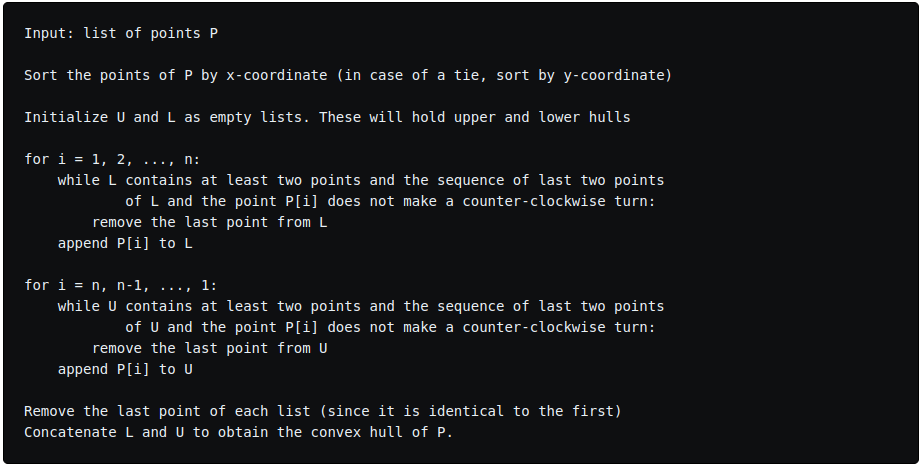
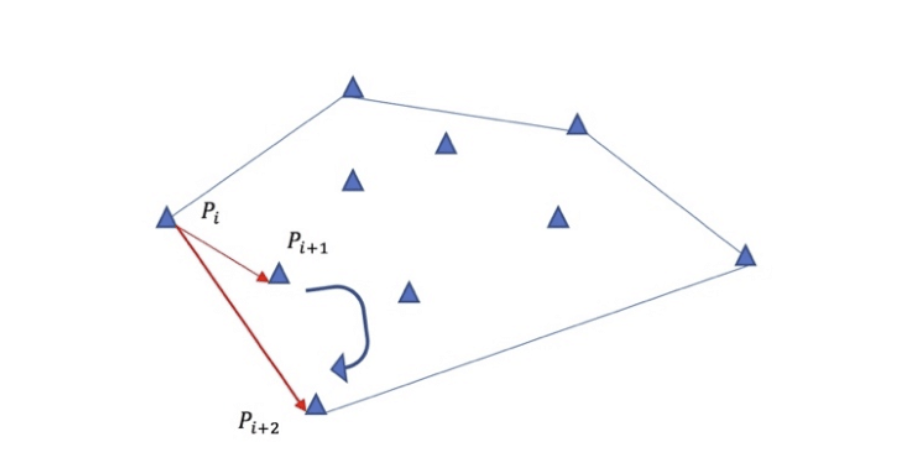
1. Monotone chain algorithms - Andrew algorithms:

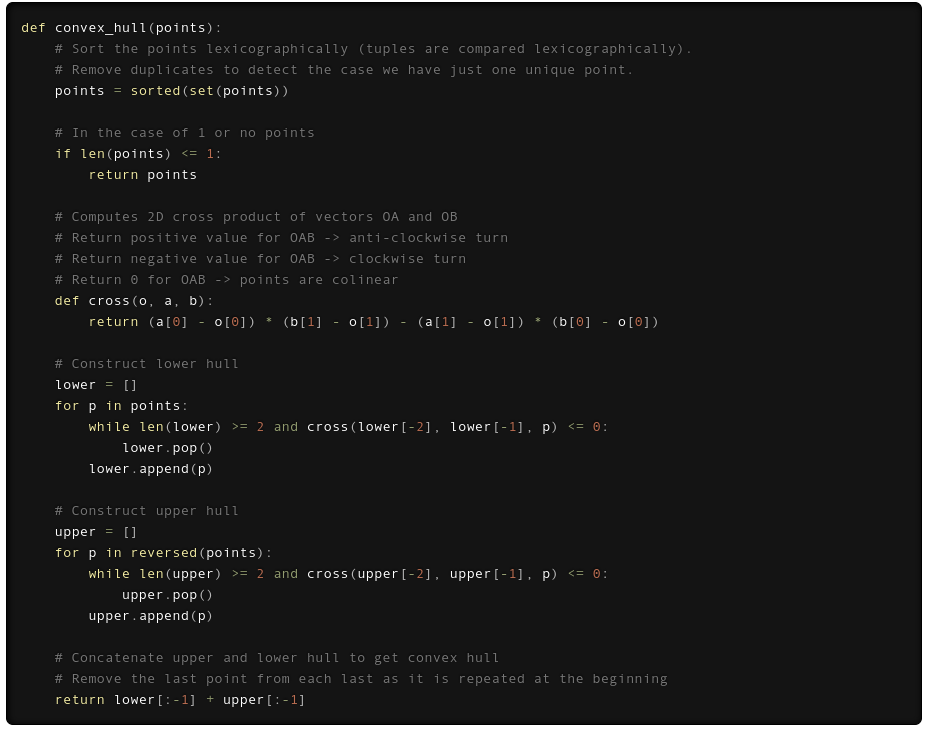
Sắp xếp các điểm theo tọa độ x của chúng (y nếu có điểm ngang nhau trên mặt phẳng x).

* Sau đó, lớp vỏ trên và dưới được tính toán trong thời gian O(n).
* Tìm điểm cực trái, xoay theo chiều kim đồng hồ để tìm điểm tiếp theo và lặp lại cho đến khi tìm thấy điểm cực phải. Xoay lại theo chiều kim đồng hồ để tìm thân tàu dưới.

