

Computational geometry is the field of informatics or computer science that deals with solving geometric problems using a computer. This field contains many areas of study, including graph drawing (given a description of a graph, how to draw it on a surface aesthetically?), algebraic modeling (solving a geometric problem as a vast set of equations, and finding fast solvers), computational topology (study of the structure of graphs on surfaces) and discrete geometry (studying the combinatorial properties of geometric problems). The core area of the field is to study algorithmic techniques and data structures in the context of geometric problems. Such techniques are widely used in computer graphics, geographical information systems, computer aided design, robot motion planning, databases and much more. In this assignment, we look at two very simple problems that are often used as black boxes in more complex algorithms.

1 Angular Sorting with Timsort

1.1 Sorting Angularly

In this problem, we will sort points in the two dimensional plane in anti-clockwise direction, based on the angle formed with the x axis. Each point is described as a tuple (x, y) , where x and y are lengths of the point on the respective coordinate axes. Note that each point can be visualized as a vector, i.e., a ray from the origin in the direction of the point (see Figure (i) below).

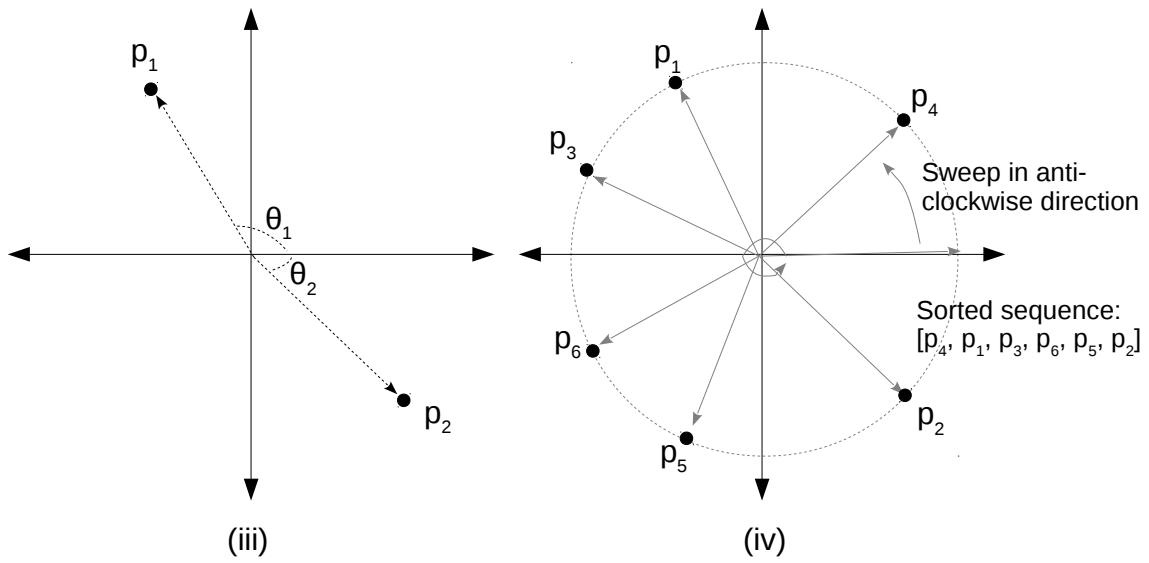
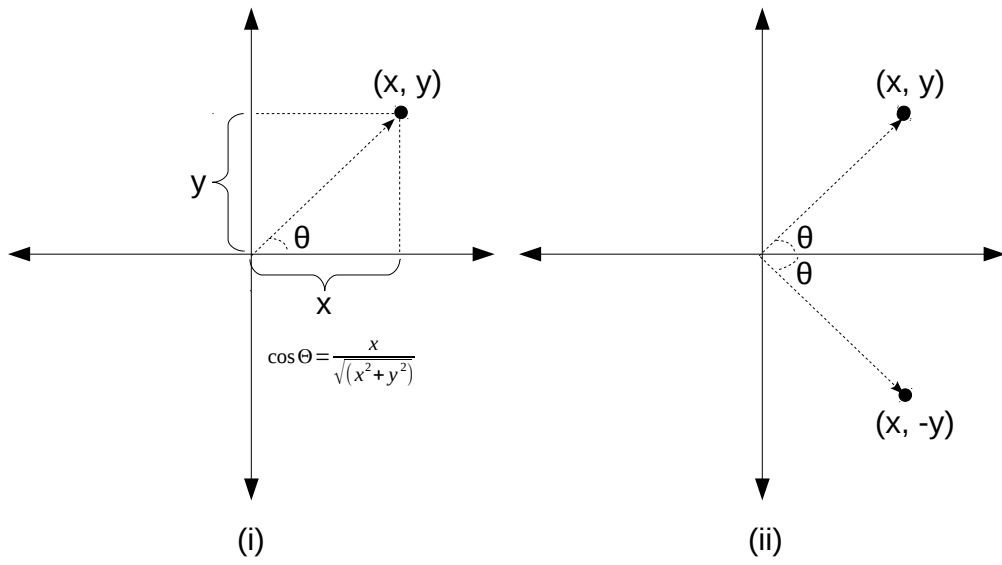
The angle θ formed with the x axis can be easily computed:

$$\begin{aligned}\cos \theta &= \frac{x}{\sqrt{x^2 + y^2}} \\ \Rightarrow \theta &= \cos^{-1} \left(\frac{x}{\sqrt{x^2 + y^2}} \right)\end{aligned}$$

The use of the cosine function is nice because it is monotonically decreasing from 1 to -1 in $[0, \pi]$. The inverse of the cosine function, also called as arc cosine, is a common mathematical function provided in the math libraries of most programming languages. This however, gives the same value of θ for points (x, y) and $(x, -y)$ (see Figure (ii) below), since the range of the function is only $[0, \pi]$. This raises an important issue in sorting points angularly: the angle should be taken in the context of the quadrant¹ in which it lies. For example, consider comparing two points p_1 and p_2 forming angles θ_1 and θ_2 respectively. Suppose θ_1 is larger than θ_2 . It is possible for p_1 to appear earlier in the sorted ordering if p_2 lies in the fourth quadrant (see Figure (iii) below).

You may imagine angular sorting with a ray extending out of the origin in the direction of the positive x axis, which sweeps in the anti-clockwise direction (see Figure (iv)). Each point is recorded in the output sorted sequence when it intersects this ray.

¹In coordinate geometry, the first quadrant contains all points with $x \geq 0$ and $y \geq 0$, the second quadrant with points $x < 0$ and $y \geq 0$, third quadrant with points $x < 0$ and $y < 0$, and finally the fourth quadrant with points $x \geq 0$ and $y < 0$.



1.2 The Point Class

Since comparing two points is now non-trivial, it will be beneficial to implement a `Point` class that stores information about a point, and one that also knows to compare two points together. Please refer to the appropriate language documentation to see how to include a comparator or a comparison function to compare two `Point` objects. Depending on the language and implementation, this function may return an integer value that is < 0 , 0 and > 0 if the first point is angularly smaller, equal and larger respectively than the second point. In some languages, it may be easier to override appropriate operators for the same task. Please make sure that you are returning the proper value (either `int` or `boolean`) depending on the implementation of the comparator function.

1.3 Timsort

Timsort is an industrial grade adaptive sorting algorithm whose running time adapts to the structure of data in the input. It is the sorting algorithm of choice in both Java and Python sorting library functions. It combines two sorting algorithms for its task – insertion sort to sort small $O(1)$ sized subproblems, and mergesort for larger subproblems. However, timsort does not divide the problem into equal sized chunks, rather it divides the problems into runs, which are consecutive elements that are non-decreasing in the input. Therefore it merges variable-sized sorted sequences into one. Timsort is also stable, and this is achieved by only merging consecutive runs. For this, it makes use of a run stack. Timsort has two distinct phases which are described below.

1.3.1 Scanning Phase

As a first step, timsort scans the input data from left to right and identifies runs. It performs a number of tasks during this process. Let the input array be a . We use the subscript notation a_i to refer to the i^{th} element in the array.

