



ALGORITHMES AVANCÉS ET COMPLEXITÉ

Deuxième découverte d'algorithmes

Étienne Gouinaud
Barthélemy Juglair

Février 2022

Table des matières

1	Tri par tas d'un tableau par Étienne Gouinaud	4
1.1	Fonctionnement de l'algorithme	4
1.2	Complexité	5
1.3	Test	5
1.3.1	Tableau vide	5
1.3.2	Tableau contenant un élément	6
1.3.3	Tableau contenant deux éléments	6
1.3.4	Tableau contenant des doublons	6
1.3.5	Tableau contenant seulement des valeurs négatives	7
1.3.6	Tableau contenant seulement des valeurs positives	7
1.3.7	Tableau contenant des valeurs limites	7
1.3.8	Tableau déjà trié	8
2	Recherche de l'enveloppe convexe d'un ensemble de points par Barthélemy Juglair	9
2.1	Introduction	9
2.2	Fonctionnement du programme	9
2.3	Description de l'algorithme	9
2.4	Complexité de l'algorithme	10
2.5	Test	10
2.5.1	Ensemble contenant un seul point	10
2.5.2	Ensemble contenant deux points	10
2.5.3	Ensemble contenant quatre points sur l'enveloppe convexe	10
2.5.4	Ensemble contenant quatre points non alignés	11
2.5.5	Ensemble contenant des points situés aléatoirement	12
Annexe		16
	Code du tri par tas	16
	Code du tri par tas	20

Table des figures

1	Représentation d'un tas grâce à un tableau	4
2	Représentation d'un tas maximal	4
3	Résultat sur un tableau vide	6
4	Résultat sur un tableau d'un réel	6
5	Résultat sur un tableau de deux réels	6
6	Résultat sur un tableau contenant des doublons	7
7	Résultat sur un tableau de réels négatifs	7
8	Résultat sur un tableau de réels positifs	7
9	Résultat avec des valeurs limites	8
10	Résultat sur un tableau déjà trié	8
11	Dernière étape exécution enveloppe convexe	9
12	Résultat sur un ensemble d'un seul point	10
13	Résultat sur un ensemble de deux points	10
14	Résultat affichage sur un ensemble de 4 points	11
15	Résultat sur un ensemble de 4 points	11
16	Résultat affichage sur un ensemble de 4 points non alignés	12
17	Résultat sur un ensemble de 4 points non alignés	12
18	Résultat affichage sur un ensemble quelconque	13
19	Résultat sur un ensemble quelconque	14

1 Tri par tas d'un tableau par Étienne Gouinaud

L'algorithme a été codé en C et Étienne s'est inspiré du livre Numerical Recipes in C : The Art of Scientific Computing, Second Edition[1] pour réaliser cet algorithme.

1.1 Fonctionnement de l'algorithme

Le tri par tas construit un tas en comparant les valeurs du tableau (cf. **Figure 1**). Puis transforme ce tas en tas maximal, c'est à dire de telle sorte qu'un nœud soit toujours plus grand que ses fils (cf. **Figure 2**). Ensuite, il place la valeur du premier nœud, donc la plus grande, à la fin du tableau. Puis, il enlève cette valeur du tas, et réitère jusqu'à avoir rangé toutes les valeurs. L'algorithme d'Étienne trie un tableau en utilisant un tas, cependant il modifie le tableau pour représenter le tas utilisé pendant les opérations, l'algorithme est alors dit **en place**. Cela permet d'optimiser la mémoire utilisée pour ne manipuler que le strict nécessaire.

