# Algoritmos para competencias

Sebastián Múnera Álvarez

## 1. Next Permutation

```
void swap(char[] c, int i, int j) {
  char t = c[i];
  c[i] = c[j];
  c[j] = t;
}

boolean nextPermutation(char[] c) {
  int n = c.length;
  int k = -1;
  for (int i = n - 2; i >= 0; --i)
    if (c[i] < c[i + 1]) {
      k = i;
      break;
    }
  if (k == -1)
      return false;
  int l = 0;
  for (int i = n - 1; i >= 0; --i)
    if (c[k] < c[i]) {
      l = i;
      break;
    }
  swap(c, k, l);
  for (int i = k + 1; i < (n + k + 1) / 2; ++i)
    swap(c, i, n + k - i);

  return true;
}
```

## 2. All permutations

```
  static void solve(int[] b, int k) {
    if (k == n)
      System.out.println(Arrays.toString(b));

    for (int i = 0; i < n; ++i) {
      if (used[i])
        continue;
      used[i] = true;
      b[k] = original[i];
      solve(b, k + 1);
      used[i] = false;
    }
  }
```

## 3. All subsets of length k

```
void subset(int[] set, int n, int k) {
  for (int i = 0; i < (1 << n); ++i)
    if (Integer.bitCount(i) == k) {
      int[] sub = new int[k];
      int ind = 0;
      for (int j = 0; j < n; ++j)
        if ((i & (1 << j)) != 0)
          sub[ind++] = set[j];
      System.out.println(Arrays.toString(sub));
    }
}
```

## 4. DFS in a grid

```
static int N;
static char[][] map;
static boolean[][] visited;
static int[] dx = {-1, -1, -1, 0, 0, 1, 1, 1};
static int[] dy = {-1, 0, 1, -1, 1, -1, 0, 1};
```

```
static boolean valid(int i, int j) {
  return i >= 0 && j >= 0 && i < N &&
      j < N && map[i][j] == '1' &&
         !visited[i][j];
}

static void dfs(int i, int j) {
  if (!valid(i, j))
    return;

  visited[i][j] = true;
  for (int d = 0; d < dx.length; ++d)
    dfs(i + dy[d], j + dx[d]);
}
```

## 5. DFS in graph

```
static List<Edge>[] G;
static boolean[] visited;
static int N;

static class Edge {
  int to;

  public Edge(int to) {
    this.to = to;
  }
}

static void dfs(int v) {
  if (visited[v])
    return;

  visited[v] = true;
  for (Edge w : G[v])
    dfs(w.to);
}
```

## 6. BFS

```
static Map<Integer, Integer> vertices;
static boolean[][] G;
static int N;

static int[] bfs(int source) {
  int[] distance = new int[N];
  boolean[] visited = new boolean[N];
  int[] parent = new int[N];

  Arrays.fill(distance, -1);
  Arrays.fill(parent, -1);

  Queue<Integer> Q = new
     LinkedList<Integer>();

  Q.offer(source);
  int i = vertices.get(source);
  visited[i] = true;
  distance[i] = 0;

  while (!Q.isEmpty()) {
    int v = Q.poll();
    i = vertices.get(v);

    for (int w : vertices.keySet()) {
      int j = vertices.get(w);
      if (G[i][j] && !visited[j]) {
        visited[j] = true;
        distance[j] = distance[i] + 1;
        parent[j] = v;
        Q.offer(w);
      }
    }
  }

  return distance;
}
```

## 7. Prim (Minimum Spanning Tree)

```
static int N;
static Vertex[] G;
static final int INF = Integer.MAX_VALUE;

static class Vertex {
```

```
  int id;
  List<Edge> adjacency = new
     ArrayList<Edge>();

  public Vertex(int id) {
    this.id = id;
  }
}

static class Edge {
  Vertex to;
  int weight;

  public Edge(Vertex to, int weight) {
    this.to = to;
    this.weight = weight;
  }
}

static class QueueItem implements
   Comparable<QueueItem> {
  Vertex v;
  int distance;

  public QueueItem(Vertex v, int distance) {
    this.v = v;
    this.distance = distance;
  }

  public int compareTo(QueueItem q) {
    return this.distance - q.distance;
  }

}

static int prim(int s) {
  boolean[] intree = new boolean[N];
  int[] distance = new int[N];
  int[] parent = new int[N];
  int cost = 0;

  Arrays.fill(distance, INF);
  Arrays.fill(parent, -1);

  distance[s] = 0;

  PriorityQueue<QueueItem> Q = new
     PriorityQueue<QueueItem>();
  Q.offer(new QueueItem(G[s], 0));

  while (!Q.isEmpty()) {
    QueueItem qi = Q.poll();
    Vertex v = qi.v;

    if (!intree[v.id]) {
      intree[v.id] = true;

      for (Edge edge : v.adjacency) {
        Vertex w = edge.to;
        int weight = edge.weight;

        if (!intree[w.id] && weight <
           distance[w.id]) {
          distance[w.id] = weight;
          parent[w.id] = v.id;
          Q.offer(new QueueItem(w, weight));
        }
      }
    }
  }

  for (int i = 0; i < N; ++i)
    cost += distance[i];

  return cost;
}
```

