

2 Pointers Method – Phương Pháp 2 Con Trỏ

Nguyễn Quân Bá Hồng*

Ngày 16 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++: 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.

Mục lục

1 Introduction to 2 Pointers Method – Giới Thiệu Phương Pháp 2 Con Trỏ	1
2 Miscellaneous	8

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/subarray with given sum](#).

4. PHAN ĐÌNH KHÔI. [VNOI/kỹ thuật 2 con trỏ](#).

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).

*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>.

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 $O(n)$.

Bài toán 1 (Monotone sequence concatenation – Ghép dãy đơn điệu). Cho 2 dãy số nguyên đã được sắp xếp không giảm $a = \{a_i\}_{i=1}^m, b = \{b_i\}_{i=1}^n$ 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 2 3 6 6 7 8 10 12 14 15
1 3 6 8 10	
2 6 7 12 14 15	

Solution. Vì 3 dãy a, b, c không giảm, i.e., $a_1 \leq a_2 \leq \dots \leq a_m, b_1 \leq b_2 \leq \dots \leq b_n, c_1 \leq c_2 \leq \dots \leq c_{m+n}$, 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} .

C++:

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

```

1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int m, n;
6      cin >> m >> n;
7      int a[m], b[n], c[m + n];
8      for (int i = 0; i < m; ++i) cin >> a[i];
9      for (int i = 0; i < n; ++i) cin >> b[i];
10     int idx_a = 0, idx_b = 0; // 2 pointers
11     for (int i = 0; i < m + n; ++i) {
12         if (a[idx_a] <= b[idx_b]) {
13             c[i] = a[idx_a];
14             ++idx_a;
15         }
16         else {
17             c[i] = b[idx_b];
18             ++idx_b;
19         }
20     }
21     for (int i = 0; i < m + n; ++i) cout << c[i] << ' ';
22 }
```

or shorter:

```

1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int m, n;
6      cin >> m >> n;
7      int a[m], b[n], c[m + n];
8      for (int i = 0; i < m; ++i) cin >> a[i];
9      for (int i = 0; i < n; ++i) cin >> b[i];
10     int idx_a = 0, idx_b = 0; // 2 pointers
11     for (int i = 0; i < m + n; ++i)
```

```

12         if (a[idx_a] <= b[idx_b]) c[i] = a[idx_a++];
13         else c[i] = b[idx_b++];
14     for (int i = 0; i < m + n; ++i) cout << c[i] << ' ';
15 }

```

□

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

– Cho 1 mảng $\{a_i\}_{i=1}^n \subset \mathbb{N}^*$ (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++:

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

```

1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int n, x;
6      cin >> n >> x;
7      int a[n];
8      for (int i = 0; i < n; ++i) {
9          cin >> a[i];
10         if (a[i] == x) {
11             cout << i << ' ' << i;
12             return 0;
13         }
14     }
15     int l = 0, r = 1; // left- & right pointers
16     int sum = a[0] + a[1];
17     while (sum != x) {
18         if (sum < x && r == n - 1) {
19             cout << "Impossible";
20             return 0;
21         }
22         if (sum < x && r < n - 1) sum += a[++r];
23         if (sum > x && l < n - 1) sum -= a[l++];
24     }
25     cout << l << ' ' << r;
26 }

```

□

A little bit different version of the above problem:

Problem 2 (Subarray sum – Tổng 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: 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 .

– **Cách tiếp cận ngây thơ: sử dụng vòng lặp lồng nhau $O(n^2)$ thời gian & $O(1)$ không gian:** 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++:

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

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  // function to find a continuous subarray whose sum = a given number
6  vector<int> subarray_sum_naive(vector<int> arr, int target) {
7      vector<int> res;
8      int n = arr.size();
9      for (int s = 0; s < n; ++s) { // pick a starting point for a subarray
10         int curr = 0; // current sum from starting & ending indices
11         // consider all ending points for the picked starting point
12         for (int e = s; e < n; ++e) {
13             curr += arr[e];
14             if (curr == target) {
15                 res.push_back(s + 1);
16                 res.push_back(e + 1);
17                 return res;
18             }
19         }
20     }
21     return {-1}; // if no subarray is found
22 }
23
24 int main() {
25     int n, x;
26     cin >> n >> x;
27     vector<int> a(n);
28     for (int& v : a) cin >> v;
29     vector<int> res = subarray_sum_naive(a, x);
30     for (int ele : res) cout << ele << " ";
31 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3
4  class GfG {
5      // function to find a continuous subarray which adds up to a given number
6      static List<int> subarray_sum(int[] arr, int target) {
7          List<int> res = new List<int>();
```

```

8         int n = arr.Length;
9         for (int s = 0; s < n; ++s) { // pick a starting point for a subarray
10             int curr = 0;
11             // consider all ending points for the picked starting point
12             for (int e = s; e < n; ++e) {
13                 curr += arr[e];
14                 if (curr == target) {
15                     res.Add(s + 1);
16                     res.Add(e + 1);
17                     return res;
18                 }
19             }
20         }
21         res.Add(-1); // if no satisfying subarray is found
22         return res;
23     }
24
25     static void Main() {
26         int[] arr = {15, 2, 4, 8, 9, 5, 10, 23};
27         int target = 23;
28         List<int> res = subarray_sum(arr, target);
29         foreach (var ele in res)
30             Console.Write(ele + " ");
31     }
32 }

```

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

