Федеральное государственное автономное образовательное учреждение высшего образования
"Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского"(ННГУ)

Институт информационных технологий, математики и механики

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

«Построение выпуклой оболочки для компонент бинарного изображения»

Выполнил:
Студент группы 381708-2
Асянин Михаил Вадимович
_____ Подпись

Проверил:
Ассистент кафедры МОСТ, магистр прикладной математики и информатики.
_____ Волокитин В. Д.

Нижний Новгород
2020.

Оглавление

1. Введение

Основная цель данной работы – реализовать алгоритм, способный выделить компоненты на бинарном изображении и далее на их основе построить выпуклую оболочку.

Для начала необходимо знать и понимать термины бинарное изображение, выпуклая оболочка, а также как с ними эффективно работать и применять.

Бинарное изображение – разновидность цифрового растрового изображения, когда каждый пиксель может представлять только один из двух цветов. Значения каждого пикселя условно кодируются как 0 и 1. Значение 0 соответствует заднему плану или фону, обычно является черным цветом; 1 – переднему плану (объекту), обычно является белым цветом. Для нас это особенно удобно, т.к. бинарное изображение можно представить в виде одномерного или двумерного массива размерностью умноженной длины на его ширины, заполненного значениями 0 и 1 в каждом пункте матрицы. Номер строчки и столбца для каждого пикселя можно интерпретировать как координаты x и y в пространстве изображения. Таким образом, можно применить алгоритмы построения выпуклой оболочки из набора связанных областей, состоящих из точек.

Компонентой в изображении считается связанная, где окрестность пикселя существует только по горизонтали и вертикали, фигура, принадлежащую объекту, являющейся 1.

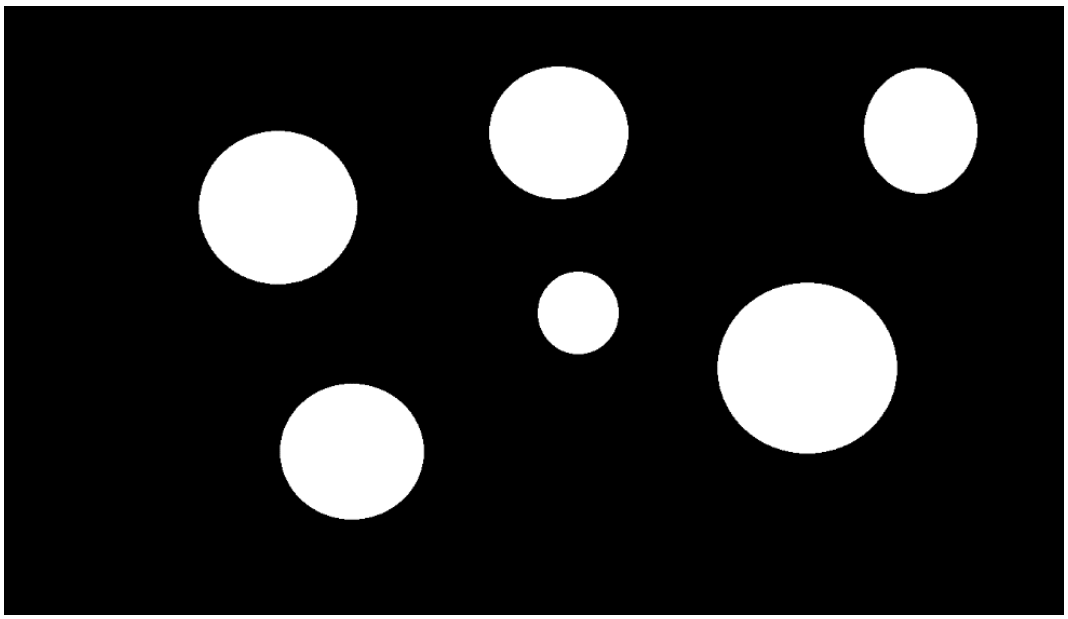

Рис. 1: Бинарное изображение с несколькими компонентами связанности

Выпуклая оболочка множества точек — это выпуклое множество точек,

где все точки компоненты также лежат на нем. Далее, строить мы будем не тривиальную выпуклую оболочку, а минимальную.

Минимальная выпуклая оболочка множества точек — это минимальная по площади выпуклая оболочка.

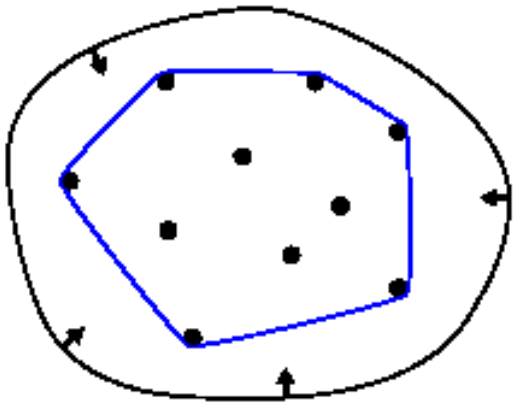

Рис. 2: Выпуклая оболочка и минимальная выпуклая оболочка

По сути, минимальная выпуклая оболочка это знакомый всем со школы выпуклый многоугольник, состоящий из точек входного множества так, что все остальные точки этого множества лежат внутри него.

2. Постановка задачи

Для входного изображения выделить компоненты связанности, построить для них внешнюю и внутреннюю выпуклые оболочки, а так же продемонстрировать результат работы программы. Полученный алгоритм необходимо распараллелить с использованием технологий OpenMP, TBB, и Threads так, чтобы он корректно работал для произвольного числа процессов, работающих с исходным бинарным изображением.

3. Описание алгоритмов

3.1 Выделение компонент на бинарном изображении

Предположим, что исходное изображение мы перевели в бинарное и занесли значения пикселей в массив. Далее необходимо пройтись по нему и отделить компоненты друг от друга (выделить связанные области).

Идея алгоритма основана на использовании уголка (маски) ABC. Проход по изображению (массиву) с ее помощью осуществляется слева направо и сверху вниз. При этом мы считаем, что за границей изображения объектов нет.

Рис. 3: Маска ABC и ее возможные положения

Рассмотрим подробнее все позиции маски:

- Позиция под номером 0, когда не размечены все три компонента маски — в этом случае мы просто пропускаем пиксель.
- Позиция под номером 1, когда помечен только элемент A — в этом случае мы говорим о создании нового объекта — новый номер.
- Позиция под номером 2, когда помечен элемент элемент B — в этом случае мы помечаем текущий пиксель A меткой, расположенной в B.
- Позиция под номером 3, когда помечен элемент элемент C — в этом случае мы помечаем текущий пиксель A меткой, расположенной в C.
- Позиция под номером 4, тогда мы говорим о том, что метки (номера объектов) B и C связаны — то есть эквивалентны и пиксель A может быть помечен либо как B либо как C.

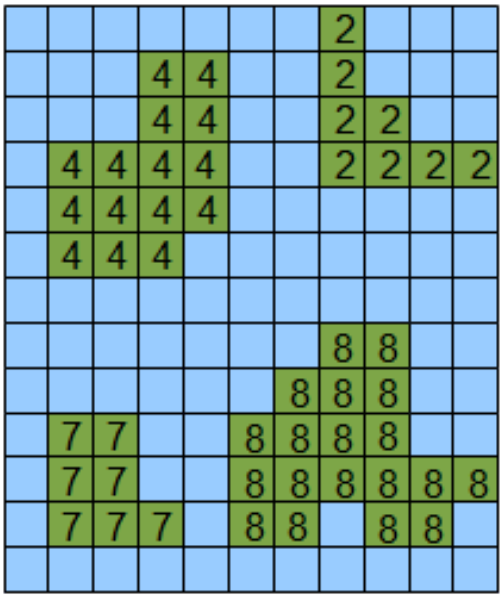

Рис. 4: Результат выделения компонент

Как видно из рисунка 4, компоненты пронумерованы непоследовательно, но для нас это и не важно. Главное, чтобы мы могли их отличать.

3.2 Построение внешней выпуклой оболочки с помощью прохода Джарвиса

Среди точек a_1, ..., a_n выберем самую нижнюю (по координате y) левую (по координате x) точку b_1, она точно будет являться вершиной выпуклой оболочки. Теперь для каждой текущей точки b_i ($1 \leq i \leq n$) ищется такая точка $b_i{+}1$ (из оставшихся + самая первая), в которой будет образовываться наибольший угол между прямыми $b_i{-}1$ b_i и b_i $b_i{+}1$. Уточним, что при поиске второй точки в качестве b_0 берется точка с координатами (x_1 – 2, y_1) (к примеру). Найденная таким образом точка $b_i{+}1$ будет следующей вершиной выпуклой оболочки. При этом нет необходимости искать сам угол. Достаточно вычислить его косинус (через скалярное произведения, используя координаты точек). При этом ищется минимальный косинус (чем он меньше, чем больше угол). Нахождение вершин продолжается до тех пор, пока $b_i{+}1 \neq b_1$. Если для нескольких точек косинус одинаковый (они лежат на одной прямой), то выбирается наиболее удаленная от текущей.

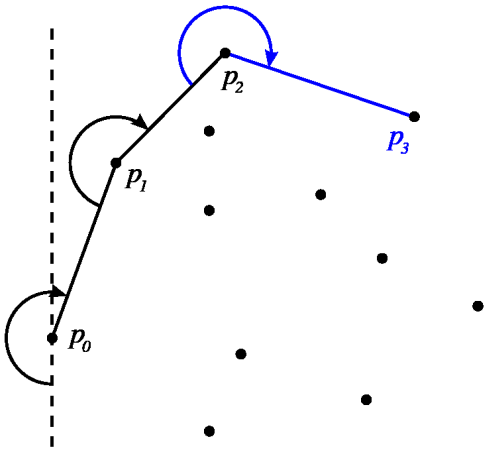

Рис. 5: Построение выпуклой оболочки проходом Джарвиса

3.3 Построение внутренней выпуклой оболочки

Внутренняя выпуклая оболочка должна соединять пиксели, принадлежащие компоненте, попавшей во внешнюю выпуклую оболочку. При этом из каждой компоненты выбирается только один пиксель.

