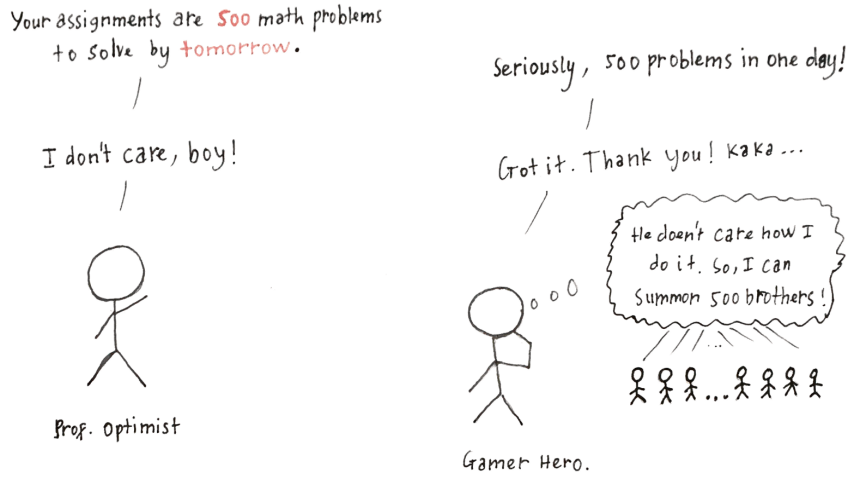
Phat Le Follow
Talk in RoR, React. Founder http://knocktocall.com
Apr 11, 2017 · 15 min read

Course 1—Algorithmic toolbox—Part 3: Divide and Conquer

We're going to talk about divide-and-conquer in this article. I'm very interested in this topic. I hope you will too. It takes me much of time to finish the assignments. I highly recommend you to take assignments seriously and try to solve 2 advanced problems.



Divider.

Divide and Conquer

Steps to do with Divide and conquer:

1. Break into non-overlapping subproblems of the same type.
2. Solve subproblems.
3. Combine results.

Binary search: Searching in sorted array.

Input: A sorted array A[low…high] (A[i] < A[i+1]) and a key k.

Output: An index, i, where A[i] = k.

Otherwise, the greatest index i, where A[i] < k.

Otherwise (k < A[low]), the result is low—1.

Sample 1:

Input: [3, 5, 9, 20, 27, 52, 65] and a key 20.

Output: 3.

Sample 2:

Input: [3, 5, 9, 20, 27, 52, 65] and a key 7.

Output: 1.

We will solve this problem by using divide and conquer algorithm. We will do step by step to solve it.

1. Break into non-overlapping subproblems of the same type.

- The input array is sorted. We will divide it half-half array. So there is no overlap elements between 2 sub-arrays.

2. Solve subproblems.

- We have 2 subproblems: A and B. We compare the key k with B[0].
- If B[0] == k => we found the result (result = index of B[0])
- If B[0] ≤ k => we choose array A. (result = index of B[0])
- If B[0] ≥ k => we choose array B.(result = index of A[0])

3. Combine results.

- Just return the result.

Pseudocode:

```
BinarySearch(A, low, high, key)

if high < low:
  return low − 1
mid ← ⌊low + (high−low)/2⌋
if key = A[mid]:
  return mid
else if key < A[mid]:
  return BinarySearch(A, low, mid − 1, key)
else:
  return BinarySearch(A, mid + 1, high, key)
```

