Câu 1:

Cho các điểm: (4,5), (6,4), (7,6), (8,7), (9,8), (5,10), (4,9), (5,9), (8,5), (8,11)  
a) Mô tả thuật toán xác định bao lồi, tìm bao lồi đấy

|  |
| --- |
| Thuật toán:  1. Chọn điểm **p** là điểm ở bên trái nhất 2. Chạy vòng lặp while cho tới khi gặp lại điểm **p**:  - chọn **q** là điểm tiếp theo, duyệt qua lại tiếp các điểm  + với mỗi điểm i, nếu i có khuynh hướng ngược chiều kim đồng hồ hơn, gán i là q  + gán điểm tiếp theo sau **p** là **q (p.next = q)**  + gán **p** = **q** 3. Output sẽ là các điểm tạo nên bao lồi  **Bước 1: Tìm Điểm Dưới Nhất**   * Xác định điểm có tọa độ y nhỏ nhất (hoặc một trong những điểm có y nhỏ nhất nếu có nhiều điểm). * Trong trường hợp này, điểm (4, 5) có y nhỏ nhất.   **Bước 2: Sắp xếp Điểm Theo Góc (Tăng dần)**   * Từ điểm dưới nhất, vẽ các tia hướng đến các điểm còn lại. * Sắp xếp các điểm theo góc tạo bởi tia và trục Ox (theo chiều kim đồng hồ), bắt đầu từ 0 độ. * Sử dụng công thức tính góc: góc = atan2(y - y\_min, x - x\_min) với (x\_min, y\_min) là điểm dưới nhất. * Sau khi sắp xếp, ta được danh sách các điểm theo thứ tự: (4, 5), (5, 9), (5, 10), (6, 4), (7, 6), (8, 5), (8, 7), (8, 11), (9, 8).   **Bước 3: Xác định Bao Lồi**   * Khởi tạo danh sách bao lồi rỗng L. * Duyệt qua danh sách các điểm đã sắp xếp:   + Thêm điểm đầu tiên vào L.   + Với mỗi điểm tiếp theo:     - Kiểm tra xem điểm hiện tại có nằm trong góc tạo bởi hai điểm cuối cùng trong L hay không.       * Nếu nằm trong góc, bỏ qua điểm hiện tại.       * Nếu không nằm trong góc, loại bỏ điểm cuối cùng trong L và thêm điểm hiện tại vào L. * Lặp lại bước 3 cho đến khi duyệt qua hết danh sách các điểm.   **Bước 4: Kết quả**  - Bao lồi gồm các điểm: (4, 5), (4, 9), (5, 10), (8, 10), (9, 8), (8, 5), (6,4). |

## b) Viết chương trình sử dụng hàm xác định bao lồi và tìm diện tích bao lồi bằng ít nhất 2 phương pháp: - Graham Scan

|  |
| --- |
| //graham scan  #include <bits/stdc++.h>  #include <iostream>  #include <stack>  #include <stdlib.h>  using namespace std;  struct Point  {  int x, y;  };  // A global point needed for sorting points with reference  // to the first point Used in compare function of qsort()  Point p0;  // A utility function to find next to top in a stack  Point nextToTop(stack<Point> &S)  {  Point p = S.top();  S.pop();  Point res = S.top();  S.push(p);  return res;  }  // A utility function to swap two points  void swap(Point &p1, Point &p2)  {  Point temp = p1;  p1 = p2;  p2 = temp;  }  // A utility function to return square of distance  // between p1 and p2  int distSq(Point p1, Point p2)  {  return (p1.x - p2.x)\*(p1.x - p2.x) +  (p1.y - p2.y)\*(p1.y - p2.y);  }  // To find orientation of ordered triplet (p, q, r).  // The function returns following values  // 0 --> p, q and r are collinear  // 1 --> Clockwise  // 2 --> Counterclockwise  int orientation(Point p, Point q, Point r)  {  int val = (q.y - p.y) \* (r.x - q.x) -  (q.x - p.x) \* (r.y - q.y);  if (val == 0) return 0; // collinear  return (val > 0)? 1: 2; // clock or counterclock wise  }  // A function used by library function qsort() to sort an array of  // points with respect to the first point  int compare(const void \*vp1, const void \*vp2)  {  Point \*p1 = (Point \*)vp1;  Point \*p2 = (Point \*)vp2;  // Find orientation  int o = orientation(p0, \*p1, \*p2);  if (o == 0)  return (distSq(p0, \*p2) >= distSq(p0, \*p1))? -1 : 1;  return (o == 2)? -1: 1;  }  // Prints convex hull of a set of n points.  vector<Point> convexHull(Point points[], int n)  {  vector<Point> hello;  // Find the bottommost point  int ymin = points[0].y, min = 0;  for (int i = 1; i < n; i++)  {  int y = points[i].y;  // Pick the bottom-most or choose the left  // most point in case of tie  if ((y < ymin) || (ymin == y &&  points[i].x < points[min].x))  ymin = points[i].y, min = i;  }  // Place the bottom-most point at first position  swap(points[0], points[min]);  // Sort n-1 points with respect to the first point.  // A point p1 comes before p2 in sorted output if p2  // has larger polar angle (in counterclockwise  // direction) than p1  p0 = points[0];  qsort(&points[1], n-1, sizeof(Point), compare);  // If two or more points make same angle with p0,  // Remove all but the one that is farthest from p0  // Remember that, in above sorting, our criteria was  // to keep the farthest point at the end when more than  // one points have same angle.  int m = 1; // Initialize size of modified array  for (int i=1; i<n; i++)  {  // Keep removing i while angle of i and i+1 is same  // with respect to p0  while (i < n-1 && orientation(p0, points[i],  points[i+1]) == 0)  i++;  points[m] = points[i];  m++; // Update size of modified array  }  // If modified array of points has less than 3 points,  // convex hull is not possible  if (m < 3) return hello;  // Create an empty stack and push first three points  // to it.  stack<Point> S;  S.push(points[0]);  S.push(points[1]);  S.push(points[2]);  // Process remaining n-3 points  for (int i = 3; i < m; i++)  {  // Keep removing top while the angle formed by  // points next-to-top, top, and points[i] makes  // a non-left turn  while (S.size()>1 && orientation(nextToTop(S), S.top(), points[i]) != 2)  S.pop();  S.push(points[i]);  }  // Now stack has the output points, print contents of stack    while (!S.empty())  {  Point p = S.top();  hello.push\_back(p);  S.pop();  }  return hello;  }  double polygonArea(vector<Point> ans, int n)  {  double S = 0;  for (int i = 0; i < n - 1; i++)  {  S += (ans.at(i).x\* ans.at(i+1).y) - (ans.at(i+1).x\*ans.at(i).y);  }  double A = 0.5\* fabs(S);  // S = (1/2)\*fabs(S);  return A;  }  void printVector(vector<Point> a)  {  cout<<"\n---------After Using Graham Scan Algorithm---------------\n";  cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CONVEX HULL \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  for (int i = 0; i < a.size(); i++)  cout << "(" << a[i].x << ", "<< a[i].y << ")\n";  }  // Driver program to test above functions  int main()  {  Point points[] = {{4, 5}, {6, 4}, {7,6}, {8, 7},  {9, 8}, {5, 10}, {4, 9}, {5, 9}, {8, 5}, {8, 10}};  int T= sizeof(points) / sizeof(points[0]);  // cout << "enter number of points: ";  // cin>>T;  // if(T<=0)  // return -1;  // Point points[T];  // for(int i=0;i<T;++i){  // cout << "Enter x cor of " << i << " point: ";  // cin >> x;  // points[i].x=x;  // cout << "Enter y cor of " << i << " point: ";  // cin >> y;  // points[i].y=y;  // }    vector<Point> ans = convexHull(points, T);  printVector(ans);  cout << "Polygon area: " << polygonArea(ans, ans.size());  return 0;  } |
| Kết quả: |

- Jarvis

|  |
| --- |
| //Jarvis  #include <bits/stdc++.h>  using namespace std;  #define ll long long  struct Point  {  ll x, y;  // ostream& operator<<(ostream& os, const Point dt)  // {  // os << "(" <<dt.x << ' - ' << dt.y << ")" << "\n";  // return os;  // }  };  // To find orientation of ordered triplet (p, q, r).  // The function returns following values  // 0 --> p, q and r are collinear  // 1 --> Clockwise  // 2 --> Counterclockwise  int orientation(Point p, Point q, Point r)  {  int val = (q.y - p.y) \* (r.x - q.x) -  (q.x - p.x) \* (r.y - q.y);  if (val == 0) return 0; // collinear  return (val > 0)? 1: 2; // clock or counterclock wise  }  // Prints convex hull of a set of n points.  vector<Point> convexHull(Point points[], int n)  {  // There must be at least 3 points  // Initialize Result  vector<Point> hull;  // Find the leftmost point  int l = 0;  for (int i = 1; i < n; i++)  if (points[i].x < points[l].x)  l = i;  // Start from leftmost point, keep moving counterclockwise  // until reach the start point again. This loop runs O(h)  // times where h is number of points in result or output.  