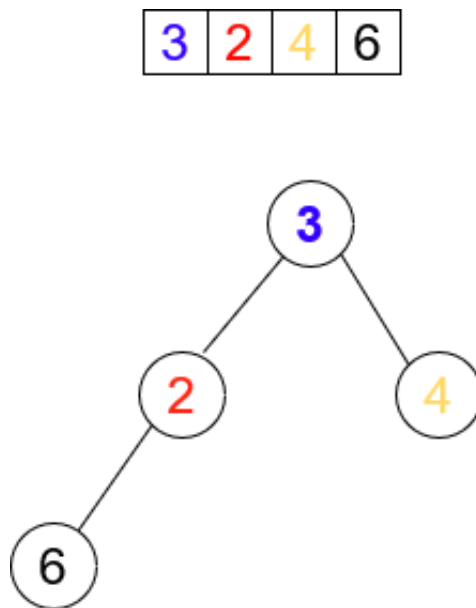


FIGURE 1 – Représentation d'un tas grâce à un tableau

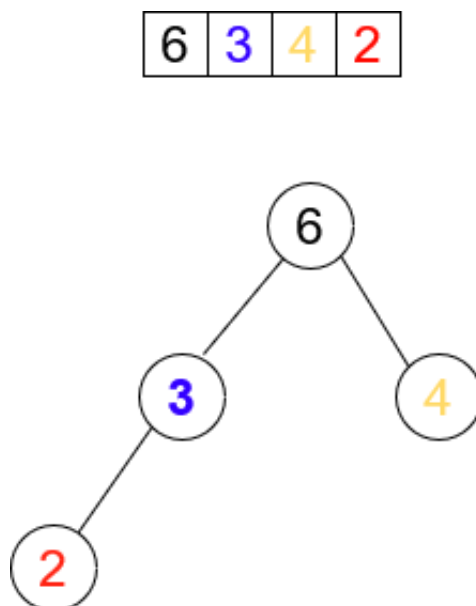


FIGURE 2 – Représentation d'un tas maximal

```

1  /*
2  * This function is made to heapsort an array.
3  * The principle is to isolate the bigger value and at the end of a cycle you
4  * put it at the end of the array.
5  * This allow to have an  $N \cdot \log_2(N)$  complexity, because when you put a value at
6  * the end the algorithm don't analyse it again.
7  * The result could be compare to heapsort from the <stdlib.h> library.
8  */
9  void myheapsort(float *tab, int n){
10     int i;
11     int boucle = 0;
12     float tmp;
13     while (boucle < n - 1){
14         i = 0;
15         // Try to isolate the biggest value at the start
16         while (i < n - boucle){
17             if (tab[i] > tab[(i - 1) / 2]){
18                 tmp = tab[i];
19                 tab[i] = tab[(i - 1) / 2];
20                 tab[(i - 1) / 2] = tmp;
21                 i = 0;
22                 continue;
23             }
24             i++;
25         }
26         // We put the biggest value at the end
27         tmp = tab[0];
28         tab[0] = tab[n - boucle - 1];
29         tab[n - boucle - 1] = tmp;
30         boucle++;
31     }
32 }

```

Extrait de code 1 – Algorithme du tri par tas d'Étienne

1.2 Complexité

Le tri par tas fait partie des meilleurs algorithmes de tri puisqu'il a une complexité temporelle quasi linéaire en $O(n) = n \log(n)$ et bat ainsi les algorithmes en complexité quadratique tel que le trie à bulle qui nécessite de comparer chaque valeur avec l'ensemble du tableau pour trouver sa juste place.

1.3 Test

1.3.1 Tableau vide

```

1  /*
2  * Test heap sort algorithm with an empty tab
3  */
4  void testHeapSortEmpty(){
5     float tab[0] = {};
6
7     printTab(tab, 0);
8     myheapsort(tab, 0);
9     printTab(tab, 0);
10 }

```

Extrait de code 2 – Test sur un tableau vide

L'Extrait de code 2 montre un exemple de résultat pour un tableau vide. La fonction affiche le tableau, puis le trie avec l'algorithme de tri par tas, et affiche enfin le résultat.



FIGURE 3 – Résultat sur un tableau vide

1.3.2 Tableau contenant un élément

```
1  /*
2  * Test heap sort algorithm with one element
3  */
4  void testHeapSortOne(){
5      float tab[1] = {8.0};
6
7      printTab(tab, 1);
8      myheapsort(tab, 1);
9      printTab(tab, 1);
10 }
```

Extrait de code 3 – Test sur un tableau contenant un élément

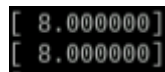


FIGURE 4 – Résultat sur un tableau d'un réel

1.3.3 Tableau contenant deux éléments

```
1  /**
2  * Test heap sort algorithm with two elements
3  */
4  void testHeapSortTwo(){
5      float tab[2] = {8.0, -16.0};
6
7      printTab(tab, 2);
8      myheapsort(tab, 2);
9      printTab(tab, 2);
10 }
```

Extrait de code 4 – Test sur un tableau contenant deux éléments



FIGURE 5 – Résultat sur un tableau de deux réels

1.3.4 Tableau contenant des doublons

```
1  /*
2  * Test heap sort algorithm with duplicated elements
3  */
4  void testHeapSortDuplicates(){
5      float tab[5] = {2.0, 5.5, 5.5, -3.0, 2.0};
6
7      printTab(tab, 5);
8      myheapsort(tab, 5);
```

```

9   printTab(tab, 5);
10  }

```

Extrait de code 5 – Test sur un tableau contenant des doublons

```

[ 2.000000, 5.500000, 5.500000, -3.000000, 2.000000]
[ -3.000000, 2.000000, 2.000000, 5.500000, 5.500000]

```

FIGURE 6 – Résultat sur un tableau contenant des doublons

1.3.5 Tableau contenant seulement des valeurs négatives

```

1  /*
2  * Test heap sort algorithm with only negative values
3  */
4  void testHeapSortNegative(){
5      float tab[5] = {-1.2, -0.6, -2.5, -3.3, -6.1};
6
7      printTab(tab, 5);
8      myheapsort(tab, 5);
9      printTab(tab, 5);
10 }

```

Extrait de code 6 – Test sur un tableau contenant seulement des valeurs négatives

```

[ -1.200000, -0.600000, -2.500000, -3.300000, -6.100000]
[ -6.100000, -3.300000, -2.500000, -1.200000, -0.600000]

```

FIGURE 7 – Résultat sur un tableau de réels négatifs

1.3.6 Tableau contenant seulement des valeurs positives

```

1  /*
2  * Test heap sort algorithm with positive values
3  */
4  void testHeapSortPositive(){
5      float tab[5] = {1.2, 0.6, 2.5, 3.3, 6.1};
6
7      printTab(tab, 5);
8      myheapsort(tab, 5);
9      printTab(tab, 5);
10 }

```

Extrait de code 7 – Test sur un tableau contenant seulement des valeurs positives

```

[ 1.200000, 0.600000, 2.500000, 3.300000, 6.100000]
[ 0.600000, 1.200000, 2.500000, 3.300000, 6.100000]

```

FIGURE 8 – Résultat sur un tableau de réels positifs

1.3.7 Tableau contenant des valeurs limites

```

1  /*
2  * Test heap sort algorithm with bounds values
3  */
4  void testHeapSortBounds(){
5      float tab[4] = {FLT_MAX, -FLT_MAX, -FLT_MIN, FLT_MIN};