Binary Search

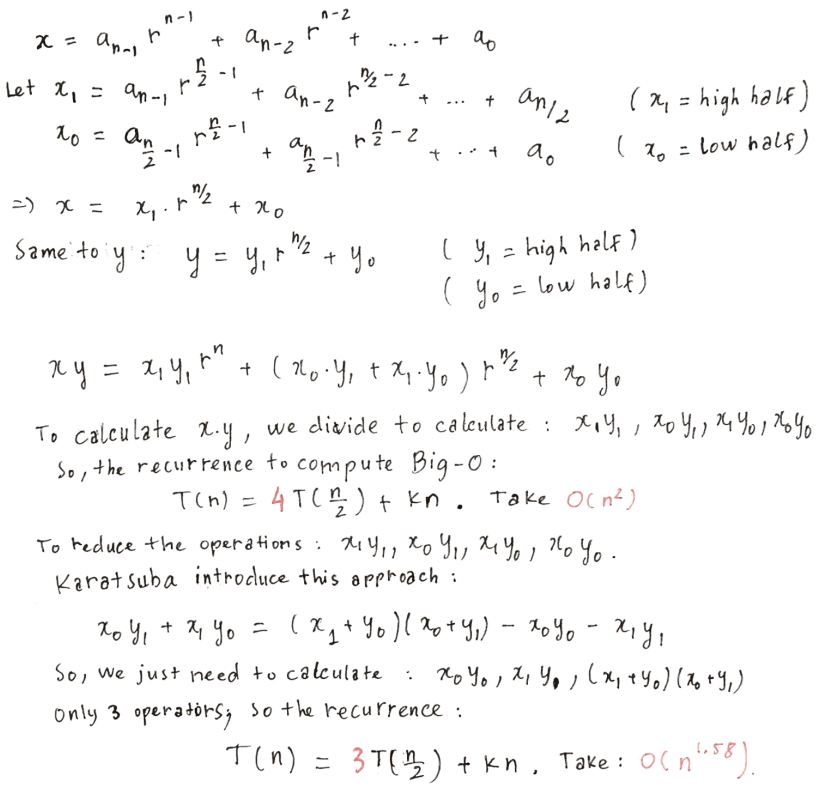
The runtime of binary search is **O(logn)**.

Multiplying Polynomials

This is very interesting section. We want to multiply large-integer numbers. How can we do that? Beside that problem, multiplying polynomials are applying for error-correcting codes, generate functions, convolution in signal processing…

We have 2 n-digit numbers: x and y (radix r = 2, 10).

We represent x and y into 2 forms that:



Karatsuba's method

Master Theorem

To calculate Big-O of recursive algorithms. We will general the formula and solve that in general cases.

If $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$ (for constants $a > 0, b > 1, d \geq 0$), then:

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Master Theorem

The proof of this formula is in [2]. We will focus on problem solving in this article. :)

MergeSort

MergeSort likes the name, break-down and merge. The concept is very simple:

- Break into 2 sub-list, break until you have only one element.
- Merge 2 sub-list by order(sorting).

Pseudocode:

```
MergeSort(A[1...n])

if n = 1:
  return A
m ← ⌊n/2⌋
B ← MergeSort(A[1...m])
C ← MergeSort(A[m+1...n])
A' ← Merge(B, C)
return A'
```

MergeSort

The key-point here is merging strategy. Just loop thought 2 sub-array, pick the smaller element, put it into a third-array.

```
Merge(B[1...p], C[1...q])

{B and C are sorted}
D ← empty array of size p+q
while B and C are both non-empty:
  b ← the first element of B
  c ← the first element of C
  if b ≤ c:
    move b from B to the end of D
  else:
    move c from C to the end of D
move the rest of B and C to the end of D
return D
```

Merge strategy

The Big-O is **O(nlogn).**

Note: There is an interesting problem below that required us to tweak the Merge strategy. By tweaking it, you will full-fill understand the MergeSort.

QuickSort

The concept of QuickSort is choosing a pivot, rearranging array that every elements on the left pivot are smaller than pivot, every elements on the right pivot are bigger than pivot.

Example: For array A = [**6**, 4, 2, 3, 9, 8, 9, 4, 7, 6, 1]

If we choose A[1] = 6 is a pivot, we need to re-arrange A into :

[1, 4, 2, 3, 4, 6, **6**, 9, 7, 8, 9]

The QuickSort algorithm is implemented into 2 steps:

- Choose pivot, re-arrange into A[left] ≤A[pivot] < A[right].
- Keep choosing pivot and re-arrange A[left] and A[right].

The pseudocode for this 2 steps:

```
QuickSort(A, ℓ, r)
if ℓ ≥ r:
  return
m ← Partition(A, ℓ, r)
{A[m] is in the final position}
QuickSort(A, ℓ, m − 1)
QuickSort(A, m + 1, r)
```

QuickSort

As you see, l is left index, r is right index.

