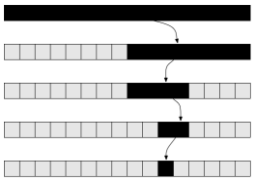
1 Binary Search

Problem Introduction



In this problem, you will implement the binary search algorithm that allows searching very efficiently (even huge) lists, provided that the list is sorted.

Problem Description

Task. The goal in this code problem is to implement the binary search algorithm.

Input Format. The first line of the input contains an integer n and a sequence $a_0 < a_1 < \cdots < a_{n-1}$ of n pairwise distinct positive integers in increasing order. The next line contains an integer k and k positive integers $b_0, b_1, \ldots b_{k-1}$.

Constraints. $1 \le k \le 10^5$; $1 \le n \le 3 \cdot 10^4$; $1 \le a_i \le 10^9$ for all $0 \le i < n$; $1 \le b_j \le 10^9$ for all $0 \le j < k$.

Output Format. For all i from 0 to $k-1$, output an index $0 \le j \le n-1$ such that $a_j = b_i$ or -1 if there is no such index.

Sample 1.

Input:

```
5 1 5 8 12 13
5 8 1 23 1 11
```

Output:

```
2 0 -1 0 -1
```

In this sample, we are given an increasing sequence $a_0 = 1, a_1 = 5, a_2 = 8, a_3 = 12, a_4 = 13$ of length five and five keys to search: $8, 1, 23, 1, 11$. We see that $a_2 = 8$ and $a_0 = 1$, but the keys 23 and 11 do not appear in the sequence a. For this reason, we output a sequence $2, 0, -1, 0, -1$.

2 Majority Element

Problem Introduction

Majority rule is a decision rule that selects the alternative which has a majority, that is, more than half the votes.

Given a sequence of elements $a_1, a_2, \ldots, a_n$, you would like to check whether it contains an element that appears more than $n/2$ times. A naïve way to do this is the following:

```
MajorityElement(a_1, a_2, ..., a_n):
for i from 1 to n:
  currentElement ← a_i
  count ← 0
  for j from 1 to n:
    if a_j = currentElement:
      count ← count + 1
  if count > n/2:
    return a_i
return "no majority element"
```

The running time of this algorithm is quadratic. Your goal is to use the divide-and-conquer technique to design an $O(n \log n)$ algorithm.

Problem Description

Task. The goal in this code problem us to check whether an input sequence contains a majority element.

Input Format. The first line contains an integer n, the next one contains a sequence of n non-negative integers $a_0, a_1, \ldots, a_{n-1}$.

Constraints. $1 \leq n \leq 10^5$; $0 \leq a_i \leq 10^9$ for all $0 \leq i < n$.

Output Format. Output 1 if the sequence contains an element that appears strictly more than $n/2$ times, and 0 otherwise.

Sample 1.

Input:

```
5
2 3 9 2 2
```

Output:

```
1
```

2 is the majority element.

Sample 2.

Input:

```
4
1 2 3 4
```

Output:

```
0
```

There is no majority element in this sequence.

Sample 3.

Input:

```
4
1 2 3 1
```

Output:

```
0
```

This sequence also does not have a majority element (note that the element 1 appears twice and hence is not a majority element).

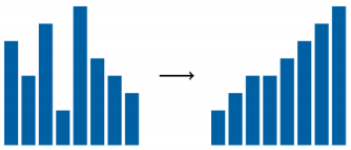
What To Do

As you might have already guessed, this problem can be solved by the divide-and-conquer algorithm in $O(n \log n)$ time. Indeed, if a sequence of length n contains a majority element, then the same element is also a majority element for one of its halves. Thus to solve this problem you first split a given sequence into halves and make two recursive calls. Do you see how to combine the results of two recursive calls?

It is interesting to note that this problem can also be solved in $O(n)$ time by a more advanced (non-divide and conquer) algorithm that just scans the given sequence twice.

3 Improving Quick Sort

Problem Introduction



The goal in this problem is to redesign a given implementation of the randomized quick sort algorithm so that it works fast even on sequences containing many equal elements.

Problem Description

Task. To force the given implementation of the quick sort algorithm to efficiently process sequences with few unique elements, your goal is to replace a 2-way partition with a 3-way partition. That is, your new partition procedure should partition the array into three parts: $< x$ part, $= x$ part, and $> x$ part.

Input Format. The first line of the input contains an integer n. The next line contains a sequence of n integers $a_0, a_1, \ldots, a_{n-1}$.

Constraints. $1 \le n \le 10^5$; $1 \le a_i \le 10^9$ for all $0 \le i < n$.

Output Format. Output this sequence sorted in non-decreasing order.