Trong mã giả, bạn có thể thấy điều này như bên dưới:







Đầu tiên, chúng ta sắp xếp các điểm đầu vào theo thứ tự từ điển và thêm một trường hợp ngoại lệ khi có 1 hoặc không có điểm nào được nhập vào. Sau đó, để quyết định cách rẽ, chúng ta tìm tích có hướng của các vectơ giữa hai điểm, nếu dương, rẽ ngược chiều kim đồng hồ và âm, rẽ theo chiều kim đồng hồ. Nếu giá trị này là 0, chúng ta có thể nêu các điểm là đồng tuyến tính. Tiếp theo, chúng ta sẽ định nghĩa và tính toán các vỏ trên và dưới của các điểm (một nửa của vỏ lồi). Sau đó nối cả hai lại để trả về vỏ lồi cuối cùng.

Ví dụ, vỏ lồi của lưới 10x10 sẽ là (0,0), (9,0), (9,9), (0,9).

Vỏ lồi của [(0,3), (2,2), (1,1), (2,1), (3,0), (0,0), (3,3)] sẽ là (0,0), (3,0), (3,3), (0,3).

Code:

*// Implementation of Andrew's monotone chain 2D convex hull algorithm.*

*// Asymptotic complexity: O(n log n).*

*// Practical performance: 0.5-1.0 seconds for n=1000000 on a 1GHz machine.*

#include *<algorithm>*

#include *<vector>*

**using** **namespace** **std**;

**typedef** double coord\_t; *// coordinate type*

**typedef** double coord2\_t; *// must be big enough to hold 2\*max(|coordinate|)^2*

**struct** **Point** {

coord\_t x, y;

bool **operator** <(**const** Point &p) **const** {

**return** x < p.x || (x == p.x && y < p.y);

}

};

*// 3D cross product of OA and OB vectors, (i.e z-component of their "2D" cross product, but remember that it is not defined in "2D").// Returns a positive value, if OAB makes a counter-clockwise turn,// negative for clockwise turn, and zero if the points are collinear.*

coord2\_t cross(**const** Point &O, **const** Point &A, **const** Point &B){

**return** (A.x - O.x) \* (B.y - O.y) - (A.y - O.y) \* (B.x - O.x);

}

*// Returns a list of points on the convex hull in counter-clockwise order.*

*// Note: the last point in the returned list is the same as the first one.*

vector<Point> convex\_hull(vector<Point> P){

size\_t n = P.size(), k = 0;

**if** (n <= 3)

**return** P;

vector<Point> H(2\*n);

*// Sort points lexicographically*

sort(P.begin(), P.end());

*// Build lower hull*

**for** (size\_t i = 0; i < n; ++i) {

**while** (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0)

k--;

H[k++] = P[i];

}

*// Build upper hull*

**for** (size\_t i = n-1, t = k+1; i > 0; --i) {

**while** (k >= t && cross(H[k-2], H[k-1], P[i-1]) <= 0)

k--;

H[k++] = P[i-1];

}

H.resize(k-1);

**return** H;

}

1. Convex Hull use Jarvis algorithms or Wrapping:

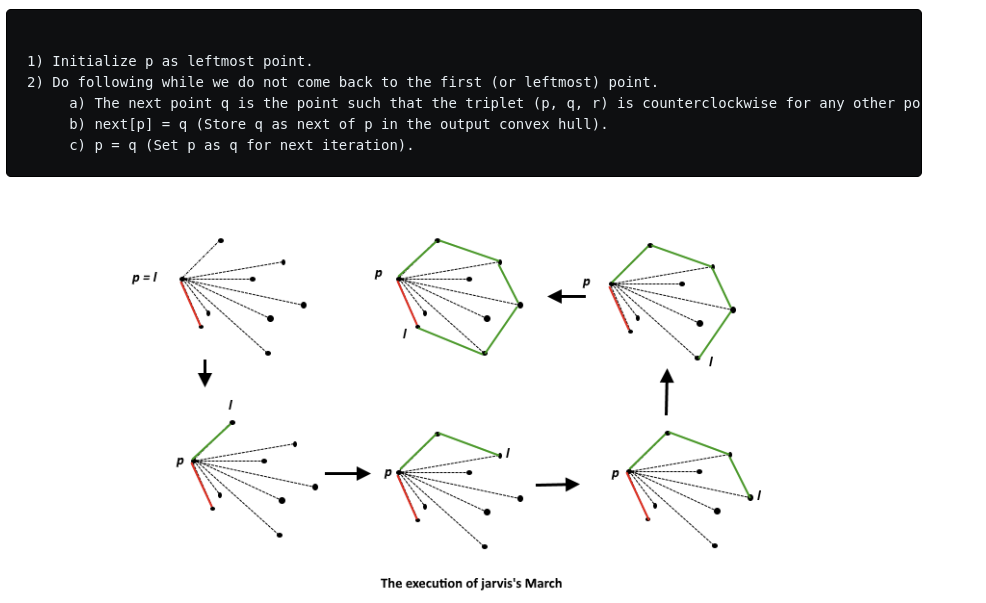
Ý tưởng của thuật toán Jarvis rất đơn giản,

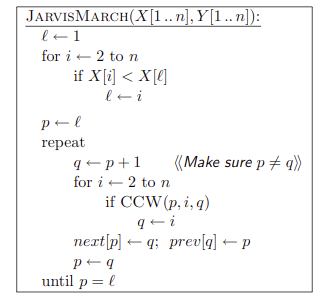
Chúng ta bắt đầu từ điểm cực trái (hoặc điểm có giá trị tọa độ x nhỏ nhất) và tiếp tục bao quanh các điểm theo hướng ngược chiều kim đồng hồ.