1. Timsort makes sure that each run is of sufficient length. Specifically, if timsort is scanning the i^{th} element a_i , it tries to decide if a_i can be included in the current run or whether a new run starts at a_i . If the number of elements in the current run is less than a pre-defined `minrun` length value, then a_i is insertion sorted into the current run. Note that a_i is not inserted into a sorted prefix of the entire input $a_1 \dots a_{i-1}$, but only the sorted prefix of the current run. If a_i is at least as large as the last element in the run, then the length of the run is increased. If a_i is smaller than the last element in the run and if the length of the current run is at least as large as `minrun`, then a new run is started.

With lots of empirical testing, it is recommended to use a `minrun` value between 32 and 64. For this problem, we will set this value to 32.

2. Once an entire run is identified, timsort adds the current run onto a stack. Each run is identified using a tuple (i, l) where i is the index of the start of the run and l is its run length. It then checks to see if the following invariants hold for three consecutive runs z , y and x on top of the stack.

- (a) $|z| \geq |x| + |y|$: If not, then z and y are merged.
- (b) $|y| \geq |x|$: If not, then y with x are merged.

Note that $|x|$ is the length of run x . It only proceeds to the next run if the above invariants hold for the three topmost runs on the stack. Otherwise, it repeatedly merges two consecutive runs (either y with z or y with x) and checks the invariant again.

Figure (v) shows the above two steps working in tandem. Steps (a) and (e) indicate start of new runs. In steps (b), (f), (h) and (i), the current element is increasing. Therefore we move on. In steps (d), (j), (k) and (m), current runs are completed. In step (k), the first invariant is checked and the bottom two runs on the stack are merged. In step (m), the second invariant is violated and the top two runs on the stack are merged. Note that there could be multiple merges after the current run is determined. The algorithm only proceeds to the next run if the current invariant on the top of the stack is satisfied. Note that this does not mean that the invariants are satisfied for any three consecutive runs. A future run could merge things on the stack in myriad different ways, thereby potentially violating the invariant further along in the stack. But we always guarantee that the invariants are satisfied on the top of the stack before finding the next run.

1.3.2 Bottom-up Merge

After the scanning phase is completed, there may be many runs on the stack. These runs are merged in a bottom-up fashion, two at a time. This can be implemented by popping two runs from the stack, merging them, and pushing the merged run onto another “helper” stack. Once the first stack is empty, we then copy all elements from the helper stack back onto the run stack and repeat.