int p = l, q;  do  {  // Add current point to result  hull.push\_back(points[p]);  // Search for a point 'q' such that orientation(p, q,  // x) is counterclockwise for all points 'x'. The idea  // is to keep track of last visited most counterclock-  // wise point in q. If any point 'i' is more counterclock-  // wise than q, then update q.  q = (p+1)%n;  for (int i = 0; i < n; i++)  {  // If i is more counterclockwise than current q, then  // update q  if (orientation(points[p], points[i], points[q]) == 2)  q = i;  }  // Now q is the most counterclockwise with respect to p  // Set p as q for next iteration, so that q is added to  // result 'hull'  p = q;  } while (p != l); // While we don't come to first point  // Print Result  // for (int i = 0; i < hull.size(); i++)  // cout << "(" << hull[i].x << ", "  // << hull[i].y << ")\n";  return hull;  }  void printVector(vector<Point> a)  {  cout << "Convex Hull: " << "\n";  for (int i = 0; i < a.size(); i++)  cout << "(" << a[i].x << ", "<< a[i].y << ")\n";  }  double polygonArea(vector<Point> ans, int n)  {  double S = 0;  for (int i = 0; i < n - 1; i++)  {  S += (ans.at(i).x\* ans.at(i+1).y) - (ans.at(i+1).x\*ans.at(i).y);  }  double A = 0.5\*fabs(S);  // S = (1/2)\*fabs(S);  return A;  }  // Driver program to test above functions  int main()  {  Point points[] = {{4, 5}, {6, 4}, {7,6}, {8, 7},  {9, 8}, {5, 10}, {4, 9}, {5, 9}, {8, 5}, {8, 10}};  int T = sizeof(points) / sizeof(points[0]);  vector<Point> ans;  if (T>3){  ans = convexHull(points, T);  }  int n = ans.size();  // double S = polygonArea(ans);  printVector(ans);  double S = polygonArea(ans, n);  cout <<"polygon area: "<< S << "\n";  return 0;  } |
| Kết quả: |

## c) Viết chương trình sử dụng hàm xác định khoảng cách ngắn nhất giữa các đỉnh của bao lồi tìm được

|  |
| --- |
| // A C++ program to find convex hull of a set of points. Refer  // https://www.geeksforgeeks.org/orientation-3-ordered-points/  // for explanation of orientation()  //graham scan  #include <bits/stdc++.h>  #include <iostream>  #include <stack>  #include <stdlib.h>  using namespace std;  struct Point  {  int x, y;  };  // A global point needed for sorting points with reference  // to the first point Used in compare function of qsort()  Point p0;  // A utility function to find next to top in a stack  Point nextToTop(stack<Point> &S)  {  Point p = S.top();  S.pop();  Point res = S.top();  S.push(p);  return res;  }  // A utility function to swap two points  void swap(Point &p1, Point &p2)  {  Point temp = p1;  p1 = p2;  p2 = temp;  }  // A utility function to return square of distance  // between p1 and p2  int distSq(Point p1, Point p2)  {  return (p1.x - p2.x)\*(p1.x - p2.x) +  (p1.y - p2.y)\*(p1.y - p2.y);  }  // To find orientation of ordered triplet (p, q, r).  // The function returns following values  // 0 --> p, q and r are collinear  // 1 --> Clockwise  // 2 --> Counterclockwise  int orientation(Point p, Point q, Point r)  {  int val = (q.y - p.y) \* (r.x - q.x) -  (q.x - p.x) \* (r.y - q.y);  if (val == 0) return 0; // collinear  return (val > 0)? 1: 2; // clock or counterclock wise  }  // A function used by library function qsort() to sort an array of  // points with respect to the first point  int compare(const void \*vp1, const void \*vp2)  {  Point \*p1 = (Point \*)vp1;  Point \*p2 = (Point \*)vp2;  // Find orientation  int o = orientation(p0, \*p1, \*p2);  if (o == 0)  return (distSq(p0, \*p2) >= distSq(p0, \*p1))? -1 : 1;  return (o == 2)? -1: 1;  }  // Prints convex hull of a set of n points.  