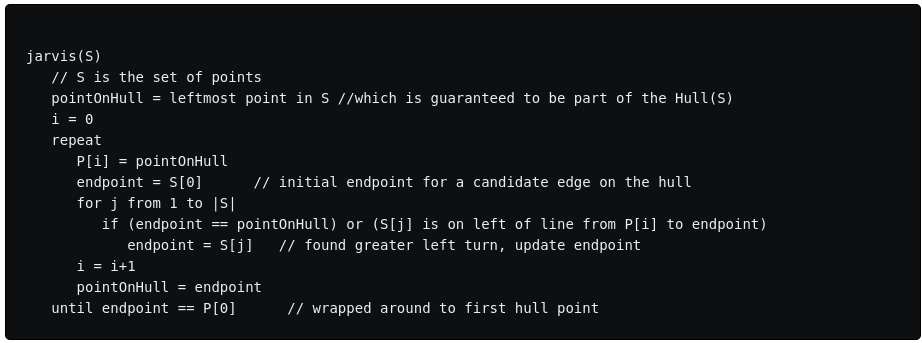
Câu hỏi lớn là, ****với điểm p là điểm hiện tại, làm thế nào để tìm điểm tiếp theo trong đầu ra?****

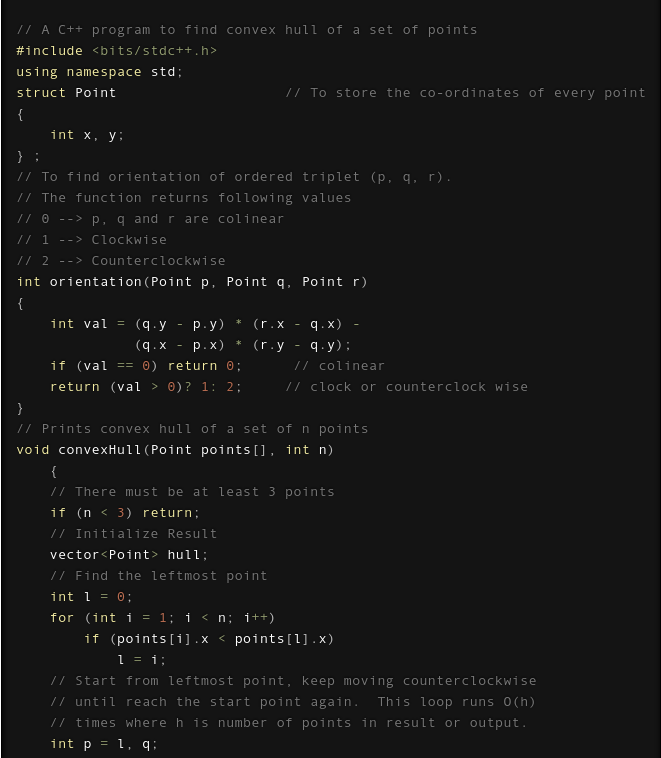
Ý tưởng là sử dụng orientation() ở đây. Điểm tiếp theo được chọn là điểm đánh bại tất cả các điểm khác theo hướng ngược chiều kim đồng hồ, tức là điểm tiếp theo là q nếu đối với bất kỳ điểm r nào khác, chúng ta có “hướng(p, r, q) = ngược chiều kim đồng hồ”.

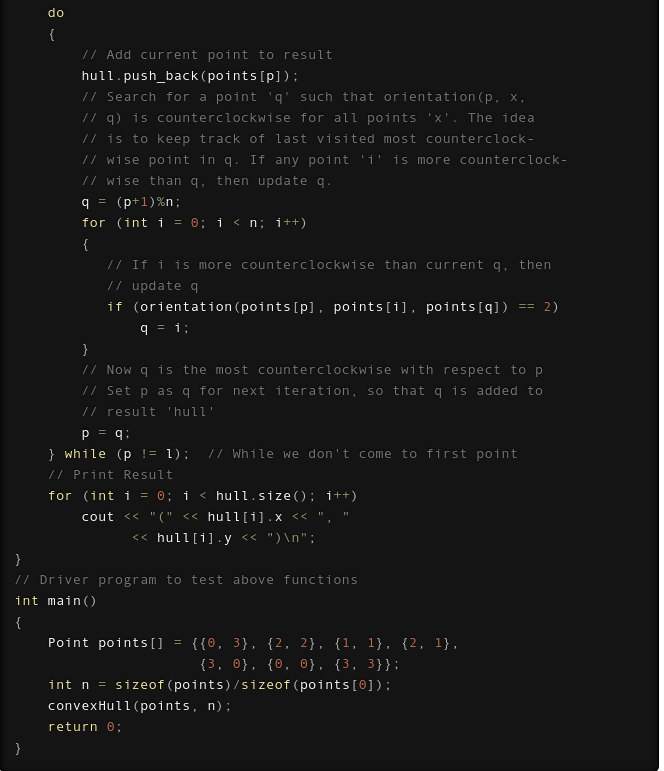
Sau đây là thuật toán chi tiết.











Complexity:

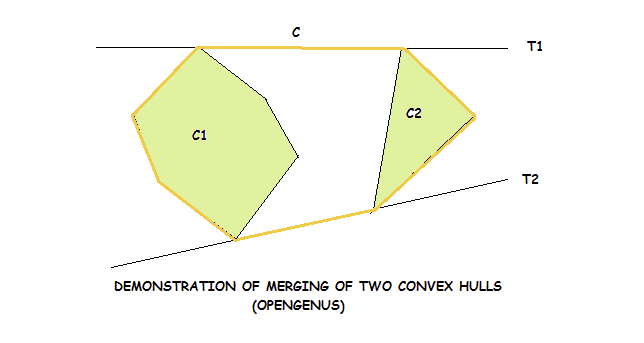
* Độ phức tạp thời gian trong trường hợp xấu nhất:****Θ(N ^ 2)****
* Độ phức tạp thời gian trường hợp trung bình:****Θ(N log N)****
* Độ phức tạp thời gian trường hợp tốt nhất:****Θ(N log N)****
* Độ phức tạp của không gian:****Θ(N)****

1. Divide and conquer algorithms:

Vậy thì vấn đề bây giờ là phải hợp nhất hai vỏ lồi nàyC1VàC2và xác định vỏ lồiCcho bộ hoàn chỉnhS.  
Điều này có thể thực hiện được bằng cách tìm ****tiếp tuyến trên và dưới của các vỏ lồi trái và phải C1 và C2**** .

Hãy để vỏ lồi bên trái làC1và vỏ lồi bên phải làC2. Sau đó các tiếp tuyến dưới và trên được đặt tên làT1VàT2tương ứng, như thể hiện trong hình.

