Oblig 5 in IN3030, Spring 2019. The convex hull of a set of points - a recursive geometrical problem.

Delivery deadline: Wednesday May 10th 2019, 23:59.

A set of points P in a plane consists of n different points pi (i =0,1,2,…,n-1) with integer coordinates (no decimals) and we can assume that *not all* points will be on the same line. The convex hull for this set of points is a polygon with lines between some of these points so that all the other points are on the inside of this polygon and also so that all the inner angles of this polygon ≤ 180. In simpler terms: there are no inwards dents on the set of lines running around the set of points (fig1).

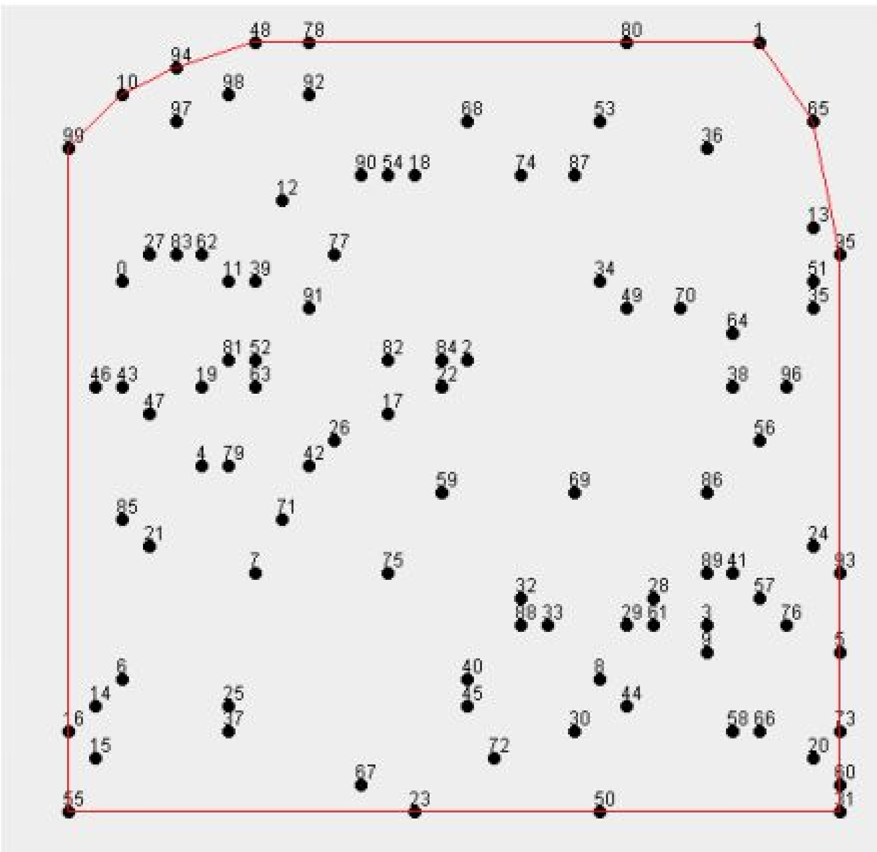


Fig1: the convex hull of 100 random points in a plane (the same that are used in the assignment)

When jotting down such a convex hull, we start with a random point along the hull and jot down the following points that make up the hull *counter-clockwise*. In fig1 it could for instance be 16, 55, 23, 50, 31, 60, 73, 5, 93, 95, 65, 1, 80, 78, 48, 94, 10, 99. Notice that if there are more points on a line (for instance, 31, 60, 73, 5, 93, and 95), all of those points should be included in the hull. You are in other words not allowed to go straight from point 31 to 95, you have to specify all the lines 31-60, 60-73, 73-5, 5-93 and 93-95 with no skips in-between. If it is used in in your solution you can always assume that it is possible to choose the points with the lowest and highest x and y values so that there are 4 different points (see point 55 and 31 in fig1).

How do we find the hull of n points?

