Formulaire BAPC 2013

Team UCooL
Auteurs : François Aubry, Guillaume Derval, Benoît Legat, Anthony Gégo.

Table des matières

1 Remarques

1.1 Attention !

1. Lire **TOUS** les énoncés avant de commencer la moindre implémentation
2. Faire attention au copier-coller bête et méchant.
3. Surveiller les overflow. Parfois, un long peut régler pas mal de problèmes

1.2 Opérations sur les bits

1. Vérification parité de n : `(n & 1) == 0`
2. 2^n : `1 << n`.
3. Tester si le $i$ème bit de n est 0 : `(n & 1 << i) != 0`
4. Mettre le $i$ème bit de n à 0 : `n &= ~(1 << i)`
5. Mettre le $i$ème bit de n à 1 : `n |= (1 << i)`
6. Union : `a | b`
7. Intersection : `a & b`
8. Soustraction bits : `a & ~b`
9. Vérifier si n est une puissance de 2 : `(x & (x−1) == 0)`
10. Passage au négatif : `0 x7fffffff ^n`

1.3 Table des complexités

n $\leq$	Complexité max
$[10, 11]$	$O(n!), O(n^6)$
$[15, 18]$	$O(2^n n^2)$
$[18, 22]$	$O(2^n n)$
100	$O(n^4)$
400	$O(n^3)$
$2K$	$O(n^2 \log(n))$
$10K$	$O(n^2)$
$1M$	$O(n \log(n))$
$10M$	$O(n), O(\log(n)), O(1)$

2 Graphes

2.1 Bases

— Adjacency matrix : $A[i][j] = 1$ if i is connected to j and 0 otherwise
— Undirected graph : $A[i][j] = A[j][i]$ for all i, j (i.e. $A = A^T$)
— Adjacency list : `LinkedList<Integer>[] g;` $g[i]$ stores all neightboors of i
— Useful alternatives :

```
HashSet<Integer >[] g; // for edge deletion
HashMap<Integer , Integer >[] g; // for weighted
      graphs
```

— Classes de base (à adapter, les notations changent)

```
class Vertex implements Comparable<Vertex>
{
  int i; long d;
  public Vertex(int i, long d)
  {
    this.i = i; this.d = d;
  }
  public int compareTo(Vertex o)
  {
    return d < o.d ? -1 : d > o.d ? 1 : 0;
  }
}

class Edge implements Comparable<Edge>
{
  int o, d, w;
  public Edge(int o, int d, int w)
  {
    this.o = o; this.d = d; this.w = w;
  }
  public int compareTo(Edge o)
  {
    return w - o.w;
  }
}
```

2.2 BFS (Parcours en largeur)

Calcule à partir d'un graphe g et d'un noeud v un vecteur d t.q. $d[u]$ réprésente le nombre d'arète min. à parcourir pour arrive au noeud u.
$d[v] = 0$, $d[u] = \infty$ si u injoignable. Si $(u, w) \in E$ et $d[u]$ connu et $d[w]$ inconnu, alors $d[w] = d[u] + 1$.

```
int[] bfsVisit(LinkedList<Integer>[] g, int v, int c
    []) //c is for connected components only
{
  Queue<Integer> Q = new LinkedList<Integer>();
  Q.add(v);
  int[] d = new int[g.length];
  c[v]=v; //for connected components
  Arrays.fill(d, Integer.MAX_VALUE);
  // set distance to origin to 0
  d[v] = 0;
  while(!Q.isEmpty())
  {
    int cur = Q.poll();
    // go over all neighboors of cur
    for(int u : g[cur])
    {
      // if u is unvisited
      if(d[u] == Integer.MAX_VALUE) //or c[u] == -1
    if we calculate connected components
      {
        c[u] = v; //for connected components
        Q.add(u);
        // set the distance from v to u
        d[u] = d[cur] + 1;
      }
    }
  }
  return d;
}
```

2.2.1 Composantes connexes

```
int[] bfs(LinkedList<Integer>[] g)
{
  int[] c = new int[g.length];
  Arrays.fill(c, -1);
  for(int v = 0; v < g.length; v++)
    if(c[v] == -1)
      bfsVisit(g, v, c);
  return c;
}
```

2.2.2 Vérifier Biparticité (Bicolorabilité)

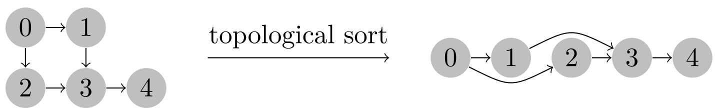
```
boolean isBipartite(LinkedList<Integer>[] g)
{
  int[] d = bfs(g);
  for(int u = 0; u < g.length; u++)
    for(Integer v: g[u])
      if((d[u]%2)!=(d[v]%2)) return false;
  return true;
}
```

2.3 DFS (Parcours en profondeur)

Soit = BFS avec *Stack* à la place de *Queue* ou implémentation récursive hyper-simple. Complexité $O(|V| + |E|)$

```
int UNVISITED = 0, OPEN = 1, CLOSED = 2;
boolean cycle; // true iff there is a cycle

void dfsVisit(LinkedList<Integer>[] g, int v,int[]
    label)
{
  label[v] = OPEN;
  for(int u : g[v])
  {
    if(label[u] == UNVISITED)
      dfsVisit(g, u, label);
    if(label[u] == OPEN)
      cycle = true;
  }
  label[v] = CLOSED;
}

void dfs(LinkedList<Integer>[] g)
{
  int[] label = new int[g.length];
  Arrays.fill(label, UNVISITED);
  cycle = false;
  for(int v = 0; v < g.length; v++)
    if(label[v] == UNVISITED)
      dfsVisit(g, v, label);
}
```

2.3.1 Ordre topologique



Le graphe doit être acyclique. On modifie légèrement DFS :

```
Stack<Integer> toposort; // add stack to global
    variables
/* ... */
void dfs(LinkedList<Integer>[] g)
{
  /* ... */
  toposort = new Stack<Integer>();
  for(int v = 0; v < g.length; v++) { /* ... */ }
}

void dfsVisit(LinkedList<Integer>[] g, int v,int[]
    label)
{
  /* ... */
  toposort.push(v); // push vertex when closing it
  label[v] = CLOSED;
}
```

2.3.2 Composantes fortement connectées

Calculer l'ordre topologique du graphe avec les arêtes inversées, puis exécuter un BFS dans l'ordre topologique (et sans repasser par un nœud déjà fait). Les nœuds parcourus à chaque execution du BFS sont fortement connectés.

```
int[] scc(LinkedList<Integer>[] g)
{
```

```
  // compute the reverse graph
  LinkedList<Integer>[] gt = transpose(g);
  // compute ordering
  dfs(gt);
  // !! last position will contain the number of scc
    's
  int[] scc = new int[g.length + 1];
  Arrays.fill(scc, -1);
  int nbComponents = 0;
  // simulate bfs loop but in toposort ordering
  while(!toposort.isEmpty())
  {
    int v = toposort.pop();
    if(scc[v] == -1)
    {
      nbComponents++;
      bfsVisit(g, v, scc);
    }
  }
  scc[g.length] = nbComponents;
  return scc;
}
```

2.4 Arbre de poids minimum (Prim)

On ajoute toujours l'arète de poids minimal parmit les noeuds déja visités.

```
double mst(LinkedList<Edge>[] g)
{
  boolean[] inTree = new boolean[g.length];
  PriorityQueue<Edge> PQ = new PriorityQueue<Edge>()
    ;
  // add 0 to the tree and initialize the priority
    queue
  inTree[0] = true;
  for(Edge e : g[0]) PQ.add(e);
  double weight = 0;
  int size = 1;
  while(size != g.length)
  {
    // poll the minimum weight edge in PQ
    Edge minE = PQ.poll();
    // if its endpoint in not in the tree, add it
    if(!inTree[minE.dest])
    {
      // add edge minE to the MST
      inTree[minE.dest] = true;
      weight += minE.w;
      size++;
      // add edge leading to new endpoints to the PQ
      for(Edge e : g[minE.dest])
        if(!inTree[e.dest]) PQ.add(e);
    }
  }
  return weight;
}
```