Для этого при проверке ненулевых пикселей изображения заводятся два вектора: корректности и запрета. Вектор корректности будет проверять, есть ли компонента, которой принадлежит текущий пиксель во внешней выпуклой оболочке, а вектор запрета – брали ли уже из этой компоненты пиксели. Итак, для каждого пикселя проверяется есть ли его компонента в векторе корректности. Если нет, то пиксель просто пропускается. Если есть и при этом его компоненты нет в векторе запрета, то пиксель добавляется в внутреннюю выпуклую оболочку, а его компонента – в вектор запрета. Если же пиксель есть и в векторе корректности и в векторе запрета, то он также пропускается.

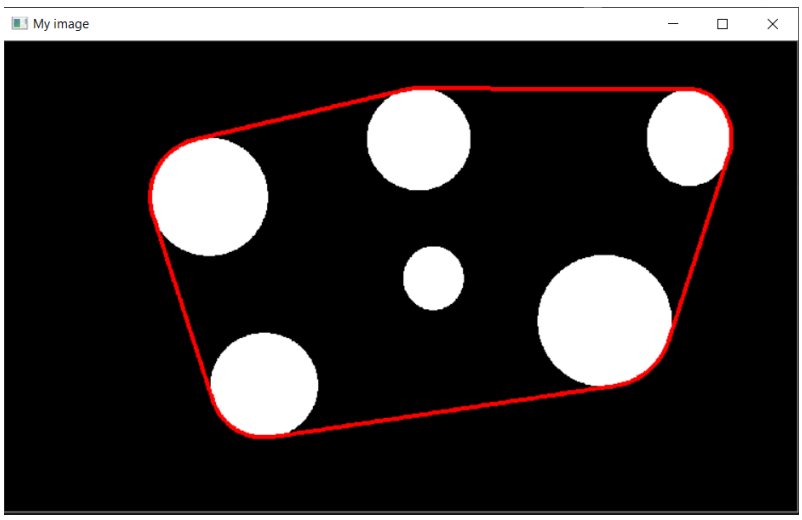

Рис. 6: Внешняя выпуклая оболочка

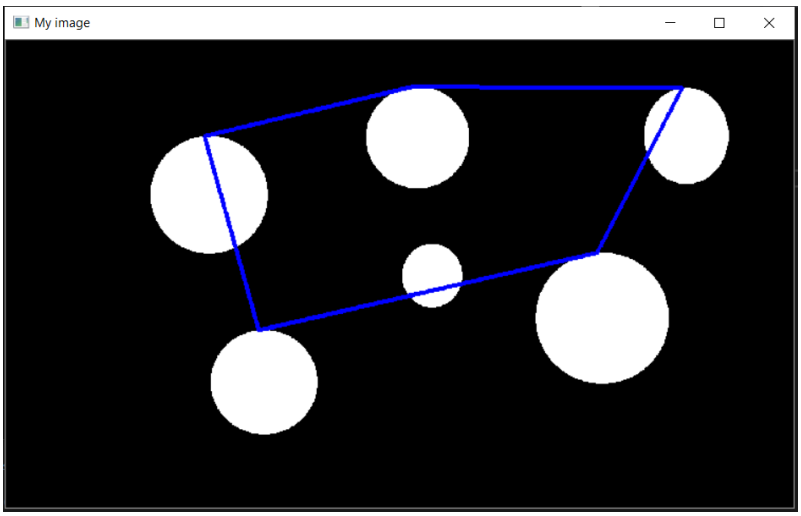

Рис. 7: Соответствующая ей внутренняя выпуклая оболочка

На рисунках 6 и 7 можно увидеть результат выделения как внешней, так и внутренней выпуклых оболочек

4. Схема распараллеливания

Из описания алгоритма выделения компонент можно увидеть, что при не совпадении значений в элементах B и C возникает необходимость изменения в уже пройденной части изображения или массива. Таких вызовов, как правило, достаточно много из-за чего они занимают существенное время. Поэтому именно эта часть и подвергается распараллеливанию с использованием обеих технологий. Принцип у них одинаковый: строчки матрицы (изображения) от начальной до текущей делятся между процессами, те находят в них пиксели, помеченные как C, и помечают их как B. Соответственно, в версии OpenMP это реализовано при помощи дерективы #pragma omp for. В версии TBB – tbb::parallel_for с использованием лямбда-выражения в качестве замены объявлению класса функтора. В версии Threads, каждому потоку достается свое количество пикселей для меток, при этом используется лямба-выражение с той же целью, что и в TBB.

5. Описание программной реализации

В программе имеется четыре модуля: task4.cpp, task5.cpp, task6.cpp, и task7.cpp. В первом из них реализована последовательная версия алгоритма, в то время как во втором по четвертую, помимо последовательной, присутствуют так же omp-, tbb- и threads-версии. Пройдем сначала по методам, присутствующих во всех этих модулях. В первую очередь это случайная генерация бинарного изображения заданного размера из целочисленных типа в бинарный (setRandom), как альтернатива загрузке реального изображения. Далее, последовательная версия выделения компонент на изображении (функция findConnectedComponents). В первом модуле она является основной, а во втором по четвертый необходима для сравнения времени работы с ее параллельным аналогом. После этого необходимо найти внешнюю выпуклую оболочку (метод Jarvis и два вспомогательных – betweenPoints для определения расстояния до точки и turnPoint для вычисления косинуса между векторами).

Что же касается второго по четвертый модули, то в них присутствуют методы findConnectedComponents_OMP, findConnectedComponents_TBB, и findConnectedC соответственно, для нахождения компонент связанности с использованием нескольких потоков.

6. Эксперименты

Эксперименты проводились на ПК с следующими параметрами:

- Операционная система: Linux 5.3.0-51-generic
- Процессор: Intel(R) Core(TM)2 Duo CPU T6600 @ 2.20GHz
- Оперативная память: 4 Gb
- Версия g++: 9.2.1 20191008

В таблице 1 приведена зависимость времени работы алгоритма при разном числе потоков.

Размер матрицы: 4500x4500.

Таблица 1: Время работы алгоритма в зависимости от числа потоков.

Количество потоков	Время работы последовательного алгоритма	Время работы OpenMP	Время работы TBB	Время работы Threads
1	15.3118	14.5277	13.216	15.2376
2	15.3491	13.2124	11.9182	13.2173
4	15.3245	13.2418	12.5256	12.3244

Исходя из этой таблицы можно сделать вывод, что программа работает эффективно, с ростом числа потоков время работы не всегда линейно уменьшается. Стоит отметить, что растет оно не так быстро, как, ожидалось. Объясняется это тем, что алгоритм выделения компонент на изображении не везде распараллелен, а только часть меняющая метки с C на B, не позволяя получить максимальное ускорение. На одном процессе обе параллельные версии работают медленнее, чем последовательная, в следствии накладных расходов на создание потока и инициализацию необходимых данных. В остальном же время работы как TBB, так и OpenMP версии сопоставимы. Они обе дают вполне неплохое ускорение. TBB работает чуть медленнее в силу использования лямбда-выражения, которое, в отличие от класса, при каждом новом заходе в него (а таких много) заново инициализирует нужные данные. Нужно обратить внимание на работу при 4 потоках: из-за ограничений возможностей процессора, оно не будет работает быстрее 2 потоков.

7. Заключение

При исследовании данной темы, были найдены различные способы реализации алгоритмов выделения компонент на изображении и построение выпуклой оболочки на ней, называемая алгоритмом Джарвиса. Данная работа позволила наиболее полно понять способы хранение и интерпретации бинарного изображения, в конкретном случае это одномерный компактный битовый массив.

Стоит отметить, что мне удалось реализовать все необходимые алгоритмы от выделения компонент до построения выпуклой оболочки специальным алгоритмом для компонент бинарного изображения. Кроме того, алгоритм удалось успешно распараллелить с использованием трех технологий – OpenMP, TBB и стандартной библиотеки Threads.

Литература

[1] Сообщество IT-специалистов Habr. Подсчет числа объектов на бинарном изображении: https://habr.com/ru/post/119244/

[2] Википедия: свободная электронная энциклопедия на английском языке: https://en.wikipedia.org/wiki/Binary_image

[3] Википедия: свободная электронная энциклопедия на английском языке: https://en.wikipedia.org/wiki/Gift_wrapping_algorithm

