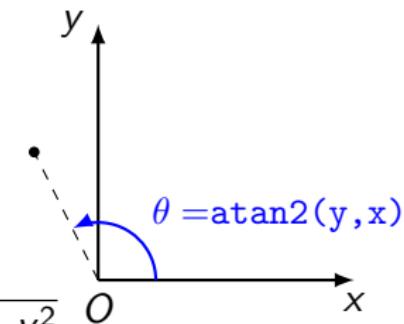


Les vecteurs

Prévoir dans son code :

- ▶ structure ad hoc,
- ▶ fonction **produit scalaire** ($xx' + yy'$),
- ▶ fonction **produit vectoriel** ($xy' - x'y$).

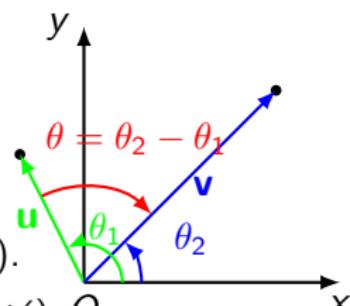


Attributs :

- ▶ Norme : `hypot(x,y)` renvoie $\sqrt{x^2 + y^2}$.
- ▶ Angle avec l'axe (Ox) :
 - ▶ Éviter `atan(y/x)` (danger : division par 0).
 - Utiliser `atan2(y,x)` (valeur dans $[-\pi, \pi]$).

Angle entre deux vecteurs **u** et **v** :

- ▶ Soustraire leur angle avec (Ox) (angle orienté).
- ▶ Calculer $\arccos(\mathbf{u} \cdot \mathbf{v} / \|\mathbf{u}\| \|\mathbf{v}\|)$ (angle non-orienté).



(Ex : UVa920, UVa10927, UVa11519)

Les droites, les segments, les coniques, . . .

Formes possibles de droites :

- ▶ équation cartésienne $ax + by + c = 0$,
- ▶ pente et ordonnée à l'origine $y = ax + b$ (sauf droite verticale),
- ▶ abscisse et ordonnée à l'origine $x/a + y/b = 1$,
- ▶ point & vecteur directeur,
- ▶ couple de points.

Coordonnées de l'**intersection** de deux droites sécantes

$D : ax + by + c$ et $D' : a'x + b'y + c'$ (via les formules de Cramer) :

$$\Delta = \begin{vmatrix} a & b \\ a' & b' \end{vmatrix}, \quad x = -\frac{\begin{vmatrix} c & b \\ c' & b' \end{vmatrix}}{\Delta} \quad \text{et} \quad y = -\frac{\begin{vmatrix} a & c \\ a' & c' \end{vmatrix}}{\Delta}.$$

Les droites sont **effectivement sécantes** ssi Δ est non-nul.

(Ex : UVa837, UVa10678)

Distances sur la sphère

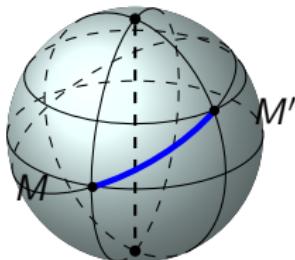
Distance orthodromique (ou distance du plus grand cercle) entre :

- ▶ premier point M de latitude δ et longitude λ ,
- ▶ second point M' de latitude δ' et longitude λ' ,

avec la formule suivante (*Haversine formula* en anglais) :

$$d = 2R \arcsin \sqrt{\sin^2\left(\frac{\delta' - \delta}{2}\right) + \cos \delta \cos \delta' \sin^2\left(\frac{\lambda' - \lambda}{2}\right)}$$

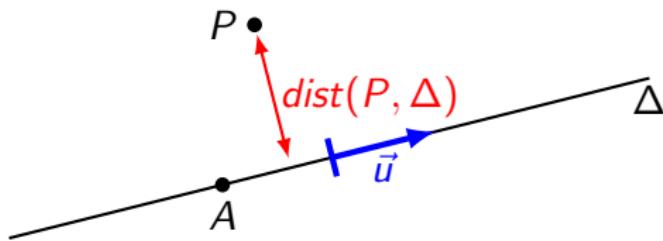
où R = rayon de la sphère.