2.5 Dijksta

Plus court chemin d'un noeud v à tout les autres. Le graphe doit être sans cycles de poids négatif.

```
double[] dijkstra(LinkedList<Edge>[] g, int v)
{
  double[] d = new double[g.length];
  Arrays.fill(d, Double.POSITIVE_INFINITY);
  // initialize distance to v and the priority queue
  d[v] = 0;
  PriorityQueue<Edge> PQ = new PriorityQueue<Edge>()
    ;
  for(Edge e : g[v])
    PQ.add(e);
  while(!PQ.isEmpty())
  {
    // poll minimum edge from PQ
    Edge minE = PQ.poll();
    if(d[minE.dest] == Double.POSITIVE_INFINITY)
    {
      // set the distance to the new found endpoint
      d[minE.dest] = minE.w;
      for(Edge e : g[minE.dest])
      {
        // add to the queue all edges leaving the
  new
        // endpoint with the increased weight
        if(d[e.dest] == Double.POSITIVE_INFINITY)
          PQ.add(new Edge(e.orig, e.dest, e.w + d[e.
  orig]));
      }
    }
  }
  return d;
}
```

2.6 Bellman-Ford

Plus court chemin d'un noeud v à tout les autres. Le graphe peut avoir des cycles de poids négatif, mais alors l'algorithme ne retourne pas les chemins les plus courts, mais retourne l'existence de tels cycles.
$d[i][u]$ = shortest path from v to u with $\leq i$ edge
$d[0][v] = 0$
$d[0][u] = \infty \quad$ for $u \neq v$
$d[i][u] = \min\{d[i-1][u], \quad \min_{(s,u)\in E} d[i-1][s] + w(s,u)\}$
Si pas de cycle, la solution est dans $d[|V|-1]$. Si cycle il y a, $d[|V|-1] = d[V]$.
$O(|V||E|)$.

```
double[] bellmanFord(LinkedList<Edge>[] gt, int v)
{
  int n = gt.length;
  double[][] d = new double[n][n];
  for(int u = 0; u < n; u++)
    d[0][u] = u == v ? 0 : Double.POSITIVE_INFINITY;
  for(int i = 1; i < n; i++)
  {
    for(int u = 0; u < n; u++)
    {
      double min = d[i - 1][u];
      for(Edge e : gt[u])
        min = Math.min(min, d[i - 1][e.dest] + e.w);
      d[i][u] = min;
    }
  }
  return d[n - 1];
}
```

2.7 Floyd-Warshall

Plus court chemin de tout les noeuds à tout les autres. Prend en argument la matrice d'adjacence. $O(|V|^3)$ en temps et $O(|V|^2)$ en mémoire.
Le graphe contient des cycles de poids négatif ssi $result[v][v] < 0$.

```
double[][] floydWarshall(double[][] A)
{
  int n = A.length;
  // initialization: base case
  double[][] d = new double[n][n];
  for(int v = 0; v < n; v++)
    for(int u = 0; u < n; u++)
      d[v][u] = A[v][u];

  for(int k = 0; k < n; k++)
    for(int v = 0; v < n; v++)
      for(int u = 0; u < n; u++)
        d[v][u] = Math.min(d[v][u], d[v][k] + d[k][u
  ]);
  return d;
```

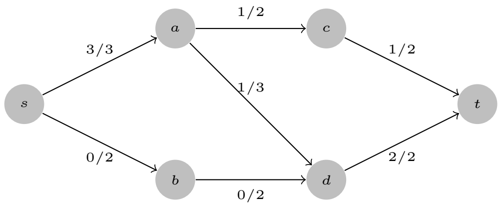
```
}
```

2.8 Flux maximum

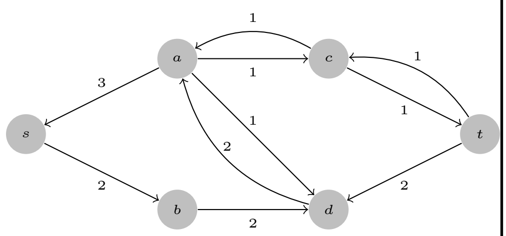
2.8.1 Bases

On cherche à calculer le flux maximum d'une source S à un puits T. Chaque arête à un débit maximum et un débit actuel (uniquement pendant la résolution). On construit le graphe résiduel comme sur les exemples.

flow graph



residual graph



L'algorithme de base fonctionne en cherchant un chemin de S à T dans le graphe résiduel.

2.8.2 Ford-Fulkerson

Si le chemin est cherché avec un DFS, la complexité est $O(|E|f^*)$ où f^* est le flux maximum. On préferera pour les problèmes l'algorithme avec un BFS (Edmonds-Karps).

2.8.3 Edmonds-Karps (BFS)

Chemin cherché avec un BFS. On a $O(|V||E|^2)$.

```
int maxFlow(HashMap<Integer, Integer>[] g, int s,
    int t)
{
  // output 0 for s = t (convention)
  if(s == t) return 0;
  // initialize maxflow
  int maxFlow = 0;
  // compute an augmenting path
  LinkedList<Edge> path = findAugmentingPath(g, s, t
    );
  // loop while augmenting paths exists and update g
  while(path != null)
  {
    int pathCapacity = applyPath(g, path);
    maxFlow += pathCapacity;
    path = findAugmentingPath(g, s, t);
  }
  return maxFlow;
}

LinkedList<Edge> findAugmentingPath(HashMap<Integer,
    Integer>[] g, int s, int t)
{
  // initialize the queue for BFS
  Queue<Integer> Q = new LinkedList<Integer>();
  Q.add(s);
  // initialize the parent array for path
    reconstruction
  Edge[] parent = new Edge[g.length];
  Arrays.fill(parent, null);
  // perform a BFS
  while(!Q.isEmpty())
  {
    int cur = Q.poll();
    for(Entry<Integer, Integer> e : g[cur].entrySet
    ())
    {
      int next = e.getKey();
      int w = e.getValue();
      if(parent[next] == null)
      {
        Q.add(next);
        parent[next] = new Edge(cur, next, w);
      }
    }
  }
  // reconstruct the path
  if(parent[t] == null) return null;
  LinkedList<Edge> path = new LinkedList<Edge>();
  int cur = t;
  while(cur != s)
  {
    path.add(parent[cur]);
    cur = parent[cur].orig;
  }
  return path;
}

int applyPath(HashMap<Integer, Integer>[] g,
    LinkedList<Edge> path)
{
  int minCapacity = Integer.MAX_VALUE;
  for(Edge e : path)
    minCapacity = Math.min(minCapacity, e.w);
  for(Edge e : path)
  {
    // treat path edge
    if(minCapacity == e.w)
    {
      // the capacity became 0, remove edge
      g[e.orig].remove(e.dest);
    }
    else
    {
      // there remains capacity, update capacity
      g[e.orig].put(e.dest, e.w - minCapacity);
    }
    // treat back edge
    Integer backCapacity = g[e.dest].get(e.orig);
    if(backCapacity == null)
    {
      // the back edge does not exist yet
      g[e.dest].put(e.orig, minCapacity);
    }
    else
    {
      // the back edge already exists, update
    capacity
      g[e.dest].put(e.orig, backCapacity+minCapacity
    );
    }
  }
  return minCapacity;
}
```

2.8.4 Coupe minimale

On cherche, avec deux noeuds s et t, V_1 et V_2 tel que $s \in V_1$, $t \in V_2$ et $\sum_{e \in E(V_1,V_2)} w(e)$ minimum.
Il suffit de calculer le flot maximum entre s et t et d'appliquer un parcours du graphe résiduel depuis s(BFS par exemple). Tout les noeuds ainsi parcourus sont dans V_1, les autres dans V_2. Le poids de la coupe est le flot maximum.

3 Programmation dynamique

3.1 Bottom-up

Répartir pour 3 personnes n objets de valeurs $v[i]$ tel que $\max_i V_i - \min_i V_i$ est minimum (V_i est la valeur totale pour la personne i).

$canDo[i][v_1][v_2] = 1$ si on peut donner les objets $0, 1, \ldots, i$ tel que v_1 va à P_1 et v_2 va à P_2, 0 sinon. v_3 déterminé à partir de la somme.