Приложение

task4.cpp

```
#include <iostream>
#include <bitset>
#include <string>
#include <vector>
#include <time.h>
#include <cmath>
#include <cfenv>
using namespace std;

#define POWER_OF_TWO 3 // 2 ^ 3
#define HIGHEST_VALUE_IN_BYTE 256 // 11111111
#define BIGGEST_SINGLE_BIT 128 // 10000000
#define INFINITE 10000000

#pragma STDC FENV_ACCESS ON
double roundFloat(double x) {
    std::fenv_t save_env;
    std::feholdexcept(&save_env);
    double result = std::rint(x);
    if (std::fetestexcept(FE_INEXACT)) {
        auto const save_round = std::fegetround();
        std::fesetround(FE_TOWARDZERO);
        result = std::rint(std::copysign(0.5 + std::fabs(x), x));
        std::fesetround(save_round);
    }
    std::feupdateenv(&save_env);
    return result;
}

typedef unsigned int IMAGE; // bit array

class BitImage {
private:
    struct Mask {
        int A, B, C;
    };

    IMAGE *binimg;
    int height, width, size, pixels;
    int amount;

    int bit = 1, bitMask = 1;
    int byte = 0, bitInByte = 0;
    int components = 0;
    int** convex_hull = NULL;

    string matrix = "";
    std::vector<int*> result;

public:
    BitImage(int x, int y) {
        if (x <= 0 || y <= 0) {
            std::cout << "Either size cannot be less or equal 0" << std::endl;
            exit(0);
        }
        height = x;
```

```
57          width = y;
58          pixels = x * y;
59
60          size = (x * y) >> POWER_OF_TWO;
61
62          if (((x * y) % 8) != 0) {
63              size += 1;
64          }
65
66          binimg = new IMAGE[size];
67
68          for (int i = 0; i < size; i++) {
69              binimg[i] = 0;
70              std::bitset<8> bits(0);
71              matrix += bits.to_string();
72          }
73
74          int redundant = (size << POWER_OF_TWO) - pixels;
75
76          matrix.erase(matrix.size() - redundant);
77      };
78
79      void setUpAccess(int x, int y) {
80          bit = ((x - 1) * width) + y;
81          byte = 0;
82          bitMask = 1;
83          bitInByte = bit % 8;
84
85          byte = (bit >> POWER_OF_TWO);
86
87          if (bitInByte != 0) {
88              bitMask = BIGGEST_SINGLE_BIT >> (bitInByte - 1);
89              byte += 1;
90          }
91      }
92
93      void setRandom() {
94          srand(time(NULL)); // use current time
95
96          int digit;
97          matrix = "";
98          for (int i = 0; i < size; i++) {
99              digit = rand() % HIGHEST_VALUE_IN_BYTE;
100             binimg[i] = digit;
101             std::bitset<8> bits(digit);
102             matrix += bits.to_string();
103         }
104     }
105
106     void printAll() {
107         printf("\nOur␣image");
108         for (int i = 0; i < pixels; i++) {
109             if ( (i % width) == 0 ) { putchar('\n'); }
110             printf("%c", matrix[i]);
111         }
112         putchar('\n');
113     }
114
115     void setPixel(int x, int y) {
116         if (x <= 0 || y <= 0 || pixels < (x * y)) {
```

```
117             cout << "You␣are␣accessing␣" << (x * y) << "th␣bit␣from␣only␣" <<
                    pixels << "␣available,␣";
118             cout << "therefore␣it's␣a␣nonexistent␣bit,␣abort." << endl;
119             return;
120         }
121
122         setUpAccess(x, y);
123
124         cout << "Accessing␣" << byte << "th␣byte:␣" << std::bitset<8>(binimg[byte
                - 1]) << ",␣";
125         cout << "setting␣up␣" << bit << "th␣bit:\t" << std::bitset<8>(bitMask) <<
                endl;
126
127         binimg[byte - 1] |= bitMask;
128         matrix[bit - 1] = '1';
129     }
130
131     void fillImage() {
132         string s;
133         matrix = "";
134         for (int i = 0; i < size; i++) {
135             binimg[i] = HIGHEST_VALUE_IN_BYTE - 1; // all bits set to 1's.
136             auto s = to_string(binimg[i]);
137             matrix += s;
138         }
139     }
140
141     void fillHalf() {
142         string s;
143         matrix = "";
144         for (int i = 0; i < size; i++) {
145             if (!(i % 2)) continue;
146             binimg[i] = HIGHEST_VALUE_IN_BYTE - 1; // all bits set to 1's.
147             auto s = to_string(binimg[i]);
148             matrix += s;
149         }
150     }
151
152     bool getPixel(int x, int y) {
153         setUpAccess(x, y);
154
155         int nthByte = 7 - ((bit - 1) % 8);
156         std::bitset<8> ourByte(binimg[byte - 1]);
157
158         if (ourByte[nthByte]) return true;
159
160         return false;
161     }
162
163     double turnPoint(int* p1, int* p2, int* p3) {
164         double ax = p2[0] - p1[0];
165         double ay = p2[1] - p1[1];
166         double bx = p3[0] - p1[0];
167         double by = p3[1] - p1[1];
168
169         return ((ax * bx + ay * by) / (sqrt(ax * ax + ay * ay) * sqrt(bx * bx + by
                * by)));
170     }
171
172     double betweenPoints(int* p1, int* p2) {
```

```
173         return sqrt((p2[0] - p1[0]) * (p2[0] - p1[0]) + (p2[1] - p1[1]) * (p2[1] -
                p1[1]));
174     }
175     void findConnectedComponents() {
176         int kWidth = 0, kHeight = 0;
177         string s; // converion
178 
179         amount = 1;
180         Mask localMask;
181 
182         for (int i = 1; i <= width; i++) {
183             for (int j = 1; j <= height; j++) {
184                 kWidth = j - 1;
185                 if (kWidth <= 0) {
186                     kWidth = 1;
187                     localMask.B = 0;
188                 } else {
189                     localMask.B = getPixel(i, kWidth);
190                 }
191 
192                 kHeight = i - 1;
193                 if (kHeight <= 0) {
194                     kHeight = 1;
195                     localMask.C = 0;
196                 } else {
197                     localMask.C = getPixel(kHeight, j);
198                 }
199 
200                 localMask.A = getPixel(i, j);
201 
202                 if (localMask.A == 0) {
203                 } else if (localMask.B == 0 && localMask.C == 0) {
204                     components += 1;
205                     amount += 1;
206                     s = to_string(amount);
207                     matrix[((i - 1) * width) + (j - 1)] = s[0];
208                 } else if (localMask.B != 0 && localMask.C == 0) {
209                     matrix[((i - 1) * width) + (j - 1)] = (char)localMask.B;
210                 } else if (localMask.B == 0 && localMask.C != 0) {
211                     matrix[((i - 1) * width) + (j - 1)] = (char)localMask.C;
212                 } else if (localMask.B != 0 && localMask.C != 0) {;
213                     matrix[((i - 1) * width) + (j - 1)] = (char)localMask.B;
214 
215                     // find pixels labeled as C, then relabel them as B.
216                     if (localMask.B != localMask.C) {
217                         for (int a = 0; a < i; a++) {
218                             for (int b = 0; b < height; b++) {
219                                 if (matrix[(a * width) + b] == localMask.C) {
220                                     s = to_string(localMask.B);
221                                     matrix[(a * width) + b] = s[0];
222                                 }
223                             }
224                         }
225                     }
226                     // fi
227                 }
228             }
229         }
230     }
231 
```

```
232     void makeConvex() {
233         int tempo = 0;
234         convex_hull = new int*[components];
235 
236         for (int i = 0; i < components; i++) {
237             convex_hull[i] = new int[3];
238         }
239 
240         for (int i = 0; i < width; i++) {
241             for (int j = 0; j < height; j++) {
242 
243                 char access = matrix[(i * width) + j];
244                 if (access != '0') {
245                     convex_hull[tempo][0] = j;
246                     convex_hull[tempo][1] = i;
247 
248                     auto s = (int)access;
249                     convex_hull[tempo][2] = s;
250 
251                     if (tempo >= components - 1) break;
252                     tempo += 1;
253                 }
254             }
255         }
256     }
257 
258     void Jarvis() {
259         int m = 0, minind = 0;
260         double mincos, cosine;
261         double len = 0, maxlen = 0;
262 
263         if (components == 1) {
264             result.push_back(convex_hull[0]);
265             return;
266         }
267         if (components == 2) {
268             result.push_back(convex_hull[0]);
269             result.push_back(convex_hull[1]);
270             return;
271         }
272 
273         double* first_elements = new double[2];
274         first_elements[0] = convex_hull[0][0];
275         first_elements[1] = convex_hull[0][1];
276 
277         for (int i = 1; i < components; i++) {
278             if (convex_hull[i][1] < first_elements[1]) {
279                 m = i;
280             } else {
281                 if ((convex_hull[i][1] == first_elements[1]) && (convex_hull[i][0]
                        < first_elements[0])) {
282                     m = i;
283                 }
284             }
285         }
286 
287         result.push_back(convex_hull[m]);
288 
289         int* last = new int[2];
290         int* beforelast = new int[2];
```

```
291
292         last = convex_hull[m];
293         beforelast[0] = convex_hull[m][0] - 2;
294         beforelast[1] = convex_hull[m][1];
295
296         for(;;) {
297             mincos = 2;
298             for (int i = 0; i < components; i++) {
299                 cosine = roundFloat(turnPoint(last, beforelast, convex_hull[i]) *
                        INFINITE) / INFINITE;
300                 if (cosine < mincos) {
301                     minind = i;
302                     mincos = cosine;
303                     maxlen = betweenPoints(last, convex_hull[i]);
304                 } else if (cosine == mincos) {
305                     len = betweenPoints(last, convex_hull[i]);
306                     if (len > maxlen) {
307                         minind = i;
308                         maxlen = len;
309                     }
310                 }
311             }
312
313             beforelast = last;
314
315             last = convex_hull[minind];
316
317             if (last == convex_hull[m])
318                 break;
319
320             result.push_back(convex_hull[minind]);
321         }
322     }
323
324     void printResult() {
325         int* temp = NULL;
326
327         for (int i = 0; i < result.size(); i++) {
328             temp = result[i];
329         }
330         putchar('\n');
331         for (int i = 0; i < sizeof(temp)/sizeof(temp[0]); i++)
332             cout << "(" << temp[i] << ")";
333
334         putchar('\n');
335     }
336
337     ~BitImage() {
338         matrix.clear();
339     }
340 };
341
342 int main() {
343     int height, width;
344
345     cout << "We represent our binary image as a packed one-dimensional array in
            which each pixel is a bit." << endl;
346     /* user input */
347     /*cout << "Enter height, then width: ";
348     cin >> height >> width;
```

```
349     putchar('\n');*/
350
351     /* predefined input */
352     width = 4500;
353     height = 4500;
354
355     /* create binary image (object) */
356     BitImage be(height, width);
357
358     be.setRandom(); // sets random image
359
360     //start measuring time
361     clock_t begin = clock();
362     //doing job
363     be.findConnectedComponents();
364     be.makeConvex();
365     be.Jarvis();
366     //stop measuring time
367     clock_t end = clock();
368     double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
369     //end.
370
371     be.printResult();
372
373     cout << "(Sequential) Time spent on finding connected components and making
            the convex hull is: " << time_spent << " seconds" << endl;
374
375     return 0;
376 }
```