Partition(A, l, r) function is containing choosing pivot, and rearrange A from l(left-index) to r(right-index) into A[left] ≤ A[pivot] ≤ A[right] and returning the position(index) of A[pivot].

To choose pivot, we have many strategies. If we know what types of data we have, we will have better choice of pivot. There are common choices of pivot:

- Choose A[0] as a pivot.

- Choose random from l -> r as a pivot
- median of medians algorithm.

The pseudocode for re-arranging array with A[0] as pivot:

```
Partition(A, ℓ, r)
x ← A[ℓ]    {pivot}
j ← ℓ
for i from ℓ + 1 to r:
  if A[i] ≤ x:
    j ← j + 1
    swap A[j] and A[i]
  {A[ℓ + 1 . . . j] ≤ x, A[j + 1 . . . i] > x}
swap A[ℓ] and A[j]
return j
```

QuickSort partition

x will be the value of pivot.

We want to re-arrange A[l…r] into A[left] ≤ A[pivot] ≤ A[right] with

A[left] = A[l…j]

A[right] = A[j+1…r]

We will use the last example to show how we can transform A = [**6**, 4, 2, 3, 9, 8, 9, 4, 7, 6, 1] into [1, 4, 2, 3, 4, 6, **6**, 9, 7, 8, 9].

left = 0, r = 10, pivot A[0] = 6.

Make a assumption that we have A[left] has only A[l] element, so j = 0.

We loop i from l+1 to r. At each step, we have to guarantee that A[left…j] ≤ A[pivot] ≤ A[j+1…r]. There are 2 cases:

- A[i] ≥ A[pivot], we do nothing because it's already A[i]≥ A[pivot].
- A[i] ≤ A[pivot], we have A[j] is the last index of element that is smaller than A[pivot]. So we need to swap A[j+1] and A[i] and mark j = j+1 to indicate that j is the last index of element that smaller than A[pivot]. By swap A[j+1] and A[i], we make sure that A[l…j] ≤ A[pivot] ≤ A[j+1…r].

If you run that algorithm manually, we will have A = [**6**, 4, 2, 3, 4, 6, 1, 9, 7, 8, 9].

So we have array: A[l] + A[1…j] + A[j+1…r] now. The last step is swap A[j] and A[0] so that we will have array: A[0…j-1] + A[pivot] + a[j+1…r].

So we got: A = [1, 4, 2, 3, 4, 6, **6**, 9, 7, 8, 9].

Randomized pivot:

To implement randomized pivot, it's very simple by doing:

- Choose random pivot $l \leq k \leq r$.
- Swap A[0] and A[k]
- Keep partition like choosing A[o] as pivot.

The Big-O of Randomized pivot QuickSort is **O(nlogn)** in average running time. The worst case running time is **$O(n^2)$**. This Big-O is a bit tricky. You can read the proof in [3].

Note: The worst case of QuickSort is **$O(n^2)$** but in the practice, QuickSort gave us better performance than MergeSort on average.

Equal Elements: QuickSort with a few uniq elements.

You can see the visualization of QuickSort here. We can observe that if the data has many equal elements, QuickSort takes long time to finish. To optimize, we should have different partition strategies:

Instead of re-arranging A into A[left] ≤ A[pivot] ≤ A[right]. We will turn A into: A[left] ≤ A[m1…m2] ≤ A[right] with A[i] == A[pivot] ($m1 \leq i \leq m2$).

So at each step, we have less A[left], A[right] items to sort.

The pseudocode for equal elements:

```
RandomizedQuickSort(A, ℓ, r)
if ℓ ≥ r:
  return
k ← random number between ℓ and r
swap A[ℓ] and A[k]
(m1, m2) ← Partition3(A, ℓ, r)
{A[m1 ... m2] is in final position}
RandomizedQuickSort(A, ℓ, m1 − 1)
RandomizedQuickSort(A, m2 + 1, r)
```

Equal elements QuickSort

In the exercises, we will have the implement details for this algorithm.