Cas de base $i = 0$:
- $canDo[0][0][0] = 1$
- $canDo[0][v[0]][0] = 1$
- $canDo[0][0][v[0]] = 1$

Cas $i \geq 1$:
$canDo[i][v_1][v_2] =$
$canDo[i-1][v_1][v_2] \lor$
$canDo[i-1][v_1 - v[i]][v_2] \lor$
$canDo[i-1][v_1][v_2 - v[i]]$

Sol. : $\min_{v_1, v_2 : canDo[n-1][v_1][v_2]} [max(v_1, v_2, S - v_1 - v_2) - min(v_1, v_2, S - v_1 - v_2)]$

```java
int solveDP() {
  boolean[][][] canDo = new boolean[v.length][sum + 1][sum + 1];
  // initialize base cases
  canDo[0][0][0] = true;
  canDo[0][v[0]][0] = true;
  canDo[0][0][v[0]] = true;
  // compute solutions using recurrence relation
  for(int i = 1; i < v.length; i++) {
    for(int a = 0; a <= sum; a++) {
      for(int b = 0; b <= sum; b++) {
        boolean giveA = a - v[i] >= 0 && canDo[i - 1][a - v[i]][b];
        boolean giveB = b - v[i] >= 0 && canDo[i - 1][a][b - v[i]];
        boolean giveC = canDo[i - 1][a][b];
        canDo[i][a][b] = giveA || giveB || giveC;
      }
    }
  }
  // compute best solution
  int best = Integer.MAX_VALUE;
  for(int a = 0; a <= sum; a++) {
    for(int b = 0; b <= sum; b++) {
      if(canDo[v.length - 1][a][b]) {
        best = Math.min(best, max(a, b, sum - a - b) - min(a, b, sum - a - b));
      }
    }
  }
  return best;
}
```

3.2 Top-down

Même problème que bottom-up. Idée principale : mémoisation (On retient les résultats intermédiaires).

```java
int solve(int i, int a, int b) {
  if(i == n) {
    memo[i][a][b] = max(a, b, sum - a - b) - min(a, b, sum - a - b);
    return memo[i][a][b];
  }
  if(memo[i][a][b] != null) {
    return memo[i][a][b];
  }
  int giveA = solve(i + 1, a + v[i], b);
  int giveB = solve(i + 1, a, b + v[i]);
  int giveC = solve(i + 1, a, b);
  memo[i][a][b] = min(giveA, giveB, giveC);
  return memo[i][a][b];
}
```

3.3 Problème du sac à dos (Knapsack)

On a n objets de valeurs $v[i]$ et de poids $w[i]$, un entier W, on veut :
- Maximiser $\sum_i x[i]v[i]$
- Avec $\sum_i x[i]w[i] \leq W$ où $x[i]$= 0 (pas pris) ou 1 (pris)

3.3.1 Un exemplaire de chaque

best[i][w]= meilleur façon de prendre les objets $0, 1, \ldots, i$ dans sac à dos de capacité w.

Cas de base :
- $best[0][w] = v[0]$ si $w[0] \leq w$
- 0 sinon

Autres cas :
$best[i][w] = \max\{best[i-1][w], best[i-1][w - w[i]] + v[i]\}$

3.3.2 Plusieurs exemplaires de chaque

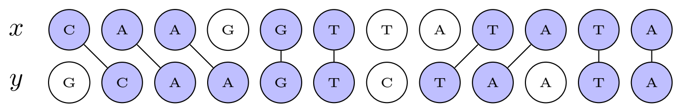
- $best[0] = 0$
- $best[w] = \max_{i : w[i] < w}\{best[w - w[i]] + v[i]\}$

3.3.3 Plusieurs knapsack

$best[i][w_1][w_2]$ = meilleur façon de prendre les objets $0, 1, \ldots, i$ dans des sacs de capacités w_1 et w_2.

3.4 Longest common subsequence (LCS)

Soit deux String x et y. Trouver la sous-séquence commune la plus longue entre x et y.



- **Formulation :** $lcs[i][j]$ = taille de $LCS(x[0]x[1]\cdots x[i-1], y[0]y[1]\cdots y[j-1])$
- **Cas de base :** $lcs[0][j] = 0$ $\quad lcs[i][0] = 0$
- **Autres cas :**
  - Si $x[i-1] = y[i-1]$ alors : $lcs[i][j] = 1 + lcs[i-1][j-1]$
  - Si $x[i-1] \neq y[i-1]$ alors : $lcs[i][j] = \max\{lcs[i-1][j], lcs[i][j-1]\}$

3.5 Matrix Chain Multiplication (MCM)

Soit une liste de matrices, trouver l'ordre qui minimise le nombres de multiplications pour calculer leur produit.
- Nombre pour multiplier une matrice $n \times m$ par une $m \times r$: $n \cdot m \cdot r$.
- Exemple : A : 10×30, B : 30×5 et C : 5×60.
  - Pour $(AB)C$: $10 \cdot 30 \cdot 5 + 10 \cdot 5 \cdot 60 = 4500$ multiplications ;
  - Pour $A(BC)$: $30 \cdot 5 \cdot 60 + 10 \cdot 30 \cdot 60 = 27000$ multiplications.
- **Formulation :** $best[i][j]$ = min cost to multiply $A_i, \ldots, A_j$
- **Cas de base :** $best[i][i] = 0$
- **Autres cas :**

$$best[i][j] = \min_{i \leq k < j} best[i][k] + best[k+1][j] + A_i.n_1 \times A_k.n_2 \times A_j.n_2$$

3.5.1 MCM généralisé

Soit une liste d'objets $x[0], \ldots, x[n-1]$ et une opération $\odot$ avec un coût associé, trouver l'ordre dans lequel effectuer les opérations pour minimiser le coût total. La multiplication des matrices est remplacée par $\odot$.

$$best[i][j] = \min_{i \le k < j} best[i][k] + best[k+1][j] + cost(i, j, k)$$

$cost(i, j, k)$ est le coût de $(x[i] \odot \cdots \odot x[k]) \odot (x[k+1] \odot \cdots \odot x[j])$.

```
int bestParenthesize() {
  int n = x.length; // x is a global variable
  int[][] best = new int[n][n];
  for(int i = 0; i < n; i++) {
    best[i][i] = 0;
  }
  for(int l = 1; l <= n; l++) {
    for(int i = 0; i < n - l; i++) {
      int j = i + l;
      int min = Integer.MAX_VALUE;
      for(int k = i; k < j; k++) {
        min = Math.min(min, best[i][k] + best[k + 1][j] + cost(i, j, k)); // cost is problem-independent
      }
      best[i][j] = min;
    }
  }
  return best[0][n - 1];
}
```

3.6 Edit distance

On a deux String x et y, en effectuant des opérations sur x, on veut obtenir le coût minimum pour transformer x en y. On peut (coût opération) :

1. Enlever un caractère (D=1)
2. Insérer un caractère (I=1)
3. Remplacer un caractère (R=2)

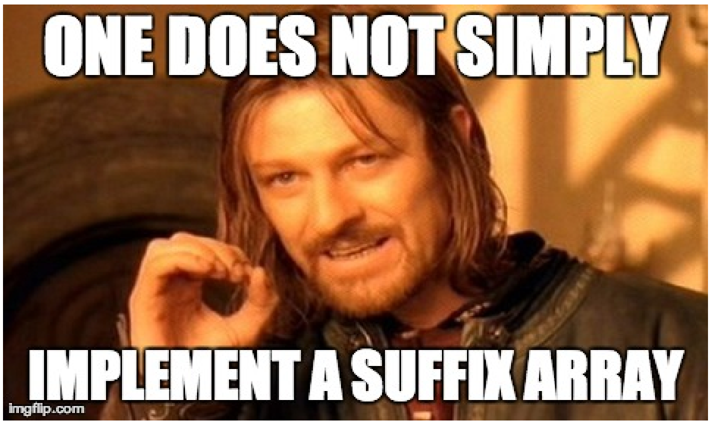
— **Formulation** :*editDist*$[i][j]$= coût min. pour transformer $x_0 \cdots x_{i-1}$ en $y_0 \cdots y_{j-1}$
— **Cas de base** :
$editDist[i][0] = i \cdot D \qquad editDist[0][j] = j \cdot I$
— **Autres cas** :

$$editDist[i][j] = \min \begin{array}{l} editDist[i-1][j] + D, \\ editDist[i][j-1] + I, \\ editDist[i-1][j-1] + R^* \end{array}$$

où $R^* = R$ si $x[i-1] \neq y[j-1]$ et 0 sinon.

```
int editDistance(String txt1, String txt2, int I,
    int D, int R){
  int[][] d = new int[txt1.length()+1][txt2.length()
    +1];
  for(int i=0; i <= txt1.length(); i++)
    d[i][0]=i*D;
  for(int j=0; j <= txt2.length(); j++)
    d[0][j]=j*I;
  for(int i=1; i <= txt1.length(); i++){
    for(int j=1; j <= txt2.length(); j++){
      int cost;
      // Evaluation du cout de non-egalite d'un
    caractere
      if(txt1.charAt(i-1)==txt2.charAt(j-1))
        cost = 0;
      else
        cost = R;
      // Suppression, insertion, substitution
      d[i][j] = Math.min(Math.min(d[i-1][j] + D, d[i
    ][j-1] + I), d[i-1][j-1] + cost);
    }
  }
  // Le dernier element calcule est la distance
  return d[txt1.length()][txt2.length()];
}
```