task5.cpp

```
1  #include <omp.h>
2  #include <iostream>
3  #include <bitset>
4  #include <string>
5  #include <vector>
6  #include <time.h>
7  #include <cmath>
8  #include <cfenv>
9  using namespace std;
10
11 #define POWER_OF_TWO 3 // 2 ^ 3
12 #define HIGHEST_VALUE_IN_BYTE 256 // 11111111
13 #define BIGGEST_SINGLE_BIT 128 // 10000000
14 #define INFINITE 10000000
15
16 #pragma STDC FENV_ACCESS ON
17 double roundFloat(double x) {
18     std::fenv_t save_env;
19     std::feholdexcept(&save_env);
20     double result = std::rint(x);
21     if (std::fetestexcept(FE_INEXACT)) {
22         auto const save_round = std::fegetround();
23         std::fesetround(FE_TOWARDZERO);
24         result = std::rint(std::copysign(0.5 + std::fabs(x), x));
25         std::fesetround(save_round);
26     }
27     std::feupdateenv(&save_env);
28     return result;
29 }
```

```
30
31  typedef unsigned int IMAGE; // bit array
32
33  class BitImage {
34  private:
35      struct Mask {
36          int A, B, C;
37      };
38
39      IMAGE *binimg;
40      int height, width, size, pixels;
41      int amount;
42
43      int bit = 1, bitMask = 1;
44      int byte = 0, bitInByte = 0;
45      int components = 0;
46      int numberOfThreads = 2;
47      int** convex_hull = NULL;
48
49      string matrix = "";
50      std::vector<int*> result;
51
52  public:
53      BitImage(int x, int y) {
54          if (x <= 0 || y <= 0) {
55              std::cout << "Either size cannot be less or equal 0" << std::endl;
56              exit(0);
57          }
58          height = x;
59          width = y;
60          pixels = x * y;
61
62          size = (x * y) >> POWER_OF_TWO;
63
64          if (((x * y) % 8) != 0) {
65              size += 1;
66          }
67
68          binimg = new IMAGE[size];
69
70          for (int i = 0; i < size; i++) {
71              binimg[i] = 0;
72              std::bitset<8> bits(0);
73              matrix += bits.to_string();
74          }
75
76          int redundant = (size << POWER_OF_TWO) - pixels;
77
78          matrix.erase(matrix.size() - redundant);
79      };
80
81      void setUpAccess(int x, int y) {
82          bit = ((x - 1) * width) + y;
83          byte = 0;
84          bitMask = 1;
85          bitInByte = bit % 8;
86
87          byte = (bit >> POWER_OF_TWO);
88
89          if (bitInByte != 0) {
```

```
90              bitMask = BIGGEST_SINGLE_BIT >> (bitInByte - 1);
91              byte += 1;
92          }
93      }
94
95      void setRandom() {
96          srand(time(NULL)); // use current time
97
98          int digit;
99          matrix = "";
100         for (int i = 0; i < size; i++) {
101             digit = rand() % HIGHEST_VALUE_IN_BYTE;
102             binimg[i] = digit;
103             std::bitset<8> bits(digit);
104             matrix += bits.to_string();
105         }
106     }
107
108     void printAll() {
109         printf("\nOur␣image");
110         for (int i = 0; i < pixels; i++) {
111             if ( (i % width) == 0 ) { putchar('\n'); }
112             printf("%c", matrix[i]);
113         }
114         putchar('\n');
115     }
116
117     void setPixel(int x, int y) {
118         if (x <= 0 || y <= 0 || pixels < (x * y)) {
119             cout << "You␣are␣accessing␣" << (x * y) << "th␣bit␣from␣only␣" <<
                    pixels << "␣available,␣";
120             cout << "therefore␣it's␣a␣nonexistent␣bit,␣abort." << endl;
121             return;
122         }
123
124         setUpAccess(x, y);
125
126         cout << "Accessing␣" << byte << "th␣byte:␣" << std::bitset<8>(binimg[byte
                - 1]) << ",␣";
127         cout << "setting␣up␣" << bit << "th␣bit:\t" << std::bitset<8>(bitMask) <<
                endl;
128
129         binimg[byte - 1] |= bitMask;
130         matrix[bit - 1] = '1';
131     }
132
133     void fillImage() {
134         string s;
135         matrix = "";
136         for (int i = 0; i < size; i++) {
137             binimg[i] = HIGHEST_VALUE_IN_BYTE - 1; // all bits set to 1's.
138             auto s = to_string(binimg[i]);
139             matrix += s;
140         }
141     }
142
143     void fillHalf() {
144         string s;
145         matrix = "";
146         for (int i = 0; i < size; i++) {
```

```
147             if (!(i % 2)) continue;
148             binimg[i] = HIGHEST_VALUE_IN_BYTE - 1; // all bits set to 1's.
149             auto s = to_string(binimg[i]);
150             matrix += s;
151         }
152     }
153
154     bool getPixel(int x, int y) {
155         setUpAccess(x, y);
156
157         int nthByte = 7 - ((bit - 1) % 8);
158         std::bitset<8> ourByte(binimg[byte - 1]);
159
160         if (ourByte[nthByte]) return true;
161
162         return false;
163     }
164
165     double turnPoint(int* p1, int* p2, int* p3) {
166         double ax = p2[0] - p1[0];
167         double ay = p2[1] - p1[1];
168         double bx = p3[0] - p1[0];
169         double by = p3[1] - p1[1];
170
171         return (ax * bx + ay * by) / (sqrt(ax * ax + ay * ay) * sqrt(bx * bx + by
                * by));
172     }
173
174     double betweenPoints(int* p1, int* p2) {
175         return sqrt((p2[0] - p1[0]) * (p2[0] - p1[0]) + (p2[1] - p1[1]) * (p2[1] -
                p1[1]));
176     }
177     void findConnectedComponents() {
178         int kWidth = 0, kHeight = 0;
179         string s; // convertion
180
181         amount = 1;
182         Mask localMask;
183
184         for (int i = 1; i <= width; i++) {
185             for (int j = 1; j <= height; j++) {
186                 kWidth = j - 1;
187                 if (kWidth <= 0) {
188                     kWidth = 1;
189                     localMask.B = 0;
190                 } else {
191                     localMask.B = getPixel(i, kWidth);
192                 }
193
194                 kHeight = i - 1;
195                 if (kHeight <= 0) {
196                     kHeight = 1;
197                     localMask.C = 0;
198                 } else {
199                     localMask.C = getPixel(kHeight, j);
200                 }
201
202                 localMask.A = getPixel(i, j);
203
204                 if (localMask.A == 0) {
```

```
205                 } else if (localMask.B == 0 && localMask.C == 0) {
206                     components += 1;
207                     amount += 1;
208                     s = to_string(amount);
209                     matrix[((i - 1) * width) + (j - 1)] = s[0];
210                 } else if (localMask.B != 0 && localMask.C == 0) {
211                     s = to_string(localMask.B);
212                     matrix[((i - 1) * width) + (j - 1)] = s[0];
213                 } else if (localMask.B == 0 && localMask.C != 0) {
214                     s = to_string(localMask.C);
215                     matrix[((i - 1) * width) + (j - 1)] = s[0];
216                 } else if (localMask.B != 0 && localMask.C != 0) {
217                     s = to_string(localMask.B);
218                     matrix[((i - 1) * width) + (j - 1)] = s[0];
219 
220                     // find pixels labeled as C, then relabel them as B.
221                     if (localMask.B != localMask.C) {
222                         for (int a = 0; a < i; a++) {
223                             for (int b = 0; b < height; b++) {
224                                 if (matrix[(a * width) + b] == localMask.C) {
225                                     s = to_string(localMask.B);
226                                     matrix[(a * width) + b] = s[0];
227                                 }
228                             }
229                         }
230                     }
231                     // fi
232                 }
233             }
234         }
235     }
236 
237     void makeConvex() {
238         int tempo = 0;
239         convex_hull = new int*[components];
240 
241         for (int i = 0; i < components; i++) {
242             convex_hull[i] = new int[3];
243         }
244 
245         for (int i = 0; i < width; i++) {
246             for (int j = 0; j < height; j++) {
247 
248                 char access = matrix[(i * width) + j];
249                 if (access != '0') {
250                     convex_hull[tempo][0] = j;
251                     convex_hull[tempo][1] = i;
252 
253                     auto s = (int)access;
254                     convex_hull[tempo][2] = s;
255 
256                     if (tempo >= components - 1) break;
257                     tempo += 1;
258                 }
259             }
260         }
261     }
262 
263     void Jarvis() {
264         int m = 0, minind = 0;
```

```
265         double mincos, cosine;
266         double len = 0, maxlen = 0;
267 
268         if (components == 1) {
269             result.push_back(convex_hull[0]);
270             return;
271         }
272         if (components == 2) {
273             result.push_back(convex_hull[0]);
274             result.push_back(convex_hull[1]);
275             return;
276         }
277 
278         double* first_elements = new double[2];
279         first_elements[0] = convex_hull[0][0];
280         first_elements[1] = convex_hull[0][1];
281 
282         for (int i = 1; i < components; i++) {
283             if (convex_hull[i][1] < first_elements[1]) {
284                 m = i;
285             } else {
286                 if ((convex_hull[i][1] == first_elements[1]) && (convex_hull[i][0]
                        < first_elements[0])) {
287                     m = i;
288                 }
289             }
290         }
291 
292         result.push_back(convex_hull[m]);
293 
294         int* last = new int[2];
295         int* beforelast = new int[2];
296 
297         last = convex_hull[m];
298         beforelast[0] = convex_hull[m][0] - 2;
299         beforelast[1] = convex_hull[m][1];
300 
301         for(;;) {
302             mincos = 2;
303             for (int i = 0; i < components; i++) {
304                 cosine = roundFloat(turnPoint(last, beforelast, convex_hull[i]) *
                        INFINITE) / INFINITE;
305                 if (cosine < mincos) {
306                     minind = i;
307                     mincos = cosine;
308                     maxlen = betweenPoints(last, convex_hull[i]);
309                 } else if (cosine == mincos) {
310                     len = betweenPoints(last, convex_hull[i]);