Sau đó, đường viền màu đỏ cho thấy lớp vỏ lồi cuối cùng.



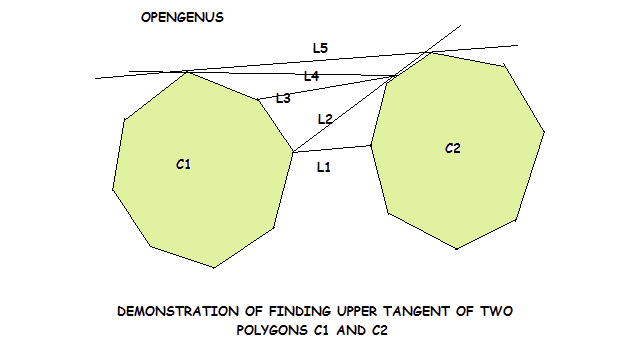
Bây giờ đệ quy xuất hiện, chúng ta chia tập hợp các điểm cho đến khi số điểm trong tập hợp rất nhỏ, chẳng hạn như 5, và chúng ta có thể tìm thấy vỏ lồi cho các điểm này bằngthuật toán vũ phu. Việc hợp nhất các nửa này sẽ tạo ra lớp vỏ lồi cho tập hợp điểm hoàn chỉnh.

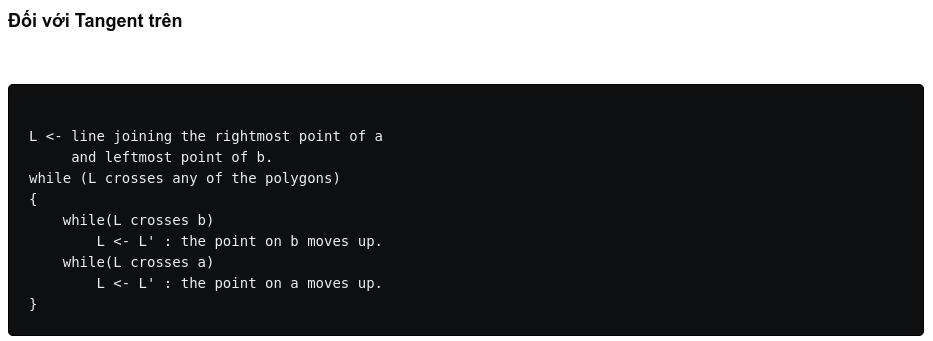
**TIẾP TUYẾN GIỮA 2 ĐA GIÁC LỒI**

Để tìm tiếp tuyến trên, chúng ta bắt đầu bằng cách lấy hai điểm.  
Điểm cực phải (ví dụMỘT) của vỏ lồi trái C1 và điểm cực trái (ví dụB) của vỏ lồi phải C2. Đường thẳng nối chúng được dán nhãn làL1.

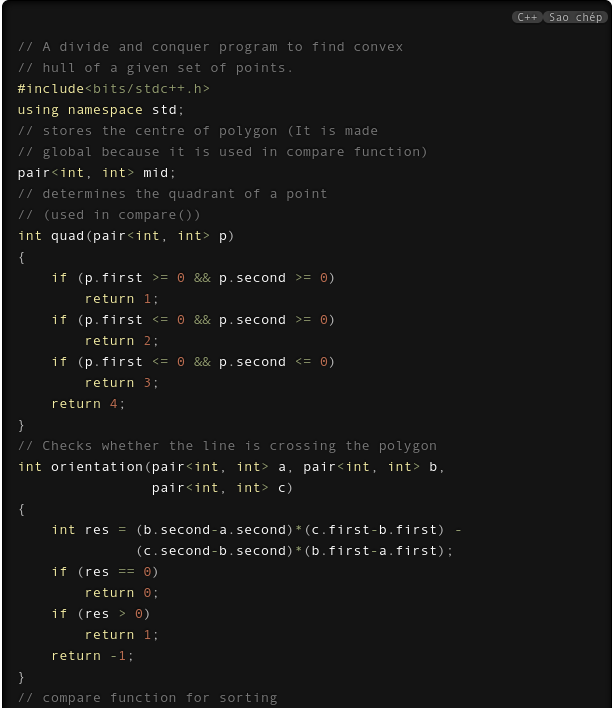
Khi đường thẳng này đi qua đa giácC2(không nằm trên đa giác b) vì vậy chúng ta lấy điểm tiếp theo ngược chiều kim đồng hồ trênC2, đường thẳng được dán nhãn là 2. Bây giờ đường thẳng nằm phía trên đa giácC2, tốt! Nhưng đường thẳng đang cắt qua đa giácC1, vì vậy chúng ta di chuyển đến điểm tiếp theo theo chiều kim đồng hồ, được dán nhãn là 3 trong hình. Điểm này lại cắt đa giác a nên chúng ta di chuyển đến đường 4. Đường này cắt b nên chúng ta di chuyển đến đường 5. Bây giờ đường này không cắt bất kỳ điểm nào trong hai điểm. Vì vậy, đây là tiếp tuyến trên của các đa giác đã cho.