vector<Point> convexHull(Point points[], int n, vector<Point> \*inner)  {  vector<Point> hello;  // Find the bottommost point  int ymin = points[0].y, min = 0;  for (int i = 1; i < n; i++)  {  int y = points[i].y;  // Pick the bottom-most or choose the left  // most point in case of tie  if ((y < ymin) || (ymin == y &&  points[i].x < points[min].x))  ymin = points[i].y, min = i;  }  // Place the bottom-most point at first position  swap(points[0], points[min]);  // Sort n-1 points with respect to the first point.  // A point p1 comes before p2 in sorted output if p2  // has larger polar angle (in counterclockwise  // direction) than p1  p0 = points[0];  qsort(&points[1], n-1, sizeof(Point), compare);  // If two or more points make same angle with p0,  // Remove all but the one that is farthest from p0  // Remember that, in above sorting, our criteria was  // to keep the farthest point at the end when more than  // one points have same angle.  int m = 1; // Initialize size of modified array  for (int i=1; i<n; i++)  {  // Keep removing i while angle of i and i+1 is same  // with respect to p0  while (i < n-1 && orientation(p0, points[i],  points[i+1]) == 0)  i++;  points[m] = points[i];  m++; // Update size of modified array  }  // If modified array of points has less than 3 points,  // convex hull is not possible  if (m < 3) return hello;  // Create an empty stack and push first three points  // to it.  stack<Point> S;  S.push(points[0]);  S.push(points[1]);  S.push(points[2]);  // Process remaining n-3 points  for (int i = 3; i < m; i++)  {  // Keep removing top while the angle formed by  // points next-to-top, top, and points[i] makes  // a non-left turn  while (S.size()>1 && orientation(nextToTop(S), S.top(), points[i]) != 2)  {  inner->push\_back(S.top());  S.pop();  }  S.push(points[i]);  }  // Now stack has the output points, print contents of stack    while (!S.empty())  {  Point p = S.top();  hello.push\_back(p);  S.pop();  }  return hello;  }  double polygonArea(vector<Point> ans, int n)  {  double S = 0;  for (int i = 0; i < n - 1; i++)  {  S += (ans.at(i).x\* ans.at(i+1).y) - (ans.at(i+1).x\*ans.at(i).y);  }  double A = 0.5\* fabs(S);  // S = (1/2)\*fabs(S);  return A;  }  void printVector(vector<Point> a)  {    for (int i = 0; i < a.size(); i++)  cout << "(" << a[i].x << ", "<< a[i].y << ")\n";  }  // Driver program to test above functions  //findClosest  int compareX(const void\* a, const void\* b)  {  Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;  return (p1->x - p2->x);  }  // Needed to sort array of points according to Y coordinate  int compareY(const void\* a, const void\* b)  {  Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;  return (p1->y - p2->y);  }  // A utility function to find the  // distance between two points  float dist(Point p1, Point p2)  {  return sqrt( (p1.x - p2.x)\*(p1.x - p2.x) +  (p1.y - p2.y)\*(p1.y - p2.y)  );  }  // A Brute Force method to return the  // smallest distance between two points  // in P[] of size n  float bruteForce(Point P[], int n)  {  float min = FLT\_MAX;  for (int i = 0; i < n; ++i)  for (int j = i+1; j < n; ++j)  if (dist(P[i], P[j]) < min)  min = dist(P[i], P[j]);  return min;  }  // A utility function to find  // minimum of two float values  float min(float x, float y)  {  return (x < y)? x : y;  }  // A utility function to find the  // distance between the closest points of  // strip of given size. All points in  // strip[] are sorted according to  // y coordinate. They all have an upper  // bound on minimum distance as d.  // Note that this method seems to be  // a O(n^2) method, but it's a O(n)  // method as the inner loop runs at most 6 times  float stripClosest(Point strip[], int size, float d)  {  float min = d; // Initialize the minimum distance as d  qsort(strip, size, sizeof(Point), compareY);  // Pick all points one by one and try the next points till the difference  // between y coordinates is smaller than d.  // This is a proven fact that this loop runs at most 6 times  for (int i = 0; i < size; ++i)  for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)  if (dist(strip[i],strip[j]) < min)  min = dist(strip[i], strip[j]);  return min;  }  // A recursive function to find the  // smallest distance. The array P contains  // all points sorted according to x coordinate  float closestUtil(Point P[], int n)  {  // If there are 2 or 3 points, then use brute force  if (n <= 3)  return bruteForce(P, n);  // Find the middle point  int mid = n/2;  Point midPoint = P[mid];  // Consider the vertical line passing  // through the middle point calculate  // the smallest distance dl on left  // of middle point and dr on right side  float dl = closestUtil(P, mid);  float dr = closestUtil(P + mid, n - mid);  // Find the smaller of two distances  float d = min(dl, dr);  // Build an array strip[] that contains  // points close (closer than d)  // to the line passing through the middle point  Point strip[n];  int j = 0;  for (int i = 0; i < n; i++)  if (abs(P[i].x - midPoint.x) < d)  strip[j] = P[i], j++;  // Find the closest points in strip.  // Return the minimum of d and closest  // distance is strip[]  return min(d, stripClosest(strip, j, d) );  }  // The main function that finds the smallest distance  // This method mainly uses closestUtil()  float closest(vector<Point> v, int n)  {  Point P[n];  for (int i = 0; i<n; i++)  {  P[i] = v.at(i);  }  qsort(P, n, sizeof(Point), compareX);  // Use recursive function closestUtil()  // to find the smallest distance  return closestUtil(P, n);  }  int main()  {  Point points[] = {{4, 5}, {6, 4}, {7,6}, {8, 7},  {9, 8}, {5, 10}, {4, 9}, {5, 9}, {8, 5}, {8, 10}};  int T= sizeof(points) / sizeof(points[0]);  // cout << "enter number of points: ";  // cin>>T;  // if(T<=0)  // return -1;  // Point points[T];  // for(int i=0;i<T;++i){  // cout << "Enter x cor of " << i << " point: ";  // cin >> x;  // points[i].x=x;  // cout << "Enter y cor of " << i << " point: ";  // cin >> y;  // points[i].y=y;  // }  vector<Point> inner;  vector<Point> ans = convexHull(points, T, &inner);  // cout<<"\n---------After Using Graham Scan Algorithm---------------\n";  // cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CONVEX HULL \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  printVector(ans);  // cout << "Inner points: " << "\n";  // cout << "Polygon area: " << polygonArea(ans, ans.size());  // printVector(inner);  cout << "closest distance among convexhull points: " << closest(ans, ans.size());  return 0;  } |
| Kết quả: |

## d) Viết chương trình tìm khoảng các ngắn nhất giữa các điểm nằm trong bao lồi, độ phức tạp nlogn

|  |
| --- |
| // A C++ program to find convex hull of a set of points. Refer  // https://www.geeksforgeeks.org/orientation-3-ordered-points/  // for explanation of orientation()  //graham scan  #include <bits/stdc++.h>  #include <iostream>  #include <stack>  #include <stdlib.h>  using namespace std;  struct Point  {  int x, y;  };  // A global point needed for sorting points with reference  // to the first point Used in compare function of qsort()  Point p0;  // A utility function to find next to top in a stack  Point nextToTop(stack<Point> &S)  {  Point p = S.top();  S.pop();  Point res = S.top();  S.push(p);  return res;  }  // A utility function to swap two points  void swap(Point &p1, Point &p2)  {  Point temp = p1;  p1 = p2;  p2 = temp;  }  // A utility function to return square of distance  // between p1 and p2  int distSq(Point p1, Point p2)  {  return (p1.x - p2.x)\*(p1.x - p2.x) +  (p1.y - p2.y)\*(p1.y - p2.y);  }  // To find orientation of ordered triplet (p, q, r).  // The function returns following values  // 0 --> p, q and r are collinear  // 1 --> Clockwise  // 2 --> Counterclockwise  int orientation(Point p, Point q, Point r)  {  int val = (q.y - p.y) \* (r.x - q.x) -  (q.x - p.x) \* (r.y - q.y);  if (val == 0) return 0; // collinear  return (val > 0)? 1: 2; // clock or counterclock wise  }  // A function used by library function qsort() to sort an array of  // points with respect to the first point  int compare(const void \*vp1, const void \*vp2)  {  Point \*p1 = (Point \*)vp1;  Point \*p2 = (Point \*)vp2;  // Find orientation  int o = orientation(p0, \*p1, \*p2);  if (o == 0)  return (distSq(p0, \*p2) >= distSq(p0, \*p1))? -1 : 1;  return (o == 2)? -1: 1;  }  // Prints convex hull of a set of n points.  vector<Point> convexHull(Point points[], int n, vector<Point> \*inner)  {  vector<Point> hello;  // Find the bottommost point  int ymin = points[0].y, min = 0;  for (int i = 1; i < n; i++)  {  int y = points[i].y;  // Pick the bottom-most or choose the left  // most point in case of tie  if ((y < ymin) || (ymin == y &&  points[i].x < points[min].x))  ymin = points[i].y, min = i;  }  // Place the bottom-most point at first position  swap(points[0], points[min]);  // Sort n-1 points with respect to the first point.  // A point p1 comes before p2 in sorted output if p2  // has larger polar angle (in counterclockwise  // direction) than p1  p0 = points[0];  qsort(&points[1], n-1, sizeof(Point), compare);  // If two or more points make same angle with p0,  // Remove all but the one that is farthest from p0  // Remember that, in above sorting, our criteria was  // to keep the farthest point at the end when more than  // one points have same angle.  int m = 1; // Initialize size of modified array  for (int i=1; i<n; i++)  {  // Keep removing i while angle of i and i+1 is same  // with respect to p0  while (i < n-1 && orientation(p0, points[i],  points[i+1]) == 0)  i++;  points[m] = points[i];  m++; // Update size of modified array  }  // If modified array of points has less than 3 points,  // convex hull is not possible  if (m < 3) return hello;  // Create an empty stack and push first three points  // to it.  stack<Point> S;  S.push(points[0]);  S.push(points[1]);  S.push(points[2]);  // Process remaining n-3 points  for (int i = 3; i < m; i++)  {  // Keep removing top while the angle formed by  // points next-to-top, top, and points[i] makes  // a non-left turn  while (S.size()>1 && orientation(nextToTop(S), S.top(), points[i]) != 2)  {  inner->push\_back(S.top());  S.pop();  }  S.push(points[i]);  }  // Now stack has the output points, print contents of stack    while (!S.empty())  {  Point p = S.top();  hello.push\_back(p);  S.pop();  }  return hello;  }  double polygonArea(vector<Point> ans, int n)  {  double S = 0;  for (int i = 0; i < n - 1; i++)  {  S += (ans.at(i).x\* ans.at(i+1).y) - (ans.at(i+1).x\*ans.at(i).y);  }  double A = 0.5\* fabs(S);  // S = (1/2)\*fabs(S);  return A;  }  void printVector(vector<Point> a)  {    for (int i = 0; i < a.size(); i++)  cout << "(" << a[i].x << ", "<< a[i].y << ")\n";  }  // Driver program to test above functions  //findClosest  int compareX(const void\* a, const void\* b)  {  Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;  return (p1->x - p2->x);  }  // Needed to sort array of points according to Y coordinate  int compareY(const void\* a, const void\* b)  {  Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;  return (p1->y - p2->y);  }  // A utility function to find the  // distance between two points  float dist(Point p1, Point p2)  {  return sqrt( (p1.x - p2.x)\*(p1.x - p2.x) +  (p1.y - p2.y)\*(p1.y - p2.y)  );  }  // A Brute Force method to return the  // smallest distance between two points  // in P[] of size n  float bruteForce(Point P[], int n)  {  float min = FLT\_MAX;  for (int i = 0; i < n; ++i)  for (int j = i+1; j < n; ++j)  if (dist(P[i], P[j]) < min)  min = dist(P[i], P[j]);  return min;  }  // A utility function to find  // minimum of two float values  float min(float x, float y)  {  return (x < y)? x : y;  }  // A utility function to find the  // distance between the closest points of  // strip of given size. All points in  // strip[] are sorted according to  // y coordinate. They all have an upper  // bound on minimum distance as d.  // Note that this method seems to be  // a O(n^2) method, but it's a O(n)  // method as the inner loop runs at most 6 times  float stripClosest(Point strip[], int size, float d)  {  float min = d; // Initialize the minimum distance as d  qsort(strip, size, sizeof(Point), compareY);  // Pick all points one by one and try the next points till the difference  // between y coordinates is smaller than d.  // This is a proven fact that this loop runs at most 6 times  for (int i = 0; i < size; ++i)  for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)  if (dist(strip[i],strip[j]) < min)  min = dist(strip[i], strip[j]);  return min;  }  // A recursive function to find the  // smallest distance. The array P contains  // all points sorted according to x coordinate  float closestUtil(Point P[], int n)  {  // If there are 2 or 3 points, then use brute force  if (n <= 3)  return bruteForce(P, n);  // Find the middle point  int mid = n/2;  Point midPoint = P[mid];  // Consider the vertical line passing  // through the middle point calculate  // the smallest distance dl on left  // of middle point and dr on right side  float dl = closestUtil(P, mid);  float dr = closestUtil(P + mid, n - mid);  // Find the smaller of two distances  float d = min(dl, dr);  // Build an array strip[] that contains  // points close (closer than d)  // to the line passing through the middle point  Point strip[n];  int j = 0;  for (int i = 0; i < n; i++)  if (abs(P[i].x - midPoint.x) < d)  strip[j] = P[i], j++;  // Find the closest points in strip.  // Return the minimum of d and closest  // distance is strip[]  return min(d, stripClosest(strip, j, d) );  }  // The main function that finds the smallest distance  // This method mainly uses closestUtil()  float closest(vector<Point> v, int n)  {  Point P[n];  for (int i = 0; i<n; i++)  {  P[i] = v.at(i);  }  qsort(P, n, sizeof(Point), compareX);  // Use recursive function closestUtil()  // to find the smallest distance  return closestUtil(P, n);  }  int main()  {  Point points[] = {{4, 5}, {6, 4}, {7,6}, {8, 7},  {9, 8}, {5, 10}, {4, 9}, {5, 9}, {8, 5}, {8, 10}};  int T= sizeof(points) / sizeof(points[0]);  // cout << "enter number of points: ";  // cin>>T;  // if(T<=0)  // return -1;  // Point points[T];  // for(int i=0;i<T;++i){  // cout << "Enter x cor of " << i << " point: ";  // cin >> x;  // points[i].x=x;  // cout << "Enter y cor of " << i << " point: ";  // cin >> y;  // points[i].y=y;  // }  vector<Point> inner;  vector<Point> ans = convexHull(points, T, &inner);  // cout<<"\n---------After Using Graham Scan Algorithm---------------\n";  // cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CONVEX HULL \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  // printVector(ans);  // cout << "Inner points: " << "\n";  // cout << "Polygon area: " << polygonArea(ans, ans.size());  printVector(inner);  cout << "closest distance among inner points: " << closest(inner, inner.size());    return 0;  } |
| Kết quả: |

## e) Viết chương trình tìm khoảng cách ngắn nhất giữa các điểm trong bao lồi với đỉnh bao lồi

|  |
| --- |