Sample 1.

Input:

```
5
2 3 9 2 2
```

Output:

```
2 2 2 3 9
```

Starter Files

In the starter files, you are given an implementation of the randomized quick sort algorithm using a 2-way partition procedure. This procedure partitions the given array into two parts with respect to a pivot x: $\le x$ part and $> x$ part. As discussed in the video lectures, such an implementation has $\Theta(n^2)$ running time on sequences containing a single unique element. Indeed, the partition procedure in this case splits the array into two parts, one of which is empty and the other one contains $n - 1$ elements. It spends cn time on this. The overall running time is then

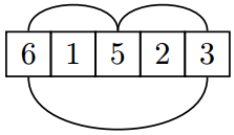
$$cn + c(n-1) + c(n-2) + \cdots = \Theta(n^2).$$

What To Do

Implement a 3-way partition procedure and then replace a call to the 2-way partition procedure by a call to the 3-way partition procedure.

4 Number of Inversions

Problem Introduction



An inversion of a sequence $a_0, a_1, \ldots, a_{n-1}$ is a pair of indices $0 \le i < j < n$ such that $a_i > a_j$. The number of inversions of a sequence in some sense measures how close the sequence is to being sorted. For example, a sorted (in non-descending order) sequence contains no inversions at all, while in a sequence in descending order any two elements constitute an inversion (for a total of $n(n-1)/2$ inversions).

Problem Description

Task. The goal in this problem is to count the number of inversions of a given sequence.

Input Format. The first line of the input contains an integer n. The next line contains a sequence of n integers $a_0, a_1, \ldots, a_{n-1}$.

Constraints. $1 \le n \le 10^5$; $1 \le a_i \le 10^9$ for all $0 \le i < n$.

Output Format. Output the number of inversions in the sequence.

Sample 1.

Input:

```
5
2 3 9 2 9
```

Output:

```
2
```

The two inversions here are $(1, 3)$ $(a_1 = 3 > 2 = a_3)$ and $(2, 3)$ $(a_2 = 9 > 2 = a_3)$.

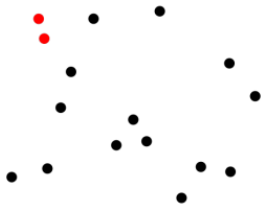
What To Do

This problem can be solved by modifying the merge sort algorithm. For this, we change both the `Merge` and `MergeSort` procedures as follows:

- `Merge(B, C)` returns the resulting sorted array and the number of pairs (b, c) such that $b \in B, c \in C$, and $b > c$.
- `MergeSort(A)` returns a sorted array A and the number of inversions in A.

5 Closest Points

Problem Introduction



In this problem, your goal is to find the closest pair of points among the given n points. This is a basic primitive in computational geometry having applications in, for example, graphics, computer vision, traffic-control systems.

Problem Description

Task. Given n points on a plane, find the smallest distance between a pair of two (different) points. Recall that the distance between points (x_1, y_1) and (x_2, y_2) is equal to $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

Input Format. The first line contains the number n of points. Each of the following n lines defines a point (x_i, y_i).

Constraints. $2 \leq n \leq 10^5$; $-10^9 \leq x_i, y_i \leq 10^9$ are integers.

Output Format. Output the minimum distance. The absolute value of the difference between the answer of your program and the optimal value should be at most 10^{-3}. To ensure this, output your answer with at least four digits after decimal point (otherwise your answer, while being computed correctly, can turn out to be wrong because of rounding issue).

Sample 1.

Input:

```
2
0 0
3 4
```

Output:

```
5.0
```

There are only two points here. The distance between them is 5.

Sample 2.

Input:

```
4
7 7
1 100
4 8
7 7
```

Output:

```
0.0
```

There are two coinciding points among the four given points. Thus, the minimum distance is zero.

Sample 3.

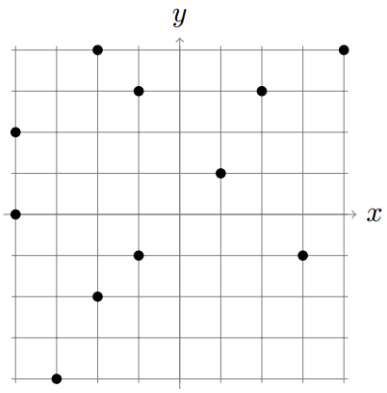
Input:

```
11
4 4
-2 -2
-3 -4
-1 3
2 3
-4 0
1 1
-1 -1
3 -1
-4 2
-2 4
```

Output:

```
1.414213
```

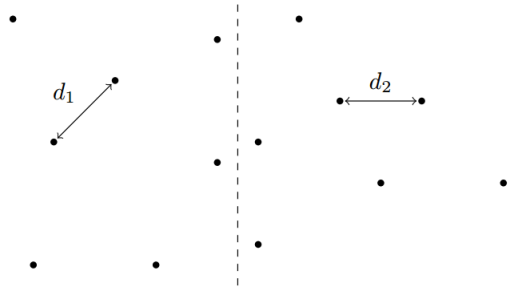
The smallest distance is $\sqrt{2}$. There are two pairs of points at this distance: $(-1,-1)$ and $(-2,-2)$; $(-2,4)$ and $(-1,3)$.