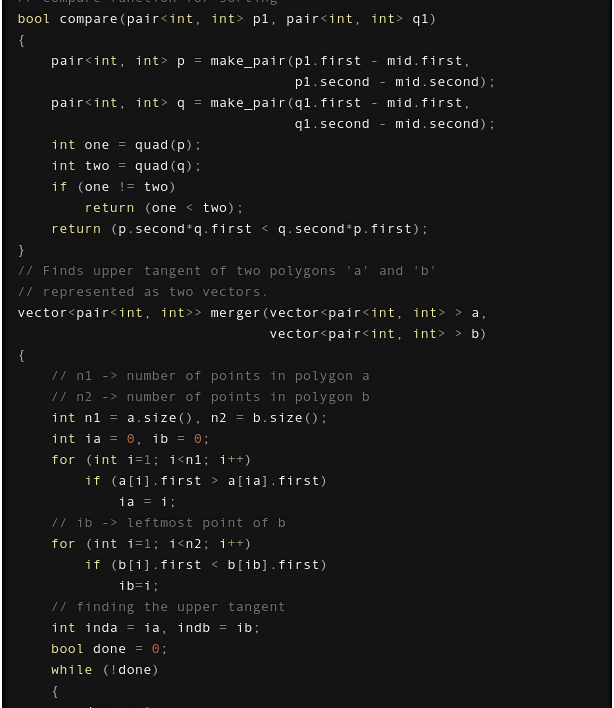
Để tìm tiếp tuyến dưới, chúng ta cần di chuyển ngược qua các đa giác, tức là nếu đường thẳng cắt đa giácC2chúng ta di chuyển theo chiều kim đồng hồ tiếp theo và ngược chiều kim đồng hồ tiếp theo nếu đường thẳng cắt đa giácC1.