| // A divide and conquer program in C++  // to find the smallest distance from a  // given set of points.  #include <bits/stdc++.h>  using namespace std;  // A structure to represent a Point in 2D plane  struct Point  {  int x, y;  };  /\* Following two functions are needed for library function qsort().  Refer: http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ \*/  // Needed to sort array of points  // according to X coordinate  int compareX(const void\* a, const void\* b)  {  Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;  return (p1->x - p2->x);  }  // Needed to sort array of points according to Y coordinate  int compareY(const void\* a, const void\* b)  {  Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;  return (p1->y - p2->y);  }  // A utility function to find the  // distance between two points  float dist(Point p1, Point p2)  {  return sqrt( (p1.x - p2.x)\*(p1.x - p2.x) +  (p1.y - p2.y)\*(p1.y - p2.y)  );  }  // A Brute Force method to return the  // smallest distance between two points  // in P[] of size n  float bruteForce(Point P[], int n)  {  float min = FLT\_MAX;  for (int i = 0; i < n; ++i)  for (int j = i+1; j < n; ++j)  if (dist(P[i], P[j]) < min)  min = dist(P[i], P[j]);  return min;  }  // A utility function to find  // minimum of two float values  float min(float x, float y)  {  return (x < y)? x : y;  }  // A utility function to find the  // distance between the closest points of  // strip of given size. All points in  // strip[] are sorted according to  // y coordinate. They all have an upper  // bound on minimum distance as d.  // Note that this method seems to be  // a O(n^2) method, but it's a O(n)  // method as the inner loop runs at most 6 times  float stripClosest(Point strip[], int size, float d)  {  float min = d; // Initialize the minimum distance as d  qsort(strip, size, sizeof(Point), compareY);  // Pick all points one by one and try the next points till the difference  // between y coordinates is smaller than d.  // This is a proven fact that this loop runs at most 6 times  for (int i = 0; i < size; ++i)  for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)  if (dist(strip[i],strip[j]) < min)  min = dist(strip[i], strip[j]);  return min;  }  // A recursive function to find the  // smallest distance. The array P contains  // all points sorted according to x coordinate  float closestUtil(Point P[], int n)  {  // If there are 2 or 3 points, then use brute force  if (n <= 3)  return bruteForce(P, n);  // Find the middle point  int mid = n/2;  Point midPoint = P[mid];  // Consider the vertical line passing  // through the middle point calculate  // the smallest distance dl on left  // of middle point and dr on right side  float dl = closestUtil(P, mid);  float dr = closestUtil(P + mid, n - mid);  // Find the smaller of two distances  float d = min(dl, dr);  // Build an array strip[] that contains  // points close (closer than d)  // to the line passing through the middle point  Point strip[n];  int j = 0;  for (int i = 0; i < n; i++)  if (abs(P[i].x - midPoint.x) < d)  strip[j] = P[i], j++;  // Find the closest points in strip.  // Return the minimum of d and closest  // distance is strip[]  return min(d, stripClosest(strip, j, d) );  }  // The main function that finds the smallest distance  // This method mainly uses closestUtil()  float closest(Point P[], int n)  {  qsort(P, n, sizeof(Point), compareX);  // Use recursive function closestUtil()  // to find the smallest distance  return closestUtil(P, n);  }  // Driver code  int main()  {  Point P[] = {{4, 5}, {6, 4}, {7,6}, {8, 7},  {9, 8}, {5, 10}, {4, 9}, {5, 9}, {8, 5}, {8, 10}};  int n = sizeof(P) / sizeof(P[0]);  cout << "The smallest distance is " << closest(P, n);  return 0;  } |
| Kết quả: |