311                     if (len > maxlen) {
312                         minind = i;
313                         maxlen = len;
314                     }
315                 }
316             }
317 
318             beforelast = last;
319             last = convex_hull[minind];
320             if (last == convex_hull[m])
321                 break;
322             result.push_back(convex_hull[minind]);
```

```
323         }
324     }
325 
326     void printResult() {
327         int* temp = NULL;
328 
329         for (int i = 0; i < result.size(); i++) {
330             temp = result[i];
331         }
332         putchar('\n');
333         for (int i = 0; i < sizeof(temp)/sizeof(temp[0]); i++)
334             cout << "(" << temp[i] << ")";
335 
336         putchar('\n');
337     }
338 
339     void findConnectedComponents__OMP() {
340         int kWidth = 0, kHeight = 0;
341         string s; // converion
342 
343         amount = 1;
344         Mask localMask;
345 
346         for (int i = 1; i <= width; i++) {
347             for (int j = 1; j <= height; j++) {
348                 kWidth = j - 1;
349                 if (kWidth <= 0) {
350                     kWidth = 1;
351                     localMask.B = 0;
352                 } else {
353                     localMask.B = getPixel(i, kWidth);
354                 }
355 
356                 kHeight = i - 1;
357                 if (kHeight <= 0) {
358                     kHeight = 1;
359                     localMask.C = 0;
360                 } else {
361                     localMask.C = getPixel(kHeight, j);
362                 }
363 
364                 localMask.A = getPixel(i, j);
365 
366                 if (localMask.A == 0) {
367                 } else if (localMask.B == 0 && localMask.C == 0) {
368                     components += 1;
369                     amount += 1;
370                     s = to_string(amount);
371                     matrix[((i - 1) * width) + (j - 1)] = s[0];
372                 } else if (localMask.B != 0 && localMask.C == 0) {
373                     matrix[((i - 1) * width) + (j - 1)] = (char)localMask.B;
374                 } else if (localMask.B == 0 && localMask.C != 0) {
375                     s = (char)localMask.C;
376                     matrix[((i - 1) * width) + (j - 1)] = s[0];
377                 } else if (localMask.B != 0 && localMask.C != 0) {
378                     matrix[((i - 1) * width) + (j - 1)] = (char)localMask.B;
379 
380                     if (localMask.B != localMask.C) {
381                     // find pixels labeled as C, then relabel them as B.
382                     int a;
```

```
383
384                     #pragma omp parallel private(a) num_threads(numberOfThreads)
385                         {
386                     #pragma omp for
387                             for (a = 0; a < i; a++) {
388                                 for (int b = 0; b < height; b++) {
389                                     #pragma omp critical
390                                     {
391                                         if (matrix[(a * width) + b] ==
                                            localMask.C) {
392                                             matrix[(a * width) + b] =
                                                (char)localMask.B;
393                                         }
394                                     }
395                                 }
396                             }
397                         }
398                     }
399                     // fi
400                 }
401             }
402         }
403     }
404
405     void Jarvis__OMP() {
406         int id, delta, rest; // for omp
407         int m = 0, minind = 0;
408         double mincos, cosine;
409         double len = 0, maxlen = 0;
410         int* last;
411         int* beforelast;
412
413         std::vector<std::vector<int*>> local_result(numberOfThreads);
414
415         if (components == 1) {
416             result.push_back(convex_hull[0]);
417             return;
418         }
419         if (components == 2) {
420             result.push_back(convex_hull[0]);
421             result.push_back(convex_hull[1]);
422             return;
423         }
424
425         result.push_back(convex_hull[m]);
426
427         for (int i = 0; i < numberOfThreads; i++)
428             local_result[i].push_back(convex_hull[m]);
429
430         // OMP
431         #pragma omp parallel private(id, delta, rest, mincos, cosine, minind,
                maxlen, last, beforelast, len) num_threads(numberOfThreads)
432         {
433             delta = components / numberOfThreads;
434             id = omp_get_thread_num();
435
436             maxlen = minind = rest = 0;
437
438             if (id == numberOfThreads - 1) {
439                 rest = components % numberOfThreads;
```

```
440             }
441
442             last = new int[2];
443             beforelast = new int[2];
444
445             last = convex_hull[m];
446             beforelast[0] = convex_hull[m][0] - 2;
447             beforelast[1] = convex_hull[m][1];
448
449             for(;;) {
450                 mincos = 2;
451                 for (int i = id * delta; i < id * delta + delta + rest; i++) {
452                     cosine = roundFloat(turnPoint(last, beforelast,
                            convex_hull[i]) * INFINITE) / INFINITE;
453                     if (cosine < mincos) {
454                         minind = i;
455                         mincos = cosine;
456                         maxlen = betweenPoints(last, convex_hull[i]);
457                     } else if (cosine == mincos) {
458
459                         len = betweenPoints(last, convex_hull[i]);
460                         if (len > maxlen) {
461                             minind = i;
462                             maxlen = len;
463                         }
464                     }
465                 }
466
467                 if (id != 0) {
468                     cosine = roundFloat(turnPoint(last, beforelast,
                            convex_hull[0]) * INFINITE) / INFINITE;
469                     if (cosine < mincos) {
470                         minind = 0;
471                         mincos = cosine;
472                         maxlen = betweenPoints(last, convex_hull[0]);
473                     }
474                     if (cosine == mincos) {
475
476                         len = betweenPoints(last, convex_hull[0]);
477                         if (len > maxlen) {
478                             minind = 0;
479                             maxlen = len;
480                         }
481                     }
482                 }
483
484                 beforelast = last;
485                 last = convex_hull[minind];
486
487                 if (last == convex_hull[m])
488                     break;
489                 local_result[id].push_back(convex_hull[minind]);
490             }
491         }
492         // END OF OMP
493
494         std::vector<int*> finale_local;
495         for (int i = 0; i < numberOfThreads; i++) {
496             int finalSize = local_result[i].size();
497
```

```
498             for (int j = 0; j < finalSize; j++)
499                 finale_local.push_back(local_result[i][j]);
500             }
501 
502             int s = finale_local.size();
503 
504             last = new int[2];
505             beforelast = new int[2];
506             last = convex_hull[m];
507             beforelast[0] = convex_hull[m][0] - 2;
508             beforelast[1] = convex_hull[m][1];
509 
510             for(;;) {
511                 mincos = 2;
512                 for (int i = 0; i < s; i++) {
513                     cosine = roundFloat(turnPoint(last, beforelast,
                        finale_local[i]) * INFINITE) / INFINITE;
514 
515                     if (cosine < mincos) {
516                         minind = i;
517                         mincos = cosine;
518                         maxlen = betweenPoints(last, finale_local[i]);
519                     }
520 
521                     if (cosine == mincos) {
522                         len = betweenPoints(last, finale_local[i]);
523                         if (len > maxlen) {
524                             minind = i;
525                             maxlen = len;
526                         }
527                     }
528                 }
529 
530                 beforelast = last;
531                 last = finale_local[minind];
532                 if (last == finale_local[m]) break;
533                 result.push_back(finale_local[minind]);
534             }
535     }
536 
537     ~BitImage() {
538         matrix.clear();
539     }
540 };
541 
542 int main() {
543     int height, width;
544 
545     cout << "We represent our binary image as a packed one-dimensional array in
            which each pixel is a bit." << endl;
546     /* user input */
547     /*cout << "Enter height, then width: ";
548     cin >> height >> width;
549     putchar('\n');*/
550 
551     /* predefined input */
552     width = 4500;
553     height = 4500;
554 
555     /* create binary image (object) */
```

```
556     BitImage be(height, width);
557     BitImage ae(height, width);
558 
559     be.setRandom(); // set random image
560     ae = be;
561 
562     //start measuring time
563     clock_t begin = clock();
564     //doing job
565     be.findConnectedComponents__OMP();
566     be.makeConvex();
567     be.Jarvis__OMP();
568     //stop measuring time
569     clock_t end = clock();
570     double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
571     //end.
572     cout << "(OMP) Time spent on finding connected components and making the
            convex hull is: " << time_spent << " seconds" << endl;
573 
574     //start measuring time
575     begin = clock();
576     //doing job
577     ae.findConnectedComponents();
578     ae.makeConvex();
579     ae.Jarvis();
580     //stop measuring time
581     end = clock();
582     time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
583 
584     cout << "(Seq) Time spent on finding connected components and making the
            convex hull is: " << time_spent << " seconds" << endl;
585     be.printResult();
586 
587     return 0;
588 }
```