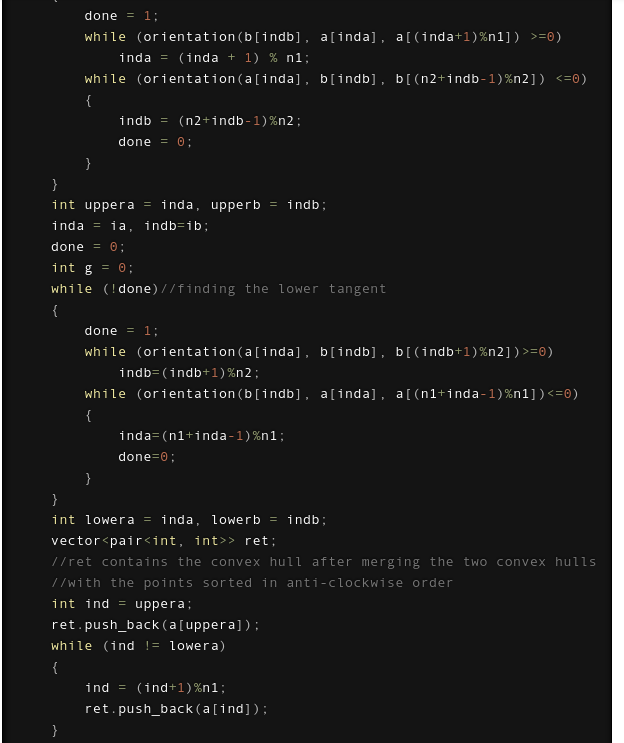


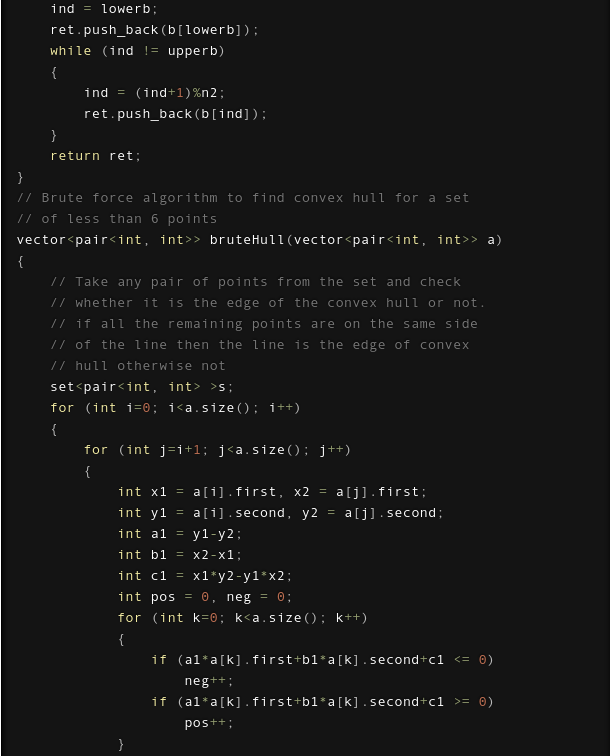


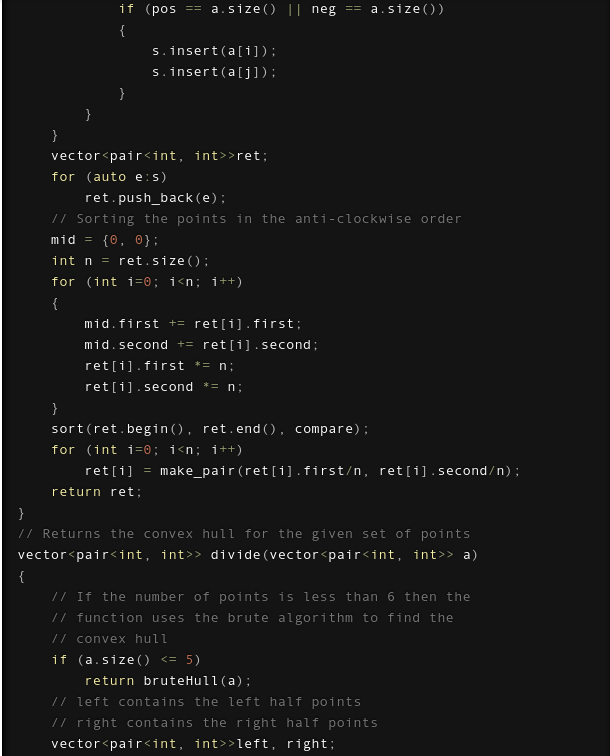


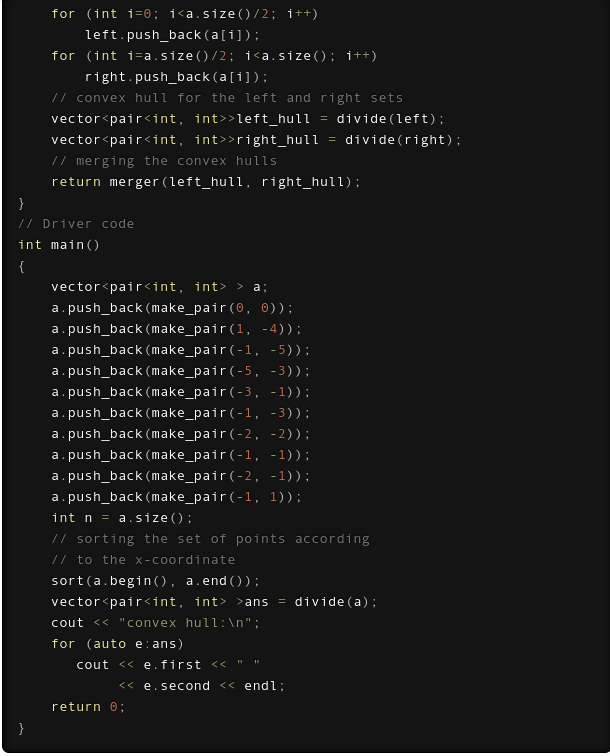












Complexity:

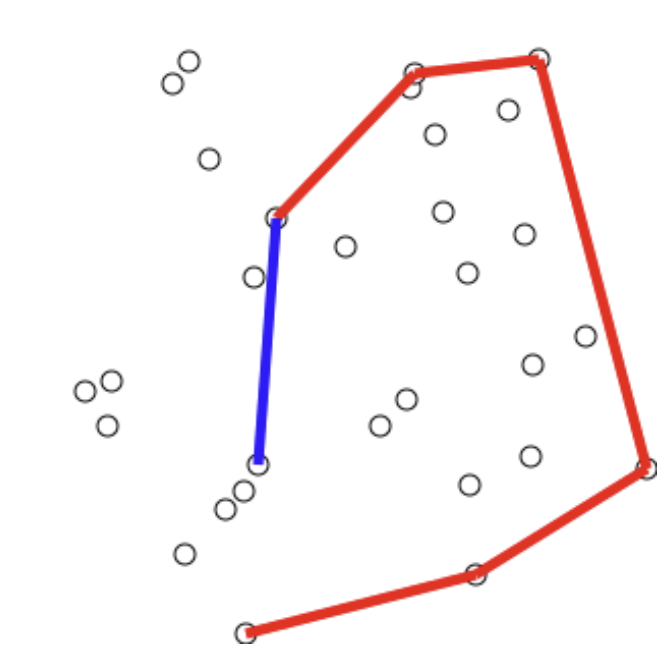
Sự hợp nhất của các vỏ lồi bên trái và bên phải diễn ra****TRÊN)****thời gian và vì chúng ta chia các điểm thành hai phần bằng nhau nên độ phức tạp về thời gian của thuật toán trên là****O(n \* log n)****.

* Độ phức tạp thời gian trong trường hợp xấu nhất:****Θ(N log N)****
* Độ phức tạp thời gian trường hợp trung bình:****Θ(N log N)****
* Độ phức tạp thời gian trường hợp tốt nhất:****Θ(N log N)****
* Độ phức tạp của không gian:****Θ(N)****

1. **Graham algorithms - scanning:**

****Thuật toán quét Graham**** là một thuật toán hiệu quả để tìm vỏ lồi của một tập hợp hữu hạn các điểm trên mặt phẳng với độ phức tạp thời gian O(N log N). Thuật toán tìm tất cả các đỉnh của vỏ lồi được sắp xếp theo ranh giới của nó. ****Thuật toán sử dụng một ngăn xếp để phát hiện và loại bỏ các phần lõm trong ranh giới một cách hiệu quả**** .

Thuật toán này được đặt theo tên của ****Ronald Graham**** , người đã công bố thuật toán gốc vào năm 1972.



Algorithms:

Trong thuật toán Jarvis cho Convex Hull. Độ phức tạp thời gian trường hợp xấu nhất của thuật toán Jarvis là O(n^2). Sử dụng thuật toán quét Graham, chúng ta có thể tìm ra Convex Hull trong thời gian O(nLogn). Sau đây là thuật toán Graham

Giả sử points[0..n-1] là mảng đầu vào.

Tìm điểm dưới cùng bằng cách so sánh tọa độ y của tất cả các điểm. Nếu có hai điểm có cùng giá trị y, thì điểm có giá trị tọa độ x nhỏ hơn sẽ được xem xét. Giả sử điểm dưới cùng là P0. Đặt P0 ở vị trí đầu tiên trong vỏ đầu ra.

Xét n-1 điểm còn lại và sắp xếp chúng theo góc polor theo thứ tự ngược chiều kim đồng hồ xung quanh các điểm[0]. Nếu góc polor của hai điểm bằng nhau, thì đặt điểm gần nhất lên trước.

Sau khi sắp xếp, kiểm tra xem hai hoặc nhiều điểm có cùng góc không. Nếu hai điểm nữa có cùng góc, thì xóa tất cả các điểm có cùng góc ngoại trừ điểm xa nhất so với P0. Giả sử kích thước của mảng mới là m.

Nếu m nhỏ hơn 3, hãy trả về (Convex Hull không khả thi)

Tạo một ngăn xếp rỗng 'S' và đẩy các điểm [0], điểm [1] và điểm [2] vào S.

