ các phân tích trên, ta có giải pháp sử dụng 2 con trỏ như sau:

- 1 con trỏ i được đặt ở đầu mảng a, con trỏ j còn lại được đặt ở cuối mảng a.
- Nếu tổng của 2 phần tử ở 2 vị trí con trỏ: $< x$: tăng vị trí con trỏ i lên 1 đơn vị, $> x$: giảm vị trí con trỏ j đi 1 đơn vị.
- Tiếp tục di chuyển đến khi 2 con trỏ gặp nhau.
- Khi con trỏ chưa gặp nhau mà tổng ở 2 vị trị con trỏ bằng x thì ta đã tìm được 2 vị trí cần tìm là i, j, kết thúc chương trình.

Độ phức tạp thuật toán. Vị trí con trỏ i luôn tăng còn vị trí con trỏ j luôn giảm & sự thay đổi vị trí 2 con trỏ này sẽ dừng lại khi tổng 2 phần tử ở 2 vị trí con trỏ có tổng bằng x hay khi $i = j$ nên việc thay đổi vị trí 2 con trỏ sẽ không quá n lần, suy ra độ phức tạp của giải pháp là $O(n)$.

C++ implementation.

```
#include <algorithm>
#include <iostream>
using namespace std;

int main() {
    int n, x;
    cin >> n >> x;
    int a[n];
    for (int i = 0; i < n; ++i) cin >> a[i];
    sort(a, a + n);
    int i = 0, j = n - 1; // 2 pointers
    while (i < j) {
        if (a[i] + a[j] == x) {
            cout << i << " " << j;
            return 0;
        }
        if (a[i] + a[j] < x) ++i;
        else --j;
    }
    cout << "No solution";
}
```

□

Problem 6 (CSES Problem Set/sum of 2 values)**.** *You are given an array of $n \in \mathbb{N}^\star$ integers. Find 2 values (at distinct positions) whose sum is x.*

Input. *The 1st input line has 2 integers $n, x \in \mathbb{N}^\star$: the array size & the target sum. The 2nd line has n integers $a_1, a_2, \ldots, a_n$: the array values.*

Output. *Print 2 integers: the positions of the values. If there are several solutions, you may print any of them. If there are no solutions, print* `IMPOSSIBLE`.

Constraints. $n \in [2 \cdot 10^5], x, a_i \in [10^9], \forall i \in [n]$.

Sample.

sum_2_value.inp	sum_2_value.out
4 8 2 7 5 1	2 4

Bài toán 6 (Tổng 2 giá trị: 2SUM cho mảng số nguyên chưa được sắp xếp)**.** *Bạn được cho 1 mảng gồm $n \in \mathbb{N}^\star$ số nguyên. Tìm 2 giá trị (ở các vị trí phân biệt) có tổng bằng x.*

Đầu vào. *Dòng đầu vào thứ nhất chứa 2 số nguyên $n, x \in \mathbb{N}^\star$: kích thước mảng & tổng mục tiêu. Dòng thứ hai chứa n số nguyên $a_1, a_2, \ldots, a_n$: các giá trị mảng.*

Đầu ra. *In 2 số nguyên: vị trí của các giá trị. Nếu có nhiều nghiệm, bạn có thể in bất kỳ nghiệm nào. Nếu không có nghiệm nào, in* `IMPOSSIBLE`.

Ràng buộc. $n \in [2 \cdot 10^5], x, a_i \in [10^9], \forall i \in [n]$.

1st solution: Naive approach $O(n^2)$ time & $O(1)$ space. The very basic approach is to generate all the possible pairs & check if any of them add up to the target value. To generate all pairs, we simply run 2 nested loops.

– Cách tiếp cận cơ bản nhất là tạo ra tất cả các cặp có thể & kiểm tra xem có cặp nào trong số chúng cộng lại với giá trị mục tiêu hay không. Để tạo ra tất cả các cặp, chúng ta chỉ cần chạy 2 vòng lặp lồng nhau.