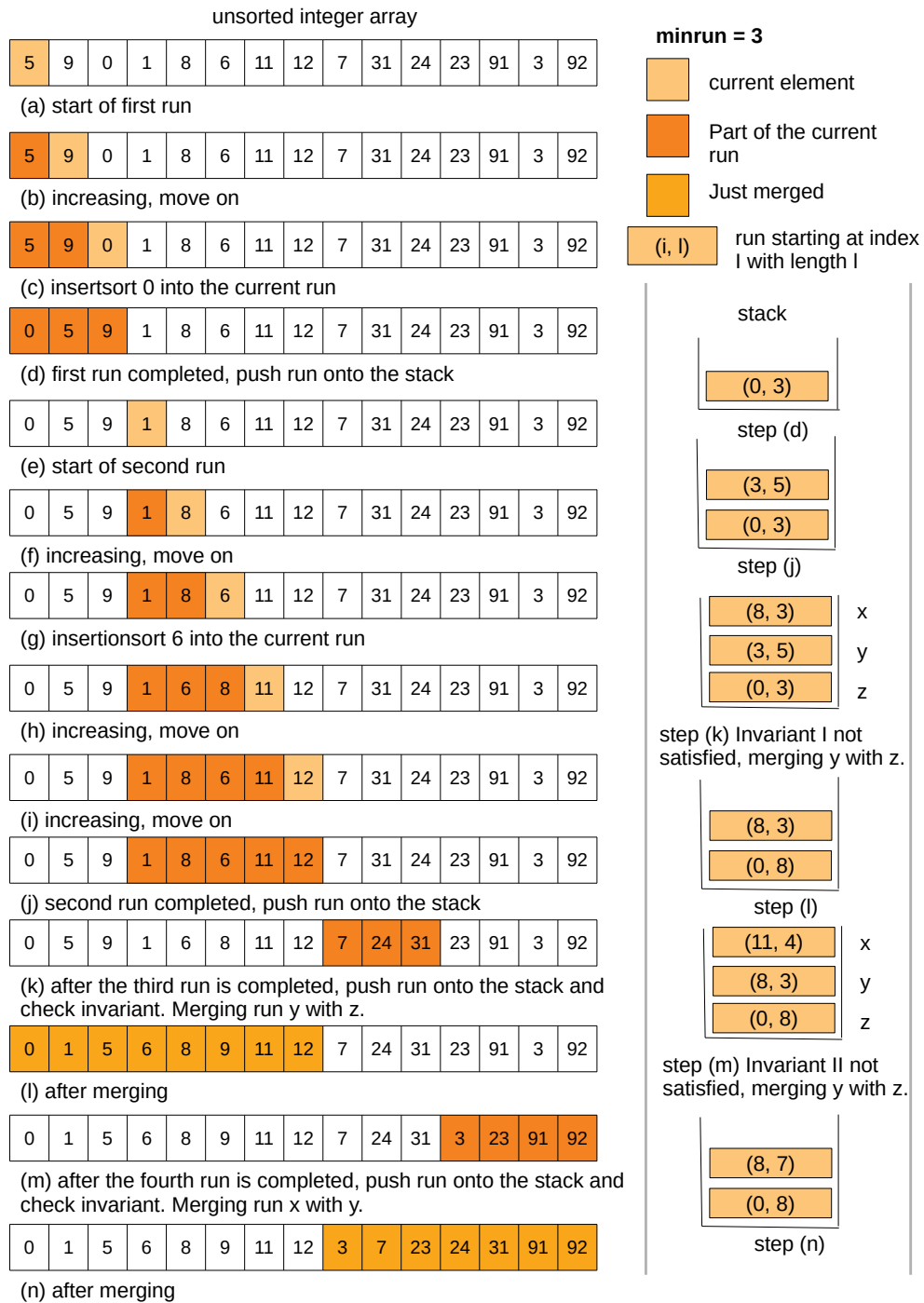
In Figure (vi), we see 5 runs still remaining on the stack after the scanning phase. The run stack is shown on the left, and the helper stack is shown on the right. In steps (a), (b), (d) and (f), we pop two runs from the run stack, merge them, and push the merged run onto the helper stack. In steps (c) and (e), there is only one run on the run stack. If the initial number of runs is even, then this stack will be empty. In steps (c) and (e), we pop all runs from the helper stack and push them onto the run stack. This merging process continues until we are left with a single run, which is the sorted array.