We can see that there are two obvious points on the hull - that with the lowest value for x and that with the highest value for x. If there are more points with the same values, just choose one of those with the lowest and one of those with the highest values for x (in the example, for instance 16 and 5). The rest of the algorithm is based on a simple geometrical sats **(editor's note: sats is the norwegian word, don’t know what it translates to. doesn’t matter though)**

Equation for a line

Each line from a point p1 (x1, y1) to p2 (x2, y2) can be written as such:

$$ax + by + x = 0$$

Where: **a** = $y1 - y2$, **b** = $x2 - x1$ and **c** = $y2 * x1 - y1 * x2$

Notice that this is a **straight line *from* p1 *to* p2.**

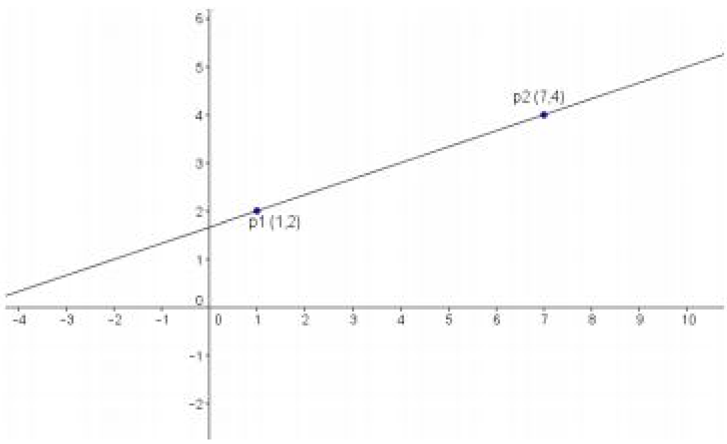


Fig2: A line from p1 (1,2) to p2 (7,4) has the line equation:
$(2 - 4)x + (7 - 1)y + (4 * 1 - 2 * 7) = 0$; i.e. $-2x + 6y - 10 = 0$

The line equation means that all points on this line fits in to the formula; if we add x and y to any point on this line we will get 0 as an answer. We can also say that this line splits the plane in two - the points to the left side seen from the direction of the line - from p1 to p2 - and the points on the right side in the direction from p1 to p2.

The distance from the line to other points

If we add points that are not on the line p1-p2, we can see that all points, for example p3 (5,1) on the right side of the line gives a number <0 in the line equation ax + by + c, og all points to the left of the line, like p4 (3,6) gives a number >0.

The further from the line the points are, the larger negatives or positive values they will have. Generally speaking we can say that the distance perpendicular down from a point p (x,y) to a line ax + by + c is:

$$d = \frac{ax + by + c}{\sqrt{a^2 + b^2}}$$

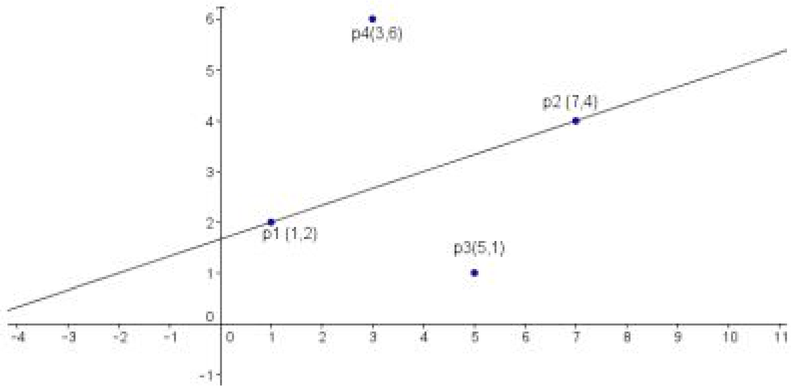


Fig3: The distance from a point to a line. The line divides the plane in two; those points with negative distance (to the right) and those with positive distance (to the left) from the line.

We look at two points in fig3: p3 (5,1) and p4 (3,6), find the distance from p4 to the line p1-p2 is:

$$\frac{-2*3+6*6-10}{\sqrt{(-2)^2+6^2}} = \frac{20}{\sqrt{40}} = 3{,}16..,$$

While the distance from p3 (5,1) to p1-p2 is:

$$\frac{-2*5+6*1-10}{\sqrt{40}} = \frac{-14}{6{,}32} = -\,2{,}21..$$

The algorithm for the convex hull

We can now formulate the algorithm for the convex hull after making another observation: The point that has the longest (largest) negative distance from a line pi-pj is itself a point on the convex hull (see fig4, point P and line I - A).