C implementation. G4G's C: sum of 2 values: naive approach:

```c
#include <stdbool.h>
#include <stdio.h>

// function to check whether
bool two_sum_naive(int arr[], int n, int target) {
    for (int i = 0; i < n; ++i) // iterate through each element in the array
        for (int j = i + 1; j < n; ++j) // for each element arr[i], check every other element arr[j] that com
            if (arr[i] + arr[j] == target) return true; // check if the sum of the current pair equals the ta
    return false; // if no pair is found after checking all possibilities
}

int main() {
    int arr[] = {0, -1, 2, -3, 1};
    int target = -2;
    int n = sizeof(arr) / sizeof(arr[0]);

    // call the two_sum_naive function & print the result
    if (two_sum_naive(arr, n, target)) printf("true\n");
    else printf("false\n");
    return 0;
}
```

C++ implementation.

1. NQBH's C++ implementation: sum of 2 values: naive approach:

```cpp
#include <iostream>
using namespace std;

int main() {
    int n, x;
    cin >> n >> x;
    int a[n];
    for (int i = 0; i < n; ++i) cin >> a[i];
    for (int i = 0; i < n - 1; ++i)
        for (int j = i + 1; j < n; ++j)
            if (a[i] + a[j] == x) {
                cout << i + 1 << " " << j + 1;
                return 0;
            }
    cout << "IMPOSSIBLE";
}
```

This naive program only passes 18/27 AC, 9/27 TLE on CSES.

Worst-case analysis. +++

2. G4G's C++ implementation: sum of 2 values: naive approach:

```
#include <iostream>
#include <vector>
using namespace std;

bool two_sum_naive(vector<int>& arr, int target) { // O(n^2) time & O(1) space
    int n = arr.size();
    for (int i = 0; i < n; ++i) // consider all pairs (arr[i], arr[j])
        for (int j = i + 1; j < n; ++j)
            if (arr[i] + arr[j] == target) return true; // check if the sum of the current pair equals the
    return false; // if no pair is found after checking all pairs
}

int main() {
    int n, x;
    cin >> n >> x;
    vector<int> a(n);
    for (int& v : a) cin >> v;
    cout << two_sum_naive(a, x);
}
```

C# implementation. G4G's C#: 2sum:

```
using System;

class GfG {
    // function to check whether any pair exists whose sum is equal to the given target value
    static bool two_sum_naive(int[] arr, int target) {
        int n = arr.Length;
        for (int i = 0; i < n; ++i) // iterate through each element in the array
            for (int j = i + 1; j < n; ++j) // for each element arr[i], check every other element arr[j] that
                if (arr[i] + arr[j] == target) return true; // check if the sum of the current pair equals th
        return false; // if no pair is found after checking all possibilities
    }

    static void Main() {
        int[] arr = { 0, -1, 2, -3, 1 };
        int target = -2;

        // call the two_sum_naive & print the result
        if (two_sum_naive(arr, target)) Console.WriteLine("true");
        else Console.WriteLine("false");
    }
}
```

Java implementation. G4G's Java: 2sum:

```
class G4G {
    // function to check whether any pair exists whose sum is equal to the given target value
    static boolean two_sum_naive(int[] arr, int target) {
        int n = arr.length;
        for (int i = 0; i < n; ++i) // iterate through each element in the array
            for (int j = i + 1; j < n; ++j) // for each element arr[i], check every other element arr[j] that
                if (arr[i] + arr[j] == target) return true; // check if the sum of the current pair equals th
        return false; // if no pair is found after checking all possibilities
    }

    public static void main(String[] args) {
        int[] arr = {0, -1, 2, -3, 1};
        int target = -2;
        // call the two_sum_naive & print the result
        if (two_sum_naive(arr, target)) System.out.println("true");
```

```
        else System.out.println("false");
    }
}
```

Java Script implementation. G4G's Java: 2sum:

```
// function to check whether any pair exists whose sum is equal to the given target value
function two_sum_naive(arr, target) {
    let n = arr.length;
    for (let i = 0; i < n; ++i) // iterate through each element in the array
        for (let j = i + 1; j < n; ++j) // for each element arr[i], check every other element arr[j] that com
            if (arr[i] + arr[j] == target) return true; // check if the sum of the current pair equals the ta
    return false; // if no pair is found after checking all possibilities
}

let arr = [0, -1, 2, -3, 1];
let target = -2;

// call the two_sum_naive function & print the result
if (twoSum(arr, target)) console.log("true");
else console.log("false");
```