The best way to fully understand these sorting algorithms and divide and conquer technique is to solve interesting problems. Let's solve it together.

Problem 1: Implementing Binary Search

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

Input Format. The first line of the input contains an integer n and a sequence a0 < a1 < . . . < a(n-1) of n pairwise distinct positive integers in increasing order. The next line contains an integer k and k positive integers b0, b1, . . . , b(k-1).

Constraints. 1 ≤ n, **k ≤ 10^5**; 1 ≤ai ≤ 10^9 for all 0 ≤i<n; 1≤bj ≤10^9 for all 0≤j<k;

Output Format. For all 0 ≤ i ≤ k-1, output an index 0≤j≤n-1 such that aj = bi or -1 if there is no such index.

Sample.

Input:

5 1 5 8 12 13

5 8 1 23 1 11

Output:

2 0 –1 0 –1

Explanation:
In this sample, we are given an increasing sequence a0 = 1, a1 = 5, a2 = 8, a3 = 12, a4 = 13 of length five and five keys to search: 8, 1, 23, 1, 11. We see that a2 = 8 and a0 = 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.

Solution:

We just follow the BinarySearch pseudocode above and implement BinarySearch algorithm. Finally, we just loop through searching keys b0, b1, …,bk and run **binary_search(a, bi)**.

```
def binary_search(a, x):
    left, right = 0, (len(a) - 1)
    return binary_search_run(a, left, right, x)

def binary_search_run(a, left, right, x):
    if right < left:
        return -1
    mid = int(left + (right - left) / 2)
    if x == a[mid]:
        return mid
```

BinarySeach

Problem 2: Finding a Majority Element

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 a1, a2, . . . , an, you would like to check whether it contains an element that appears more than n/2 times. A naive way to do this is the following.

```
MajorityElement(a1, a2, . . . , an):
    for i from 1 to n:
        currentElement = ai
        count = 0
        for j from 1 to n:
            if aj == currentElement:
                count = count + 1
```

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.

Input Format. The first line contains an integer n, the next one contains a sequence of n non-negative integers a0, a1, . . . , an 1.

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

Explanation:
2 is the majority element.

Sample 2.

Input: 4

1 2 3 1

Output:

0

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

As you might have already guessed, this problem can be solved by the divide-and-conquer algorithm in time O(nlogn). 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.

Solution:

This is a super-interesting problem if you solve it by using divide and conquer.

Firstly, we need to divide into subproblems. At each step, we divide sequence into 2 left_half and right_half sequences. The main job is writing merge strategy:

```
def get_majority_element(a, left, right):
    if left + 1 == right:
        return [[a[left]], []]

    mid = int(left + (right - left) / 2)
    left_half = get_majority_element(a, left, mid)
```

To determine the majority of **sequence**, we will represent **left_half** or **right_half** into array of 2 elements: **A[majorities, others]**.

majorities = A[0] is containing all elements(same value)that has majority in this sequence. Called **A_major_elements**

others = A[1] is all elements that has no A_major_elements inside.

Example:

left_half = [[2, 2, 2], [3, 5, 7, 7, 9]]

right_half = [[5, 5, 5, 5],[4, 8, 10, 34, 2, 10, 10]]

the **count_merge** function will return the **M[majorities, others]** bases on **left_half** and **right_half**.

To do that, we do:

- Take all **left_half_major_elements**(ex: 2) from **right_half[1]**([4, 8, 10, 34, 2, 10, 10]), put it into **left_half[0].**
- Take all **right_half_major_elements(ex: 5)** from **left_half[1]**([3, 5, 7, 7, 9]), put it into **right_half[0].**

So we got :

left_half = [[2, 2, 2, **2**], [3, 7, 7, 9]] (add 2, remove 5)

right_half = [[5, 5, 5, 5, **5**],[4, 8, 10, 34, 10, 10]](add 5, remove 2).