3.7 Suffix array



— Suffix array de *algorithm* = tableau trié des suffixes. Exemple :algorithm, gorithm, hm, ithm, lgorithm, m, orithm, rithm, thm
— Caractérisé par son index de départ
Exemple : Suffix array de *algorithm* :

$$[0,\ 2,\ 7,\ 5,\ 1,\ 8,\ 3,\ 4,\ 6]$$

Exemple : Soit suf_j le suffixe commençant à l'index j. Soit $C(i, j, k)$ le résultat de la comparaison de suf_j et suf_k sur les 2^i premiers caractères.

$$C(i,j,k) = \begin{cases} C(i-1,j,k) & \text{si } C(i-1,j,k) \neq 0 \\ C(i-1, j+2^{i-1}, k+2^{i-1}) & \text{sinon} \end{cases}$$

— On définit une matrice *so* telle que :

$$so[i][j] = so[i][k] \Leftrightarrow C(i,j,k) = 0$$
$$so[i][j] < so[i][k] \Leftrightarrow C(i,j,k) < 0$$
$$so[i][j] > so[i][k] \Leftrightarrow C(i,j,k) > 0$$

$so[i]$ est l'ordre des suffixes triés sur les 2^i premiers caractères.
— **Cas de base :** $so[0][j] = (int)s.charAt(i)$
Exemple : pour $s = ccacab$ on a
$s[0] = [97, 97, 95, 97, 95, 96]$
— Pour chaque j on définit un triplet (l, r, j) :

$$\begin{cases} (s[i-1][j], s[i-1][j+2^{i-1}], j) & \text{si } j + 2^{i-1} < n \\ (s[i-1][j], -1, j) & \text{si } j + 2^{i-1} \geq n \end{cases}$$

```
class Triple implements Comparable<Triple> {
  int l, r, index;
  public Triple(int half1, int half2, int index) {
    this.l = half1;
    this.r = half2;
    this.index = index;
  };
  public int compareTo(Triple other) {
```

```java
    if(l != other.l) {
      return l - other.l;
    }
    return r - other.r;
  }
}


int[][] suffixOrder(String s) {
  int n = s.length();
  int lg = (int)Math.ceil((Math.log(n) / Math.log(2)
    )) + 1;
  int[][] so = new int[lg][n];
  // initialize so[0] with character order
  for(int i = 0; i < n; i++) {
    so[0][i] = s.charAt(i);
  }
  Triple[] next = new Triple[n];
  for(int i = 1; i < lg; i++) {
    // build the next array
    for(int j = 0; j < n; j++) {
      int k = j + (1 << (i - 1));
      next[j] = new Triple(so[i - 1][j], k < n ? so[
  i - 1][k] : -1, j);
    }
    // sort next array
    Arrays.sort(next);
    // build so[i]
    for(int j = 0; j < n; j++) {
      if(j == 0) {
      // smallest elements gets value 0
      so[i][next[j].index] = 0;
     } else if(next[j].compareTo(next[j - 1]) == 0)
    {
      // equal to previous so it gets the same value
      so[i][next[j].index] = so[i][next[j - 1].index
    ];
     } else {
      // largest than previous so get + 1
      so[i][next[j].index] = so[i][next[j - 1].index
    ] + 1;
     }
    }
  }
 return so;
}

//Calcule le Suffix Array pour un so donne:
int[] suffixArray(int[][] so) {
  int[] sa = new int[so[0].length];
  for(int j = 0; j < so[0].length; j++) {
    sa[so[so.length - 1][j]] = j;
  }
  return sa;
}

//Retourne le plus long prefixe commun de suf_j (le
    suffixe de s commencant a j = s.substr(j)) et
    suf_k pour un so donne:
int lcp(int[][] so, int j, int k) {
  int lcp = 0;
  for(int i = so.length - 1; i >= 0; i--) {
    if(so[i][j] == so[i][k]) {
      lcp += (1 << i);
      j += (1 << i);
      k += (1 << i);
    }
  }
  return lcp;
}

//Quelques exemples
String maxStrRepeatedKTimes(String s, int k) {
  int[][] so = suffixOrder(s);
  int[] SA = suffixArray(so);
  int n = s.length();
  int max = Integer.MIN_VALUE;
  int j = 0;
  for(int i = 0; i <= n - k; i++) {
    int lcp = lcp(so, SA[i], SA[i + k - 1]);
    if(lcp > max) {
      max = lcp;
      j = SA[i];
    }
  }
  return s.substring(j, j + max);
}

String minLexicographicRotation(String s) {
  int n = s.length();
  s += s;
  int[] SA = suffixArray(suffixOrder(s));
  int i = 0;
  while(!(0 <= SA[i] && SA[i] < n)) {
    i++;
  }
  return s.substring(SA[i], SA[i] + n);
}

class MaxLexConc implements Comparator<String> {
 public int compare(String x, String y) {
    String xy = x + y;
    String yx = y + x;
    if(xy.compareTo(yx) < 0 ||
      (xy.equals(yx) && x.length()<y.length())) {
      return 1;
    }
    return -1;
}
```

4 Géométrie

Attention aux arrondis. Définir E en fonction du problème.

```java
boolean eq(double a, double b){ return Math.abs(a -
    b) <= E; }
boolean le(double a, double b) { return a < b - E; }
boolean leq(double a, double b) { return a <= b + E;
    }
```

4.1 Points *non-testé*

```java
public static class Point
{
  double x, y;
}

boolean eq(Point p1, Point p2) { return eq(p1.x, p2.
    x) && eq(p2.y, p2.y); }
Point subtract(Point p0, Point p1) { return new
    Point(p0.x - p1.x, p0.y - p1.y); }

class horizontalComp implements Comparator<Point>
{
  public int compare(Point a, Point b)
  {
    if(a.x < b.x) return -1;
    if(a.x > b.x) return 1;
    if(a.y < b.y) return -1;
    if(a.y > b.y) return 1;
    return 0;
  }
}
```

4.1.1 Ordonner selon angle *non-testé*

```java
LinkedList<Point> sortPolar(Point[] P, Point o)
{
  LinkedList<Point> above = new LinkedList<Point>();
  LinkedList<Point> samePos = new LinkedList<Point
    >();
  LinkedList<Point> sameNeg = new LinkedList<Point
    >();
  LinkedList<Point> bellow = new LinkedList<Point>()
    ;
  for(Point p : P)
  {
    if(p.y > o.y)
```

```
            above.add(p);
        else if(p.y < o.y)
            bellow.add(p);
        else
        {
            if(p.x < o.x)
                sameNeg.add(p);
            else
                samePos.add(p);

        }
    }
    PolarComp comp = new PolarComp(o);
    Collections.sort(samePos, comp);
    Collections.sort(sameNeg, comp);
    Collections.sort(above, comp);
    Collections.sort(bellow, comp);
    LinkedList<Point> sorted = new LinkedList<Point>()
        ;
    for(Point p : samePos) sorted.add(p);
    for(Point p : above) sorted.add(p);
    for(Point p : sameNeg) sorted.add(p);
    for(Point p : bellow) sorted.add(p);
    return sorted;
}

class PolarComp implements Comparator<Point>
{
    Point o;
    public PolarComp(Point o)
    {
        this.o = o;
    }
    @Override
    public int compare(Point p0, Point p1)
    {
        double pE = prodE(subtract(p0,o), subtract(p1,o)
        );
        if(pE < 0)
            return 1;
        else if(pE > 0)
            return -1;
        else
            return Double.compare(squareDist(p0, o),
        squareDist(p1, o));
    }
}
```