## 8. Dijkstra

```
static int N;
static Vertex[] G;
static final int INF = Integer.MAX_VALUE;

static class Vertex {
  int id;
  List<Edge> adjacency = new
     ArrayList<Edge>();
```

```
  public Vertex(int id) {
    this.id = id;
  }
}

static class Edge {
  Vertex to;
  int weight;

  public Edge(Vertex to, int weight) {
    this.to = to;
    this.weight = weight;
  }
}

static class QueueItem implements
   Comparable<QueueItem> {
  Vertex v;
  int distance;

  public QueueItem(Vertex v, int distance) {
    this.v = v;
    this.distance = distance;
  }

  public int compareTo(QueueItem q) {
    return this.distance - q.distance;
  }
}

static int dijkstra(int s, int t) {
  boolean[] visited = new boolean[N];
  int[] distance = new int[N];
  PriorityQueue<QueueItem> Q = new
     PriorityQueue<QueueItem>();

  Arrays.fill(distance, INF);

  distance[s] = 0;
  Q.offer(new QueueItem(G[s], 0));

  while (!Q.isEmpty()) {
    QueueItem q = Q.poll();
    Vertex v = q.v;

    if (!visited[v.id]) {
      visited[v.id] = true;

      for (Edge edge : G[v.id].adjacency) {
        Vertex w = edge.to;
        int weight = edge.weight;

        if (distance[w.id] > distance[v.id] +
           weight) {
          distance[w.id] = distance[v.id] +
             weight;
          Q.offer(new QueueItem(w,
             distance[w.id]));
        }
      }
    }
  }

  return distance[t];
}
```

## 9. Bellman-Ford (Negative cycles)

```
static int N, M;
static final int INF = Integer.MAX_VALUE;
static Edge[] G;

static class Edge {
  int from;
  int to;
  int time;

  public Edge(int from, int to, int time) {
    this.from = from;
    this.to = to;
    this.time = time;
  }
}

static boolean bellmanford(int s) {
  int[] distance = new int[N];
```

```
    Arrays.fill(distance, INF);
    distance[s] = 0;

    for (int i = 0; i < N; ++i)
      for (int j = 0; j < M; ++j)
        if (distance[G[j].to] >
            distance[G[j].from] + G[j].time)
          distance[G[j].to] =
              distance[G[j].from] + G[j].time;
    for (int j = 0; j < M; ++j)
      if (distance[G[j].to] >
          distance[G[j].from] + G[j].time)
        return true;
    return false;
  }
```

## 10. Number of Divisors of an integer

```
  int divisors(int x) {

    int nDiv = 1;
    for (int p = 2; p * p <= x; ++p) {
      int cnt = 0;
      while (x % p == 0) {
        ++cnt;
        x /= p;
      }
      nDiv *= cnt + 1;
    }
    if (x > 1)
      nDiv *= 2;

    return nDiv;
}
```

## 11. All the prime factors of a number

```
  void primeFactors(int N) {
    for (long p = 2; p * p <= N; ++p)
      while (N % p == 0) {
        System.out.println(p);
        N /= p;
      }
    if (N > 1)
      System.out.println(N);
}
```

## 12. Primality Test

```
  boolean isPrime(int x) {
    if (x < 2)
      return false;
    if (x == 2)
      return true;
    if (x % 2 == 0)
      return false;

    for (int i = 3; i * i <= x; i += 2)
      if (x % i == 0)
        return false;

    return true;
  }
```