```

```

6   printTabExponent(tab, 4);
7   myheapsort(tab, 4);
8   printTabExponent(tab, 4);
9 }

```

Extrait de code 8 – Test sur un tableau contenant des valeurs limites

```

[ 3.402823e+38, -3.402823e+38, -1.175494e-38, 1.175494e-38]
[ -3.402823e+38, -1.175494e-38, 1.175494e-38, 3.402823e+38]

```

FIGURE 9 – Résultat avec des valeurs limites

1.3.8 Tableau déjà trié

```

1  /*
2   * Test heap sort algorithm with already sorted values
3   */
4  void testHeapSortSorted(){
5      float tab[5] = {-5.0, -2.3, 0.0, 5.0, 36.0};
6      printTab(tab, 5);
7      myheapsort(tab, 5);
8      printTab(tab, 5);
9  }

```

Extrait de code 9 – Test sur un tableau déjà trié

```

[-5.000000, -2.300000, 0.000000, 5.000000, 36.000000]
[-5.000000, -2.300000, 0.000000, 5.000000, 36.000000]

```

FIGURE 10 – Résultat sur un tableau déjà trié

2 Recherche de l'enveloppe convexe d'un ensemble de points par Barthélemy Juglair

L'algorithme a été codé en C++ et la partie graphique utilise la librairie SMFL. J'ai récupéré certaines fonctions de Slimane Fakani et les ai adaptées à mon problème pour dessiner des points, des lignes et définir les limites de la fenêtre.

2.1 Introduction

L'algorithme `convex_hul` a pour vocation de déterminer la sous liste de points qui compose l'enveloppe convexe d'un nuage de point. L'enveloppe convexe peut se définir comme étant le sous-ensemble contenant le minimum de points qui une fois reliés contiennent tout l'ensemble.

2.2 Fonctionnement du programme

Barthélémy a réalisé un programme graphique en c++ via la bibliothèque SMFL. On renseigne l'ensemble de points sur lequel appliqué l'algorithme dans le tableau `tab` du `main`. Lorsque l'on compile et qu'on exécute on peut observer pas à pas dans la fenêtre qui s'ouvre l'exécution de l'algorithme et les points qui se relie au fil des étapes. A la sortie on observe l'enveloppe convexe et on récupère via la ligne de commande le contenu de l'enveloppe.

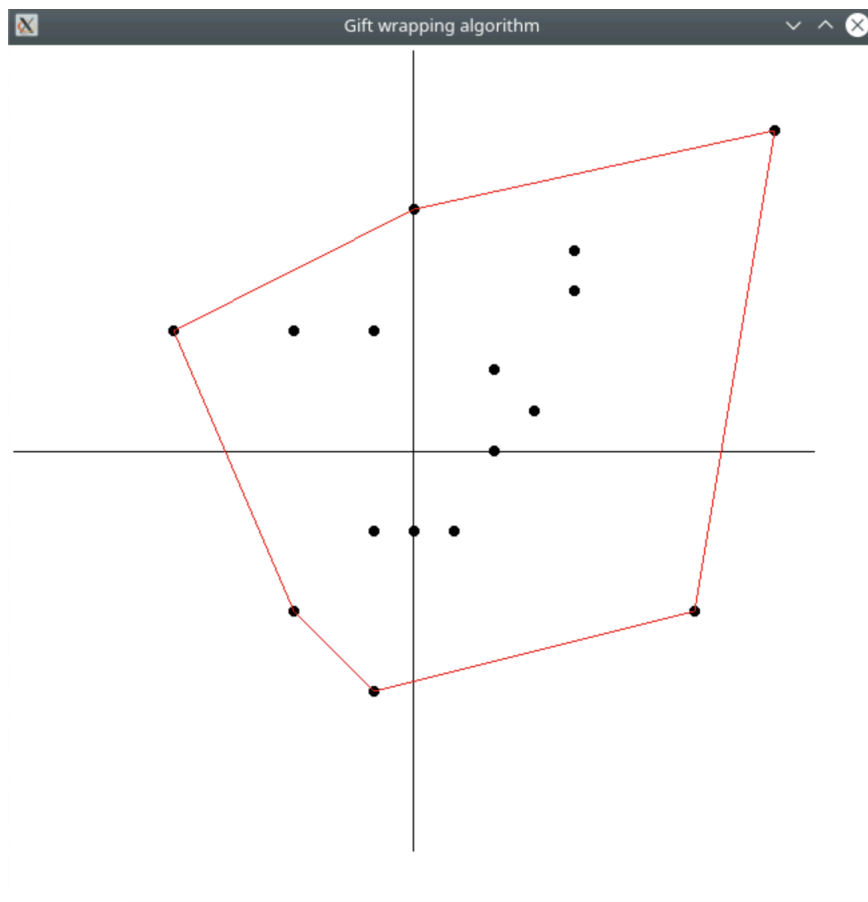


FIGURE 11 – Dernière étape exécution enveloppe convexe

2.3 Description de l'algorithme

Le principe de l'algorithme est assez simple. Il parcourt l'ensemble des points et mesure entre 3 points le produit scalaire de l'angle formé par ces 3 points. Si l'angle est obtu le point se situe

donc à l'extérieur de l'enveloppe convexe actuelle, on le rajoute donc à la liste et on passe au point suivant.