task6.cpp

```
1  #include <iostream>
2  #include <bitset>
3  #include <string>
4  #include <vector>
5  #include <time.h>
6  #include <cmath>
7  #include <cfenv>
8  #include <tbb/tbb.h>
9  using namespace std;
10 
11 #define POWER_OF_TWO 3 // 2 ^ 3
12 #define HIGHEST_VALUE_IN_BYTE 256 // 11111111
13 #define BIGGEST_SINGLE_BIT 128 // 10000000
14 #define INFINITE 10000000
15 
16 #pragma STDC FENV_ACCESS ON
17 double roundFloat(double x) {
18     std::fenv_t save_env;
19     std::feholdexcept(&save_env);
20     double result = std::rint(x);
21     if (std::fetestexcept(FE_INEXACT)) {
22         auto const save_round = std::fegetround();
23         std::fesetround(FE_TOWARDZERO);
```

```
24          result = std::rint(std::copysign(0.5 + std::fabs(x), x));
25          std::fesetround(save_round);
26      }
27      std::feupdateenv(&save_env);
28      return result;
29  }
30
31  typedef unsigned int IMAGE; // bit array
32
33  class BitImage {
34  private:
35      struct Mask {
36          int A, B, C;
37      };
38
39      IMAGE *binimg;
40      int height, width, size, pixels;
41      int amount;
42
43      int bit = 1, bitMask = 1;
44      int byte = 0, bitInByte = 0;
45      int components = 0;
46      int numberOfThreads = 4;
47      int** convex_hull = NULL;
48
49      string matrix = "";
50      std::vector<int*> result;
51
52  public:
53      BitImage(int x, int y) {
54          if (x <= 0 || y <= 0) {
55              std::cout << "Either size cannot be less or equal 0" << std::endl;
56              exit(0);
57          }
58          height = x;
59          width = y;
60          pixels = x * y;
61
62          size = (x * y) >> POWER_OF_TWO;
63
64          if (((x * y) % 8) != 0) {
65              size += 1;
66          }
67
68          binimg = new IMAGE[size];
69
70          for (int i = 0; i < size; i++) {
71              binimg[i] = 0;
72              std::bitset<8> bits(0);
73              matrix += bits.to_string();
74          }
75
76          int redundant = (size << POWER_OF_TWO) - pixels;
77
78          matrix.erase(matrix.size() - redundant);
79      };
80
81      void setUpAccess(int x, int y) {
82          bit = ((x - 1) * width) + y;
83          byte = 0;
```

```
84          bitMask = 1;
85          bitInByte = bit % 8;
86
87          byte = (bit >> POWER_OF_TWO);
88
89          if (bitInByte != 0) {
90              bitMask = BIGGEST_SINGLE_BIT >> (bitInByte - 1);
91              byte += 1;
92          }
93      }
94
95      void setRandom() {
96          srand(time(NULL)); // use current time
97
98          int digit;
99          matrix = "";
100         for (int i = 0; i < size; i++) {
101             digit = rand() % HIGHEST_VALUE_IN_BYTE;
102             binimg[i] = digit;
103             std::bitset<8> bits(digit);
104             matrix += bits.to_string();
105         }
106     }
107
108     void printAll() {
109         printf("\nOur␣image");
110         for (int i = 0; i < pixels; i++) {
111             if ( (i % width) == 0 ) { putchar('\n'); }
112             printf("%c", matrix[i]);
113         }
114         putchar('\n');
115     }
116
117     void setPixel(int x, int y) {
118         if (x <= 0 || y <= 0 || pixels < (x * y)) {
119             cout << "You␣are␣accessing␣" << (x * y) << "th␣bit␣from␣only␣" <<
                    pixels << "␣available,␣";
120             cout << "therefore␣it's␣a␣nonexistent␣bit,␣abort." << endl;
121             return;
122         }
123
124         setUpAccess(x, y);
125
126         cout << "Accessing␣" << byte << "th␣byte:␣" << std::bitset<8>(binimg[byte
                - 1]) << ",␣";
127         cout << "setting␣up␣" << bit << "th␣bit:\t" << std::bitset<8>(bitMask) <<
                endl;
128
129         binimg[byte - 1] |= bitMask;
130         matrix[bit - 1] = '1';
131     }
132
133     void fillImage() {
134         string s;
135         matrix = "";
136         for (int i = 0; i < size; i++) {
137             binimg[i] = HIGHEST_VALUE_IN_BYTE - 1; // all bits set to 1's.
138             auto s = to_string(binimg[i]);
139             matrix += s;
140         }
```

```
141     }
142
143     void fillHalf() {
144         string s;
145         matrix = "";
146         for (int i = 0; i < size; i++) {
147             if (!(i % 4)) continue;
148             binimg[i] = HIGHEST_VALUE_IN_BYTE - 1; // all bits set to 1's.
149             auto s = to_string(binimg[i]);
150             matrix += s;
151         }
152     }
153
154     bool getPixel(int x, int y) {
155         setUpAccess(x, y);
156
157         int nthByte = 7 - ((bit - 1) % 8);
158         std::bitset<8> ourByte(binimg[byte - 1]);
159
160         if (ourByte[nthByte]) return true;
161
162         return false;
163     }
164
165     double turnPoint(int* p1, int* p2, int* p3) {
166         double ax = p2[0] - p1[0];
167         double ay = p2[1] - p1[1];
168         double bx = p3[0] - p1[0];
169         double by = p3[1] - p1[1];
170
171         return (ax * bx + ay * by) / (sqrt(ax * ax + ay * ay) * sqrt(bx * bx + by
                * by));
172     }
173
174     double betweenPoints(int* p1, int* p2) {
175         return sqrt((p2[0] - p1[0]) * (p2[0] - p1[0]) + (p2[1] - p1[1]) * (p2[1] -
                p1[1]));
176     }
177     void findConnectedComponents() {
178         int kWidth = 0, kHeight = 0;
179         string s; // convertion
180
181         amount = 1;
182         Mask localMask;
183
184         for (int i = 1; i <= width; i++) {
185             for (int j = 1; j <= height; j++) {
186                 kWidth = j - 1;
187                 if (kWidth <= 0) {
188                     kWidth = 1;
189                     localMask.B = 0;
190                 } else {
191                     localMask.B = getPixel(i, kWidth);
192                 }
193
194                 kHeight = i - 1;
195                 if (kHeight <= 0) {
196                     kHeight = 1;
197                     localMask.C = 0;
198                 } else {
```

```
199                     localMask.C = getPixel(kHeight, j);
200                 }
201 
202                 localMask.A = getPixel(i, j);
203 
204                 if (localMask.A == 0) {
205                 } else if (localMask.B == 0 && localMask.C == 0) {
206                     components += 1;
207                     amount += 1;
208                     s = to_string(amount);
209                     matrix[((i - 1) * width) + (j - 1)] = s[0];
210                 } else if (localMask.B != 0 && localMask.C == 0) {
211                     s = to_string(localMask.B);
212                     matrix[((i - 1) * width) + (j - 1)] = s[0];
213                 } else if (localMask.B == 0 && localMask.C != 0) {
214                     s = to_string(localMask.C);
215                     matrix[((i - 1) * width) + (j - 1)] = s[0];
216                 } else if (localMask.B != 0 && localMask.C != 0) {
217                     s = to_string(localMask.B);
218                     matrix[((i - 1) * width) + (j - 1)] = s[0];
219 
220                     // find pixels labeled as C, then relabel them as B.
221                     if (localMask.B != localMask.C) {
222                         for (int a = 0; a < i; a++) {
223                             for (int b = 0; b < height; b++) {
224                                 if (matrix[(a * width) + b] == localMask.C) {
225                                     s = to_string(localMask.B);
226                                     matrix[(a * width) + b] = s[0];
227                                 }
228                             }
229                         }
230                     }
231                     // fi
232                 }
233             }
234         }
235     }
236 
237     void makeConvex() {
238         int tempo = 0;
239         convex_hull = new int*[components];
240 
241         for (int i = 0; i < components; i++) {
242             convex_hull[i] = new int[3];
243         }
244 
245         for (int i = 0; i < width; i++) {
246             for (int j = 0; j < height; j++) {
247 
248                 char access = matrix[(i * width) + j];
249                 if (access != '0') {
250                     convex_hull[tempo][0] = j;
251                     convex_hull[tempo][1] = i;
252 
253                     auto s = (int)access;
254                     convex_hull[tempo][2] = s;
255 
256                     if (tempo >= components - 1) break;
257                     tempo += 1;
258                 }
```

```
259             }
260         }
261     }
262
263     void Jarvis() {
264         int m = 0, minind = 0;
265         double mincos, cosine;
266         double len = 0, maxlen = 0;
267
268         if (components == 1) {
269             result.push_back(convex_hull[0]);
270             return;
271         }
272         if (components == 2) {
273             result.push_back(convex_hull[0]);
274             result.push_back(convex_hull[1]);
275             return;
276         }
277
278         double* first_elements = new double[2];
279         first_elements[0] = convex_hull[0][0];
280         first_elements[1] = convex_hull[0][1];
281
282         for (int i = 1; i < components; i++) {
283             if (convex_hull[i][1] < first_elements[1]) {
284                 m = i;
285             } else {
286                 if ((convex_hull[i][1] == first_elements[1]) && (convex_hull[i][0]
                        < first_elements[0])) {
287                     m = i;
288                 }
289             }
290         }
291
292         result.push_back(convex_hull[m]);
293
294         int* last = new int[2];
295         int* beforelast = new int[2];
296
297         last = convex_hull[m];
298         beforelast[0] = convex_hull[m][0] - 2;
299         beforelast[1] = convex_hull[m][1];
300
301         for(;;) {
302             mincos = 2;
303             for (int i = 0; i < components; i++) {
304                 cosine = roundFloat(turnPoint(last, beforelast, convex_hull[i]) *
                        INFINITE) / INFINITE;
305                 if (cosine < mincos) {
306                     minind = i;
307                     mincos = cosine;
308                     maxlen = betweenPoints(last, convex_hull[i]);
309                 } else if (cosine == mincos) {
310                     len = betweenPoints(last, convex_hull[i]);
311                     if (len > maxlen) {
312                         minind = i;
313                         maxlen = len;
314                     }
315                 }
316             }
```

```
317
318             beforelast = last;
319             last = convex_hull[minind];
320             if (last == convex_hull[m])
321                 break;
322             result.push_back(convex_hull[minind]);
323         }
324     }
325
326     void printResult() {
327         int* temp = NULL;
328
329         for (int i = 0; i < result.size(); i++) {
330             temp = result[i];
331         }
332         putchar('\n');
333         for (int i = 0; i < sizeof(temp)/sizeof(temp[0]); i++)
334             cout << "(" << temp[i] << ")";
335
336         putchar('\n');
337     }
338
339     void findConnectedComponents_TBB() {
340         tbb::task_scheduler_init init(numberOfThreads);
341         int kWidth = 0, kHeight = 0;
342         string s; // converion
343
344         amount = 1;
345         Mask localMask;
346
347         for (int i = 1; i <= width; i++) {
348             for (int j = 1; j <= height; j++) {
349                 kWidth = j - 1;
350                 if (kWidth <= 0) {
351                     kWidth = 1;
352                     localMask.B = 0;
353                 } else {
354                     localMask.B = getPixel(i, kWidth);
355                 }
356
357                 kHeight = i - 1;
358                 if (kHeight <= 0) {
359                     kHeight = 1;
360                     localMask.C = 0;
361                 } else {
362                     localMask.C = getPixel(kHeight, j);
363                 }
364
365                 localMask.A = getPixel(i, j);
366
367                 if (localMask.A == 0) {
368                 } else if (localMask.B == 0 && localMask.C == 0) {
369                     components += 1;
370                     amount += 1;
371                     s = to_string(amount);
372                     matrix[((i - 1) * width) + (j - 1)] = s[0];
373                 } else if (localMask.B != 0 && localMask.C == 0) {
374                     matrix[((i - 1) * width) + (j - 1)] = (char)localMask.B;
375                 } else if (localMask.B == 0 && localMask.C != 0) {
376                     matrix[((i - 1) * width) + (j - 1)] = (char)localMask.C;
```

```
377                 } else if (localMask.B != 0 && localMask.C != 0) {
378                     matrix[((i - 1) * width) + (j - 1)] = (char)localMask.B;
379 
380                     if (localMask.B != localMask.C) {
381                     // find pixels labeled as C, then relabel them as B.
382 
383                         tbb::parallel_for(
384                             tbb::blocked_range<int>(0, i),
385                             [=](const tbb::blocked_range<int>& t) {
386                             int begin = t.begin(), end = t.end();
387                             for (int a = begin; a != end; a++) {
388                                 for (int b = 0; b < height; b++) {
389                                     if (matrix[(a * width) + b] == localMask.C) {
390                                         matrix[(a * width) + b] =
                                                (char)localMask.B;
391                                     }
392                                 }
393                             }
394                         });
395                     }
396                     // fi
397                 }
398             }
399         }
400     }
401 
402     ~BitImage() {
403         matrix.clear();
404     }
405 };
406 
407 int main() {
408     int height, width;
409 
410     cout << "We represent our binary image as a packed one-dimensional array in
            which each pixel is a bit." << endl;
411     /* user input */
412     /*cout << "Enter height, then width: ";
413     cin >> height >> width;
414     putchar('\n');*/
415 
416     /* predefined input */
417     width = 4500;
418     height = 4500;
419 
420     /* create binary image (object) */
421     BitImage be(height, width);
422     BitImage ae(height, width);
423 
424     be.setRandom(); // set random image
425     ae = be;
426 
427     //start measuring time
428     clock_t begin = clock();
429     //doing job
430     be.findConnectedComponents__TBB();
431     be.makeConvex();
432     be.Jarvis();
433     //stop measuring time
434     clock_t end = clock();
```

```
435     double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
436     //end.
437 
438     cout << "(TBB) Time spent on finding connected components and making the
            convex hull is: " << time_spent << " seconds" << endl;
439 
440     //start measuring time
441     begin = clock();
442     //doing job
443     ae.findConnectedComponents();
444     ae.makeConvex();
445     ae.Jarvis();
446     //stop measuring time
447     end = clock();
448     time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
449 
450     cout << "(Seq) Time spent on finding connected components and making the
            convex hull is: " << time_spent << " seconds" << endl;
451 
452 
453     return 0;
454 }
```