## 13. Sieve of Erathostenes

```
  static boolean[] sieve(int N) {
    boolean[] prime = new boolean[N + 1];
    Arrays.fill(prime, true);

    prime[0] = prime[1] = false;
    for (int p = 2; p * p <= N; ++p)
      if (prime[p])
        for (int i = p * p; i <= N; i += p)
          prime[i] = false;
    return prime;
  }
```

## 14. GCD (Euclid Algorithm)

```
  int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
  }
```

## 15. LCM

```
int lcm(int a, int b) {
  return a * (b / gcd(a, b));
}
```

## 16. Fast Exponentiation

```
static int fastPow(int B, int P, int M) {
  if (P == 0) return 1;
  if (P == 1) return B;
  if ((P & 1) == 1)
    return ((B % M) * fastPow(((B % M) * (B %
       M)) % M,
                (P - 1) / 2, M) % M) % M;
  else
    return fastPow(((B % M) * (B % M)) % M, P
       / 2, M);
}
```

## 17. Floyd-Warshall (All pairs shortest paths)

```
for (int k = 0; k < n; ++k)
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      adj[i][j] = Math.min(adj[i][j],
         adj[i][k] + adj[k][j]);
```

## 18. Binary Search

```
int binarySearch(int lo, int hi) {
  while (lo < hi) {
    int mid = lo + (hi - lo) / 2;
    if (p(mid))
      hi = mid;
    else
      lo = mid + 1;
  }
  if (!p(lo))
    return -1;
  return lo;
}
```

## 19. Longest Increasing Subsequence

```
int lis(int[] sequence) {
  int n = sequence.length;
  int[] q = new int[n];

  for (int i = 0; i < n; ++i) {
    int max = 0;
    for (int j = 0; j < i; ++j)
      if (sequence[i] > sequence[j])
        max = Math.max(max, q[j]);
    q[i] = max + 1;
  }
  int max = 0;
  for (int i = 0; i < n; ++i)
    max = Math.max(max, q[i]);
  return max;
}
```

## 20. Convex Hull (Graham Scan)

```
static class Point {
  int x;
  int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public String toString() {
    return this.x + " " + this.y;
  }
}

static int signedTriangleArea(Point a, Point b,
   Point c) {
  return a.x * b.y - a.y * b.x + a.y * c.x -
         a.x * c.y + b.x * c.y - c.x * b.y;
```

```
}

// Counterclockwise predicate. Says whether c
   is to
// the right of the line formed by a and b
static boolean ccw(Point a, Point b, Point c) {
  return signedTriangleArea(a, b, c) > 0;
}

// Clockwise predicate. Says whether c is to
   the left
// of the line formed by a and b
static boolean cw(Point a, Point b, Point c) {
  return signedTriangleArea(a, b, c) < 0;
}

// Says whether a, b, c are collinear
static boolean collinear(Point a, Point b,
   Point c) {
  return signedTriangleArea(a, b, c) == 0;
}

static double distance(Point p1, Point p2) {
  int dx = p1.x - p2.x;
  int dy = p1.y - p2.y;
  return Math.sqrt(dx * dx + dy * dy);
}

// First and last points are different
static Point[] graham(Point[] in) {
  int N = in.length;
  final Point first;

  // Choose the point with the least y
     coordinate, or
  // least x coordinate in case of a tie
  int min = 0;
  for (int i = 1; i < N; ++i) {
    if (in[i].y < in[min].y)
      min = i;
    else if (in[i].y == in[min].y)
      if (in[i].x < in[min].x)
        min = i;
  }

  first = in[min];
  in[min] = in[0];
  in[0] = first;

  // Sort by angle with first
  Arrays.sort(in, 1, N, new Comparator<Point>()
     {
    public int compare(Point p1, Point p2) {
      if (collinear(first, p1, p2))
        return Double.compare(distance(first,
           p1),
                              distance(first,
                                 p2));
      if (ccw(first, p1, p2))
        return -1;
      else
        return 1;
    }
  });
  if (N < 3)
    return in;

  Stack<Point> S = new Stack<Point>();
  S.push(in[0]);
  S.push(in[1]);

  for (int i = 2; i < N; ++i) {
    while (S.size() > 1) {
      Point top = S.pop();
      Point nextTop = S.pop();
      S.push(nextTop);
      S.push(top);

      if (!ccw(nextTop, top, in[i]))
        S.pop();
      else
        break;
    }
    S.push(in[i]);
  }

  int M = S.size();
  Point[] hull = new Point[M + 1];
```

```
  int k = M - 1;
  while (!S.isEmpty())
    hull[k--] = S.pop();
  hull[M] = hull[0];

  return hull;
}
```