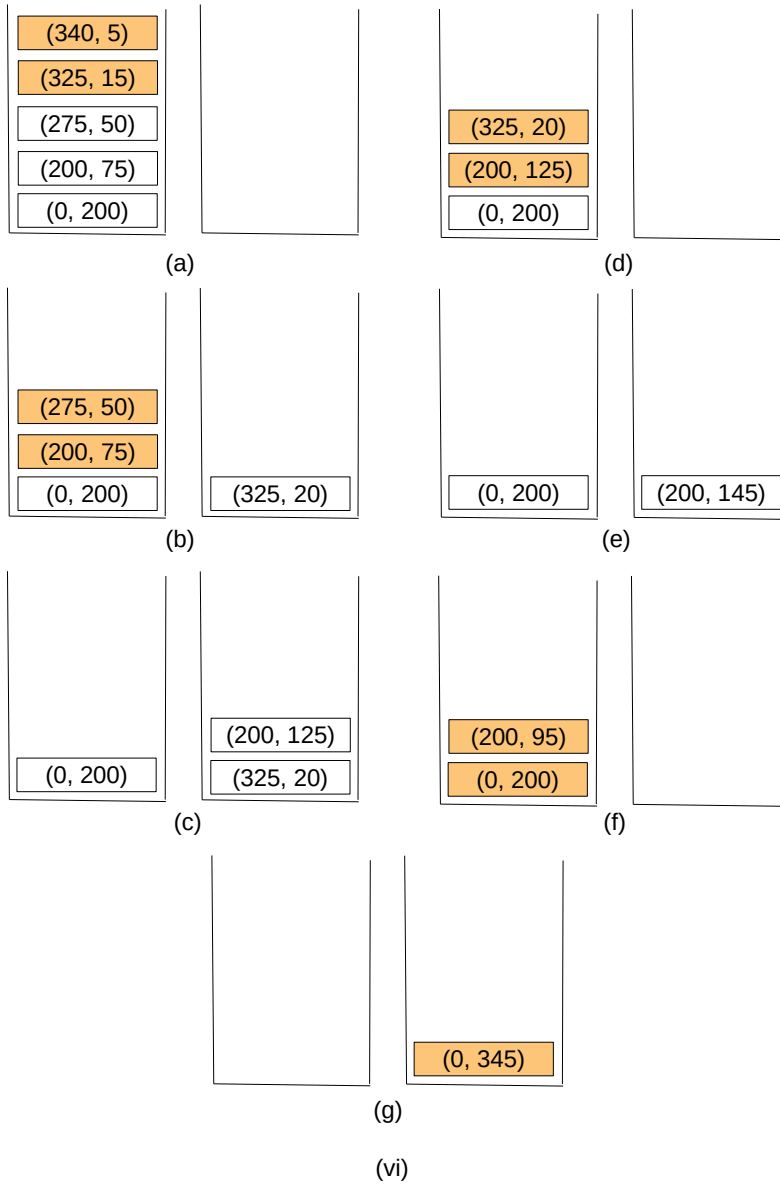
1.4 Sensible Testing

As there are many moving blocks, consider working with integer input as described in this handout. This allows you to test and debug your timsort routines. Also use good code design to modularize your code. For example, you can have separate functions for the scanning and the bottom-up parts. Moreover, you can have separate functions that perform insertion sort and mergesort. After you have thoroughly tested timsort for integers, you can work on the comparison function that compares two point objects. For this, you can use the small input instance provided in this handout and in the file `input-small.txt`. You may also use the library sorting function with your comparator function and test the sorted output, which is given in the file `output-small.txt`. Please also test for different input-output cases.

1.5 Input and Output

For this program, we have one input file, and two output files. The file points `input-points.txt` contains 10000 points. The input file contains the number of points n on the first line fol-





lowed by n lines that describe the x and y coordinates of a point. The sorted sequence is in `sorted-points.txt`. A sample input and output for 6 points is provided below.

Sample Input:

```
6
0.5 -0.8660254
-0.5 -0.8660254
-0.5 0.8660254
0.5 0.8660254
-1.0 0.0
1.0 0.0
```

Sample Sorted Output:

```
1.0 0.0
0.5 0.8660254
-0.5 0.8660254
-1.0 0.0
-0.5 -0.8660254
0.5 -0.8660254
```

In addition to the sorted sequence, we would also like to obtain information about the scanning and bottom-up merge phases. During the scanning phase, we would like to print each run (of length at least `minrun`) that was identified and inserted on the stack, which invariant was not satisfied, and which runs on the stack are merged. After the scanning phase, we print the runstack (from top to bottom), and the merges of the runs during the bottom-up merging phase. After the array is sorted, we print the total number of runs and merges the algorithm performs. Note that the number of merges is always one less than the total number of runs found in the scanning phase. The relevant parts of the transcript output for the points in `input-points.txt` are shown below. Please see the full transcript in `points-timsort-info.txt`.