4.1.2 Paire de points la plus proche *non-testé*

```
double closestPair(Point[] points)
{
    if(points.length == 1) return 0;
    Arrays.sort(points, new horizontalComp());
    double min = distance(points[0], points[1]);
    int leftmost = 0;
    SortedSet<Point> candidates = new TreeSet<Point>(
        new verticalComp());
    candidates.add(points[0]);
    candidates.add(points[1]);
    for (int i = 2; i < points.length; i++)
    {
        Point cur = points[i];
        while (cur.x - points[leftmost].x > min)
        {
            candidates.remove(points[leftmost]);
            leftmost++;
        }
        Point low = new Point(cur.x-min, (int)(cur.y-min
        ));
        Point high = new Point(cur.x, (int)(cur.y+min));
        for(Point point:candidates.subSet(low,high))
        {
            double d = distance(cur, point);
            if (d < min)
                min = d;
        }
        candidates.add(cur);
    }
    return min;
}
```

4.2 Lignes *non-testé*

```
class Line
{
    double a;
    double b;
    double c;
    public Line(double a, double b, double c)
    {
        this.a = a;
        this.b = b;
        this.c = c;
    }
    public Line(Point p1, Point p2) {
        if(p1.x == p2.x) {
            a = 1;
            b = 0;
            c = -p1.x;
        } else {
            b = 1;
            a = -(p1.y - p2.y) / (p1.x - p2.x);
            c = -(a * p1.x) - (b * p1.y);
        }
    }
    public Line(Point p, double m) {
        a = -m;
        b = 1;
        c = -((a*p.x) + (b*p.y));
    }
}

boolean areParallel(Line l1, Line l2) {
    return (eq(l1.a, l2.a) && eq(l1.b, l2.b));
}

boolean areEqual(Line l1, Line l2) {
    return areParallel(l1, l2) && eq(l1.c, l2.c);
}

boolean contains(Line l, Point p) {
    return eq(l.a*p.x + l.b*p.y + l.c, 0);
}

Point intersection(Line l1, Line l2) {
    if(areEqual(l1, l2) || areParallel(l1, l2)) {
        return null;
    }
    double x = (l2.b * l1.c - l1.b * l2.c) /
            (l2.a * l1.b - l1.a * l2.b);
    double y;
    if(Math.abs(l1.b) > E) {
        y = -(l1.a * x + l1.c) / l1.b;
    } else {
        y = -(l2.a * x + l2.c) / l2.b;
    }
    return new Point(x, y);
}

double angle(Line l1, Line l2) {
    double tan = (l1.a * l2.b - l2.a * l1.b) /
        (l1.a * l2.a + l1.b * l2.b);
    return Math.atan(tan);
}

Line getPerp(Line l, Point p) {
    return new Line(p, 1 / l.a);
}

Point closest(Line l, Point p) {
    double x;
    double y;
    if(isVertical(l)) {
        x = -l.c;
        y = p.y;
        return new Point(x, y);
    }
    if(isHorizontal(l)) {
        x = p.x;
```

```
    y = -l.c;
    return new Point(x, y);
  }
  Line perp = getPerp(l, p);
  return intersection(l, perp);
}

boolean isVertical(Line l) {
  return eq(l.b, 0);
}

boolean isHorizontal(Line l) {
  return eq(l.a, 0);
}
```

4.3 Segments *non-testé*

```
boolean onSegment(Segment s, Point p) {
  return Math.min(s.p1.x, s.p2.x) <= p.x &&
         Math.max(s.p1.x, s.p2.x) >= p.x &&
         Math.min(s.p1.y, s.p2.y) <= p.y &&
         Math.max(s.p1.y, s.p2.y) >= p.y;
}

double direction(Segment s, Point p) {
  return prodE(subtract(p,s.p1), subtract(s.p2,s.p1)
    );
}

boolean intersects(Segment s1, Segment s2) {
  double d1 = direction(s2, s1.p1);
  double d2 = direction(s2, s1.p2);
  double d3 = direction(s1, s2.p1);
  double d4 = direction(s1, s2.p2);
  if(((d1 > 0 && d2 < 0) || (d1 < 0 && d2 > 0)) &&
      ((d3 > 0 && d4 < 0) || (d3 < 0 && d4 > 0))) {
    return true;
  } else if(eq(d1, 0) && onSegment(s2, s1.p1)) {
    return true;
  } else if(eq(d2, 0) && onSegment(s2, s1.p2)) {
    return true;
  } else if(eq(d3, 0) && onSegment(s1, s2.p1)) {
    return true;
  } else if(eq(d4, 0) && onSegment(s1, s2.p2)) {
    return true;
  }
  return false;
}

boolean segmentIntersection(Segment[] S) {
  Point[] P = new Point[S.length * 2];
  for(int i = 0; i < S.length; i++) {
    S[i].p1.i = i; S[i].p1.isLeft = true;
    S[i].p2.i = i; S[i].p2.isLeft = false;
  }
  int j = 0;
  for(Segment s : S) {
    P[j++] = s.p1;
    P[j++] = s.p2;
  }
  Arrays.sort(P, new SegIntPointComp());
  SegmentComp comp = new SegmentComp();
  TreeSet&lt;Segment&gt; T = new TreeSet&lt;Segment&gt;(comp);
  for(int i = 0; i < P.length; i++) {
    Segment s = S[P[i].i];
    if(P[i].isLeft) {
      comp.x = P[i].x;
      T.add(s);
      Segment above = T.higher(s);
      Segment bellow = T.lower(s);
      if((above != null && intersects(above, s)) ||
          (bellow != null && intersects(bellow, s)))
      {
        return true;
      }
    } else {
      Segment above = T.higher(s);
      Segment bellow = T.lower(s);
      if(above != null && bellow != null &&
        intersects(above, bellow)) {
      return true;
      }
      T.remove(s);
    }
  }
  return false;
}

class SegIntPointComp implements Comparator<Point> {
  @Override
  public int compare(Point p0, Point p1) {
    int xc = Double.compare(p0.x, p1.x);
    if(xc == 0) {
      if(p0.isLeft && !p1.isLeft) {
        return -1;
      }
      if(!p0.isLeft && p1.isLeft) {
    return 1;
      } else {
    return Double.compare(p0.y, p1.y);
      }
    }
    return xc;
  }
}

class SegmentComp implements Comparator&lt;Segment&gt; {
  double x;
  @Override
  public int compare(Segment s1, Segment s2) {
    if(s1.p1.i == s2.p1.i && s1.p2.i == s2.p2.i) {
      return 0;
    }
    Segment toAdd = null;
    Segment o = null;
    if(eq(s1.p1.x, x)) {
      toAdd = s1;
      o = s2;
    } else if(eq(s2.p1.x, x)){
      toAdd = s2;
      o = s1;
    } else {
      return 0;
    }
    double y = Math.min(o.p1.y, o.p2.y);
    Segment v = new Segment(new Point(x, y),
                            toAdd.p1);
    if(eq(s1.p1.x, x)) {
      if(intersects(v, o)) {
        return 1;
      } else {
        return -1;
      }
    } else if(eq(s2.p1.x, x)) {
   if(intersects(v, o)) {
      return -1;
     } else {
      return 1;
    }
    }
    return 0;
  }
}

// r > 0: a droite, r < 0: a gauche, r==0:
    colineiare
public static int positionFromSegment(Point
    segmentFrom, Point segmentTo, Point p)
{
  //Cross product of vectors segmentFrom->segmentTo
    and segmentFrom->p
  return (segmentTo.x-segmentFrom.x)*(p.y-
    segmentFrom.y)-(segmentTo.y-segmentFrom.y)*(p.x-
    segmentFrom.x);
}
```