We call this process is **chunk_process**:

```
def chunk_process(majors, eles):
    others = []
    for ele in eles:
        if majors[0] == ele:
            majors.append(ele)
        else:
```

In **count_merge** function, we just need to choose majority basing on **chunk_left** and **chunk_right**.

```
def count_merge(left_half, right_half):
    [left_majors, right_others] = chunk_process(left_ha
    [right_majors, left_others] = chunk_process(right_h
    if left_majors[0] == right_majors[0]:
        return [left_majors + right_majors, left_others
    elif len(left_majors) > len(right_majors):
        return [left_majors, right_majors + left_others
```

In the end, we will have a result array: **[majorities, others]**. We just check the majorities has length over n/2 to give the answer.

```
# Uses python3
import sys

def get_majority_element(a, left, right):
    if left + 1 == right:
        return [[a[left]], []]

    mid = int(left + (right - left) / 2)
    left_half = get_majority_element(a, left, mid)
    right_half = get_majority_element(a, mid, right)
    return count_merge(left_half, right_half)

def count_merge(left_half, right_half):
    [left_majors, right_minors] = chunk_process(left_h
    [right_majors, left_minors] = chunk_process(right_
    if left_majors[0] == right_majors[0]:
        return [left_majors + right_majors, left_minor
    elif len(left_majors) > len(right_majors):
        return [left_majors, right_majors + left_minor
    else:
        return [right_majors, left_majors + right_minc

def chunk_process(majors, eles):
    left = []
    for ele in eles:
```

Problem 3: Improving Quick Sort

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 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 a0, a1, . . . , a(n-1).

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

Solution:

As we talk above, we need to optimize QuickSort for data that has many equal elements. So we will partition array into 3 array:

$A[left] \leq A[m1 \ldots m2] \leq A[right]$ with $A[i] == A[pivot]$ $(m1 \leq i \leq m2)$.

```
def randomized_quick_sort(a, l, r):
    if l >= r:
        return
    k = random.randint(l, r)
    a[l], a[k] = a[k], a[l]
    #use partition3
    [m1, m2] = partition3(a, l, r)
```

Our main job is implementing partition3 function to re-arrange array A and return position m1, m2.

```python
# Uses python3
import sys
import random

def partition3(a, l, r):
    x = a[l]
    begin = l+1
    end = l

    for i in range(l + 1, r + 1):
        if a[i] <= x:
            end += 1
            a[i], a[end] = a[end], a[i]
            if a[end] < x:
                a[begin], a[end] = a[end], a[begin]
                begin += 1

    a[l], a[begin-1] = a[begin-1], a[l]

    return [begin, end]

def randomized_quick_sort(a, l, r):
    if l >= r:
        return
    k = random.randint(l, r)
```

We loop through all element i from l + 1 to r. At each step, we make sure that we re-arrange a[i] to right position.

For example, if **a[i] ≤ pivot**(a[l]), we relocate a[i] to a[begin…end] (has same value to pivot). We continue to check if **a[i] == pivot**, we keep that position by doing nothing, if **a[i] < pivot**, we need to move a[i] before a[begin].

In the end, we just swap a[0](pivot) and a[begin] to get final step of partition(re-arranging) array a.

It seems to be tricky to understand but if you run it manually, it will be very clear under your hand(I promised).

Advanced Problem 4: How Close a Data is To Being Sorted?

An inversion of a sequence a0, a1,…, a(n-1) is a pair of indices 0 ≤i < j < n such that ai > aj. 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 sorted 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 contains an integer n, the next one contains a sequence of integers a0, a1,…, a(n-1).

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

Sample 1.

Input: 5

2 3 9 2 9

Output: 2

Explanation:
The two inversions here are(1, 3)(a1 =3 > 2=a3) and (2, 3)(a2 =9 > 2=a3).

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 ∈ B, c ∈ C, and b > c
- MergeSort(A) returns a sorted array A and the number of inversions in A.