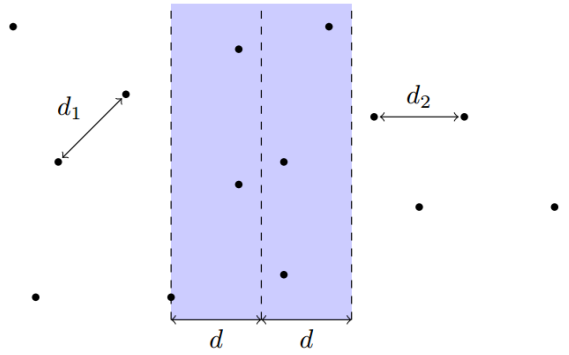
What To Do

This computational geometry problem has many applications in computer graphics and vision. A naïve algorithm with quadratic running time iterates through all pairs of points to find the closest pair. Your goal is to design an $O(n \log n)$ time divide and conquer algorithm.

To solve this problem in time $O(n \log n)$, let's first split the given n points by an appropriately chosen vertical line into two halves S_1 and S_2 of size $\frac{n}{2}$ (assume for simplicity that all x-coordinates of the input points are different). By making two recursive calls for the sets S_1 and S_2, we find the minimum distances d_1 and d_2 in these subsets. Let $d = \min\{d_1, d_2\}$.

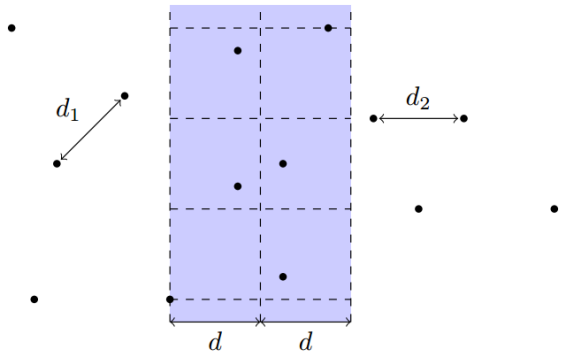


It remains to check whether there exists points $p_1 \in S_1$ and $p_2 \in S_2$ such that the distance between them is smaller than d. We cannot afford to check all possible such pairs since there are $\frac{n}{2} \cdot \frac{n}{2} = \Theta(n^2)$ of them. To check this faster, we first discard all points from S_1 and S_2 whose x-distance to the middle line is greater than d. That is, we focus on the following strip:



Stop and think: Why can we narrow the search to this strip? Now, let's sort the points of the strip by their y-coordinates and denote the resulting sorted list by $P = [p_1, \dots, p_k]$. It turns out that if $|i - j| > 7$, then the distance between points p_i and p_j is greater than d for sure. This follows from the Exercise Break below.

Exercise Break: Partition the strip into $d \times d$ squares as shown below and show that each such square contains at most four input points.



This results in the following algorithm. We first sort the given n points by their x-coordinates and then split the resulting sorted list into two halves S_1 and S_2 of size $\frac{n}{2}$. By making a recursive call for each of the sets S_1 and S_2, we find the minimum distances d_1 and d_2 in them. Let $d = \min\{d_1, d_2\}$. However, we are not done yet as we also need to find the minimum distance between points from different sets (i.e., a point from S_1 and a point from S_2) and check whether it is smaller than d. To perform such a check, we filter the initial point set and keep only those points whose x-distance to the middle line does not exceed d. Afterwards, we sort the set of points in the resulting strip by their y-coordinates and scan the resulting list of points. For each point, we compute its distance to the seven subsequent points in this list and compute d', the minimum distance that we encountered during this scan. Afterwards, we return $\min\{d, d'\}$.

$$T(n) = 2 \cdot T(n/2) + O(n \log n).$$

The $O(n \log n)$ term comes from sorting the points in the strip by their y-coordinates at every iteration.

Exercise Break: Prove that $T(n) = O(n(\log n)^2)$ by analyzing the recursion tree of the algorithm.

Exercise Break: Show how to bring the running time down to $O(n \log n)$ by avoiding sorting at each recursive call.