2.4 Complexité de l'algorithme

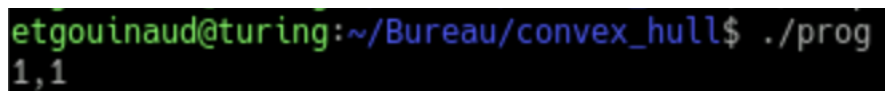
La complexité d'un tel algorithme est de $O(nh)$ où n est le nombre de point total et h le nombre de point contenu dans l'enveloppe convexe.

2.5 Test

2.5.1 Ensemble contenant un seul point

```
1 // Create points
2 const int N = 1; // Number of points
3 std::vector<Point> S;
4 int tab[N*2] = {1,1};
```

Extrait de code 10 – Test sur un ensemble d'un seul point



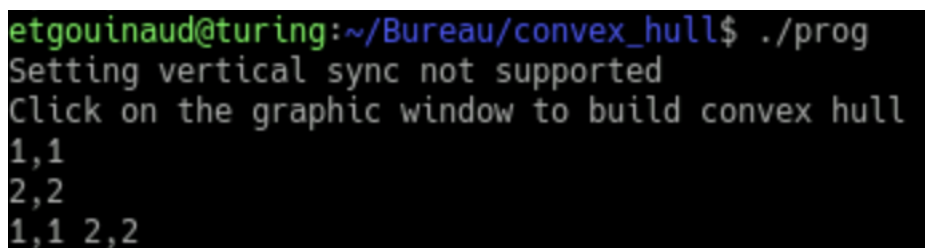
```
etgouinaud@turing:~/Bureau/convex_hull$ ./prog
1,1
```

FIGURE 12 – Résultat sur un ensemble d'un seul point

2.5.2 Ensemble contenant deux points

```
1 // Create points
2 const int N = 2; // Number of points
3 std::vector<Point> S;
4 int tab[N*2] = {1,1,2,2};
```

Extrait de code 11 – Test sur un ensemble d'un seul point



```
etgouinaud@turing:~/Bureau/convex_hull$ ./prog
Setting vertical sync not supported
Click on the graphic window to build convex hull
1,1
2,2
1,1 2,2
```

FIGURE 13 – Résultat sur un ensemble de deux points

2.5.3 Ensemble contenant quatre points sur l'enveloppe convexe

```
1 // Create points
2 const int N = 4; // Number of points
3 std::vector<Point> S;
4 int tab[N*2] = {1,1,2,2,1,2,2,1};
```

Extrait de code 12 – Test sur un ensemble de 4 points sur enveloppe

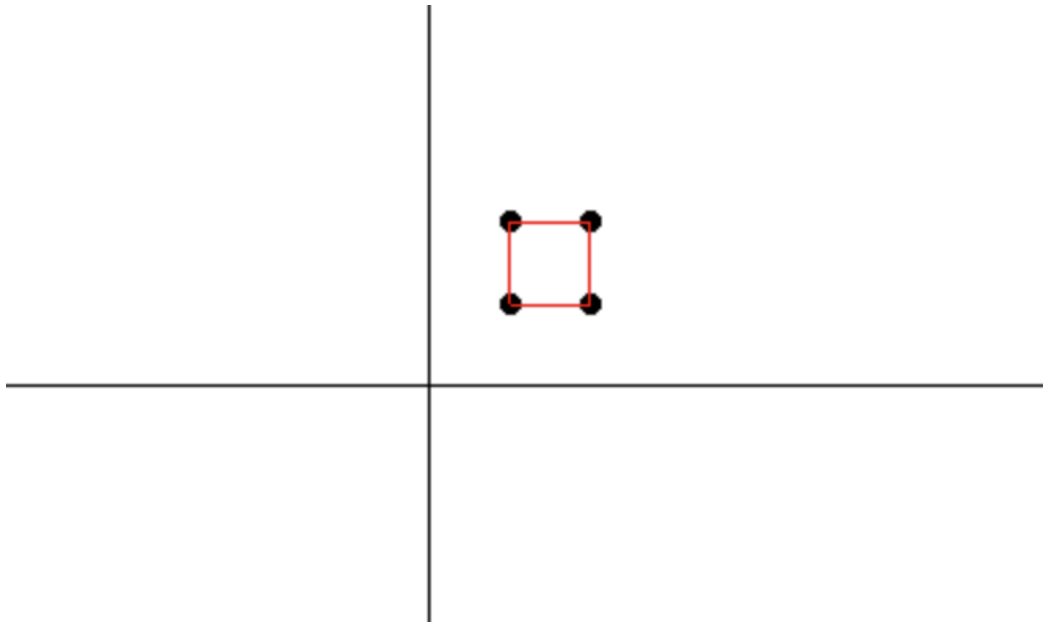


FIGURE 14 – Résultat affichage sur un ensemble de 4 points

```

etgouinaud@turing:~/Bureau/convex_hull$ ./prog
Setting vertical sync not supported
Click on the graphic window to build convex hull
1,1
1,2
2,2
2,1
End reached. Quit the graphic window to print convex hull
1,1 1,2 2,2 2,1

```

FIGURE 15 – Résultat sur un ensemble de 4 points

2.5.4 Ensemble contenant quatre points non alignés

```

1 // Create points
2 const int N = 4; // Number of points
3 std::vector<Point> S;
4 int tab[N*2] = {1,1,4,1,3,2,4,4};