Solution:

Our goal is printing number of inversions ai > aj (0 ≤ i ≤ j ≤ n). We will divide array A into **left_array** and **right_array** with format: **[count_inversions, sorted_elements].**

In **merge** function, we can implement like MergeSort and we need to add **count_inversions** for step that has **left_array[1][i] > right_array[1][i]**.

```python
# Uses python3
import sys

def get_number_of_inversions(a):
    [count_inversions, sorted_result] = mergesort_inve
    return count_inversions

def mergesort_inversions_count(a):
    if len(a) == 1:
        return [0, a]
    mid = int(len(a) / 2)
    left = mergesort_inversions_count(a[0:mid])
    right = mergesort_inversions_count(a[mid:])
    return merge(left, right)

def merge(left, right):
    count_inversions = left[0] + right[0]
    left_array = left[1]
    right_array = right[1]

    result = []
    while len(left_array) > 0 and len(right_array) > 0
        if left_array[0] > right_array[0]:
            result.append(left_array[0])
            # number of inversions are the number of r
```

Advanced Problem 5: Organizing a Lottery

You are organizing an online lottery. To participate, a person bets on a single integer. You then draw several ranges of consecutive integers at random. A participant's payoff then is proportional to the number of ranges that contain the participant's number minus the number of ranges that does not contain it. You need an efficient algorithm for computing the payoffs for all participants. A naive way to do this is to simply scan, for all participants, the list of all ranges. However, you lottery is very popular: you have thousands of participants and thousands of ranges. For this reason, you cannot afford a slow naive algorithm.

Problem Description

Task. You are given a set of points on a line and a set of segments on a line. The goal is to compute, for each point, the number of segments that contain this point.

Input Format. The first line contains two non-negative integers s and p defining the number of segments and the number of points on a line, respectively. The next s lines contain two integers ai, bi defining the i-th segment [ai, bi]. The next line contains p integers defining points x1, x2, . . . , xp.

Output Format. Output p non-negative integers k0, k1, . . . , k(p-1) where ki is the number of segments which contain xi.

Sample 1.

Input:

2 3

0 5

7 10

1 6 11

Output:

1 0 0

Explanation:
Here, we have two segments([0, 5], [7, 10]) and three points([1, 6, 11]). The first point lies only in the first segment while the remaining two points are outside of all the given segments.

Sample 2.

Input:

3 2

0 5

-3 2

7 10

1 6

Output:

2 0

As you might have already guessed, you goal is to first sort the given segments somehow (so, this is a sorting problem, not a divide-and-conquer problem).

Solution:

This problem is very challenging. To solve this problem, we have to sort given segments by some strategies.

We try to sort [ai, LEFT] + [aj, RIGHT] + [pk, POINT]. To compare 2 elements [x, LEFT/RIGHT/POINT] and [y, LEFT/RIGHT/POINT], we

compare x and y, if x == y, we compare LEFT/RIGHT/POINT basing on the value of LEFT = 1, POINT = 2, RIGHT = 3.

```
def less_than_or_equal(num1, num2, let1, let2):
    return less_than(num1, num2, let1, let2) or \
    equal(num1, num2, let1, let2)

def less_than(num1, num2, let1, let2):
    return num1 < num2 or \
            (num1 == num2 and let1 < let2)
```

For example, we have segments: [0, 5], [-3, 2], [7, 10] and points: [1, 6]. We turn these into: [0, LEFT], [5, RIGHT], [-3, LEFT], [2, RIGHT], [1, POINT], [6, POINT]. Then we sort these items using RandomizedQuickSort that we did in problem 3, we will have a sorted array:

[-3, LEFT], [0, LEFT], [1, POINT], [2, RIGHT], [5, RIGHT], [6, POINT], [10, RIGHT].