4.4 Triangles *non-testé*

Loi des sinus : $\frac{a}{\sin(A)} = \frac{b}{\sin(B)} = \frac{c}{\sin(C)} = 2r$ Loi des cosinus :
$a^2 = b^2 + c^2$ˇ$2bc\cos(A)$
$b^2 = a^2 + c^2$ˇ$2ac\cos(B)$
$c^2 = a^2 + b^2$ˇ$2ab\cos(C)$
Formule de Héron : Aire= $\sqrt{(s-a)(s-b)(s-c)}$ avec $s = \frac{a+b+c}{2}$

```
class Triangle
{
  Segment a, b, c;
  public Triangle(Segment a,Segment b,Segment c)
  {
    this.a = a;
    this.b = b;
    this.c = c;
  }
  public Triangle(Point p1, Point p2, Point p3)
  {
    a = new Segment(p1, p2);
    b = new Segment(p1, p3);
    c = new Segment(p2, p3);
  }
}

//Triangle degenere si result==0
//Sinon, si result>0, dans le sens de a.
//Sinon, -a.
double signedTriangleArea(Triangle t)
{
  return (t.p1.x * t.p2.y - t.p1.y * t.p2.x +
          t.p1.y * t.p3.x - t.p1.x * t.p3.y +
          t.p2.x * t.p3.y - t.p3.x * t.p2.y) / 2.0;
}

double triangleArea(Triangle t)
{
  return Math.abs(signedTrinangleArea(t));
}

boolean isInTriangle(Point p, Triangle t)
{
  Triangle a = new Triangle(p, t.p1, t.p2);
  Triangle b = new Triangle(p, t.p1, t.p3);
  Triangle c = new Triangle(p, t.p2, t.p3);
  double total = triangleArea(a) +
      triangleArea(b) +
      triangleArea(c);
  return eq(total, triangleArea(t));
}

boolean isInTriangle2(Point p, Triangle t)
{
  return !(cw(t.p1, t.p2, p) ||
           cw(t.p2, t.p3, p) ||
           cw(t.p3, t.p1, p));
}

boolean ccw(Point a, Point b, Point c)
{
  return signedTrinangleArea(new Triangle(a, b, c))>
    E;
}

boolean cw(Point a, Point b, Point c)
{
  return signedTrinangleArea(new Triangle(a, b, c))<
    E;
}

boolean collinear(Point a, Point b, Point c)
{
  return Math.abs(signedTrinangleArea(
         new Triangle(a, b, c))) <= E;
}
```

4.5 Cercles *non-testé*

Aire de l'intersection entre deux cercles de rayon r et R à une distance d : $A = r^2 \arccos(X) + R^2 \arccos(Y) - \frac{\sqrt{(Z)}}{2}$
$X = \frac{d^2+r^2-R^2}{2dr}$
$Y = \frac{d^2+R^2-r^2}{2dR}$
$Z = (-d+r+R)*(d+r-R)*(d-r+R)*(d+r+R)$

```
class Circle
{
  Point c;
  double r;
  public Circle(Point c, double r)
  {
    this.c = c;
    this.r = r;
  }
}

//Centre du cercle circonscrit
Point circumcenter(Point p1, Point p2, Point p3)
{
  if(eq(p1.x, p2.x))
    return circumcenter(p1, p3, p2);
  else if(eq(p2.x, p3.x))
    return circumcenter(p2, p1, p3);
  double ma = (p2.y - p1.y) / (p2.x - p1.x);
  double mb = (p3.y - p2.y) / (p3.x - p2.x);
  double x = (ma*mb*(p1.y - p3.y) +
              mb*(p1.x + p2.x) -
              ma*(p2.x + p3.x)) /
             (2 * mb - 2 * ma);
  double y = 0.0;
  if(eq(ma, 0)) {
    y = (-1/mb)*(x-(p2.x + p3.x)/2) +
        (p2.y+p3.y)/2;
  } else {
    y = (-1/ma)*(x-(p1.x + p2.x)/2) +
        (p1.y + p2.y)/2;
  }
  return new Point(x, y);
}

//Point d'intersection avec la tangente au cercle
    passant par le point p
Point[] tangentPoints(Point p, Circle c)
{
  double alfa = 0.0;
  if(!eq(p.x, c.c.x)) {
    alfa = Math.atan((p.y - c.c.y) /
                     (p.x - c.c.x));
    if(p.x < c.c.x) {
      alfa += Math.PI;
    }
  } else {
    alfa = Math.PI / 2;
    if(p.y < c.c.y) {
      alfa += Math.PI;
    }
  }
  double d = distance(p, c.c);
  double beta = Math.acos(c.r / d);
  double x1 = c.c.x +c.r * Math.cos(alfa + beta);
  double y1 = c.c.y +c.r * Math.sin(alfa + beta);
  double x2 = c.c.x +c.r * Math.cos(alfa - beta);
  double y2 = c.c.y +c.r * Math.sin(alfa - beta);
  return new Point[] {new Point(x1, y1),
                      new Point(x2, y2)};
}
```

4.6 Polygones *non-testé*

```
boolean turnSameSide(Point[] polygon)
{
  Point u = subtract(polygon[1], polygon[0]);
  Point v = subtract(polygon[2], polygon[1]);
  double first = prodE(u ,v);
  int n = polygon.length;
```

```java
  for(int i = 1; i < n; i++)
  {
    u = subtract(polygon[(i+1)%n], polygon[i]);
    v = subtract(polygon[(i+2)%n], polygon[(i+1)%n])
    ;
    double pe = prodE(u, v);
    if(Math.signum(first) * Math.signum(pe) < 0)
      return false;
  }
  return true;
}

boolean convex(Point[] polygon)
{
  if(!turnSameSide(polygon)) {return false;}
  int n = polygon.length;
  Point l = subtract(polygon[1], polygon[0]);
  Point r = subtract(polygon[n - 1], polygon[0]);
  Point u = subtract(polygon[1], polygon[0]);
  Point v = subtract(polygon[2], polygon[0]);
  double last = prodE(u, v);
  for(int i = 2; i < n - 1; i++)
  {
    u = subtract(polygon[i], polygon[0]);
    v = subtract(polygon[i + 1], polygon[0]);
    Point s = subtract(polygon[i], polygon[0]);
    if(between(l, s, r))
      return false;
    double pe = prodE(u, v);
    if(Math.signum(last) * Math.signum(pe) < 0)
      return false;
    last = pe;
  }
  return true;
}

double area(ArrayList<Point> polygon)
{
  double total = 0.0;
  for(int i = 0; i < polygon.size(); i++)
  {
    int j = (i + 1) % polygon.size();
    total += polygon.get(i).x * polygon.get(j).y-
        polygon.get(j).x * polygon.get(i).y;
  }
  return total / 2.0;
}

//Il faut ordonner les points dans le sens inverse
    des aiguilles d'une montre (traduit du portugais
    ...
boolean ear(int i, int j, int k, ArrayList<Point>
    polygon)
{
  int m;
  Triangle t = new Triangle(polygon.get(i),
                            polygon.get(j),
                            polygon.get(k));
  if(cw(t.p1, t.p2, t.p3))
    return false;
  for(m = 0; m < polygon.size(); m++)
    if(m != i && m != j && m != k)
      if(isInTriangle2(polygon.get(m), t))
        return false;
  return true;
}
```

4.6.1 Polygone convexe : Gift Wrapping

But : créer un polygône convexe comprenant un ensemble de points On "enroule une corde" autour des points. $O(n^2)$.

```java
public static List<Point> giftWrapping(ArrayList<
    Point> points)
{
  //Cherchons le point le plus a gauche
  Point pos = points.get(0);
  for(Point p: points)
    if(pos.x > p.x)
      pos = p;
  //L'algo proprement dit
  Point fin;
  List<Point> result = new LinkedList<Point>();
  do
  {
    result.add(pos);
    fin = points.get(0);
    for(int j = 1; j < points.size(); j++)
      if (fin == pos || positionFromSegment(pos, fin
    , points.get(j)) < 0)
        fin = points.get(j);
    pos = fin;
  } while(result.get(0) != fin);
  return result;
}
```

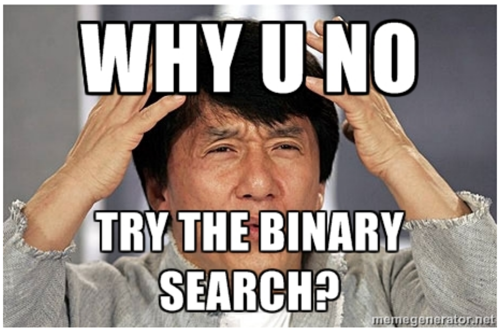
4.6.2 Polygone convexe : Graham Scan *non-testé*