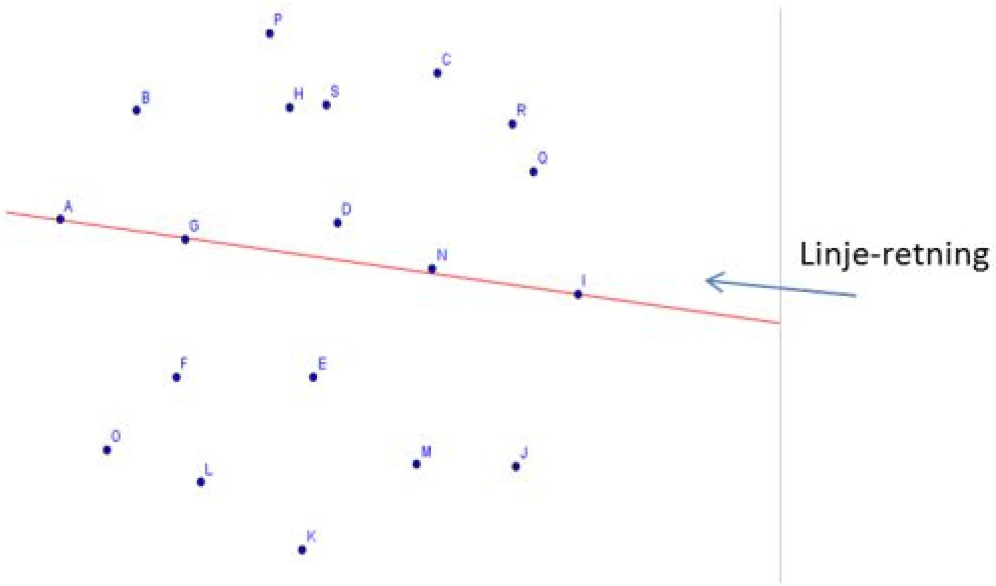


Fig.4. Starten på å finne innhyllinga fra maxx (I) til minx (A).

Fig4: The start to finding the convex hull from max_x (I) to min_x (A). **(editor's note: "line-retning" translates to "direction of line"**

Algorithm:

1. Draw a line between the two points we know are on the hull from max_x to min_x (I to A in fig4)
2. Find the point with the largest negative distance (or 0) from the line (in fig4 it's P). If multiple points have the same distance, just choose one.
3. Draw a line from the two points on the line to this new point on the hull (in fig5: I-P and P-A)
4. Continue recursively from the two new lines and for each of them find a new point on the hull with the largest non-positive distance (less than or equal to 0)
5. Repeat step 3 and 4 until there are no more points on the outside of the lines
6. Repeat steps 2-4 for the line min_x-max_x, and find all points on the hull under it.

This process is illustrated in fig5. You can see the process of finding the points on the hull above the line I-A. Du can expect to find about 1.4 * sqrt(n) elements in the convex hull from n randomly chosen points on the plane, although all values from 3 to n are possible (we get n if all points are on a circle).

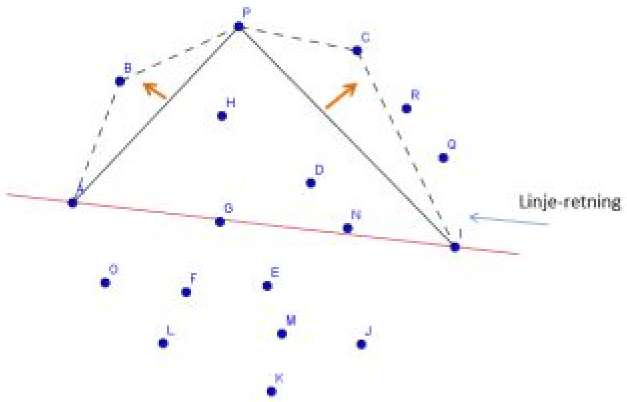


Fig5: We recursively find the point P on the hull from the line I-A (that with the largest negative distance from max_x-min_x), then C from P-I, R from I-C and lastly Q from I-R.