task7.cpp

```
1  #include <iostream>
2  #include <bitset>
3  #include <string>
4  #include <vector>
5  #include <thread>
6  #include <time.h>
7  #include <cmath>
8  #include <cfenv>
9  using namespace std;
10 
11 #define POWER_OF_TWO 3 // 2 ^ 3
12 #define HIGHEST_VALUE_IN_BYTE 256 // 11111111
13 #define BIGGEST_SINGLE_BIT 128 // 10000000
14 #define INFINITE 10000000
15 
16 #pragma STDC FENV_ACCESS ON
17 double roundFloat(double x) {
18     std::fenv_t save_env;
19     std::feholdexcept(&save_env);
20     double result = std::rint(x);
21     if (std::fetestexcept(FE_INEXACT)) {
22         auto const save_round = std::fegetround();
23         std::fesetround(FE_TOWARDZERO);
24         result = std::rint(std::copysign(0.5 + std::fabs(x), x));
25         std::fesetround(save_round);
26     }
27     std::feupdateenv(&save_env);
28     return result;
29 }
30 
31 typedef unsigned int IMAGE; // bit array
32 
33 class BitImage {
34 private:
35     struct Mask {
36         int A, B, C;
```

```
37      };
38
39      IMAGE *binimg;
40      int height, width, size, pixels;
41      int amount;
42
43      int bit = 1, bitMask = 1;
44      int byte = 0, bitInByte = 0;
45      int components = 0;
46      int** convex_hull = NULL;
47
48      string matrix = "";
49      std::vector<int*> result;
50
51  public:
52      BitImage(int x, int y) {
53          if (x <= 0 || y <= 0) {
54              std::cout << "Either size cannot be less or equal 0" << std::endl;
55              exit(0);
56          }
57          height = x;
58          width = y;
59          pixels = x * y;
60
61          size = (x * y) >> POWER_OF_TWO;
62
63          if (((x * y) % 8) != 0) {
64              size += 1;
65          }
66
67          binimg = new IMAGE[size];
68
69          for (int i = 0; i < size; i++) {
70              binimg[i] = 0;
71              std::bitset<8> bits(0);
72              matrix += bits.to_string();
73          }
74
75          int redundant = (size << POWER_OF_TWO) - pixels;
76
77          matrix.erase(matrix.size() - redundant);
78      };
79
80      void setUpAccess(int x, int y) {
81          bit = ((x - 1) * width) + y;
82          byte = 0;
83          bitMask = 1;
84          bitInByte = bit % 8;
85
86          byte = (bit >> POWER_OF_TWO);
87
88          if (bitInByte != 0) {
89              bitMask = BIGGEST_SINGLE_BIT >> (bitInByte - 1);
90              byte += 1;
91          }
92      }
93
94      void setRandom() {
95          srand(time(NULL)); // use current time
96
```

```
97          int digit;
98          matrix = "";
99          for (int i = 0; i < size; i++) {
100             digit = rand() % HIGHEST_VALUE_IN_BYTE;
101             binimg[i] = digit;
102             std::bitset<8> bits(digit);
103             matrix += bits.to_string();
104         }
105     }
106
107     void printAll() {
108         printf("\nOur␣image");
109         for (int i = 0; i < pixels; i++) {
110             if ( (i % width) == 0 ) { putchar('\n'); }
111             printf("%c", matrix[i]);
112         }
113         putchar('\n');
114     }
115
116     void setPixel(int x, int y) {
117         if (x <= 0 || y <= 0 || pixels < (x * y)) {
118             cout << "You␣are␣accessing␣" << (x * y) << "th␣bit␣from␣only␣" <<
                    pixels << "␣available,␣";
119             cout << "therefore␣it's␣a␣nonexistent␣bit,␣abort." << endl;
120             return;
121         }
122
123         setUpAccess(x, y);
124
125         cout << "Accessing␣" << byte << "th␣byte:␣" << std::bitset<8>(binimg[byte
                - 1]) << ",␣";
126         cout << "setting␣up␣" << bit << "th␣bit:\t" << std::bitset<8>(bitMask) <<
                endl;
127
128         binimg[byte - 1] |= bitMask;
129         matrix[bit - 1] = '1';
130     }
131
132     void fillImage() {
133         string s;
134         matrix = "";
135         for (int i = 0; i < size; i++) {
136             binimg[i] = HIGHEST_VALUE_IN_BYTE - 1; // all bits set to 1's.
137             auto s = to_string(binimg[i]);
138             matrix += s;
139         }
140     }
141
142     void fillHalf() {
143         string s;
144         matrix = "";
145         for (int i = 0; i < size; i++) {
146             if (!(i % 4)) continue;
147             binimg[i] = HIGHEST_VALUE_IN_BYTE - 1; // all bits set to 1's.
148             auto s = to_string(binimg[i]);
149             matrix += s;
150         }
151     }
152
153     bool getPixel(int x, int y) {
```

```
154         setUpAccess(x, y);
155 
156         int nthByte = 7 - ((bit - 1) % 8);
157         std::bitset<8> ourByte(binimg[byte - 1]);
158 
159         if (ourByte[nthByte]) return true;
160 
161         return false;
162     }
163 
164     double turnPoint(int* p1, int* p2, int* p3) {
165         double ax = p2[0] - p1[0];
166         double ay = p2[1] - p1[1];
167         double bx = p3[0] - p1[0];
168         double by = p3[1] - p1[1];
169 
170         return (ax * bx + ay * by) / (sqrt(ax * ax + ay * ay) * sqrt(bx * bx + by
            * by));
171     }
172 
173     double betweenPoints(int* p1, int* p2) {
174         return sqrt((p2[0] - p1[0]) * (p2[0] - p1[0]) + (p2[1] - p1[1]) * (p2[1] -
            p1[1]));
175     }
176     void findConnectedComponents() {
177         int kWidth = 0, kHeight = 0;
178         string s; // convertion
179 
180         amount = 1;
181         Mask localMask;
182 
183         for (int i = 1; i <= width; i++) {
184             for (int j = 1; j <= height; j++) {
185                 kWidth = j - 1;
186                 if (kWidth <= 0) {
187                     kWidth = 1;
188                     localMask.B = 0;
189                 } else {
190                     localMask.B = getPixel(i, kWidth);
191                 }
192 
193                 kHeight = i - 1;
194                 if (kHeight <= 0) {
195                     kHeight = 1;
196                     localMask.C = 0;
197                 } else {
198                     localMask.C = getPixel(kHeight, j);
199                 }
200 
201                 localMask.A = getPixel(i, j);
202 
203                 if (localMask.A == 0) {
204                 } else if (localMask.B == 0 && localMask.C == 0) {
205                     components += 1;
206                     amount += 1;
207                     s = to_string(amount);
208                     matrix[((i - 1) * width) + (j - 1)] = s[0];
209                 } else if (localMask.B != 0 && localMask.C == 0) {
210                     s = to_string(localMask.B);
211                     matrix[((i - 1) * width) + (j - 1)] = s[0];
```

```
212                 } else if (localMask.B == 0 && localMask.C != 0) {
213                     s = to_string(localMask.C);
214                     matrix[((i - 1) * width) + (j - 1)] = s[0];
215                 } else if (localMask.B != 0 && localMask.C != 0) {
216                     s = to_string(localMask.B);
217                     matrix[((i - 1) * width) + (j - 1)] = s[0];
218 
219                     // find pixels labeled as C, then relabel them as B.
220                     if (localMask.B != localMask.C) {
221                         for (int a = 0; a < i; a++) {
222                             for (int b = 0; b < height; b++) {
223                                 if (matrix[(a * width) + b] == localMask.C) {
224                                     s = to_string(localMask.B);
225                                     matrix[(a * width) + b] = s[0];
226                                 }
227                             }
228                         }
229                     }
230                     // fi
231                 }
232             }
233         }
234     }
235 
236     void makeConvex() {
237         int tempo = 0;
238         convex_hull = new int*[components];
239 
240         for (int i = 0; i < components; i++) {
241             convex_hull[i] = new int[3];
242         }
243 
244         for (int i = 0; i < width; i++) {
245             for (int j = 0; j < height; j++) {
246 
247                 char access = matrix[(i * width) + j];
248                 if (access != '0') {
249                     convex_hull[tempo][0] = j;
250                     convex_hull[tempo][1] = i;
251 
252                     auto s = (int)access;
253                     convex_hull[tempo][2] = s;
254 
255                     if (tempo >= components - 1) break;
256                     tempo += 1;
257                 }
258             }
259         }
260     }
261 
262     void Jarvis__STD(const int num_threads) {
263         int m = 0, minind = 0;
264         double mincos, cosine;
265         double len = 0, maxlen = 0;
266 
267         if (components == 1) {
268             result.push_back(convex_hull[0]);
269             return;
270         }
271         if (components == 2) {
```

```
272             result.push_back(convex_hull[0]);
273             result.push_back(convex_hull[1]);
274             return;
275         }
276 
277         double* first_elements = new double[2];
278         first_elements[0] = convex_hull[0][0];
279         first_elements[1] = convex_hull[0][1];
280 
281         for (int i = 1; i < components; i++) {
282             if (convex_hull[i][1] < first_elements[1]) {
283                 m = i;
284             } else {
285                 if ((convex_hull[i][1] == first_elements[1]) && (convex_hull[i][0]
                        < first_elements[0])) {
286                     m = i;
287                 }
288             }
289         }
290 
291         result.push_back(convex_hull[m]);