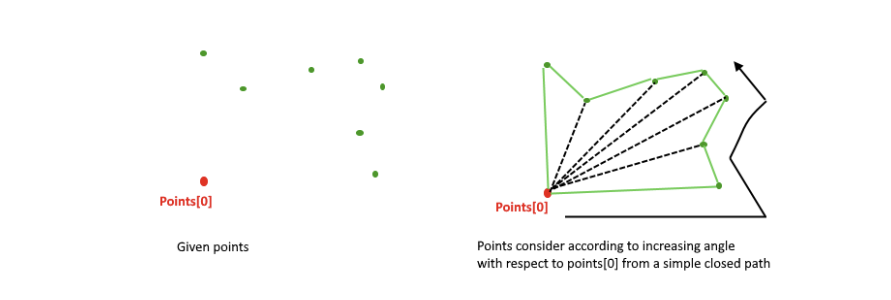
Xử lý từng điểm m-3 còn lại. Thực hiện các bước sau cho mọi điểm 'points[i]'  
4.1) Tiếp tục xóa các điểm khỏi ngăn xếp trong khi hướng của 3 điểm sau không ngược chiều kim đồng hồ (hoặc chúng không rẽ trái).  
a) Điểm bên cạnh đỉnh trong ngăn xếp  
b) Điểm ở đỉnh ngăn xếp  
c) points[i]  
4.2) Đẩy các điểm[i] vào S

In nội dung của S

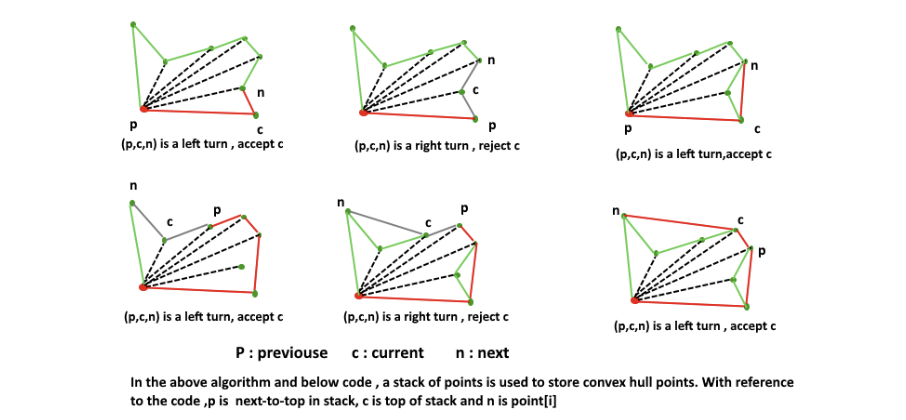
Thuật toán trên có thể được chia thành hai giai đoạn.

****Giai đoạn 1**** (Sắp xếp điểm): Đầu tiên chúng ta tìm điểm dưới cùng. Ý tưởng là xử lý trước các điểm bằng cách sắp xếp chúng theo điểm dưới cùng. Sau khi các điểm được sắp xếp, chúng tạo thành một đường dẫn khép kín đơn giản (Xem sơ đồ sau).

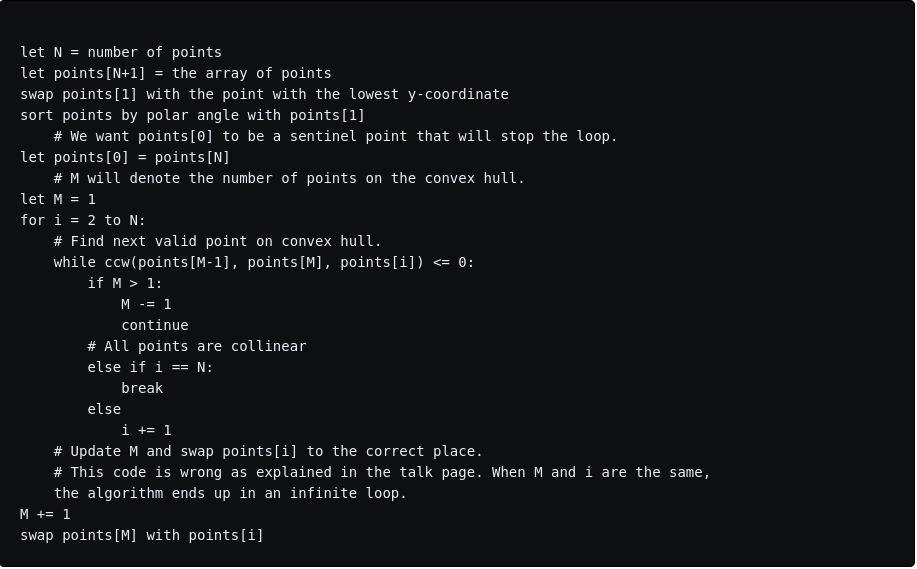
Tiêu chuẩn sắp xếp nên là gì? tính toán các góc thực tế sẽ không hiệu quả vì các hàm lượng giác không dễ để đánh giá. Ý tưởng là sử dụng hướng để so sánh các góc mà không thực sự tính toán chúng (Xem hàm compare() bên dưới)



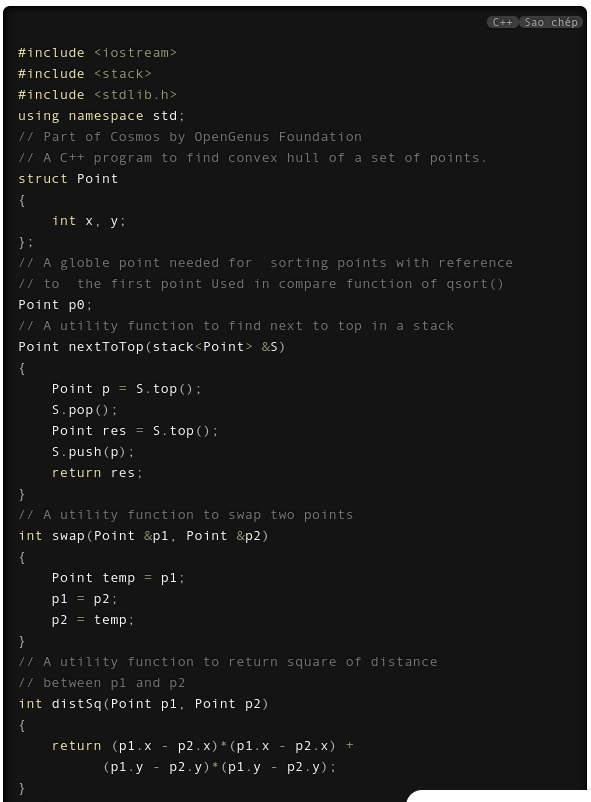
****Giai đoạn 2**** (Chấp nhận hoặc Từ chối Điểm): Khi đã có đường đi khép kín, bước tiếp theo là đi qua đường đi và loại bỏ các điểm lõm trên đường đi này. Làm thế nào để quyết định điểm nào cần loại bỏ và điểm nào cần giữ lại? Một lần nữa, định hướng sẽ giúp ích ở đây. Hai điểm đầu tiên trong mảng được sắp xếp luôn là một phần của Convex Hull. Đối với các điểm còn lại, chúng ta theo dõi ba điểm gần đây và tìm góc tạo bởi chúng. Giả sử ba điểm là prev(p), curr(c) và next(n). Nếu hướng của các điểm này (xét theo cùng thứ tự) không ngược chiều kim đồng hồ, chúng ta loại bỏ c, nếu không, chúng ta giữ nguyên. Sơ đồ sau đây cho thấy quy trình từng bước của giai đoạn này