```

Extrait de code 13 – Test sur un ensemble de 4 points non alignés

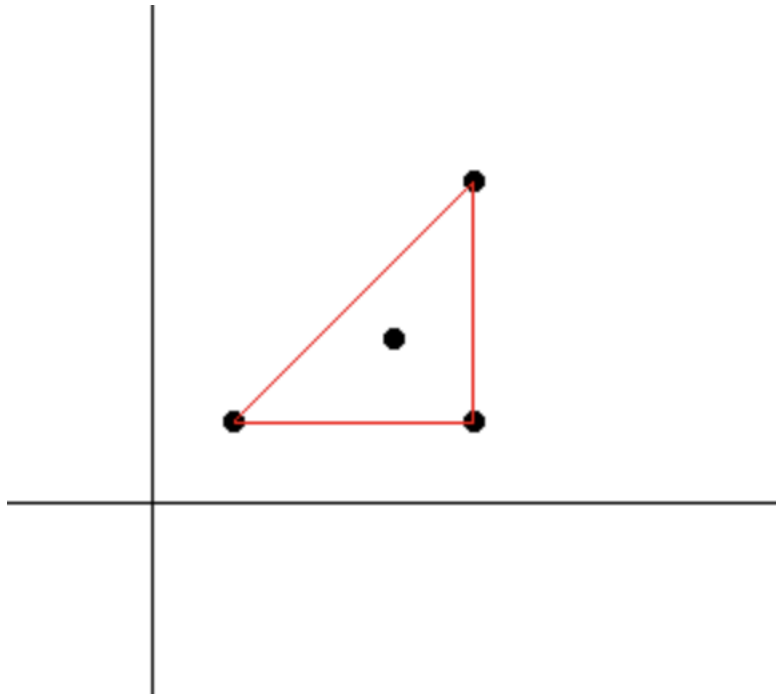


FIGURE 16 – Résultat affichage sur un ensemble de 4 points non alignés

```

etgouinaud@turing:~/Bureau/convex_hull$ ./prog
Setting vertical sync not supported
Click on the graphic window to build convex hull
1,1
4,4
4,1
End reached. Quit the graphic window to print convex hull
1,1 4,4 4,1

```

FIGURE 17 – Résultat sur un ensemble de 4 points non alignés

2.5.5 Ensemble contenant des points situés aléatoirement

```

1 // Create points
2 const int N = 16; // Number of points
3 std::vector<Point> S;
4 int tab[N*2] = {-6,3, 3,1, 2,0, -3,-4, -3,3, -1,-2, 0,6, -1,3, 4,5, 4,4,
1,-2, 0,-2, -1,-6, 9,8, 7,-4, 2,2};