To count the covering segments, we just need to count how many ("LEFT"—"RIGHT") items appeared before "POINT" item.

```python
# Uses python3
import sys
import random

def fast_count_segments(starts, ends, points):
    cnt = [0] * len(points)
    LEFT = 1
    POINT = 2
    RIGHT = 3

    starts_l = [LEFT] * len(starts)
    ends_r = [RIGHT] * len(ends)
    points_p = [POINT] * len(points)

    pairs_number = starts + ends + points
    pairs_letter = starts_l + ends_r + points_p

    randomized_quick_sort(pairs_number, pairs_letter,

    count_left = 0

    point_counts = {}
    for p in points:
        point_counts[p] = 0

    for i in range(len(pairs_number)):
        if pairs_letter[i] == LEFT:
            count_left += 1
        elif pairs_letter[i] == RIGHT:
            count_left -= 1
        elif pairs_letter[i] == POINT:
            if point_counts[pairs_number[i]] == 0:
                point_counts[pairs_number[i]] += count

    for i in range(len(points)):
        cnt[i] = point_counts[points[i]]

    return cnt

def partition3(a, b, l, r):
    x = a[l]
    letx = b[l]
    begin = l+1
    end = l

    for i in range(l + 1, r + 1):
        if less_than_or_equal(a[i], x, b[i], letx):
            end += 1
            a[i], a[end] = a[end], a[i]
            b[i], b[end] = b[end], b[i]
            if less_than(a[end], x, b[end], letx):
```

```
                a[begin], a[end] = a[end], a[begin]
                b[begin], b[end] = b[end], b[begin]
                begin += 1

    a[l], a[begin-1] = a[begin-1], a[l]
    b[l], b[begin-1] = b[begin-1], b[l]

    return [begin, end]

```

To pass this problem, I have to submit 11 times with a lot of changing strategy. So, don't give up, you're better than you think!

Advanced Problem 6: Finding the Closest Pair of Points

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.

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

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 $1/10^3$. To ensure this, output your answer with at least four digits after the decimal point (otherwise your answer, while being computed correctly, can turn out to be wrong because of rounding issues).

Sample 1.

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

Explanation: The smallest distance is √2. There are two pairs of points at this distance: (1, 1) and (2, 2); (2, 4) and (1, 3).

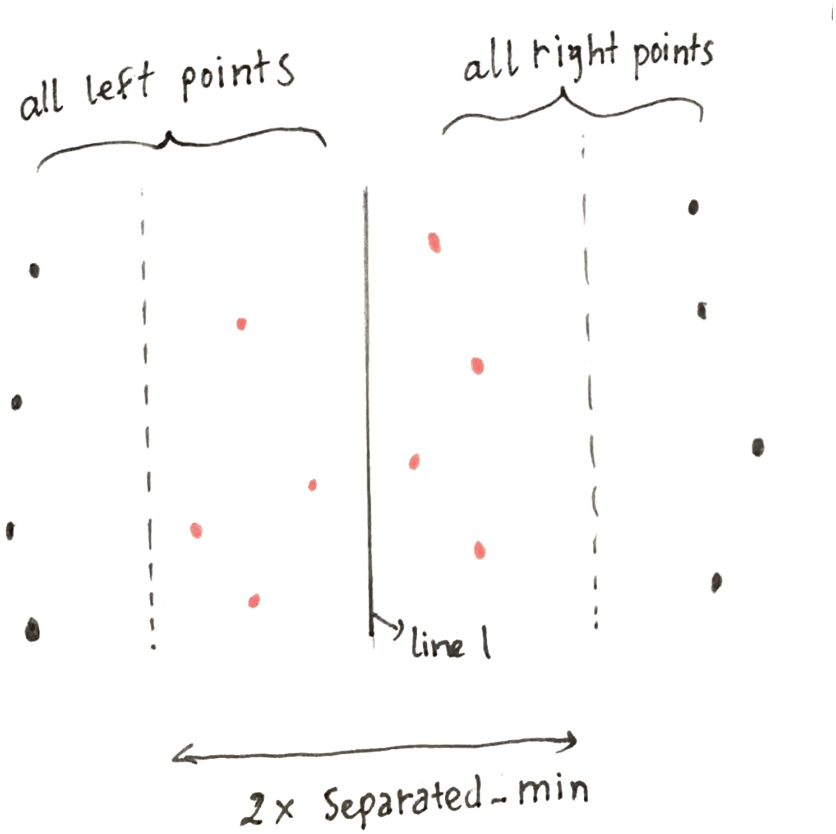
Solution:

There is a chapter in [CLRS] discuss about this problem, I think this problem is hardest in this course. I usually encounter with Failed case #22/23: time limit exceeded. So, be sure that you read [CLRS] textbook, [section 33.4] about this problem first.