292 
293         int* last = new int[2];
294         int* beforelast = new int[2];
295 
296         last = convex_hull[m];
297         beforelast[0] = convex_hull[m][0] - 2;
298         beforelast[1] = convex_hull[m][1];
299 
300         for(;;) {
301             mincos = 2;
302             int step_i = 0;
303             auto fq = [&]() {
304                 int core = step_i++;
305                 const int start = (core * components) / num_threads;
306                 const int finish = (core + 1) * components / num_threads;
307 
308                 for (int i = start; i < finish; i++) {
309                     cosine = roundFloat(turnPoint(last, beforelast,
                            convex_hull[i]) * INFINITE) / INFINITE;
310                     if (cosine < mincos) {
311                         minind = i;
312                         mincos = cosine;
313                         maxlen = betweenPoints(last, convex_hull[i]);
314                     } else if (cosine == mincos) {
315                         len = betweenPoints(last, convex_hull[i]);
316                         if (len > maxlen) {
317                             minind = i;
318                             maxlen = len;
319                         }
320                     }
321                 }
322 
323                 beforelast = last;
324                 last = convex_hull[minind];
325             };
326 
327             thread* ourThreads = new thread[num_threads];
328             for (int i = 0; i < num_threads; i++) {
329                 ourThreads[i] = thread(fq);
```

```
330             }
331             for (int i = 0; i < num_threads; i++) {
332                 ourThreads[i].join();
333             }
334 
335             delete[] ourThreads;
336 
337             if (last == convex_hull[m])
338                 break;
339             result.push_back(convex_hull[minind]);
340         }
341     }
342 
343     void printResult() {
344         int* temp = NULL;
345 
346         for (int i = 0; i < result.size(); i++) {
347             temp = result[i];
348         }
349         putchar('\n');
350         for (int i = 0; i < sizeof(temp)/sizeof(temp[0]); i++)
351             cout << "(" << temp[i] << ")";
352 
353         putchar('\n');
354     }
355 
356     void findConnectedComponents__Threads(const int num_threads) {
357         int kWidth = 0, kHeight = 0;
358         string s; // converion
359 
360         amount = 1;
361         Mask localMask;
362 
363             for (int i = 1; i <= width; i++) {
364                 for (int j = 1; j <= height; j++) {
365                     kWidth = j - 1;
366                     if (kWidth <= 0) {
367                         kWidth = 1;
368                         localMask.B = 0;
369                     } else {
370                         localMask.B = getPixel(i, kWidth);
371                     }
372 
373                     kHeight = i - 1;
374                     if (kHeight <= 0) {
375                         kHeight = 1;
376                         localMask.C = 0;
377                     } else {
378                         localMask.C = getPixel(kHeight, j);
379                     }
380 
381                     localMask.A = getPixel(i, j);
382 
383                     if (localMask.A == 0) {
384                     } else if (localMask.B == 0 && localMask.C == 0) {
385                         components += 1;
386                         amount += 1;
387                         s = to_string(amount);
388                         matrix[((i - 1) * width) + (j - 1)] = s[0];
389                     } else if (localMask.B != 0 && localMask.C == 0) {
```

```
                        matrix[((i - 1) * width) + (j - 1)] = (char)localMask.B;
                    } else if (localMask.B == 0 && localMask.C != 0) {
                        matrix[((i - 1) * width) + (j - 1)] = (char)localMask.C;
                    } else if (localMask.B != 0 && localMask.C != 0) {
                        matrix[((i - 1) * width) + (j - 1)] = (char)localMask.B;

                        // find pixels labeled as C, then relabel them as B.
                        if (localMask.B != localMask.C) {
                            int step_i = 0;
                            auto f = [&]() {
                                int core = step_i++;
                                const int start = (core * i) / num_threads;
                                const int finish = (core + 1) * i / num_threads;
                                for (int a = start; a < finish; a++) {
                                    for (int b = 0; b < height; b++) {
                                        if (matrix[(a * width) + b] ==
                                            localMask.C) {
                                            matrix[(a * width) + b] =
                                                (char)localMask.B;
                                        }
                                    }
                                }
                            };
                            thread* ourThreads = new thread[num_threads];
                            for (int i = 0; i < num_threads; i++) {
                                ourThreads[i] = thread(f);
                            }
                            for (int i = 0; i < num_threads; i++) {
                                ourThreads[i].join();
                            }
                            delete[]ourThreads;
                        }

                        // fi
                    }
                }
            }
        }
        void Jarvis() {
            int m = 0, minind = 0;
            double mincos, cosine;
            double len = 0, maxlen = 0;

            if (components == 1) {
                result.push_back(convex_hull[0]);
                return;
            }
            if (components == 2) {
                result.push_back(convex_hull[0]);
                result.push_back(convex_hull[1]);
                return;
            }

            double* first_elements = new double[2];
            first_elements[0] = convex_hull[0][0];
            first_elements[1] = convex_hull[0][1];

            for (int i = 1; i < components; i++) {
                if (convex_hull[i][1] < first_elements[1]) {
                    m = i;
```

```
448             } else {
449                 if ((convex_hull[i][1] == first_elements[1]) && (convex_hull[i][0]
                        < first_elements[0])) {
450                     m = i;
451                 }
452             }
453         }
454 
455         result.push_back(convex_hull[m]);
456 
457         int* last = new int[2];
458         int* beforelast = new int[2];
459 
460         last = convex_hull[m];
461         beforelast[0] = convex_hull[m][0] - 2;
462         beforelast[1] = convex_hull[m][1];
463 
464         for(;;) {
465             mincos = 2;
466             for (int i = 0; i < components; i++) {
467                 cosine = roundFloat(turnPoint(last, beforelast, convex_hull[i]) *
                        INFINITE) / INFINITE;
468                 if (cosine < mincos) {
469                     minind = i;
470                     mincos = cosine;
471                     maxlen = betweenPoints(last, convex_hull[i]);
472                 } else if (cosine == mincos) {
473                     len = betweenPoints(last, convex_hull[i]);
474                     if (len > maxlen) {
475                         minind = i;
476                         maxlen = len;
477                     }
478                 }
479             }
480 
481             beforelast = last;
482             last = convex_hull[minind];
483             if (last == convex_hull[m])
484                 break;
485             result.push_back(convex_hull[minind]);
486         }
487     }
488     ~BitImage() {
489         matrix.clear();
490     }
491 };
492 
493 int main() {
494     int height, width;
495 
496     cout << "We represent our binary image as a packed one-dimensional array in
            which each pixel is a bit." << endl;
497     /* user input */
498     /*cout << "Enter height, then width: ";
499     cin >> height >> width;
500     putchar('\n');*/
501 
502     /* predefined input */
503     width = 4500;
504     height = 4500;
```

```
505
506     /* create binary image (object) */
507     BitImage be(height, width);
508     BitImage ae(height, width);
509
510     be.setRandom(); // set random image
511     ae = be;
512
513     int num_threads = 1;
514
515     //start measuring time
516     clock_t begin = clock();
517     //doing job
518     be.findConnectedComponents__Threads(num_threads);
519     be.makeConvex();
520     be.Jarvis();
521     //stop measuring time
522     clock_t end = clock();
523     double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
524     //end.
525
526     be.printResult();
527
528     cout << "(Threads) Time spent on finding connected components and making the
            convex hull is: " << time_spent << " seconds" << endl;
529
530     //start measuring time
531     begin = clock();
532     //doing job
533     ae.findConnectedComponents();
534     ae.makeConvex();
535     ae.Jarvis();
536     //stop measuring time
537     end = clock();
538     time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
539
540     ae.printResult();
541     cout << "(Seq) Time spent on finding connected components and making the
            convex hull is: " << time_spent << " seconds" << endl;
542
543     return 0;
544 }
```