Python implementation. G4G's Python: 2sum:

```
# function to check whether any pair exists whose sum is equal to the given target value
def two_sum_naive(arr, target):
    n = len(arr)
    for i in range(n): # iterate through each element in the array
        for j in range(i + 1, n): # for each element arr[i], check every other element arr[j] that comes afte
            if arr[i] + arr[j] == target: return True # check if the sum of the current pair equals the targe
    return False # if no pair is found after checking all possibilities

arr = [0, -1, 2, -3, 1]
target = -2

# call the two_sum_naive function & print the result
if two_sum_naive(arr, target): print("true")
else: print("false")
```

□

2nd solution: 2 pointers method. To solve the problem, we 1st sort the array values in increasing order. After that, we iterate through the array using 2 pointers. The left pointer starts at the 1st value & moves 1 step to the right on each turn. The right pointer begins at the last value & always moves to the left until the sum of the left & right value is at most x. If the sum is exactly x, a solution has been found.

– Để giải bài toán, trước tiên chúng ta sắp xếp các giá trị mảng theo thứ tự tăng dần. Sau đó, chúng ta duyệt qua mảng bằng 2 con trỏ. Con trỏ bên trái bắt đầu từ giá trị đầu tiên & dịch chuyển 1 bước sang phải mỗi lần. Con trỏ bên phải bắt đầu từ giá trị cuối cùng & luôn dịch chuyển sang trái cho đến khi tổng của giá trị bên trái & bên phải lớn nhất là x. Nếu tổng bằng đúng x, thì bài toán đã tìm được nghiệm.

The running time of the algorithm is $O(n \log_2 n)$, because it 1st sorts the array in $O(n \log_2 n)$ time, & then both pointers move $O(n)$ steps.

– Thời gian chạy của thuật toán là $O(n \log_2 n)$, vì trước tiên nó sắp xếp mảng trong thời gian $O(n \log_2 n)$, & sau đó cả hai con trỏ di chuyển $O(n)$ bước.

C implementation. G4G's C: 2sum: 2 pointers method:

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// comparison function for qsort
int compare(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
}

// function to check whether any pair exists whose sum is equal to the given target value
bool two_sum_2_pointer(int arr[], int n, int target) {
```

```
    // sort array if not sorted yet
    int left = 0, right = n - 1;
    while (left < right) { // iterate while left pointer < right pointer
        int sum = arr[left] + arr[right];
        if (sum == target) return true; // check if the sum matches the target
        else if (sum < target) ++left; // move left pointer to the right
        else --right; // move right pointer to the left
    }
    return false; // if no pair is found
}

int main() {
    int arr[] = {0, -1, 2, -3, 1};
    int target = -2;
    int n = sizeof(arr) / sizeof(arr[0]);

    // call the two_sum_naive function & print the result
    if (two_sum_2_pointer(arr, n, target)) printf("true\n");
    else printf("false\n");
}
```

C++ implementation.

1. G4G's C++: 2sum: 2 pointers method:

```
#include <iostream>
#include <vector>
using namespace std;

bool two_sum_2_pointers(vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
    while (left < right) {
        int sum = arr[left] + arr[right];
        if (sum == target) return true;
        else if (sum < target) ++left; // move toward a possibly higher sum
        else --right; // move toward a lower sum
    }
    return false; // if no pair is found
}

int main() {
    int n, x;
    cin >> n >> x;
    vector<int> a(n);
    for (int& v : a) cin >> v;
    cout << two_sum_2_pointers(a, x);
}
```

Python implementation. G4G's Python: 2sum: 2 pointers method

```
# function to check whether any pair exists whose sum is equal to the given target value
def two_sum_2_pointer(arr, target):
    arr.sort() # sort array if not sorted yet
    left, right = 0, len(arr) - 1
    while left < right: # iterate while left pointer < right pointer
        sum = arr[left] + arr[right]
        if sum == target: return True # check if the sum matches the target
        elif sum < target: left += 1 # move left pointer to the right
        else: right -= 1 # move right pointer to the left

arr = [0, -1, 2, -3, 1]
target = -2

# call the two_sum_naive function & print the result
if two_sum_naive(arr, target): print("true")
else: print("false")
```