scanning phase:

```
run: [0, 32]
run: [32, 32]
run: [64, 32]
fixing invariant 1: merging runs [0, 32] [32, 32]
run: [96, 32]
run: [128, 32]
fixing invariant 1: merging runs [64, 32] [96, 32]
fixing invariant 1: merging runs [0, 64] [64, 64]
...
```

after scanning phase, stack contents are

```
[9993, 7]
[9865, 128]
[7814, 2051]
[4066, 3748]
[0, 4066]
```

bottom-up merging phase:

```
merging runs [9865, 128] [9993, 7]
merging runs [4066, 3748] [7814, 2051]
merging runs [4066, 5799] [9865, 135]
merging runs [0, 4066] [4066, 5934]
```

```
total number of runs found = 313
total number of merges performed = 312
```

please name your solution `angular-sort.py`. Then we will run your program by passing in the name of the timsort information file as a command line argument, but redirect both the standard input and output.

```
python3 angular-sort.py timsort-info.txt < input-points.txt > out
```

Since timsort is a deterministic sorting algorithm, the outputs of both files should match exactly. You may test the given input-output files using the `diff` Unix command.

```
diff sorted-points.txt out
```

1.6 Timsort Analysis

Let $c = O(1)$ be the `minrun` length. Then insertion sorting elements in each run takes only $O(1)$ time. We also spend $O(n)$ time to scan the input in the scanning phase. In the worst case, we end up with $\rho = n/c$ runs and need to merge them. Since each run has length at least c , every time we merge two runs together, we double the size of the run. The invariants also make sure that we always merge only similar sized chunks. So from the perspective of a given run, it participates in only $O(\log \rho)$ merges, and each of these take $O(n)$ time over all runs. So the total running time is $O(n \log \rho)$. Note that this running time does not violate the comparison based sorting lower bound, since if $\rho = O(n)$, it runs in $O(n \log n)$.

1.7 Further Optimizations

Apart from high-level ideas described above, timsort performs a sleuth of optimizations. We describe some of these below. Implementing these optimizations is beyond the scope of this assignment, but they do offer some food for thought. Feel encouraged to attempt these if you are in need of a challenge.

1. **Optimum Minrun Length:** Since the minrun length is recommended to be between 32 and 64, it is encouraged that the number of runs that can be formed to be close to a power of two. So the minrun length c is chosen in $[32, 64]$ such that n/c is closest to a power of two.
2. **Decreasing Runs:** Runs can also be formed using both increasing and decreasing sequences. However, in order to maintain stability, a decreasing run is strict, so it does not include equal elements. When scanning element a_i , we can perform a binary search within the run to see if a_i is already present. If it is not present, then we can go ahead and insert a_i within the run. Otherwise, we change the decreasing run into an increasing one by reversing it and then

performing insertion sort on a_i . Taking into account both decreasing and increasing runs reduces the number of times we perform insertion sort.

3. **Merging Blocks:** Classical mergesort requires n units of extra space for merging. However, timsort can be implemented with at most $n/2$ units of memory. When merging two runs, we can copy the smaller run into the buffer. This therefore frees up space within the original array to perform the merge more efficiently. Also note that merging can be done in both directions. So when we consume the run stack, and move things from the helper stack back onto the run stack, we can perform merges by popping two runs at a time. This avoids unnecessary copying of runs from the helper stack to the run stack. This in effect, sweeps through the entire input data in one direction at a time until the data is sorted. This makes for better caching performance.
4. **Gallopping:** When timsort notices that certain blocks of runs always get merged together, it enters gallopping mode to move these together as a single unit in subsequent runs. This has the effect of speeding up future runs.

For more information on timsort and its many optimizations, please see this Wikipedia article or this white paper from the original author Tim Peters.