## 21. Divisibility by small numbers in interval [1, 12]

```
boolean[] isMultipleOf = new boolean[13];

int sum = 0;
for (char c : M.toCharArray())
  sum += c - '0';
int lastDigit = M.charAt(n - 1) - '0';

isMultipleOf[1] = true;

isMultipleOf[2] = lastDigit % 2 == 0;

isMultipleOf[3] = sum % 3 == 0;

if (n > 1)
  isMultipleOf[4] =
     Integer.parseInt(M.substring(n - 2, n)) %
     4 == 0;
else
  isMultipleOf[4] = lastDigit % 4 == 0;

isMultipleOf[5] = lastDigit == 0 || lastDigit
   == 5;

isMultipleOf[6] = isMultipleOf[2] &&
   isMultipleOf[3];

int altSum = 0;
int[] pattern = {1, 3, 2, -1, -3, -2};
int j = 0;
for (int i = n - 1; i >= 0; --i) {
  altSum += pattern[j] * (M.charAt(i) - '0');
  j = (j + 1) % 6;
}
isMultipleOf[7] = Math.abs(altSum) % 7 == 0;

if (n > 2)
  isMultipleOf[8] =
     Integer.parseInt(M.substring(n - 3, n)) %
     8 == 0;
else
  isMultipleOf[8] = Integer.parseInt(M) % 8 ==
     0;

isMultipleOf[9] = sum % 9 == 0;

isMultipleOf[10] = lastDigit == 0;

altSum = 0;
int s = 1;
for (int i = n - 1; i >= 0; --i) {
  altSum += s * (M.charAt(i) - '0');
  s = -s;
}
isMultipleOf[11] = Math.abs(altSum) % 11 == 0;

isMultipleOf[12] = isMultipleOf[3] &&
   isMultipleOf[4];
```

## 22. Heron's Formula (Area of a triangle given vertices coordinates)

```
double heron(double x1, double y1, double x2,
   double y2, double x3, double y3) {

  double a = Math.sqrt((x1 - x2) * (x1 - x2) +
     (y1 - y2) * (y1 - y2));
  double b = Math.sqrt((x1 - x3) * (x1 - x3) +
     (y1 - y3) * (y1 - y3));
  double c = Math.sqrt((x3 - x2) * (x3 - x2) +
     (y3 - y2) * (y3 - y2));

  double s = (a + b + c) / 2.0;
  double A = Math.sqrt(s * (s - a) * (s - b) *
     (s - c));
```

```
  return A;
}
```

## 23. Radius of circle that passes through three points (circumradius)

```
double circumradius(double x1, double y1,
    double x2, double y2, double x3, double y3) {

  double a = Math.sqrt((x1 - x2) * (x1 - x2) +
      (y1 - y2) * (y1 - y2));
  double b = Math.sqrt((x1 - x3) * (x1 - x3) +
      (y1 - y3) * (y1 - y3));
  double c = Math.sqrt((x3 - x2) * (x3 - x2) +
      (y3 - y2) * (y3 - y2));

  double A = heron(x1, y1, x2, y2, x3, y3);
  double r = (a * b * c) / (4.0 * A);

  return r;
}
```

## 24. Maximum circle inscribed in a triangle (incircle)

```
void inradius(double x1, double y1, double x2,
    double y2, double x3, double y3) {

  double a = Math.sqrt((x1 - x2) * (x1 - x2) +
      (y1 - y2) * (y1 - y2));
  double b = Math.sqrt((x1 - x3) * (x1 - x3) +
      (y1 - y3) * (y1 - y3));
  double c = Math.sqrt((x3 - x2) * (x3 - x2) +
      (y3 - y2) * (y3 - y2));

  double r = 0.0;

  if (Math.abs(a) >= 1e-9 && Math.abs(b) >=
      1e-9 && Math.abs(c) >= 1e-9) {
    double A = heron(x1, y1, x2, y2, x3, y3);
    r = 2.0 * A / (a + b + c);
  }
  double xc = (c * x1 + b * x2 + a * x3) / (a +
      b + c);
  double yc = (c * y1 + b * y2 + a * y3) / (a +
      b + c);

  System.out.println("(" + xc + ", " + yc + ")
      " + r);
}
```