You are supposed to program this recursive algorithm first sequentially, and then make a parallel version and get speedups for n=100, 1000, 10 000, …, 10 million.

On generating n different points

a) The data structure for your n points should be two int arrays x and y, each of them with length = n.

```
int []x = new int[n], y = new int[n];
```

In order to fill these arrays with x and y values that does not create equal points, you will use the pre-written class NPunkter which you will find in the Oblig5 file.

b) For each value of n in your program, you should first create an object of the class NPunkter **(editor's note: NPunkter translates to NPoints)**:

```
NPunkter p = new NPunkter(n);
```

and then fill x[] and y[] with a call to the method in the object p:

```
p.fyllArrayer(int [] x, int [] y);
```

(editor's note: "fyllArrayer" translates to "fillArrays)

The reason why you are supposed to use this class, which you should not alter, is that all the deliveries for a given value of n should have the same points. Then it will be much easier for the correctors to correct the obligs (additionally, it is not easy to relatively quickly generate up to 10 million random points that you are certain are not equal, but that task is not a part of Oblig5).

c) The process of generating n points should be included in the reported runtimes.

Efficiency

a) You are not really interested in the actual distance from a point to a line, it is just used as a measure to see which point “wins”. The formula for the distance d can as such be simplified.
b) It will be more time-consuming if you have to run through the entire set of points P to find the few points that are on the outside of a line. That is why you should let your program test on smaller and smaller sets of points
c) Remember to be critical regarding only using threads instead of recursion (think “layers”, like quicksort). Threads should only be used at the top of the recursive tree.
d) Is ArrayList<Integer> fast enough to hold large sets of points, or should you use the class IntList with the “same” methods you need, and where the integers you are supposed to store resides in a simple integer-array?

The assignment

In Oblig 5 you are supposed to.

- Program a sequential version of the algorithm described.
- Parallelize the sequential implementation according to method 1 or 2, as described in week 14 lecture slides **(editor’s note: last year’s slides, remember.)**
- Use the precode *NPunkter17.java* and *TegnUt.java* (published on the course page **editor’s note: and on my github. note that I might make slight alterations to the precode so that they function similarly to oblig 3 and 4 precodes - i.e. ‘addPoint’ similar to ‘addFactor’ from the oblig 3 precode)**

Requirements

Oblig 5 is to be delivered on Devilry by Friday May 10th 23:59. The delivery should consist of:

a) The code for both the sequential and parallel solution, wrapped in a .zip file.
b) An output of the convex hull for n = 1000
c) A report with both a table and a curve showing speedup as a function of n (= 100, 1000, … 10 million) and your assessment as to why your runtimes are as low as they are, and why you get the speedups you get. NOTE: runtime for n = 100 milion should be less than 14s and n = 10 million should be less than 2s. Include specifics of the computer you ran your tests on (modell, clock frequency, number of cores) and the size of the main memory, and if possible the size of the caches and delay (run the program latency.exe if you are on a Windows computer) and gladly a picture created by TegnUt when n=100.

Oblig 5 report recommendations have been published on the course page **(editor’s note: eric will make and publish this at the same time as the assignment for 2020 is published)**

Tips

The following points are not requirements but describes areas where you might run into some trouble.

1) **The Sequential solution**
    a) **How to represent a point**
    Use the index in x[] and y[] - the contents of those will not change during runtime; they are only read.
    b) **Debugging**
    Very few, if any, manage to get their code right on the first run. We need to debug the code and that might be difficult on a graphical problem like this one. You can for instance use TegnUt to draw your lines and use a small number for n while debugging (n < 200). The call to TegnUt is made as such:

    TegnUt tu = new TegnUt (this, koHyll);

    The first line is to draw the points and the convex hull that is expected to be added in to an IntList koHyll **(editor's note: I've named the IntList 'CoHull' in my example solution)**
    c) **Find points on the convex hull in the correct order**
    Tip: du are using two methods for the sequential solution (assuming here that you are using ArrayList, but this can trivially be changed by swapping it with a faster IntList):