```
if two_sum_2_pointer(arr, target): print("true")
else: print("false")
```

□

It is also possible to solve the problem in another way in $O(n \log_2 n)$ time using binary search. In such a solution, we iterate through the array & for each array value, we try to find another value that yields the sum x. This can be done by performing n binary searches, each of which takes $O(\log_2 n)$ time.

– Cũng có thể giải bài toán theo cách khác trong thời gian $O(n \log_2 n)$ bằng cách sử dụng tìm kiếm nhị phân. Trong giải pháp này, chúng ta lặp qua mảng & với mỗi giá trị mảng, chúng ta cố gắng tìm một giá trị khác cho tổng x. Điều này có thể được thực hiện bằng cách thực hiện n tìm kiếm nhị phân, mỗi lần mất $O(\log_2 n)$ thời gian.

We can use more methods like binary search & hashing to solve 2SUM problem in better time complexity but 2 pointers method is the best solution for 2SUM problem that works well for sorted arrays.

– Chúng ta có thể sử dụng nhiều phương pháp hơn như tìm kiếm nhị phân & băm để giải quyết vấn đề 2SUM với độ phức tạp thời gian tốt hơn nhưng phương pháp 2 con trỏ là giải pháp tốt nhất cho vấn đề 2SUM hoạt động tốt với các mảng đã sắp xếp.

3rd solution: binary search & hashing. □

A more difficult problem is the 3SUM problem that asks to find 3 array values whose sum is x. Using the idea of the above algorithm, this problem can be solved in $O(n^2)$ time.

– 1 bài toán khó hơn là bài toán 3SUM, yêu cầu tìm 3 giá trị mảng có tổng bằng x. Sử dụng ý tưởng của thuật toán trên, bài toán này có thể được giải trong thời gian $O(n^2)$.

Problem 7 (3SUM)**.** *Given an array of $n \in \mathbb{N}^\star$ numbers & a target sum $x \in \mathbb{N}$, find 3 array values such that their sum is x, or report that no such values exist.*

– Cho 1 mảng gồm $n \in \mathbb{N}^\star$ số & tổng mục tiêu $x \in \mathbb{N}$, tìm 3 giá trị mảng sao cho tổng của chúng là x hoặc báo cáo rằng không tồn tại giá trị nào như vậy.

Given $x \in \mathbb{N}$ & an array $\{a_i\}_{i=1}^n \subset \mathbb{N}$, find pairwisely distinct $i, j, k \in [n]$ s.t. $a_i + a_j + a_k = x$.

1st solution: Naive approach. C++ implementation:

1. NQBH's C++ implementation: 3sum: naive approach:

```
#include <iostream>
using namespace std;