```

Extrait de code 14 – Test sur un ensemble quelconque

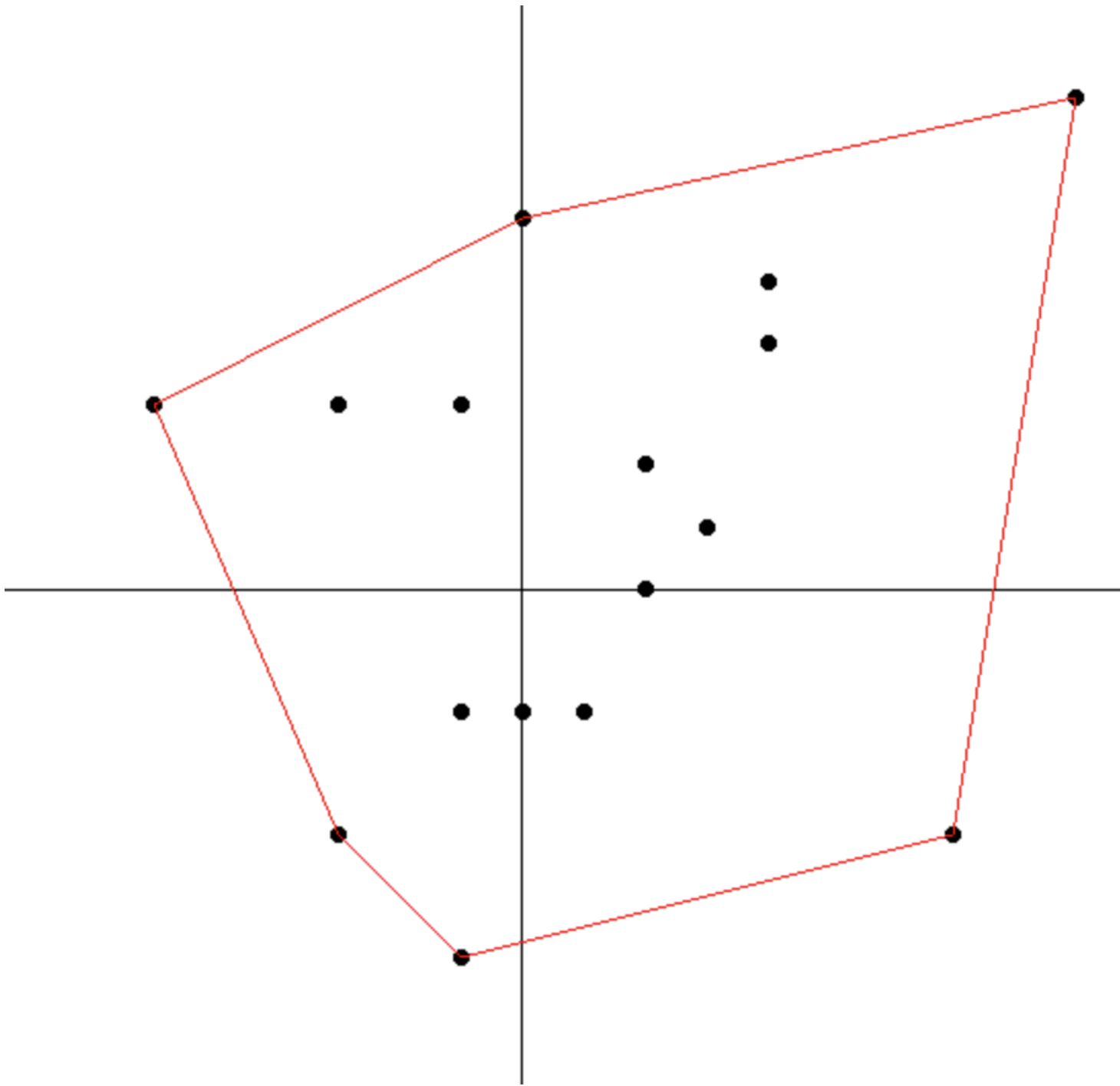


FIGURE 18 – Résultat affichage sur un ensemble quelconque

```
etgouinaud@turing:~/Bureau/convex_hull$ ./prog
Setting vertical sync not supported
Click on the graphic window to build convex hull
-6,3
0,6
9,8
7,-4
-1,-6
-3,-4
End reached. Quit the graphic window to print convex hull
-6,3 0,6 9,8 7,-4 -1,-6 -3,-4
```

FIGURE 19 – Résultat sur un ensemble quelconque

Bibliographie

- [1] William H. PRESS, Saul A. TEUKOLSKY, William T. VETTERLING et Brian P. FLANNERY. *Numerical Recipes in C : The Art of Scientific Computing, Second Edition*. Cambridge University Press, oct. 1992. URL : https://www.cec.uchile.cl/cinetica/pcordero/MC_libros/NumericalRecipesinC.pdf.

Annexe

Code du convex hull

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <iostream>
5 #include <vector>
6 #include <sstream>
7 #include <math.h>
8 #include <SFML/Graphics.hpp>
9
10 #include "Point.hpp"
11
12 // Constants
13 const int INF = 1e6;
14 const int WINDOW_WIDTH = 600;
15 const int WINDOW_HEIGHT = 600;
16 const int WINDOW_DELTA = 50;
17 const int POINT_RADIUS = 4;
18
19 /**
20  * @brief Return x scaled with WINDOW_WIDTH and bounds
21  *
22  * @param x      : value to scale up
23  * @param bounds : bounds of window
24  * @return int    : scaled value
25  */
26 int get_scaled_x(int x, const std::array<int, 4>& bounds) {
27     return WINDOW_WIDTH * ((x - bounds[0]) / (float)(abs(bounds[2]) + abs(
28         bounds[0])));
29 }
30
31 /**
32  * @brief Return y scaled with WINDOW_HEIGHT and bounds
33  *
34  * @param y      : value to scale up
35  * @param bounds : bounds of window
36  * @return int    : scaled value
37  */
38 int get_scaled_y(int y, const std::array<int, 4>& bounds) {
39     return WINDOW_HEIGHT * (1 - ((y - bounds[1]) / (float)(abs(bounds[3]) +
40         abs(bounds[1]))));
41 }
42
43 /**
44  * @brief Draw a line on window using coordinates, bounds and color
45  *
46  * @param window : graphic window
47  * @param x0      : x-axis of first point
48  * @param y0      : y-axis of first point
49  * @param x1      : x-axis of second point
50  * @param y1      : y-axis of second point
51  * @param bounds  : bounds of window
52  * @param color   : color of the line
53  */
54 void draw_line(sf::RenderWindow& window, int x0, int y0, int x1, int y1, const
55     std::array<int, 4>& bounds, sf::Color color = sf::Color::Black) {
56     sf::VertexArray line(sf::LinesStrip, 2);
57     line[0].position = sf::Vector2f(get_scaled_x(x0, bounds) + POINT_RADIUS,
```



```

    get_scaled_y(y0, bounds) + POINT_RADIUS);
55     line[0].color = color;
56     line[1].position = sf::Vector2f(get_scaled_x(x1, bounds) + POINT_RADIUS,
    get_scaled_y(y1, bounds) + POINT_RADIUS);
57     line[1].color = color;
58
59     window.draw(line);
60 }
61
62 /**
63  * @brief Draw a point on window using coordinates, bounds and color
64  *
65  * @param window : graphic window
66  * @param x       : x-axis
67  * @param y       : y-axis
68  * @param bounds  : bounds of window
69  * @param color   : color of the point
70  */
71 void draw_point(sf::RenderWindow& window, int x, int y, const std::array<int,
    4>& bounds, sf::Color color = sf::Color::Red) {
72     auto circleShape = sf::CircleShape(POINT_RADIUS);
73     int xd = get_scaled_x(x, bounds);
74     int yd = get_scaled_y(y, bounds);
75
76     circleShape.setFillColor(color);
77     circleShape.setPosition(xd, yd);
78     window.draw(circleShape);
79 }
80
81 /**
82  * @brief Clear the window
83  *
84  * @param window : graphic window
85  */
86 void draw_plane(sf::RenderWindow& window) {
87     window.clear(sf::Color::White);
88 }
89
90 /**
91  * @brief Return minimal x between points of S
92  *
93  * @param S      : vector of points
94  * @return int   : value of minimal x
95  */
96 int getMinX(std::vector<Point> S){
97     int min = 0;
98     for(int i = 1; i < S.size(); i++){
99         if(S.at(i).getX() < S.at(min).getX()){
100             min = i;
101         }
102     }
103     return min;
104 }
105
106 /**
107  * @brief Return orientation of p2 compared to (p1, p3)
108  *
109  * @param p1 : first point
110  * @param p2 : second point
111  * @param p3 : third point
112  * @return int <0 if p2 is on the left of (p1, p3)

```