## 25. Circle that passes through three points (circumcircle)

```
void circumcircle(double x1, double y1, double
    x2, double y2, double x3, double y3) {

  double h = 0.0;
  double k = 0.0;
  double r = 0.0;

  boolean isRight = false;

  double d12 = Math.sqrt((x2 - x1) * (x2 - x1)
      + (y2 - y1) * (y2 - y1));
  double d13 = Math.sqrt((x3 - x1) * (x3 - x1)
      + (y3 - y1) * (y3 - y1));
  double d23 = Math.sqrt((x2 - x3) * (x2 - x3)
      + (y2 - y3) * (y2 - y3));

  if (Math.abs(d12 * d12 + d13 * d13 - d23 *
      d23) < 1e-6) {
    isRight = true;
    h = (x2 + x3) / 2.0;
    k = (y2 + y3) / 2.0;
  }
  if (Math.abs(d12 * d12 + d23 * d23 - d13 *
      d13) < 1e-6) {
    isRight = true;
    h = (x1 + x3) / 2.0;
    k = (y1 + y3) / 2.0;
  }
  if (Math.abs(d23 * d23 + d13 * d13 - d12 *
      d12) < 1e-6) {
```

```
    isRight = true;
    h = (x2 + x1) / 2.0;
    k = (y2 + y1) / 2.0;
  }

  if (!isRight) {
    if (Double.compare(x2, x1) == 0 ||
        Double.compare(y2, y1) == 0) {
      double t = x2;
      x2 = x3;
      x3 = t;

      t = y2;
      y2 = y3;
      y3 = t;
    }

    // Line between A and B
    double m12 = (y2 - y1) / (x2 - x1);
    double b12 = y1 - m12 * x1;

    // Line bisector to 12
    double m12b = -1.0 / m12;
    double xm = (x1 + x2) / 2.0;

    double b12b = b12 + m12 * xm - m12b * xm;


    if (Double.compare(x3, x2) == 0 ||
        Double.compare(y3, y2) == 0) {
      double t = x2;
      x2 = x1;
      x1 = t;

      t = y2;
      y2 = y1;
      y1 = t;
    }

    // Line between B and C
    double mBC = (y3 - y2) / (x3 - x2);
    double bBC = y2 - mBC * x2;

    // Line bisector to BC
    double mBCb = -1.0 / mBC;
    xm = (x2 + x3) / 2.0;

    double bBCb = bBC + mBC * xm - mBCb * xm;


    // Where both bisectors intersect is the
       center of the circle
    h = (bBCb - b12b) / (m12b - mBCb);
    k = m12b * h + b12b;
  }
  r = Math.sqrt((h - x1) * (h - x1) + (k - y1)
      * (k - y1));

  System.out.println("(" + h + ", " + k + ") "
      + r);
}
```

## 26. Area of polygon

```
  int signedTriangleArea(Point a, Point b,
      Point c) {
    return a.x * b.y - a.y * b.x + a.y * c.x -
           a.x * c.y + b.x * c.y - c.x * b.y;
  }

  boolean ccw(Point a, Point b, Point c) {
    return signedTriangleArea(a, b, c) > 0;
  }

  boolean collinear(Point a, Point b, Point c) {
    return signedTriangleArea(a, b, c) == 0;
  }

  double distance(Point p1, Point p2) {
    double dx = p1.x - p2.x;
    double dy = p1.y - p2.y;

    return Math.sqrt(dx * dx + dy * dy);
  }

  double area(Polygon poly) {
    int N = poly.npoints;
```

```
  int[] x = poly.xpoints;
  int[] y = poly.ypoints;

  Point[] p = new Point[N];
  for (int i = 0; i < N; ++i)
    p[i] = new Point(x[i], y[i]);

  final Point first;

  int min = 0;
  for (int i = 1; i < N; ++i)
    if (p[i].y < p[min].y)
      min = i;
    else if (p[i].y == p[min].y)
      if (p[i].x < p[min].x)
        min = i;

  first = p[min];
  p[min] = p[0];
  p[0] = first;

  Arrays.sort(p, 1, N, new
     Comparator<Point>() {

    public int compare(Point p1, Point p2) {
      if (collinear(first, p1, p2))
        return Double.compare(distance(first,
           p1), distance(first, p2));
      if (ccw(first, p1, p2))
        return -1;
      else
        return 1;
    }

  });

  double A = 0.0;
  for (int i = 0; i < N; ++i) {
    int j = (i + 1) % N;
    A += p[i].x * p[j].y - p[j].x * p[i].y;
  }

  return A / 2.0;
}
```

