Preamble

As before. Submit the `KDTree.java` and `TestSqDist.java`.

Introduction

This topic deals with *k-dimensional trees* , or *kd trees*. This data structure allows one to efficiently find, among a set of points of $\mathbb{R}^k$, the nearest neighbor of a given point.

Kd trees are similar to binary search trees, except that the order used to distribute the descendants of a node n between the left and right subtrees depends on the depth of n in the tree: the nodes are sorted alternately according to each of their k coordinates.

More formally, a k-dimensional tree is a binary tree whose nodes each contain a point of $\mathbb{R}^k$. Moreover, if $p = (p_0, p_1, \ldots, p_{k-1})$ is a point placed in a node n at depth i and denoting r the remainder of the Euclidean division of i by k:

- for any point $q = (q_0, q_1, \ldots, q_{k-1})$ placed in the left subtree from n , we have $q_r < p_r$,
- for any point $q = (q_0, q_1, \ldots, q_{k-1})$ of the right subtree, we have $q_r \geq p_r$.

(We agree that the depth of the root is 0.)

Conventionally, a tree or subtree is represented by its root node. A node contains a point of $\mathbb{R}^k$, of type `double[]`, and pointers to the left and right subtrees. Each node also stores its depth in the tree (field `depth`).

```
class KDTree {
    int depth;
    double[] point;
    KDTree left;
    KDTree right;

    // construct a leaf
    KDTree(double[] point, int depth) {
        this.point = point;
        this.depth = depth;
    }
}
```

As in the course, the empty tree is represented by `null`.

1 Insertion

We want to add a method that inserts a point `a` into a tree while preserving the property defining kd trees. To do this, we need to know whether to place the point `a` in the left subtree or the right subtree.

1.1 Comparison

In the class `KDTree`, complete the method `boolean compare(double[] a))` that returns `true` whether the point `a` should be inserted into the right subtree, and `false` whether it should be inserted into the left subtree.

In order to reuse it later, we can isolate in a separate method the calculation of the difference in coordinates which is involved in the choice.

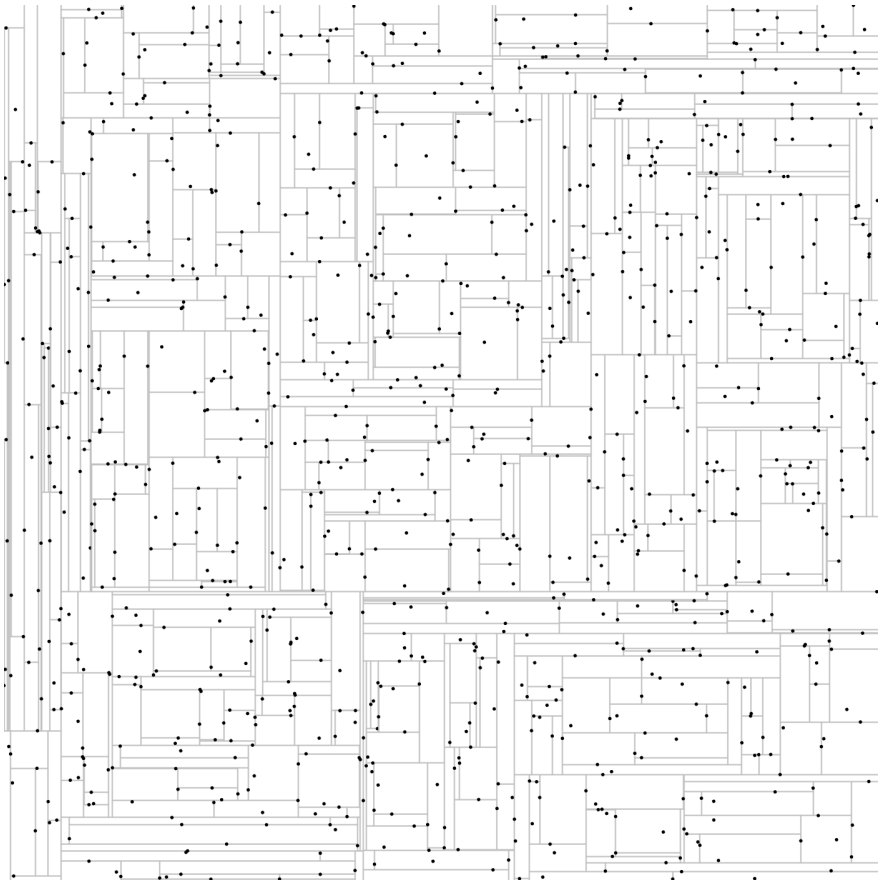
Test your code with `TestCompare`.

1.2 Insertion

In the class `KDTree`, complete the method `KDTree insert(KDTree tree, double[] a)` that adds a node containing the point `a` to the tree `tree` and returns the root of the resulting tree. The insertion method must preserve the property defining kd trees. (Be careful to correctly set the depth of the inserted point!) We are allowed to insert multiple copies of the same point.

Test your code with `TestInsert` and `InteractiveClosest`. The window produced by `InteractiveClosest` represents the 2-dimensional tree obtained after successively inserting 50 points chosen uniformly in $[0, 1] \times [0, 1]$. Type `'+'` or `'-'` to have more or fewer points.

Here is for example what we get with 1016 points:



2 Nearest point

We are now interested in finding the point of a kd tree which is closest to a given point `a`, in the sense of Euclidean distance.

2.1 Distance

Complete the method `double sqDist(double[] a, double[] b)` that calculates the square of the distance between two points.

In the class `TestSqDist`, write some tests to verify that `sqDistit` works correctly.

For example, the square of the distance between the points $(-1, 1)$ and $(1, -1)$ is equal to 8. Test some other representative examples of the possible inputs. Remember that `sqDist` must work in all dimensions ($k = 0, k = 1, k = 1000, \dots$).

2.2 Naive version

The naive algorithm for finding the nearest point simply consists of traversing all the points in the tree, maintaining in a variable `champion` the closest point encountered so far.

Complete the method `static double[] closestNaive(KDTree tree, double[] a)` that returns the closest point to the point `a` in the tree `tree`, or `null` if the tree is empty.

(An auxiliary method may be useful.)

Test your code with `TestClosestNaive`.

2.3 A more efficient algorithm

The complexity of the previous method `closestNaive` is linear in the size of the tree. Much better can be done using the kd tree structure.

We observe that for any subtree `t` depth-dependent i in our kd tree, the points in the left subtree of `t` are separated from the points in the right subtree of `t` by the cutoff hyperplane defined by `x[i%k] = t.point[i%k]`.

To find the nearest neighbor of a point `a` in `t`, we can therefore start by looking for the nearest neighbor `p` in the subtree of `t` located on the same side of the cut-off hyperplane as `a`. If `a` is closer to than `p` to the cut-off hyperplane, then there is no point in exploring the other subtree of `t`(situation 1). Otherwise, we must also consider `t.point` and the points in the right subtree of `t` (situation 2).

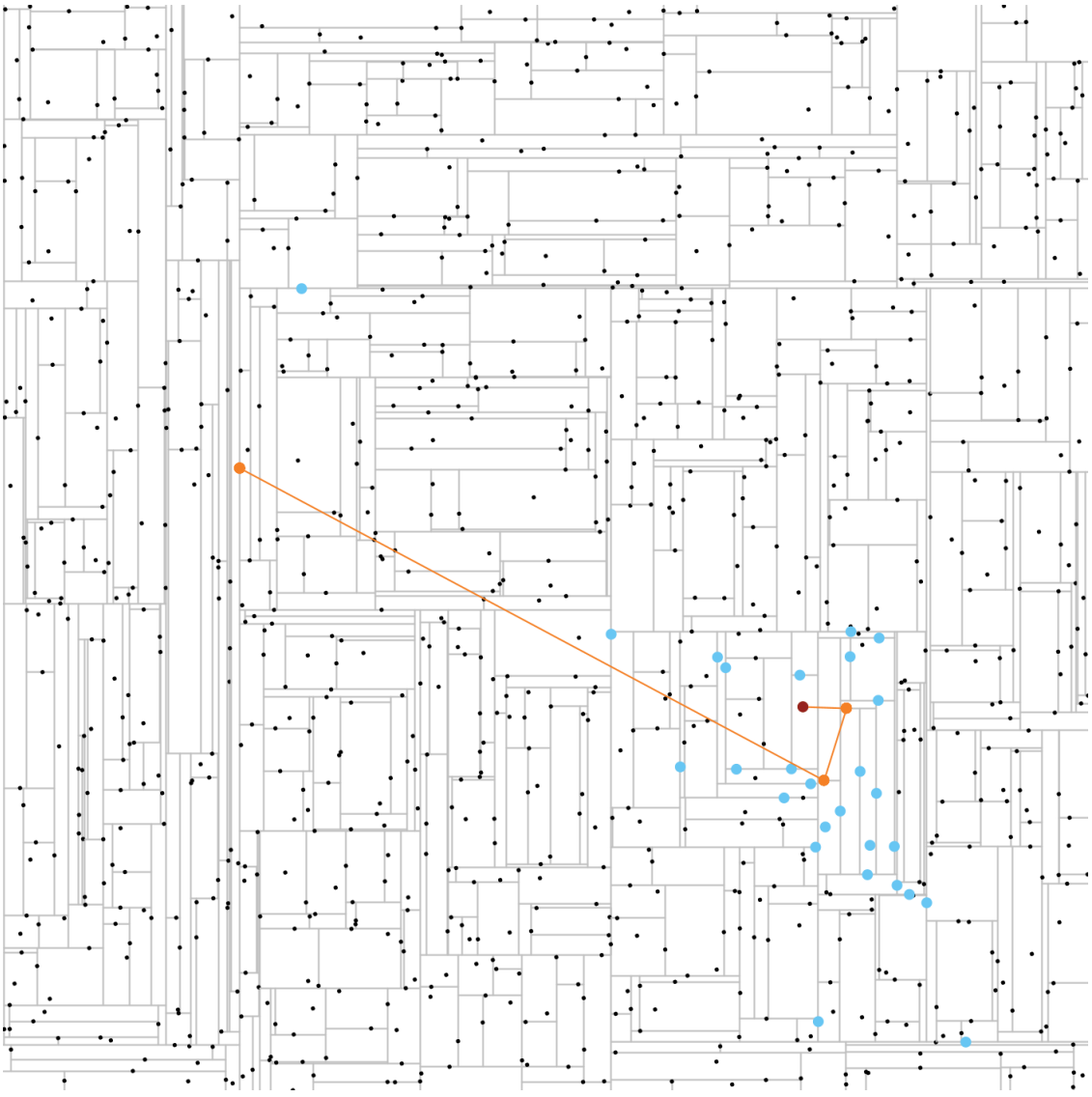
Complete the methods:

- `static double[] closest(KDTree tree, double[] a, double[] champion)` which searches for the closest point to point `a` in the set consisting of the tree `tree` and `champion` as described above;
- `static double[] closest(KDTree tree, double[] a)` which returns the closest point to `a`; if the tree is empty, we return `null`.

At the start of each recursive call, the three-parameter version of `closest` should call `InteractiveClosest.trace(tree.point, champion)` with `champion` the closest point known at that point as the parameter (see the provided code).

Test your code with the classes `TestClosestOptimized` and `InteractiveClosest`.

The class `InteractiveClosest` allows you to visualize the algorithm's progress. Click to find the point closest to the cursor. Hold down to continuously refresh. The nodes traversed are displayed in blue, and the successive champions in orange. If the orange line is very broken (say, more than 10 points), there is a problem.



3 Selecting an optimized color palette

For common applications, the colors of images viewed on a screen are encoded with 24 bits, 8 bits for each of the three colors red, green, and blue, which gives one point. To compress an image, the GIF image format selects 256 colors, from the 16777216 possible, to best represent the image. The

method of fixing 256 colors independently of the image, such as the web palette, gives very poor results. It is necessary to select a color palette appropriate to the image.

The goal of this part is to write the function `palette` that returns an object of type `Vector<double[]>` containing a prescribed number of points capable of approximating the points of a given kd tree.