We need to pre-sort points by x coordinate and divide points into 2 arrays: **left_points** and **right_points.** Our goal is calculate min distance of **left_points** called **min_left**, min distance of **right_points** called **min_right**. And min distance between **left_points** and **right_points** called **hybrid_min.**

```
def minimum_distance(points):
    sorted_p_x = sorted(points, key=lambda p: p.x)

    return large_size_min_distance(sorted_p_x)

def small_size_min_distance(points):
    result = sys.maxsize
    for i in range(len(points)):
        for j in range(i+1, len(points)):
            result = min(result, distance(points[i], p
    return result

def large_size_min_distance(p_x):
    size = len(p_x)
    half_size = int(len(p_x)/2)

    if size <= 3:
        return small_size_min_distance(p_x)

    left_p_x = p_x[0:half_size]
```

For small size of array(size < 3), we use compute min distance by brute-force. We try to compute **left_min** and **right_min** first. We call min distance of **left_min** and **right_min** is **separated_min**. Because left_points and right_points are sorted by x and we did compute **separated_min**, so instead of computing **hybrid_min** of **all left_points** and **all right_points**, we just care the points that has x coordinate within **separated_min** radius from the middle-line **line_l = (left_points[last].x + right_points[first].x)/2.**



Red points are in separated radius.

```
def filter_x_coordinate(left_x, right_x, line_l, wide)
    left = []
    for i in range(len(left_x)):
        if abs(left_x[i].x - line_l) <= wide:
            left.append(left_x[i])
    right = []
    for i in range(len(right_x)):
```

We reduced number of points the has x coordinate between **separated_min** radius. We call these reduced points is **reduced_total** points. We can see that, to compute min distance of a point to all points in reduced_total points, we just need to compute 7 points in reduced_total points that have y coordinate within **separated_min**. It means the boundary of point x from **reduced_total** points is a rectangle with **wide = 2 x separated_min**, **height = separated_min**. So to do that, we need to sort **reduced_total** points by y_coordinate and loop through all sorted reduced_total points, compute min distance of 7-points-boundary. Then we can finally compute hybrid min distance.

```python
#Uses python3
import sys
import math
import random

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return str([self.x, self.y])

def distance(p1, p2):
    return ((p1.x - p2.x) ** 2 + (p1.y - p2.y) ** 2) *

def construct_points(x, y):
    points = []
    for i in range(len(x)):
        points.append(Point(x[i], y[i]))
    return points

def minimum_distance(x, y):
    points = construct_points(x, y)
    sorted_p_x = sorted(points, key=lambda p: p.x)

    return large_size_min_distance(sorted_p_x)

def small_size_min_distance(points):
    result = sys.maxsize
    for i in range(len(points)):
        for j in range(i+1, len(points)):
            result = min(result, distance(points[i], p
    return result

def large_size_min_distance(p_x):
    size = len(p_x)
    half_size = int(len(p_x)/2)

    if size <= 3:
        return small_size_min_distance(p_x)

    left_p_x = p_x[0:half_size]
    right_p_x = p_x[half_size:size]

    left_min = large_size_min_distance(left_p_x)
    right_min = large_size_min_distance(right_p_x)
    separated_min = min(left_min, right_min)

    line_l = (left_p_x[-1].x + right_p_x[0].x)/2
    hybrid_min = compute_hybrid_min(left_p_x, right_p_
```

It’s very difficult to get right direction of implementing this algorithms. Once you implement right, you will pass this problem. I did submit 17 times for this problems. So keep calm and keep submitting. :D

Resources:

[1]: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6_006F11_lec11.pdf

[2] Master Theorem: Section 2.2 of [DPV08]

[3] Quick sort: Chapter 7 of [CLRS]

Polynomial multiplication: Section 2.1 of [DPV08]

Merge sort and lower bound for comparison based sorting: Section 2.3 of [DPV08]

Quick sort: Chapter 7 of [CLRS]

References:

[DPV08] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. Algorithms (1st Edition). McGraw-Hill Higher Education. 2008.

[CLRS] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms (3rd Edition). MIT Press and McGraw-Hill. 2009.