int main() {
    int n, x;
    cin >> n >> x;
    int a[n];
    for (int i = 0; i < n; ++i) cin >> a[i];
    for (int i = 0; i < n - 2; ++i)
        for (int j = i + 1; j < n - 1; ++j)
            for (int k = j + 1; k < n; ++k)
                if (a[i] + a[j] + a[k] == x) {
                    cout << i + 1 << " " << j + 1 << " " << k + 1;
                    return 0;
                }
    cout << "IMPOSSIBLE";
}
```

Worst-case analysis. +++

□

Problem 8 (k-sum, $\star$)**.** *Given an array of $n \in \mathbb{N}^\star$ numbers & a target sum $x \in \mathbb{N}$, find $k \in \mathbb{N}^\star$ array values such that their sum is x, or report that no such values exist.*

– Cho 1 mảng gồm $n \in \mathbb{N}^\star$ số & tổng mục tiêu $x \in \mathbb{N}$, tìm $k \in \mathbb{N}^\star$ giá trị mảng sao cho tổng của chúng là x hoặc báo cáo rằng không tồn tại giá trị nào như vậy.

A mathematical formulation of the k-sum problem:

Given $x \in \mathbb{N}$ & an array $\{a_i\}_{i=1}^n \subset \mathbb{N}$, find $\{i_1, i_2, \ldots, i_k\} \subset [n]$ s.t. $\sum_{j=1}^k a_{i_j} = a_{i_1} + a_{i_2} + \cdots + a_{i_k} = x$.

Solution: Use bitmask. +++ □

2 Problems: 2 Pointers Method – Bài Tập: Phương Pháp Phương Pháp 2 Con Trỏ

Bài toán 7. *Cho 1 dãy số nguyên* $\{x_n\}_{n=0}^{\infty}$ *được định nghĩa bởi*

$$\begin{cases} x_0 = 1, \\ x_{n+1} = (ax_n + x_n \ div \ b) \mod c. \end{cases}$$

Tìm $n \in \mathbb{N}$ *nhỏ nhất sao cho tồn tại* $m < n$ *&* $x_m = x_n$*. Dữ liệu đảm bảo* $n \leq 2 \cdot 10^7$*.*

Giới hạn. $a \in [10^4], b, c \in [10^{14}]$.

Sample. $a = 8, b = 2, c = 31$, $\{x_n\}_{n=0}^{\infty} = 1, 8, 6, 20, 15, 3, 25, 26, 4, 3, 25, 26, 4, 3, 25, 26, 4, \ldots$ *giá trị cần tìm là* 9 *vì số* 3 *là giá trị lặp lại đầu tiên trong dãy &* $x_9 = 3$.

1st solution: 2 pointers method. Kết quả của bài toán có ý nghĩa toán học là nếu xảy ra lần lặp lại giá trị đầu tiên của dãy $\{x_n\}_{n=0}^{\infty}$, e.g., $x_m = x_n$ thì xuất ra chỉ số n của số hạng sau. Xét hàm $f : \mathbb{Z} \to \mathbb{Z}$, $f(x) = (ax + x \text{ div } b) \mod c$. Dãy số đã cho có dạng $x_0 = 1, x_n = f(x_{n-1}), \forall n \in \mathbb{N}^{\star}$. Với phép chia lấy dư cho c thì $x_n \in \overline{0, c-1}, \forall n \in \mathbb{N}$ & vì dãy số có vô hạn phần tử nên luôn tồn tại ít nhất 1 cặp chỉ số $m, n \in \mathbb{N}$, $m < n$ thỏa $x_m = x_n$ theo nguyên lý Dirichlet. Hơn nữa, nếu dãy tồn tại $x_m = x_n$ thì dãy sẽ xuất hiện chu kỳ do giá trị lặp được reset lại ở vị trí n y như vị trí m. Cụ thể hơn, gọi $n \in \mathbb{N}$ là chỉ số nhỏ nhất thỏa mãn tồn tại $m \in \mathbb{N}$, $m < x$ để $x_m = x_n$. Khi đó dãy sẽ có chu kỳ độ dài $n - m - 2$ lặp lại các phần tử từ x_m đến x_{n-1}:

$$x_0, x_1, \ldots, x_{m-1}, x_m, x_{m+1}, \ldots, x_{n-1}, x_m, x_{m+1}, \ldots, x_{n-1}, x_m, x_{m+1}, \ldots, x_{n-1}, \ldots$$

Ta có thể giải quyết bài toán nếu biết phần tử bắt đầu chu kỳ x_μ & độ dài chu kỳ λ. Ta có thể tính được giá trị n thỏa mãn bằng bởi công thức $n = \mu + \lambda$.

Giải pháp. Để xác định 2 giá trị μ, λ, ta sử dụng thuật toán Floyd's tortoise & hare, thuật toán rùa & thỏ của Floyd – 1 trong các thuật toán của cycle detection/cycle finding – tìm/phát hiện chu kỳ trong Khoa học Máy tính. Để biết thêm các thuật toán khác cho cycle detection/cycle finding, see, e.g., Wikipedia/cycle detection.