Distance point – droite

- Distance entre le point P et la droite Δ passant par A dirigée par \vec{u} :

$$dist(A, \Delta) = \|\overrightarrow{AP} \times \vec{u}\| / \|\vec{u}\|.$$

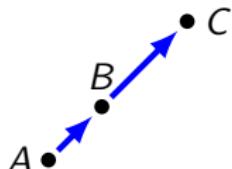


(Ex : UVa10263)

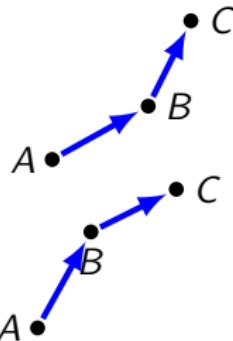
Alignements et virages

- ▶ Trois points A, B, C sont

1. colinéaires ssi $\overrightarrow{AB} \times \overrightarrow{BC} = 0$,



2. en virage à gauche ssi $\overrightarrow{AB} \times \overrightarrow{BC} > 0$,

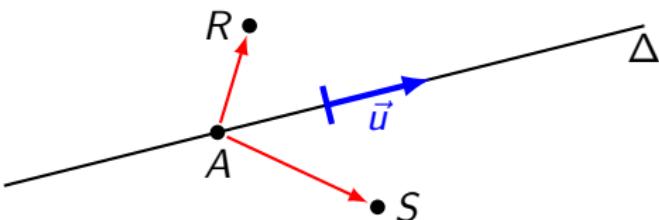


3. en virage à droite ssi $\overrightarrow{AB} \times \overrightarrow{BC} < 0$.

Intersection de segments

- ▶ Un bipoint (R, S) est **de part et d'autre** de la droite Δ passant par A et dirigée par le vecteur \vec{u} si

$$(\overrightarrow{AR} \times \vec{u}) \cdot (\overrightarrow{AS} \times \vec{u}) < 0.$$



- ▶ Deux segments $[AB]$ et $[RS]$ **s'intersectent**ssi le bipoint (A, B) est de part et d'autre de la droite (RS) et le bipoint (R, S) est de part et d'autre de la droite (AB) .

Pour calculer l'intersection, voir transparent 9.

(Ex : UVa191, UVa10902)

Grandeurs du triangle

- Longueur des côtés (Al-Kashi) :

$$c^2 = a^2 + b^2 - 2ab \cos \gamma.$$

- Aire du triangle ABC :

$$A = \frac{\text{base} \times \text{hauteur}}{2} = \frac{\left\| \overrightarrow{AB} \times \overrightarrow{AC} \right\|}{2} = \sqrt{s(s-a)(s-b)(s-c)}$$

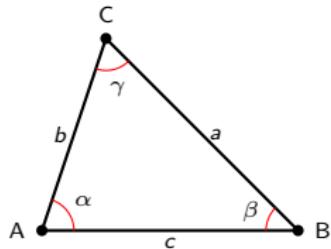
où $s = (a + b + c)/2$ (formule d'Héron).

- Centre de gravité, intersection des médianes :

$$G = \text{barycentre}((A, 1), (B, 1), (C, 1)).$$

- Orthocentre, intersection des hauteurs :

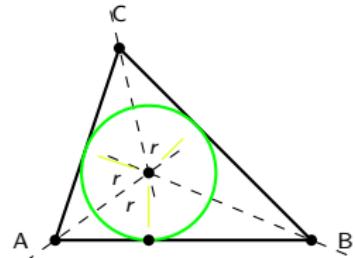
$$H = \text{barycentre}((A, \tan \alpha), (B, \tan \beta), (C, \tan \gamma)).$$



Cercles des triangles

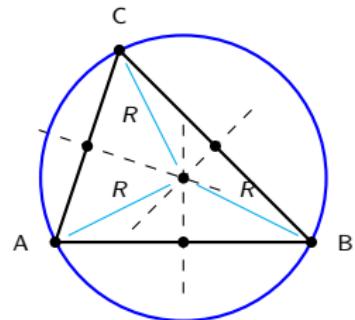
► Cercle inscrit :

- Centre I : intersection des bissectrices :
 $I = \text{barycentre}((A, a), (B, b), (C, c)).$
- Rayon $r = \frac{A}{s}$ où A = aire, s = demi-périmètre.



► Cercle circonscrit :

- Centre O : intersection des médiatrices :
 $O = \text{barycentre}((A, \tan \beta + \tan \gamma), (B, \tan \alpha + \tan \gamma), (C, \tan \alpha + \tan \beta)).$
- Rayon $R = \frac{a \cdot b \cdot c}{4A} = \frac{a}{2 \sin \alpha} = \frac{b}{2 \sin \beta} = \frac{c}{2 \sin \gamma}.$

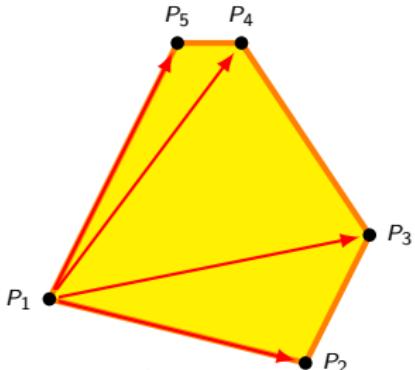


(Ex : UVa11152, UVa10195)

Aire d'un polygone

Polygone $\Pi = (P_1 P_2 \cdots P_k)$:
(même formule que le triangle)

$$\begin{aligned} A &= \frac{1}{2} \left\| \sum_{i=2}^{k-1} \overrightarrow{P_1 P_k} \times \overrightarrow{P_1 P_{k+1}} \right\| \\ &= \frac{1}{2} \left| \begin{array}{c} x_1 & y_1 \\ \vdots & \vdots \\ x_n & y_n \\ x_1 & y_1 \end{array} \right| = \frac{1}{2} \left| \sum_{i=1}^n x_i y_{i+1} - x_{i+1} y_i \right|. \end{aligned}$$



Valable que le polygone soit convexe ou non.

(Ex : UVa11265)

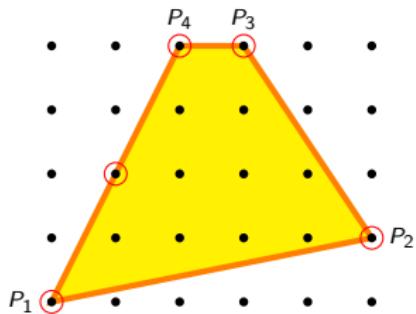
Formule de Pick

Théorème

Soit Π un polygone à sommets entiers, a son aire, b le nombre de points entiers sur sa frontière et i le nombre de points entiers dans son intérieur strict, alors

$$a = i + \frac{b}{2} - 1.$$

(Ex : UVa10088)



$$\begin{aligned}a &= \frac{1}{2} \left\| \binom{5}{1} \times \binom{3}{4} + \binom{3}{4} \times \binom{2}{4} \right\| \\&= 9 + \frac{5}{2} - 1 = \frac{21}{2}\end{aligned}$$

Enveloppe convexe : la marche de Jarvis

(Ex : UVa10652, UVa596)

Entrée : points $(M_i)_{i \leq n}$.

Sortie : Enveloppe convexe $(P_k)_{k \leq h}$.

Complexité : $O(nh)$.

Emballage de cadeaux

$P_0 \leftarrow \min_i\{M_i\}$ (ordre lexicographique).

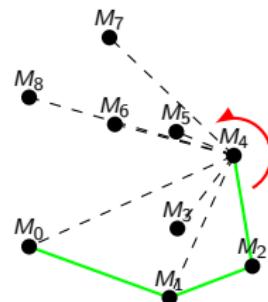
Pour $k \geq 0$,

$P_{k+1} \leftarrow$ point le plus à droite de P_k .

Si $P_{k+1} = P_0$, s'arrêter.

NB : P_{k+1} est le plus à droite de P_k si :

$$\forall M \neq P_k, P_{k+1}, \overrightarrow{P_k P_{k+1}} \times \overrightarrow{P_k M} > 0.$$



Enveloppe convexe : le parcours de Graham

Entrée : points $(M_i)_{i \leq n}$.

Sortie : Enveloppe convexe $(P_k)_{k \leq h}$.

Complexité : $O(n \log n)$.

Graham

P.push($\min_i\{M_i\}$) (ordre lexicographique).

Trier $\{M_i\}$ par pente croissante de $(P_0 M_i)$.

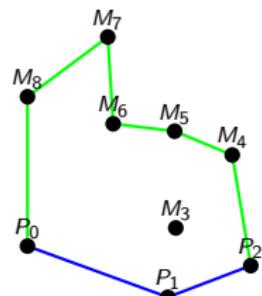
P.push(M_1).

Pour $i \leq n$,

Tant que $P_{-2} \rightarrow P_{-1} \rightarrow M_i$ en virage à droite,

P.pop().

P.push(M_i).



Plus proches points du plan euclidien

Entrée : points $(M_i)_{i \leq n}$.

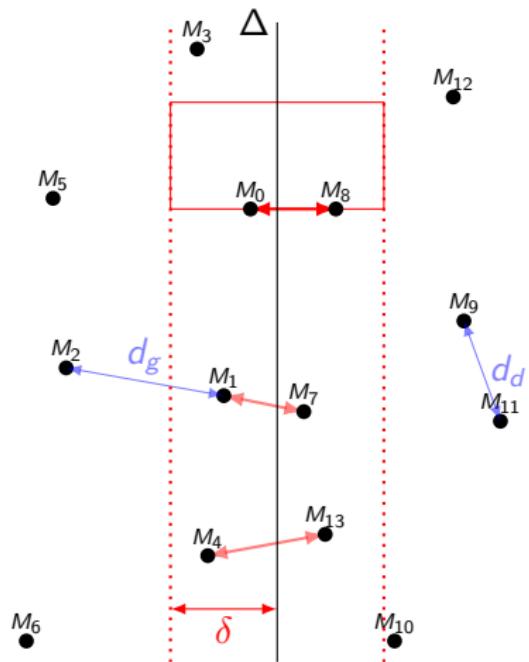
Sortie : $\min_{i \neq j} d(M_i, M_j)$.

Complexité : $O(n \log n)$.

Algorithme diviser pour régner

- ▶ Classer les points par abscisse croissante.
Les scinder par la droite médiane Δ .
- ▶ Résoudre à gauche et à droite récursivement. Minima : d_g et d_d .
- ▶ Dans une fenêtre balayante de taille $(2\delta, \delta)$, où $\delta = \min(d_g, d_d)$, centrée en Δ , chercher la distance minimum d_Δ entre un point gauche et droit.
- ▶ Renvoyer $\min(d_g, d_d, d_\Delta)$.

(Ex : UVa10245, UVa11378)



Data Structures

Several options to represent graphs:

- Adjacency matrix:
 - `bool G[MAXN][MAXN];`
 - `G[x][y]` is `true` if an edge between node `x` and `y` exists
 - Replace `bool` by `int` to represent weighted edges
- Adjacency list:
 - `vector<int> Adj[MAXN];`
 - `y` is in `Adj[x]` if an edge between node `x` and `y` exists
 - Pairs to represent weights
- Edge list:
 - `vector<pair<int, int>> Edges;`
 - `Edges` contains a pair of nodes if an edge exists between them
- Nodes and edges may also be custom structs or classes

Depth-First Search

Visit each node in the graph once:

- Recursive implementation below
- Manage stack yourself for iterative version
- First visit child nodes then siblings

```
bool Visited[MAXN] = {};
void DFS(int u) {
    if (Visited[u])
        return;
    Visited[u] = true;
    // maybe do something with u (pre-order) ...
    for (auto v : Adj[u])
        DFS(v);
    // or do something here (post-order)
}
```

Breadth-First Search

Visit each node in the graph once:

- Similar to DFS, but replaces **stack** by **queue**

```
queue<int> Q;
bool Visited[MAXN] = {};
void BFS(int root) {
    Q.push(root);
    while (!Q.empty()) {
        int u = Q.front();
        Q.pop();
        if (Visited[u])
            continue;
        Visited[u] = true;
        for (auto v : Adj[u])
            Q.push(v);           // usually do something with v
    }
}
```

Dijkstra's Algorithm

```
unsigned int Dist[MAXN];
typedef pair<unsigned int, int> WeightNode;           // weight goes first
priority_queue<WeightNode, std::vector<WeightNode>,
               std::greater<WeightNode> > Q;
void Dijkstra(int root) {
    fill_n(Dist, MAXN, MAXLEN);
    Dist[root] = 0;
    Q.push(make_pair(0, root));
    while (!Q.empty()) {
        int u = Q.top().second;                      // get node with least priority
        Q.pop();
        for (auto tmp : Adj[u]) {
            int v = tmp.first;
            unsigned int weight = tmp.second;
            if (Dist[v] > Dist[u] + weight)           // shorter path found?
                Dist[v] = Dist[u] + weight;
            Q.push(make_pair(Dist[v], v));           // simply push, no update here
        }
    }
}
```

<https://visualgo.net/sssp>

Bellman-Ford Algorithm

```
unsigned int Dist[MAXN];
void BellmanFord(int root) {
    fill_n(Dist, MAXN, MAXLEN);
    Dist[root] = 0;
    for(int u=0; u < N - 1; u++) { // just iterate N - 1 times
        for (auto tmp : Adj[u]) {
            int v = tmp.first;           // same as Dijkstra before
            unsigned int weight = tmp.second;
            Dist[v] = min(Dist[v], Dist[u] + weight);
        }
    }
}
```

<https://visualgo.net/sssp>

Floyd-Warshall Algorithm

```
int Dist[MAXN][MAXN];
void FloydWarshall() {
    fill_n((int*)Dist, MAXN*MAXN, MAXLEN);
    for(int u=0; u < N; u++) {
        Dist[u][u] = 0;
        for (auto tmp : Adj[u])
            Dist[u][tmp.first] = tmp.second;
    }
    for(int k=0; k < N; k++)           // check sub-path combinations
        for(int i=0; i < N; i++)
            for(int j=0; j < N; j++)           // concatenate paths
                Dist[i][j] = min(Dist[i][j], Dist[i][k] + Dist[k][j]);
}
```

Hierholzer's Algorithm for Eulerian Paths (assuming they exist)

```
vector<int> Circuit;
void Hierholzer() {
    int v = 0;           // find node with odd degree, else start with node 0
    for (int u=0; u < N && v == 0; u++)
        if (Adj[u].size() & 1)
            v = u;                                // node with odd degree
    stack<int> Stack;
    Stack.push(v);
    while (!Stack.empty()) {
        if (!Adj[v].empty()) {                  // follow edges until stuck
            Stack.push(v);
            int tmp = *Adj[v].begin();
            Adj[v].erase(tmp);                  // remove edge, modifying graph
            Adj[tmp].erase(v);
            v = tmp;
        } else {                                // got stuck: stack contains a circuit
            Circuit.push_back(v);             // append node at the end of circuit
            v = Stack.top();                 // backtrack using stack, find larger circuit
            Stack.pop();
        }
    }
}
```

<https://www-m9.ma.tum.de/graph-algorithms/hierholzer>

Union-Find using Ranks and Path Compression

```
map<int, pair<int, unsigned int> > Sets; // map to parent & rank
void MakeSet(int x) {
    Sets.insert(make_pair(x, make_pair(x, 0)));
}
int Find(int x) {
    if (Sets[x].first == x) return x; // Parent == x ?
    else return Sets[x].first = Find(Sets[x].first); // Get Parent
}
void Union(int x, int y) {
    int parentX = Find(x), parentY = Find(y);
    int rankX = Sets[parentX].second, rankY = Sets[parentY].second;
    if (parentX == parentY) return;
    else if (rankX < rankY)
        Sets[parentX].first = parentY;
    else
        Sets[parentY].first = parentX;
    if (rankX == rankY)
        Sets[parentX].second++;
}
```

Kruskal's Algorithm

```
vector<pair<int, pair<int, int>> > Edges;
set<pair<int,int>> A; // Final minimum spanning tree

void Kruskal() {
    for(int u=0; u < N; u++)
        MakeSet(u); // Initialize Union-Find DS

    sort(Edges.begin(), Edges.end()); // Sort edges by weight

    for(auto tmp : Edges) {
        auto edge = tmp.second;
        if (Find(edge.first) != Find(edge.second)) {
            Union(edge.first, edge.second); // update Union-Find DS
            A.insert(edge); // include edge in MST
        }
    }
}
```

Ford-Fulkerson Algorithm

```
// find path from s to t in G, return true if such a path exists
bool DFS(int G[MAXN][MAXN], int s, int t, int Predecessor[MAXN]);

int FordFulkerson(int G[MAXN][MAXN], int s, int t) {
    int GRes[MAXN][MAXN];                                // residual graph
    copy_n((int*)G, MAXN*MAXN, (int*)GRes);             // copy original graph

    int Predecessor[MAXN];
    int Maxflow = 0;
    while (DFS(GRes, s, t, Predecessor)) {              // find residual path
        int Bottleneck = MAXFLOW;                         // get minimal flow of residual path
        for(int v = t, u = Predecessor[t]; v != s; v = u, u = Predecessor[u])
            Bottleneck = min(Bottleneck, GRes[u][v]);

        for(int v = t, u = Predecessor[t]; v != s; v = u, u = Predecessor[u])
            GRes[u][v] -= Bottleneck; // decrease capacity along residual path
            GRes[v][u] += Bottleneck;
    }
    Maxflow += Bottleneck;
}
return Maxflow;
```

Hopcroft-Karp (data)

```
// Artificial node (unused otherwise) -- end of augmenting path
#define NIL 0

// "Infinity", i.e., value larger than min(|X|, |Y|)
#define INF numeric_limits<unsigned int>::max()

// Partitions X and Y
vector<int> X, Y;
// Neighbors in Y of nodes in X
vector<int> Adj[MAXX];

// Matching X-Y and Y-X
int PairX[MAXX];
int PairY[MAXY];

// Augmenting path lengths
unsigned int Dist[MAXX];
```

Hopcroft-Karp (main)

```
int HopcroftKarp() {
    fill_n(PairX, X.size(), NIL);      // initialize: empty matching
    fill_n(PairY, Y.size(), NIL);

    int Matching = 0;                  // count number of edges in matching

    while (BFS()) {                   // find all shortest augmenting paths
        for(auto x : X)              // update matching cardinality
            if (PairX[x] == NIL &&      // node not yet in matching?
                DFS(x))               // does an augmenting path start at x?
                Matching++;
    }
    return Matching;
}
```

Hopcroft-Karp (BFS)

```
bool BFS() {  
    queue<int> Q;  
    Dist[NIL] = INF;  
    for(auto x : X) // start from nodes that are not yet matched  
        Dist[x] = (PairX[x] == NIL) ? 0 : INF;  
    if (PairX[x] == NIL)  
        Q.push(x);  
  
    while (!Q.empty()) { // find all shortest paths to NIL  
        int x = Q.front(); Q.pop();  
        if (Dist[x] < Dist[NIL]) // can this become a shorter path?  
            for (auto y : Adj[x])  
                if (Dist[PairY[y]] == INF) {  
                    Dist[PairY[y]] = Dist[x] + 1; // update path length  
                    Q.push(PairY[y]);  
                }  
    }  
    return Dist[NIL] != INF; // any shortest path to NIL found?  
}
```

Hopcroft-Karp (DFS)

```
bool DFS(int x) {
    if (x != NIL) {
        for (auto y : Adj[x])
            if (Dist[PairY[y]] == Dist[x] + 1 &&
                DFS(PairY[y])) {           // follow trace of BFS
                PairX[x] = y;           // add edge from x to y to matching
                PairY[y] = x;
                return true;
            }
        Dist[x] = INF;
        return false;               // no augmenting path found
    }
    return true;                  // reached NIL
}
```

PGCD, relation de Bezout

- ▶ Algorithme d'Euclide :

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

- ▶ Relation de Bezout : $ua + vb = d$ où d est le PGCD :

```
pair<int, pair<int, int> > bezout(int a, int b) {
    if (b == 0) return make_pair(a, make_pair(1, 0));
    pair<int, pair<int, int> > p = bezout(b, a%b);
    int d = p.first;
    int u = p.second.first, v = p.second.second;
    return make_pair(d, make_pair(v, u - (a/b) * v));
}
```

- ▶ Application : $u = a^{-1} \pmod{b}$ (**inversion modulaire**).

(Ex : UVa10407, UVa10090, UVa10673)

Nombres premiers

- ▶ On génère la liste des premiers par le **crible d'Ératosthène** :

```
bitset<10000001> P;
vector<long long> premiers;
P.set(); // initialisation
P[0] = P[1] = 0;
for (long long i = 2; i < pmax; i++)
    if (P[i]) {
        for (long long j=i*i; j < pmax; j+=i)
            P[j]=0;
        premiers.push_back(i);
    }
```

- ▶ Factorisation d'un entier n : par divisions successives, chercher les diviseurs de n parmi les premiers $\leq \sqrt{n}$.

Exemple : sudoku

```
int g[9][9];
int solve(int i) {
    int x = i/9, y = i%9;
    if (x >= 9) return 1;
    if (g[x][y] != 0)
        return solve(i+1);
    for (int k = 1; k <= 9; k++)
        if (acceptable(x, y, k)) {
            g[x][y] = k;
            if (solve(i+1))
                return 1;
            g[x][y] = 0;
        }
    return 0;
}
```

Exemple : minimax

```
int minimax(int i, int player) {  
    int winner;  
    if (winner = game_over())  
        return winner;  
    for (int m = 0; m < nmoves; m++)  
        if (admissible(m, player)) {  
            do_move(m, player);  
            int ret = minimax(i+1, -player);  
            if (ret == player)  
                return player;  
            undo_move(m, player);  
        }  
    return -player;  
}
```

Exemple : Fibonacci (amélioré)

```
int T[3];  
  
int fib(int n) {  
    T[0] = 0;  
    T[1] = 1;  
    for (int i = 2; i <= n; i++)  
        T[i%3] = T[(i-1)%3] + T[(i-2)%3];  
    return T[n%3];  
}
```

- Mieux qu'un récursif mémoisé !

Exemple : Levenshtein (amélioré)

```
int M[100][100];
char *s, *t;

int dist(int ni, int nj) {
    for (int j = 0; j <= nj; j++)
        M[0][j] = j;
    for (int i = 1; i <= ni; i++) {
        M[i%2][0] = i;
        for (int j = 1; j <= nj; j++)
            M[i%2][j] = min(min(1 + M[i%2][j-1], 1 + M[(i-1)%2][j]),
                             M[(i-1)%2][j-1] + ((s[i-1] == t[j-1]) ? 0 : 1));
    }
    return M[ni%2][nj];
}
```

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}

```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T	-1	0	0	0	1	2	1	2	3	4	5	0	
i													↑
cnd													↑

Knuth–Morris–Pratt: recherche

```
char p[MAXN]; int T[MAXN+1];
int np = strlen(p), ns = strlen(s);
[...]           // ici, construire la table T comme précédemment
int cnd = 0;           // position courante dans le motif p
for (int i = 0; i <= ns; i++) {    // tant qu'on ne lit pas
    while (cnd >= 0 && p[cnd] != s[i]) // le prochain char de p
        cnd = T[cnd];                  // on recule dans p
    cnd++; // maintenant que le prochain char convient, avancer
    if (cnd == np) {
        // on a atteint la fin de p, donc on a trouvé un match
        printf("match at %d\n", i - np + 1);
        // on recule dans p au cas où le prochain match chevauche
        cnd = T[cnd];
    }
}
```

Aho-Corasick : Construction du trie

```
int nw;                                // nombre de mots du dictionnaire
char w[MAXW][MAXL];                    // contenu des mots du dictionnaire
int trie[MAXW*MAXL][ALPHA];  // trie des mots du dictionnaire
int fs;                                // prochain index libre dans trie
int endw[MAXW*MAXL]; // mot qui se finit à cet index de trie

// insérer le mot numéro i de contenu s dans le trie
// en partant du nœud d'index n dans le trie
void insert_trie(int i, char *s, int n) {
    unsigned char x = s[0];
    if (!x) { endw[n] = i; return; }      // fin du mot atteinte
    if (!trie[n][x]) // si pas d'arête étiquetée x depuis n...
        trie[n][x] = fs++; // ... on crée un nouveau nœud cible
    return insert_trie(i, s+1, trie[n][x]);
}
```

Aho-Corasick : Début du code

```
void aho_corasick() {
    // l'index 0 dans le trie indique les nœuds inexistantes
    // la racine du trie est 1
    // donc initialement le prochain index libre est 2
    fs = 2;
    // les mots sont numérotés à partir de 1
    // 0 dans endw indique qu'aucun mot ne se termine
    for (int i = 1; i <= nw; i++)
        insert_trie(i, w[i], 1); // insère chaque mot dans le trie
    [...]
}
```

// ici, on calcule les pointeurs et les raccourcis

Aho-Corasick : Calcul des pointeurs et raccourcis (1/2)

```
int pointer[MAXW*MAXL];    // pointeurs (en bleu), cf T de KMP
int shortcut[MAXW*MAXL];      // raccourcis (en vert)
queue<int> q;
q.push(1);          // explorer en BFS depuis la racine du trie
while (!q.empty()) {
    int n = q.front();
    q.pop();
    for (unsigned char x = 0; x < ALPHA; x++) {
        int n2 = trie[n][x];
        if (!n2)
            continue;      // pas d'arête pour cette lettre depuis n
        [...]           // on traite n2, cf prochain transparent
        q.push(n2);       // continuer le BFS avec n2
    }
}
```

Aho-Corasick : Calcul des pointeurs et raccourcis (2/2)

```
// nœud n2 de parent n avec étiquette x, i.e., n -x-> n2
// == calcul du pointeur ==
int p = pointer[n];                                // p est le pointeur de n
while (p && !trie[p][x]) // tant que p n'a pas d'arête x...
    p = pointer[p];                                // ... on recule p via le pointeur
if (!p)                                         // si p = 0, on est sorti du trie...
    pointer[n2] = 1;                            // ... donc on met pointeur = racine
else
    pointer[n2] = trie[p][x];                    // ... le x-sucesseur de p

// == calcul du raccourci ==
if (endw[pointer[n2]])                         // si un mot finit en n2...
    shortcut[n2] = pointer[n2];                  // ... raccourci = pointeur
else
    shortcut[n2] = shortcut[pointer[n2]];        // sinon le raccourci est celui du pointeur
```

Aho-Corasick : Utilisation

```
int pos = 1;                                // position courante dans le trie
char s[MAXS]; int ns = strlen(s);           // chaîne à parcourir

for (int i = 0; i < ns; i++) {
    unsigned char x = s[i];                  // x = nouveau caractère lu
    while (pos && !trie[pos][x])            // tant que pas d'arête x depuis p...
        pos = pointer[pos];                 // ... on recule pos suivant pointeur
    pos = trie[pos][x];                    // maintenant on essaie d'avancer par x
    if (!pos)                               // si on est sorti du trie...
        pos = 1;                           // ... on revient à la racine

    int posb = pos; // on va énumérer les matches possibles depuis pos...
    do {                                     // fin de mot possible en i
        if (endw[posb])          // on a vraiment une fin de mot à posb (en i):
            printf("match of %s at %d\n",      // afficher
                w[endw[posb]], i - strlen(endw[posb]) + 1); // le match
        posb = shortcut[posb];             // suivre le raccourci de posb
    } while (endw[posb]);                  // ... tant que des mots se terminent
}
```