```

1  import java.util.ArrayList;
2  import java.util.List;
3
4  class GfG {
5      // function to find a continuous subarray whose sum = a given number
6      static ArrayList<Integer> subarray_sum(int[] arr, int target) {
7          ArrayList<Integer> res = new ArrayList<>();
8          int n = arr.length;
9          for (int s = 0; s < n; ++s) { // pick a starting point for a subarray
10             int curr = 0;
11             // consider all ending points for the picked starting point
12             for (int e = s; e < n; ++e) {
13                 curr += arr[e];
14                 if (curr == target) {
15                     res.add(s + 1);
16                     res.add(e + 1);
17                     return res;
18                 }
19             }
20         }
21         res.add(-1); // if no satisfying subarray is found
22         return res;
23     }
24
25     public static void main(String[] args) {
26         int[] arr = {15, 2, 4, 8, 9, 5, 10, 23};
27         int target = 23;
28         ArrayList<Integer> res = subarray_sum(arr, target);
29         for (int ele : res)
30             System.out.print(ele + " ");
31     }
32 }

```

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

```

1  // function to find a continuous subarray whose sum = a given number
2  function subarraySum(arr, target) {

```

```

3     let res = [];
4     let n = arr.length;
5
6     // Pick a starting point for a subarray
7     for (let s = 0; s < n; s++) {
8         let curr = 0;
9
10        // Consider all ending points
11        // for the picked starting point
12        for (let e = s; e < n; e++) {
13            curr += arr[e];
14            if (curr === target) {
15                res.push(s + 1);
16                res.push(e + 1);
17                return res;
18            }
19        }
20    }
21    // If no subarray is found
22    return [-1];
23 }
24
25 // Driver Code
26 let arr = [15, 2, 4, 8, 9, 5, 10, 23];
27 let target = 23;
28 let res = subarraySum(arr, target);
29
30 console.log(res.join(' '));

```

Python:

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

```

1  # function to find a continuous subarray which adds up to a given number
2  def subarray_sum(arr, target):
3      res = []
4      n = len(arr)
5      for s in range(n): # pick a starting point for a subarray
6          curr = 0 # initialize current calculated sum
7          for e in range(s, n): # consider all ending points for the picked starting point
8              curr += arr[e]
9              if curr == target:
10                 res.append(s + 1)
11                 res.append(e + 1)
12                 return res
13      return [-1] # if no subarray is found
14
15  if __name__ == "__main__":
16      n, x = map(int, input().split())
17      arr = list(map(int, input().split()))
18      res = subarray_sum(arr, x)
19      for ele in res:
20          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 một 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 một cách tiếp cận tương tự như cửa sổ trượt.

- Bắt đầu với một 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, hãy xóa các phần tử khỏi đầu cửa sổ hiện tại.
- Nếu tổng hiện tại bằng tổng mục tiêu, hãy trả về kết quả.

C++:

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

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  vector<int> subarray_sum_sliding_window(vector<int>& arr, int target) {
6      int s = 0, e = 0; // initialize window
7      vector<int> res;
8      int curr = 0; // current calculated sum
9      for (size_t i = 0; i < arr.size(); ++i) {
10         curr += arr[i];
11         // if current sum becomes >=, set end & try adjusting start
12         if (curr >= target) {
13             e = i;
14             // while current is greater, remove starting elements of current window
15             while (curr > target && s < e)
16                 curr -= arr[s++];
17             if (curr == target) { // if found a satisfying subarray
18                 res.push_back(s + 1);
19                 res.push_back(e + 1);
20                 return res;
21             }
22         }
23     }
24     return {-1}; // if no satisfying subarray is found
25 }
26
27 int main() {
28     int n, x;
29     cin >> n >> x;
30     vector<int> a(n);
31     for (int& v : a) cin >> v;
32     vector<int> res = subarray_sum_sliding_window(a, x);
33     for (int ele : res) cout << ele << " ";
34 }
```

□

Problem 3 (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

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

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

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

3rd solution. **Hashing + prefix sum:** $O(n)$ time & $O(n)$ space. 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.

– **Băm + tiền tố tổng:** $O(n)$ thời gian & $O(n)$ không gian. Giải pháp của 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 một 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, hãy kiểm tra mọi chỉ mục $i \in [n]$, & tổng đến chỉ mục đó là **curr_sum**. Nếu có một tiền tố có tổng bằng **curr_sum - target**, thì mảng con có tổng đã cho sẽ được tìm thấy. □

Remark 1 (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.

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

Problem 4 (2sum). Given an array of $n \in \mathbb{N}^*$ 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 $n \in \mathbb{N}^*$ 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. $a_i + a_j = x$.

Solution. 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.

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. □

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 5 (3sum). Given an array of $n \in \mathbb{N}^*$ 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 $n \in \mathbb{N}^*$ 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.

Problem 6 (k -sum, \star). Given an array of $n \in \mathbb{N}^*$ numbers & a target sum $x \in \mathbb{N}$, find $k \in \mathbb{N}^*$ array values such that their sum is x , or report that no such values exist.

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

2 Miscellaneous