## 27. Topological Sort (DFS)

```
static int N;
static boolean[][] adj;
static boolean[] visited;
static String sort;

static void dfs(int s) {
  visited[s] = true;

  for (int i = 0; i < N; ++i)
    if (adj[s][i] && !visited[i])
      dfs(i);

  if (!sort.isEmpty())
    sort = " " + sort;
  sort = (s + 1) + sort;
}

static void topsort() {
  visited = new boolean[N];
  sort = "";

  for (int i = 0; i < N; ++i)
    if (!visited[i])
      dfs(i);
  System.out.println(sort);
}
```

## 28. Strongly Connected Components (Directed Graphs)

```
static int N;
static boolean[][] adj;
static boolean[] visited;
static List<Integer> sort;
static Map<Integer, String> map;

static void dfs2(int s, boolean first) {
  visited[s] = true;

  if (!first)
    System.out.print(", ");
```

```
    System.out.print(map.get(s));

    for (int i = 0; i < N; ++i)
      if (adj[i][s] && !visited[i])
        dfs2(i, false);
  }

  static void scc() {
    topsort();

    visited = new boolean[N];
    for (int x : sort)
      if (!visited[x]) {
        dfs2(x, true);
        System.out.println();
      }
  }
```

## 29. Longest (shortest) path in DAG

```
  static final int INF = Integer.MAX_VALUE;
  static int N;
  static boolean[][] adj;
  static boolean[] visited;
  static List<Integer> sort;
  static int[] d;
  static int[] parent;

  static void dagLongestPath(int s) {
    topsort();

    d = new int[N];
    parent = new int[N];

    Arrays.fill(d, -INF);
    Arrays.fill(parent, -1);

    d[s] = 0;

    for (int u : sort)
      for (int v = 0; v < N; ++v)
        if (adj[u][v])
          if (d[v] < d[u] + 1) { // If graph is
              weighted: d[v] < d[i] + w;
            d[v] = d[u] + 1; // If looking for
               shortest path: d[v] > d[u] + 1;
            parent[v] = u;
          }
  }
```

## 30. Binomial Coefficients

$$\binom{n}{k} = \begin{cases} 1 & \text{if k = 0 or n = k} \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{otherwise} \end{cases}$$

```
  int N = 30;
  long[][] C = new long[N + 1][N + 1];
  for (int i = 0; i <= N; ++i) C[i][0] =
     C[i][j] = 1;
  for (int i = 1; i <= N; ++i)
    for (int j = 1; j < i; ++j)
      C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
```

## 31. Longest Common Subsequence

```
int lcs(char[] s, char[] t) {
  int m = s.length;
  int n = t.length;

  if (m == 0 || n == 0)
    return 0;

  int[][] dp = new int[m + 1][n + 1];
  for (int i = 0; i <= m; ++i)
    dp[i][0] = 0;
  for (int j = 1; j <= n; ++j)
    dp[0][j] = 0;
  for (int i = 0; i < m; ++i)
    for (int j = 0; j < n; ++j)
      if (s[i] == t[j])
        dp[i + 1][j + 1] = dp[i][j] + 1;
      else
        dp[i + 1][j + 1] = Math.max(dp[i +
           1][j], dp[i][j + 1]);
```

```
  return dp[m][n];
}
```

## 32. Center of Mass of a Polygon P

$$C_x = \frac{\int\int_R x \mathrm{d}A}{M} = \frac{1}{6M}\sum_{i=1}^{n}(y_{i+1} - y_i)(x_{i+1}^2 + x_{i+1} \cdot x_i + x_i^2) \quad (1)$$

$$C_y = \frac{\int\int_R y \mathrm{d}A}{M} = \frac{1}{6M}\sum_{i=1}^{n}(x_i - x_{i+1})(y_{i+1}^2 + y_{i+1} \cdot y_i + y_i^2) \quad (2)$$

Where $M$ is the polygon area.