```

113 *           0 if p1, p2 and p3 are collinear
114 *           >0 if p2 is on the right of (p1, p3)
115 */
116 int orientation(Point p1, Point p2, Point p3){
117     int x1 = p1.getX() - p2.getX();
118     int x2 = p1.getX() - p3.getX();
119     int y1 = p1.getY() - p2.getY();
120     int y2 = p1.getY() - p3.getY();
121     return y2*x1 - y1*x2;
122 }
123
124 /**
125  * @brief Realize one iteration of the gift wrapping algorithm
126  *
127  * @param p      : current point of the convex hull
128  * @param S      : all points
129  * @return Point : next point of the convex hull
130  */
131 Point giftWrapping(Point p, std::vector<Point> S){
132     double min_angle = INF;
133     double angle;
134     Point p_next;
135
136     if(S.at(0) != p){
137         p_next = S.at(0);
138     }
139     else{
140         p_next = S.at(1);
141     }
142
143     for(auto& w : S){
144         // If candidate point is not current point
145         if(w != p){
146             // If candidate is oriented counterclockwise of (p, p_next) line
147             if(orientation(p, w, p_next) < 0){
148                 // Candidate becomes p_next
149                 p_next = w;
150             }
151         }
152     }
153     return p_next;
154 }
155
156 /**
157  * @brief Build convex hull of vector of points
158  *
159  * @param S      : points
160  * @param bounds  : window bounds
161  * @return std::vector<Point> : convex hull
162  */
163 std::vector<Point> getConvexHull(std::vector<Point> S, std::array<int, 4>
164     bounds){
165     if(S.size() < 2){
166         return S;
167     }
168
169     // Create graphic window
170     sf::RenderWindow window(
171         sf::VideoMode(WINDOW_WIDTH + WINDOW_DELTA, WINDOW_HEIGHT +
172             WINDOW_DELTA),
173         "Gift wrapping algorithm",

```

```

172     sf::Style::Default
173 );
174 sf::Event event;
175
176 std::vector<Point> L;
177 int min = INF;
178 Point p0 = S[getMinX(S)];
179 Point p_draw;
180 Point p = p0;
181 // Write user indication
182 std::cout << "Click on the graphic window to build convex hull" << std::
endl;
183
184 // Show window
185 while (window.isOpen()) {
186     // Clear window and draw axis
187     draw_plane(window);
188     draw_line(window, bounds[0], 0, bounds[2], 0, bounds);
189     draw_line(window, 0, bounds[1], 0, bounds[3], bounds);
190
191     // Draw points of S
192     for (const auto& point : S) {
193         auto color = sf::Color::Black;
194         draw_point(window, point.getX(), point.getY(), bounds, color);
195     }
196
197     // Draw convex hull
198     if(L.size() > 0){
199         auto color = sf::Color::Red;
200         p_draw = L.at(0);
201         draw_line(window, p_draw.getX(), p_draw.getY(), L.at(L.size()-1).
getX(), L.at(L.size()-1).getY(), bounds, color);
202         for(int i = 1; i < L.size(); i++){
203             draw_line(window, p_draw.getX(), p_draw.getY(), L.at(i).getX()
, L.at(i).getY(), bounds, color);
204             p_draw = L.at(i);
205         }
206     }
207
208     while (window.pollEvent(event)){
209         // If window is closed
210         if (event.type == sf::Event::Closed){
211             window.close();
212         }
213         // If a mouse button is released
214         else if (event.type == sf::Event::MouseButtonReleased){
215             // If the convex hull is built
216             if(L.size() > 1 && p == p0){
217                 std::cout << "End reached. Quit the graphic window to
print convex hull" << std::endl;
218             }
219             else{
220                 // Find next point
221                 std::cout << p.getX() << "," << p.getY() << std::endl;
222                 L.push_back(p);
223                 p = giftWrapping(p, S);
224             }
225         }
226     }
227     window.display();
228 }

```

```

229     return L;
230 }
231
232
233
234 int main(){
235     // Create points
236     const int N = 16; // Number of points
237     std::vector<Point> S;
238     int tab[N*2] = {-6,3, 3,1, 2,0, -3,-4, -3,3, -1,-2, 0,6, -1,3, 4,5, 4,4,
239 1,-2, 0,-2, -1,-6, 9,8, 7,-4, 2,2};
240     for(int i = 0; i < N*2; i+=2){
241         Point p(tab[i], tab[i+1]);
242         S.push_back(p);
243     }
244
245     // Definition of screen bounds.
246     // (!) points outside of the bounds will not appear on the window
247     std::array<int, 4> bounds;
248     bounds[0] = -10; // x min
249     bounds[1] = -10; // y min
250     bounds[2] = 10;  // x max
251     bounds[3] = 10;  // y max
252
253     // Build convex hull
254     auto L = getConvexHull(S, bounds);
255
256     // Print convex hull
257     for(auto& point : L){
258         std::cout << point.getX() << "," << point.getY() << " ";
259     }
260     std::cout << std::endl;
261     return EXIT_SUCCESS;
262 }

```

Extrait de code 15 – Code complet du convex hull réalisé par Barthélémy.

Code du tri par tas