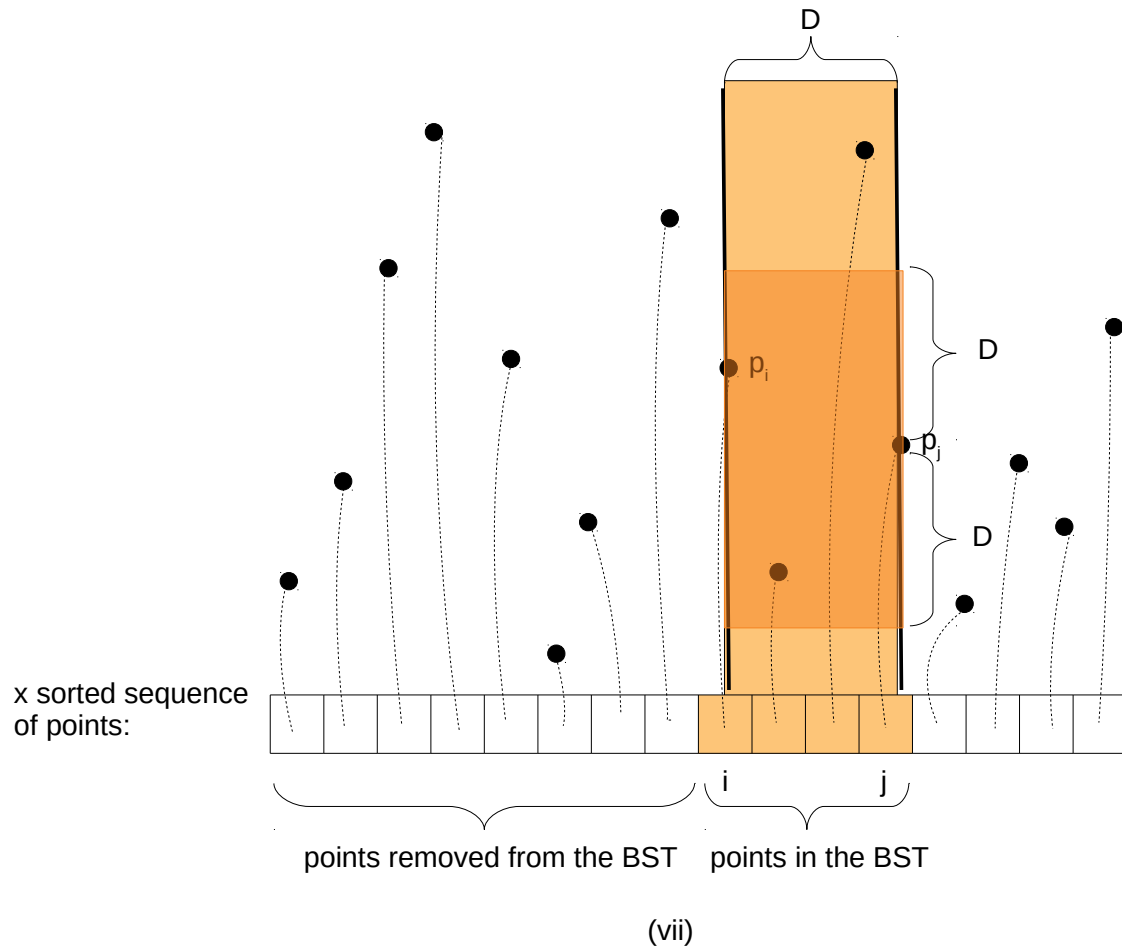
2 Closest Pair of Points

In this problem, we will compute the closest pair of points among n points on the 2D plane. Finding the closest pair of points is a common black-box in computer graphics applications, and in solving many problems in computational geometry. Other applications include collision detection in games, and finding nearest neighbors for machine learning classification.

2.1 Sweep Line Algorithms

This is a very common and powerful algorithm design technique used to solve many problems in computational geometry. It involves sweeping one or more lines across a high dimensional scene, while maintaining and querying several data structures.

Given n points on the 2D plane, a line can be swept from left to right if we sort all the points based on the x coordinate. In fact, for this problem, we will scan two lines i and j across the point set from left to right (see Figure (vii)). Every time line j advances to a new point p_j , line i advances to a point p_i such that the difference between the x coordinates of p_j and p_i is at most the current minimum distance D . Only points between p_i and p_j in the x coordinate are “relevant” in comparing p_j , since all points before p_i are farther than length D with p_j . If we store all the relevant points in a binary search tree data structure keyed on the y coordinate, then we can query this data structure to find all points close to p_j . This binary search tree needs to be updated every time the lines move. Specifically, we insert point p_j every time line j moves, and remove point p_i when line i moves. It is enough if we only compare p_j against points in the shaded $D \times 2D$ box shown in Figure (vi) because only these points differ in both x and y coordinates with p_j by at most D . Since the binary search tree supports successor and predecessor queries, we can iterate through all the successors (and predecessors) until we reach a point beyond the shaded box. If a closer pair is found, the minimum distance D is updated.