Meilleure complexité (théoriquement)

```java
static Point firstP;
Point[] convexHull(Point[] in, int n) {
  Point[] hull = new Point[n];
  int i;
  int top;
  if(n <= 3) {
    for(i = 0; i < n; i++) {
      hull[i] = in[i];
    }
    return hull;
  }
  Arrays.sort(in, new leftlowerC());
  firstP = in[0];
  in=sort(Arrays.copyOfRange(in,1,in.length),in);
  hull[0] = firstP;
  hull[1] = in[1];
  top = 1;
  i = 2;
  while(i <= n) {
    if(!ccw(hull[top - 1], hull[top], in[i])) {
      top--;
    } else {
      top++;
      hull[top] = in[i];
      i++;
    }
  }
  return Arrays.copyOfRange(hull, 0, top);
}

Point[] sort(Point[] end, Point[] in) {
  Point[] res = new Point[in.length + 1];
  Arrays.sort(end, new smallerAngleC());
  int i = 1;
  for(Point p : end) {
    res[i] = p;
    i++;
  }
  res[0] = in[0];
  res[res.length - 1] = in[0];
  return res;
}

class smallerAngleC implements Comparator<Point>{
  public int compare(Point p1, Point p2) {
    if(collinear(firstP, p1, p2)) {
      if(distance(firstP, p1) <=
         distance(firstP, p2)){
        return -1;
      } else {
  return 1;
      }
    }
    if(ccw(firstP, p1, p2)) {
      return -1;
    }
    return 1;
  }
}

class leftlowerC implements Comparator<Point> {
```

```
  public int compare(Point p1, Point p2) {
    if(p1.x < p2.x) {return -1;}
    if(p1.x > p2.x) {return 1;}
    if(p1.y < p2.y) {return -1;}
    if(p1.y > p2.y) {return 1;}
    return 0;
  }
}

boolean pointInPolygon(Point[] pol, Point p) {
  boolean c = false;
  int n = pol.length;
  for(int i = 0, j = n - 1; i < n; j = i++)
  {
    double r = (pol[j].x - pol[i].x) * (p.y - pol[i
    ].y) / (pol[j].y - pol[i].y) + pol[i].x;
    if ((((pol[i].y <= p.y) && (p.y < pol[j].y)) ||
         ((pol[j].y <= p.y) && (p.y < pol[i].y))) &&
          (p.x < r))
    {
      c = !c;
    }
  }
  return c;
}
```

5 Autres



5.1 Permutations, Combinaisons, Arrangements... *non-testé*

```
void nextPerm(int[] p) {
  int n = p.length;
  int k = n - 2;
  while(k >= 0 && p[k] >= p[k + 1]) {k--;}
  int l = n - 1;
  while(p[k] >= p[l]) {l--;}
  swap(p, k, l);
  reverse(p, k + 1, n);
}

LinkedList<Integer> getIPermutation(int n, int index
    ) {
  LeftRightArray lr = new LeftRightArray(n);
  lr.freeAll();
  LinkedList<Integer> perm = new
  LinkedList<Integer>();
  getPermutation(lr, index, fact(n), perm);
  return perm;
}

void getPermutation(LeftRightArray lr, int i, long
    fact, LinkedList<Integer> perm) {
  int n = lr.size();
  if(n == 1) {
    perm.add(lr.freeIndex(0, false));
  } else {
    fact /= n;
    int j = (int)(i / fact);
    perm.add(lr.freeIndex(j, true));
    i -= j * fact;
    getPermutation(lr, i, fact, perm);
  }
}

int[] getICombinadic(int n, int k, long i) {
  int[] comb = new int[k];
  int j = 0;
  for(int z = 1; z <= n; z++) {
    if ( k == 0 ) {
      break;
    }
    long threshold = C(n - z, k - 1);
    if (i < threshold) {
      comb[j] = z - 1;
      j++;
      k = k - 1;
    } else if (i >= threshold) {
      i = i - threshold;
    }
  }
  return comb;
}

void combinations(int n, int k) {
  combinations(n, 0, new int[k], 0);
}

void combinations(int n, int j, int[] comb, int k) {
  if(k == comb.length) {
    System.out.println(Arrays.toString(comb));
  } else {
    for(int i = j; i < n; i++) {
      comb[k] = i;
      combinations(n, i + 1, comb, k + 1);
    }
  }
}
void subsets(int[] set) {
  int n = (1 << set.length);
  for(int i = 0; i < n; i++) {
    int[] sub = new int[Integer.bitCount(i)];
    int k = 0, j = 0;
    while((1 << j) <= i) {
      if((i & (1 << j)) == (1 << j)) {
        sub[k++] = set[j];
      }
      j++;
    }
    System.out.println(Arrays.toString(sub));
  }
}
```

5.2 Décomposition en fractions unitaires *non-testé*

Ecrire $0 < \frac{p}{q} < 1$ sous forme de sommes de $\frac{1}{k}$

```
void expandUnitFrac(long p, long q)
{
  if(p != 0)
  {
    long i = q % p == 0 ? q/p : q/p + 1;
    System.out.println("1/" + i);
    expandUnitFrac(p*i-q, q*i);
  }
}
```

5.3 Combinaison

Nombre de combinaison de taille k parmi n (C_n^k)
Cas spécial : $C_n^k \ mod \ 2 = n \oplus m$

```
long C(int n, int k)
{
  double r = 1;
  k = Math.min(k, n - k);
  for(int i = 1; i <= k; i++)
    r /= i;
  for(int i = n; i >= n - k + 1; i--)
    r *= i;
  return Math.round(r);
}
```

5.4 Suite de fibonacci *non-testé*

$f(0) = 0$, $f(1) = 1$ et $f(n) = f(n-1) + f(n-2)$.
On a la relation suivante, qui permet de calculer tout nombre de la suite en $O(log(n))$:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$

```
long fib(n)
{
  int i=1; int h=1; int j=0; int k=0; int t;
  while(n > 0)
  {
    if(n % 2 == 1)
    {
      t = j * h;
      j=i * h + j * k + t;
      i=i * k + t;
       }
      t = h * h;
      h = 2 * k * h + t;
      k = k * k + t;
  }
  n = (int)n / 2;
  return j;
}
```

5.5 Strings *non-testé et non-relu*

5.5.1 Palindrome maximum

```
int[] calculateAtCenters(String s) {
  int n = s.length();
  int[] L = new int[2 * n + 1];
  int i = 0, palLen = 0, k = 0;
  while(i < n) {
    if((i > palLen) &&
       (s.charAt(i - palLen - 1)==s.charAt(i))) {
      palLen += 2;
      i += 1;
      continue;
    }
    L[k++] = palLen;
    int e = k - 2 - palLen;
    boolean found = false;
    for(int j = k - 2; j > e; j--) {
      if(L[j] == j - e - 1) {
        palLen = j - e - 1;
        found = true;
        break;
      }
      L[k++] = Math.min(j - e - 1, L[j]);
    }
    if(!found) {
      i += 1;
      palLen = 1;
    }
  }
  L[k++] = palLen;
  int e = 2 * (k - n) - 3;
  for(i = k - 2; i > e; i--) {
    int d = i - e - 1;
    L[k++] = Math.min(d, L[i]);
  }
  return L;
}

String getPalindrome(String s, int[] L) {
  int max = L[0];
  int maxI = 0;
  for(int i = 1; i < L.length; i++) {
    if(L[i] > max) {
      max = L[i];
      maxI = i;
    }
  }
  int b = 0, e = 0;
  b = maxI / 2 - L[maxI] / 2;
  e = maxI / 2 + L[maxI] / 2;
  e += maxI % 2 == 0 ? 0 : 1;
  return s.substring(b, e);
}

String getPalindrome(String s)
{
  return getPalindrome(s, calculateAtCenters(s));
}
```