```

1 // This algorithm has been inspired from the book Numerical Recipes in C
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <float.h>
6
7 /*
8  * Print an array
9  */
10 void printTab(float * tab,int n){
11     int i;
12     printf("[ ");
13     for(i = 0 ; i < n-1 ; i++){
14         printf("%f, ",tab[i]);
15     }
16     printf("%f]\n", tab[i]);
17 }
18
19 /*
20  * Print an array using exponent notation
21  */
22 void printTabExponent(float * tab,int n){

```

```

23     int i;
24     printf("[ ");
25     for(i = 0 ; i < n-1 ; i++){
26         printf("%e, ", tab[i]);
27     }
28     printf("%e]\n", tab[i]);
29 }
30
31 /*
32  * Initializes the array randomly.
33  */
34 void initRandTab(float * tab, int n){
35     srand(n);
36     for(int i = 0 ; i < n ; i++){
37         //tab[i] = rand() * 1.0;
38         tab[i] = n - i ; /* rand()%10;
39     }
40 }
41
42 /*
43  * This function is made to heapsort an array.
44  * The principle is to isolate the bigger value and at the end of a cycle you
45  * put it at the end of the array.
46  * This allow to have an  $N \log_2(N)$  complexity, because when you put a value at
47  * the end the algorithm don't analyse it again.
48  * The result could be compare to heapsort from the <stdlib.h> library.
49  */
50 void myheapsort(float *tab, int n){
51     int i;
52     int boucle = 0;
53     float tmp;
54     while (boucle < n - 1){
55         i = 0;
56         // Try to isolate the biggest value at the start
57         while (i < n - boucle){
58             if (tab[i] > tab[(i - 1) / 2]){
59                 tmp = tab[i];
60                 tab[i] = tab[(i - 1) / 2];
61                 tab[(i - 1) / 2] = tmp;
62                 i = 0;
63                 continue;
64             }
65             i++;
66         }
67         // We put the biggest value at the end
68         tmp = tab[0];
69         tab[0] = tab[n - boucle - 1];
70         tab[n - boucle - 1] = tmp;
71         boucle++;
72     }
73 }
74
75 /*
76  * Test heap sort algorithm with an empty tab
77  */
78 void testHeapSortEmpty(){
79     float tab[0] = {};
80
81     printTab(tab, 0);
82     myheapsort(tab, 0);
83     printTab(tab, 0);

```

```

82 }
83
84 /*
85  * Test heap sort algorithm with one element
86  */
87 void testHeapSortOne(){
88     float tab[1] = {8.0};
89
90     printTab(tab, 1);
91     myheapsort(tab, 1);
92     printTab(tab, 1);
93 }
94
95 /*
96  * Test heap sort algorithm with two elements
97  */
98 void testHeapSortTwo(){
99     float tab[2] = {8.0, -16.0};
100
101     printTab(tab, 2);
102     myheapsort(tab, 2);
103     printTab(tab, 2);
104 }
105
106 /*
107  * Test heap sort algorithm with duplicated elements
108  */
109 void testHeapSortDuplicates(){
110     float tab[5] = {2.0, 5.5, 5.5, -3.0, 2.0};
111
112     printTab(tab, 5);
113     myheapsort(tab, 5);
114     printTab(tab, 5);
115 }
116
117 /*
118  * Test heap sort algorithm with only negative values
119  */
120 void testHeapSortNegative(){
121     float tab[5] = {-1.2, -0.6, -2.5, -3.3, -6.1};
122
123     printTab(tab, 5);
124     myheapsort(tab, 5);
125     printTab(tab, 5);
126 }
127
128 /*
129  * Test heap sort algorithm with positive values
130  */
131 void testHeapSortPositive(){
132     float tab[5] = {1.2, 0.6, 2.5, 3.3, 6.1};
133
134     printTab(tab, 5);
135     myheapsort(tab, 5);
136     printTab(tab, 5);
137 }
138
139 /*
140  * Test heap sort algorithm with bounds values
141  */
142 void testHeapSortBounds(){

```

```

143     float tab[4] = {FLT_MAX, -FLT_MAX, -FLT_MIN, FLT_MIN};
144     printTabExponent(tab, 4);
145     myheapsort(tab, 4);
146     printTabExponent(tab, 4);
147 }
148
149 /*
150  * Test heap sort algorithm with already sorted values
151  */
152 void testHeapSortSorted(){
153     float tab[5] = {-5.0, -2.3, 0.0, 5.0, 36.0};
154     printTab(tab, 5);
155     myheapsort(tab, 5);
156     printTab(tab, 5);
157 }
158
159 int main(){
160     testHeapSortEmpty();
161     testHeapSortOne();
162     testHeapSortTwo();
163     testHeapSortDuplicates();
164     testHeapSortNegative();
165     testHeapSortPositive();
166     testHeapSortBounds();
167     testHeapSortSorted();
168 }

```

Extrait de code 16 – Code complet du tri par tas d'Étienne

Remarque

L'ensemble du code est disponible sur le dépôt public situé sur le lien suivant : [ici](#).