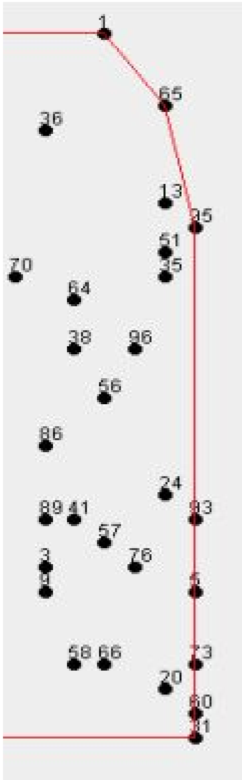
    seqMethod() which finds min_x, max_x, and starts the recursion. If you start on max_x-min_x, add max_x to the list of hull points (but not min_x yet), and start your recursion with two calls to the recursive method:

    seqRec(int p2, int p2, int p3, ArrayList m, ArrayList koHyll) which receives a set of points m (containing all points above or below the line p1-p2) and p3 which has already been found as the point with the largest negative or positive distance from p1-p2. koHyll is the set to which you should add the points from the convex hull in the correct order.

    Notice that you first "do a recursion" on the right side (line I-P in fig5) and then on the left side (line P-A) because we want the points to be added in a counter-clockwise order.

    You can let each call to seqRec add one point: p3 to the list of points on the hull, but where in your code should you do so? When do you add min_x to the list?

d) **Include all points on the hull in cases where multiple points are on the same line (distance = 0), like on the right side in fig6, and in correct order.**



Remember that if you find that the largest negative distance == 0, there is no need to include p1 or p2 as possible new points (they have already been found). Say that you have found p1=35 and p2=5 and you want to find all the points on the line between p1 and p2 (60 and 73) and you then have to test if a new point p3 has both x and y coordinates corresponding to those of p1 and p2. Then you can find one of the points with a call to seqRec above the line p1-p2 (31-5). Repeat recursively until there no longer are any points between the new p1 and p2.
The other points on the line (for instance, 5-95) will be found recursively similarly to 60 and 73.

2) The parallel solution

a) You can/should not use the model2 code directly, but should instead divide the parallel case in two methods. The first parallel call from the main-thread (between the two lines where you time the parallel implementation). It will then find out how far down the call tree we should use threads. If we have k cores on our computer, it might be reasonable to move down to a layer in the call tree where there are approximately k nodes horizontally in the tree. Example: if we have 8 cores, a call the top of the tree level 1, then level 4 will have 8 nodes. Up until this level new threads will replace recursive calls, and from that level regular recursive calls can be used from each of the threads we started.

b) The first parallel method called from and being ran from the main-method creates two threads. The following threads are born from these two threads and their offspring-threads. These are started in the parallel method, which is called from the run()-methods.

c) The other parallel method has the same parameters as the sequential one: p1, p2, p3, and the set of points to look within in addition to two more: the level of the tree and the set (IntList) of points on the hull that the thread has found.

d) The other parallel, recursive method decides whether to keep on creating threads or if we are going to start using recursive calls. Prior to that, we have found the point that has the largest negative distance and the set of points for the next call and we have also declared and generate to sets; one for each of those to recursions that are supposed to hold the convex hull that particular call finds.

e) This method strongly resembles the sequential method, and after we have reached the level where we seize to make new threads, we can actually just use the sequential recursive method.

f) Remember that when we start threads we have to first create both of them, and then wait (for instance by using join()) for them. If we do not, we do not get paralyzation.

Finding the points of the convex hull in the correct order (parallel solution)

Another problem we will face is adding points in the correct order. Were it is more clear when to add points to the convex hull in the sequential solution, it is not as predictable in the parallel solution. A way to solve this is to use:

1. Threads started at the top of the recursion tree only traverse down to a certain level for all nodes of the tree (the levels depends on the amount of cores at our disposal)..
2. After we can no longer start new threads, each thread moves on to the sequential solution (on its side of data)
3. From each of these sequential executions we will get the points in the correct order (counter-clockwise)
4. The problem then is to join these sets of points that are individually in the correct order. Do this: add first the points the right-hand thread has produced, then the point found by the node itself, and then the points from the left-hand thread.
5. Point 4 is then executed (upon return after finding its points) upwards until we reach the top and is done.