2.2 Note on Implementation

We can store the points in a class `Point2`². Note that we need to sort this set of points two different ways – one along the x -axis and the other along the y -axis. This can be achieved several different ways, for instance, using two different comparator functions. In Python, you can do the following.

```
from functools import cmp_to_key

def compare(l, r):
    '''
    Compares points l and r based on the x co-ordinates.
```

²The choice of using `Point2` is to differ from the `Point` class written in Problem 1.

```
'''
...
```

```
points.sort(key=cmp_to_key(compare))
```

The list of points in `points` is sorted using the `compare` function. Once sorted on the x co-ordinates, we can then iteratively scan this array using two indices i and j from left to right.

To successfully iterate over the predecessors or the successors, we need to store the points also in a binary search tree, keyed on the y co-ordinates. Unfortunately, Python does not have a built-in library that implements a binary search tree. Therefore, you need to build a binary search tree that is capable of storing point objects. Your binary search tree should include the following functions.

<code>insert(k)</code>	inserts a new object with key k into the binary search tree ³ .
<code>remove(k)</code>	removes an object with key k . Does nothing if k is not present in the binary search tree.
<code>predecessor(k)</code>	returns the container (node) of the largest object smaller than or equal to k , or <code>None</code> if k is smaller than the smallest object in the binary search tree.
<code>strict_predecessor(k)</code>	returns the container (node) of the largest object strictly smaller than k , or <code>None</code> if k is at most as small as the smallest object in the binary search tree.
<code>successor(k)</code>	returns the container (node) of the smallest object larger than or equal to k , or <code>None</code> if k is larger than the largest object in the binary search tree.
<code>strict_successor(k)</code>	returns the container (node) of the smallest object strictly larger than k , or <code>None</code> if k is at least as large as the largest object in the binary search tree.

Note that the predecessor and successor functions are needed to enumerate the relevant points for comparison within the $D \times 2D$ box as described above. In order for the binary search tree to work properly, you may overload the `<`, `<=`, and the `==` operators in your `Point2` class, that compares two point objects based on their y co-ordinates. Since the values of the x and y co-ordinates are floating-point values, the safest way to compare two floating values is using the `math.isclose` function.

```
import math

if math.isclose(l.x, r.x, rel_tol=1e-8):
    # l and r can be considered equal.
    ...
```

Before implementing your closest solution, please thoroughly implement and test your binary search tree. Without a correct implementation of a binary search tree, you will not be able to code the solution to the closest pair of points problem. To import your binary search tree (say in file `bst.py`) in your closest program, please put all relevant files in the same working directory, and put the following `import` statement in your closest program.

³The key is usually same as the object, if the object has an in-built comparison function, i.e., an object that overloads the `<` operator.

```
import bst
```

Please name your closest program `closest.py`

2.3 Input and Output

You are given a point set with 1 million points in the file `points.txt`. Each point is described on a single line. Please produce a single line of output that indicates the distance of the closest pair. Please use the `diff` command to check if your output matches exactly with the one shown in file `closest.out`.

```
python3 closest.py < points.txt > out
diff out closest.out
```

In the above statement, note that the input file is acquired through standard input (using input redirection), and the output is sent to standard output.

3 Postamble

3.1 Submission

Please submit two python programs, named `angular-sort.py` and `closest.py` that are the solution to the two respective problems. Please submit them through Canvas.

3.2 Proper Use of Libraries

The following Python packages are allowed. In general, you are allowed to use libraries for input and output, math and random functions, but not packages that provide you substantially more code for the assignment. If you are in doubt, please consult your instructor.

Python tuples, lists, sets and dicts as defined in the Python language. Also allowed are `deque` and `heapq` from `collections`.

You may use these **Geeks for Geeks** resources for more information, or please check out the official language documentations.

Python: <https://www.geeksforgeeks.org/python-programming-language/>