Figure 1: witness photo

Figure 2: web palette

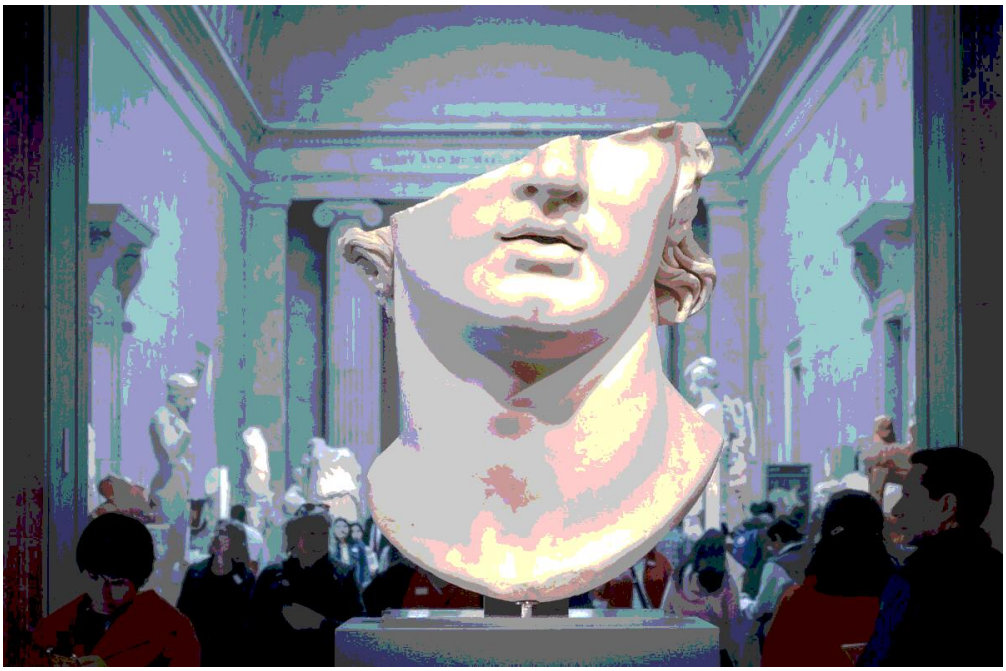
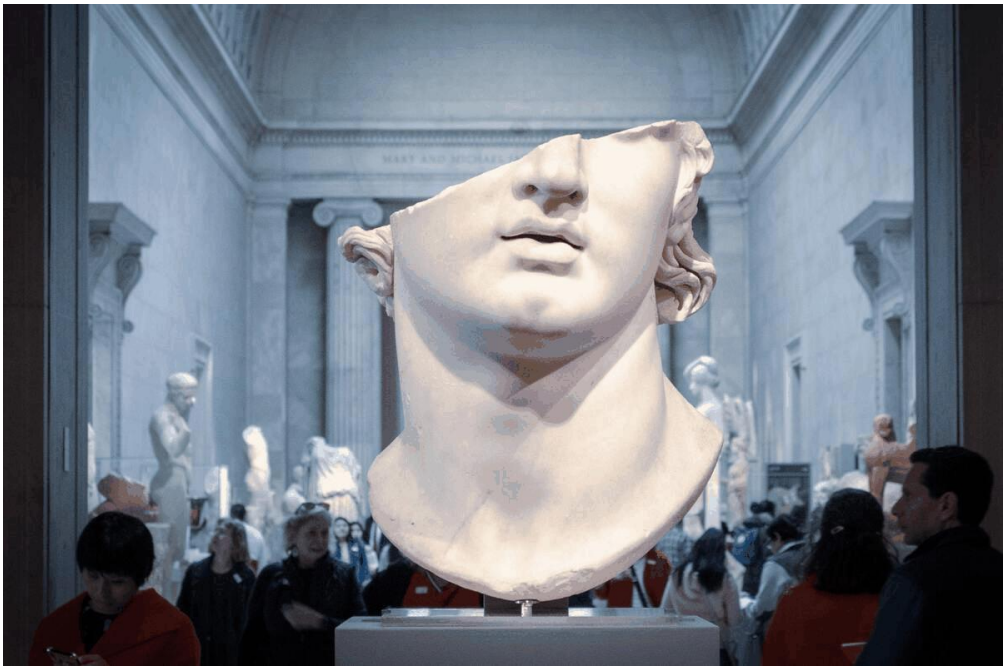


Figure 3: optimized palette



3.1 Some auxiliary functions

Complete the following methods

- `int size(KDTree tree)` which returns the number of points in the tree;
- `void sum(KDTree tree, double[] acc)` which calculates the sum of the points of the tree, adds it to `acc`;
- `double[] average(KDTree tree)` which returns the isobarycentre point (i.e. the average value) of the points in the tree, or `null` for an empty tree.

Test your code with `TestAverage`.

3.2 The method `palette`

The palette is composed from a 3D tree containing the colors of $20,000$ pixels randomly chosen from the image. The 256 colors will be obtained by calculating the isobarycentre of the points of well-chosen subtrees.

Complete the method `static Vector<double[]> palette(KDTree tree, int maxpoints)` that returns an array of `maxpoints` colors based on tree. Several strategies are possible.

Test your code with the class `ColorPalette`. A computation time of more than 1 second or a score above 10 indicates a problem. Your teachers have gotten down to around 4.3. Share your method if you do less!

It's okay to experiment. Start with a naive method for choosing `maxpoints` subtrees whose isobarycenters we'll take, then refine. At first, consider `maxpoints` in an order of magnitude rather

than a strict constraint. Note that in our application, the three-dimensional tree `tree` from which we extract a palette is not necessarily balanced (it's a random tree).

Perspectives

Beyond palette selection, dither methods are essential to avoid the flatness inherent in methods that replace a color with its nearest neighbor in a fixed palette. But that's another story...