5.6 La réponse

```
int reponse() { return 42; }
```

5.7 Occurences dans une chaine

KMP(s,w) renvoie la position des occurences de w dans s.

```
LinkedList<Integer> KMP(String s, String w) {
  LinkedList<Integer> matches = new
  LinkedList<Integer>();
  int k = 0, i = -1;
  int[] t = KMPtable(w);
  do {
    i = KMP(s, w, k, t);
    if(i != -1) {
      matches.add(i);
      // change to i+len(w) disalow overlap
      k = i + 1;
    }
  } while(i != -1 && k < s.length());
  return matches;
}

int KMP(String s, String w, int k, int[] t) {
  int i = 0;
  int n = s.length(), m = w.length();
  while(k + i < n) {
    if(w.charAt(i) == s.charAt(k + i)) {
      i++;
      if(i == m) {return k;}
    } else {
      k += i - t[i];
      i = t[i] > -1 ? t[i] : 0;
    }
  }
  return -1;
}

int[] KMPtable(String w) {
  int m = w.length();
  int[] t = new int[m];
  int pos = 2, cnd = 0;
  t[0] = -1;
  t[1] = 0;
  while(pos < m) {
    if(w.charAt(pos - 1) == w.charAt(cnd)) {
      t[pos++] = ++cnd;
    } else if(cnd > 0) {
      cnd = t[cnd];
    } else {
      t[pos++] = 0;
    }
  }
  return t;
}
```

5.8 Algorithmes de tri *non-testé*

```
int findKth(int[] A, int k, int n) {
  if(n <= 10) {
    Arrays.sort(A, 0, n);
    return A[k];
  }
  int nG = (int)Math.ceil(n / 5.0);
  int[][] group = new int[nG][];
  int[] kth = new int[nG];
  for(int i = 0; i < nG; i++) {
    if(i == nG - 1 && n % 5 != 0) {
      group[i] = Arrays.copyOfRange(A, (n/5)* 5, n);
```

```
      kth[i] = findKth(group[i], group[i].length /
    2,
                          group[i].length);
    } else {
      group[i] = Arrays.copyOfRange(A, i*5, (i+1)*5)
    ;
      kth[i] = findKth(group[i], 2, group[i].length)
    ;
    }
  }
  int M = findKth(kth, nG / 2, nG);
  int[] S = new int[n];
  int[] E = new int[n];
  int[] B = new int[n];
  int s = 0, e = 0, b = 0;
  for(int i = 0; i < n; i++) {
    if(A[i] < M) {
      S[s++] = A[i];
    } else if(A[i] > M) {
      B[b++] = A[i];
    } else {E[e++] = A[i];}
  }
  if(k < s) {
    return findKth(S, k, s);
  } else if(k >= s + e) {
    return findKth(B, k - s - e, b);
  }
  return M;
}

int[] countSort(int[] A, int k) { // O(n + k)
  int[] C = new int[k];
  for(int j = 0; j < A.length; j++) {
    C[A[j]]++;
  }
  for(int j = 1; j < k; j++) {
    C[j] += C[j - 1];
  }
  int[] B = new int[A.length];
  for(int j = A.length - 1; j >= 0; j--) {
    B[C[A[j]] - 1] = A[j];
    C[A[j]]--;
  }
  return B;
}

int[][] radixSort(int[][] nums, int k) { // O(d*(n+k
    ))
  int n = nums.length;
  int m = nums[0].length;
  int[][] B = null;
  for(int i = m - 1; i >= 0; i--) {
    int[] C = new int[k];
    for(int j = 0; j < n; j++) {
      C[nums[j][i]]++;
    }
    for(int j = 1; j < k; j++) {
      C[j] += C[j - 1];
    }
    B = new int[n][];
    for(int j = n - 1; j >= 0; j--) {
      B[C[nums[j][i]] - 1] = nums[j];
      C[nums[j][i]] = C[nums[j][i]] - 1;
    }
    nums = B;
  }
  return nums;
}

int mergeSort(int[] a) {
  int n = a.length;
  if(n == 1) {return 0;}
  int m = n / 2;
  int[] left = Arrays.copyOfRange(a, 0, m);
  int[] right = Arrays.copyOfRange(a, m, n);
  int inv = mergeSort(left);
  inv += mergeSort(right);
  inv += merge(left, right, a);
  return inv;
}

int merge(int[] left, int[] right, int[] a) {
  int i = 0, l = 0, r = 0, inv = 0;
  while(l < left.length && r < right.length) {
    if(left[l] <= right[r]) {
      a[i++] = left[l++];
    } else {
      inv += left.length - l;
      a[i++] = right[r++];
    }
  }
  for(int j = l; j < left.length; j++) {
    a[i++] = left[j];
  }
  for(int j = r; j < right.length; j++) {
    a[i++] = right[j];
  }
  return inv;
}

int countMinSwapsToSort(int[] a) {
  int[] b = a.clone();
  Arrays.sort(b);
  int nSwaps = 0;
  for(int i = 0; i < a.length; i++) {
    // cuidado com elementos repetidos!
    int j = Arrays.binarySearch(b, a[i]);
    if(b[i] == a[j] && i != j) {
      nSwaps++;
      swap(a, i, j);
    }
  }
  for(int i = 0; i < a.length; i++) {
    if(a[i] != b[i]) {
      nSwaps++;
    }
  }
  return nSwaps;
}

//Count (i, j):h[i] <= h[k] <= h[j], k = i+1,...,j
    -1.
int countVisiblePairs(int[] h) { // O(n)
  int n = h.length;
  int[] p = new int[n];
  int[] r = new int[n];
  Stack<Integer> S = new Stack<Integer>();
  for(int i = 0; i < n; i++) {
    int c = 0;
    if(S.isEmpty()) {
      S.push(h[i]);
      p[i] = 0;
    } else {
      if(S.peek() == h[i]) {
        p[i] = p[i - 1] + 1 - r[i - 1];
      } else {
        while(!S.isEmpty() && S.peek() < h[i]) {
     S.pop();
     c++;
   }
   p[i] = c;
   r[i] = c;
   if(!S.isEmpty()) {
     p[i]++;
   }
      }
    S.push(h[i]);
    }
  }
  return sum(p);
}

void shuffle(Object[] a)
{
  int N = a.length;
  for (int i = 0; i < N; i++) {
    int r = i + (int) (Math.random() * (N-i));
    swap(a, i, r);
```

```
  }
}
```

5.9 Huffman (compression)

Normalement utilisé pour des caractères, mais utilisables avec tout ce dont on peux compter les occurences.
Construit un arbre (de préfixe) qu'on utilise pour décoder et qu'on dépile pour encoder.

```
class HuffmanNode implements Comparable<HuffmanNode>
{
  public boolean isLeaf;
  public int occurences;
  public int charIndex;
  public HuffmanNode left,right;
  public HuffmanNode(HuffmanNode left, HuffmanNode
    right)
  {
    this.occurences = left.occurences+right.
    occurences;
    this.left = left;
    this.right = right;
    isLeaf = false;
  }
  public HuffmanNode(int charIndex, int occurences)
  {
    this.charIndex = charIndex;
    this.occurences = occurences;
    isLeaf = true;
  }
  @Override
  public int compareTo(HuffmanNode o) {
    return occurences-o.occurences;
  }
}
HuffmanNode getHuffmanTree(int[] occurences) {
  PriorityQueue<HuffmanNode> q = new PriorityQueue<
    HuffmanNode>();
  for(int i = 0; i < occurences.length; i++)
    q.add(new HuffmanNode(i, occurences[i]));
  while(q.size() != 1) {
    HuffmanNode right = q.poll();
    HuffmanNode left = q.poll();
    q.add(new HuffmanNode(left, right));
  }
  return q.poll();
}
void getHuffmanTable(HuffmanNode tree, BitSet[]
    result, BitSet current, int pos){
  if(tree.isLeaf) {
    BitSet finalBitSet = new BitSet();
    for(int i = 0; i < pos; i++)
      finalBitSet.set(i, current.get(pos-i-1));
    result[tree.charIndex] = finalBitSet;
  } else {
    BitSet leftBitSet = new BitSet();
    leftBitSet.or(current);
    leftBitSet.set(pos, false);
    getHuffmanTable(tree.left, result, leftBitSet,
    pos+1);

    BitSet rightBitSet = new BitSet();
    rightBitSet.or(current);
    rightBitSet.set(pos, true);
    getHuffmanTable(tree.right, result, rightBitSet,
     pos+1);
  }
}

//n=occurences.length
static BitSet[] getHuffmanTable(int n, HuffmanNode
    tree) {
  BitSet[] result = new BitSet[n];
  getHuffmanTable(tree, result, new BitSet(), 0);
  return result;
}
```