Khởi tạo 2 con trỏ, `tortoise` (rùa) & `hare` (thỏ). Tại mỗi thời điểm, ta tịnh tiến 2 con trỏ này như sau:

- `tortoise` (rùa): tịnh tiến 1 "bước". Nếu hiện tại con trỏ `tortoise` đang là x, rùa sẽ được tịnh tiến đến $f(x)$, i.e., vết đường đi của rùa sẽ là $x_0 \to x_1 \to x_2 \to x_3 \to \cdots$ vì $x_n = f(x_{n-1}), \forall n \in \mathbb{N}^{\star}$. Vì dãy số đã cho có chu kỳ nên ta có công thức tính giá trị của con trỏ `tortoise` sau t lần tịnh tiến:

$$\texttt{tortoise} = \begin{cases} x_t & \text{if } t < \mu, \\ x_{\mu+(t-\mu) \mod \lambda} & \text{if } t \geq \mu. \end{cases}$$

- `hare` (thỏ): tịnh tiến 2 "bước". Nếu hiện tại con trỏ `hare` đang là x, thỏ sẽ được tịnh tiến đến $f(f(x))$, i.e., vết đường đi của thỏ sẽ là $x_0 \to x_2 \to x_4 \to x_6 \to \cdots$. Vì dãy số đã cho có chu kỳ nên ta có công thức tính giá trị của con trỏ `hare` sau t lần tịnh tiến:

$$\texttt{hare} = \begin{cases} x_{2t} & \text{if } t < \dfrac{\mu}{2}, \\ x_{\mu+(2t-\mu) \mod \lambda} & \text{if } t \geq \dfrac{\mu}{2}. \end{cases}$$

Ngoài lúc ban đầu tại vị trí 0, 2 con trỏ `tortoise, hare` sẽ luôn gặp nhau tại thời điểm nào đó. Thật vậy:

- Nếu $t < \frac{\mu}{2}$: Sau t lần tịnh tiến, `tortoise` $= x_t$, `hare` $= x_{2t}$. Tuy nhiên, $\mu + \lambda$ mới bắt đầu lại chu kỳ nên các phần tử từ x_0 đến $x_{\mu+\lambda-1}$ phải đôi một khác nhau, nên $x_t \neq x_{2t}$, `tortoise, hare` chưa gặp nhau lúc này.
- $t \in \left[\frac{\mu}{2}, \mu\right)$: Sau t lần tịnh tiến, `tortoise` $= x_t$, `hare` $= x_{\mu+(2t-\mu) \mod \lambda}$. Tuy nhiên, $\mu + \lambda$ mới bắt đầu lại chu kỳ nên các phần tử từ x_0 đến $x_{\mu+\lambda-1}$ phải đôi một khác nhau, nên $x_t \neq x_{\mu+(2t-\mu) \mod \lambda}$, `tortoise, hare` chưa gặp nhau lúc này.
- $t \geq \mu$: Sau t lần tịnh tiến, `tortoise` $= x_{\mu+(t-\mu) \mod \lambda}$, `hare` $= x_{\mu+(2t-\mu) \mod \lambda}$. Giả sử `tortoise, hare` gặp nhau thì 2 chỉ số phải giống nhau: $\mu + (t - \mu) \mod \lambda = \mu + (2t - \mu) \mod \lambda \Leftrightarrow (2t - \mu) - (t - \mu) \mod \lambda = t \mod \lambda = 0$. Nên `tortoise, hare` sẽ gặp nhau sau t lần tịnh tiến, với $t \geq \mu$ & $t \vdots \lambda$. Trừ lúc khởi tạo, 2 con trỏ `tortoise, hare` sẽ gặp nhau khi giá trị của 2 con trỏ bằng $x_{\mu+(\lambda-\mu \mod \lambda) \mod \lambda}$.

Cách cài đặt để rùa & thỏ gặp nhau:

```
int tortoise = 1, hare = 1;
while (true) {
    tortoise = f(tortoise);
    hare = f(f(hare));
    if (tortoise == hare) break;
}
```

Tìm μ. Khởi tạo 1 con trỏ mới `p` $= x_0 = 1$, con trỏ này được tịnh tiến giống như con trỏ `tortoise`.

Lemma 2. *Tịnh tiến cùng lúc 2 con trỏ* `p, tortoise` *& dừng lại cho đến khi chúng gặp nhau. Số lần tịnh tiến bằng* μ.

Chứng minh. □

□

Problem 9 (Subarray Sums I)**.**

Problem 10 (Sum of Three Values)**.**

Problem 11 (Paired Up)**.**

Problem 12 (Cellular Network)**.**

Problem 13 (They Are Everywhere)**.**

Problem 14 (Quiz Master)**.**

Problem 15 (Diamond Collector)**.**

Problem 16 (Sleepy Cow Herding)**.**

Problem 17 (An impassioned circulation of affection)**.**

VNOJ - NKSGAME CODEFORCES - 1251C CODEFORCES - 1036D

3 Some Extensions of 2 Pointers Method – Vài Mở Rộng của Phương Pháp 2 Con Trỏ

4 Miscellaneous