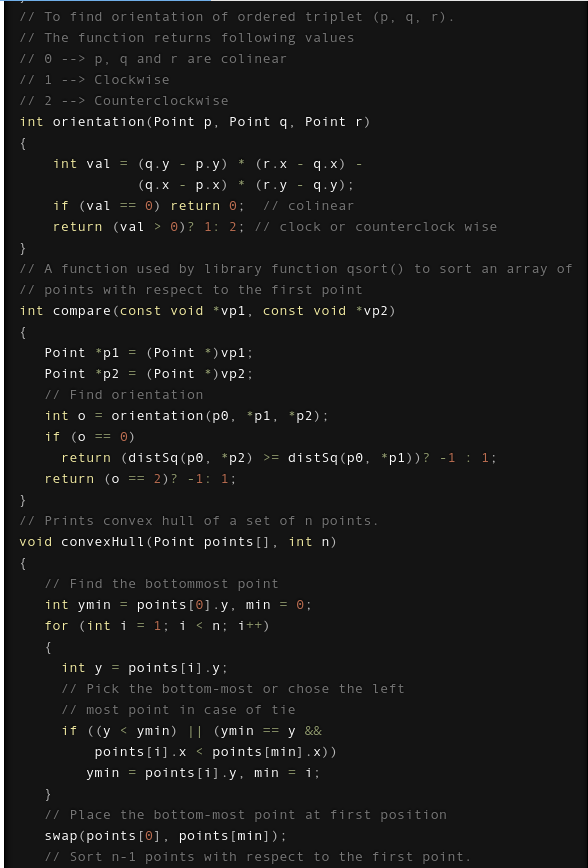


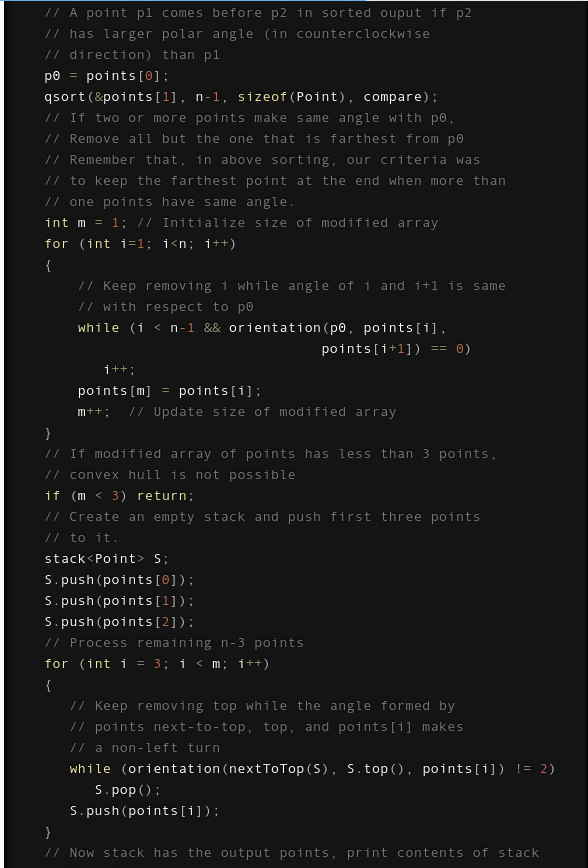
Pseudo Code:

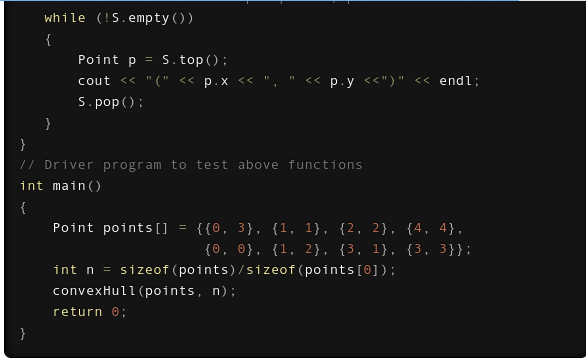


Code:









Complexity:

* Độ phức tạp thời gian trong trường hợp xấu nhất:****Θ(N log N)****
* Độ phức tạp thời gian trường hợp trung bình:****Θ(N log N)****
* Độ phức tạp thời gian trường hợp tốt nhất:****Θ(N log N)****
* Độ phức tạp của không gian:****Θ(N)****

Thuật toán mất thời gian O(nLogn) nếu chúng ta sử dụng thuật toán sắp xếp O(nLogn).

Bước đầu tiên (tìm điểm dưới cùng) mất thời gian O(n). Bước thứ hai (sắp xếp các điểm) mất thời gian O(nLogn). Bước thứ ba mất thời gian O(n). Ở bước thứ ba, mỗi phần tử được đẩy và bật ra nhiều nhất một lần. Vì vậy, bước thứ sáu để xử lý từng điểm một mất thời gian O(n), giả sử rằng các hoạt động ngăn xếp mất thời gian O(1). Độ phức tạp tổng thể là O(n) + O(nLogn) + O(n) + O(n) tức là O(nLogn)

1. Chan algorithms: