Référence d'Équipe

Université de Picardie Jules Verne

October 29, 2023

Contents

1 Dynamic Programming **1**
1.1 dice proba 1

2 Geometry **1**
2.1 all 1
2.2 center 2 points + radius 2
2.3 closest pair 2
2.4 convex hull 3
2.5 rotating calipers 3
2.6 split convex polygon 3
2.7 triangles 4

3 Graphs **4**
3.1 Dijkstra 4
3.2 Recherche en Largeur 4
3.3 Rechernche en profondeur 5
3.4 euler formula 6

4 Math **6**
4.1 Proba 6
4.2 fibonacci 6
4.3 lucas 6

5 Structure **6**
5.1 arbre 6
5.2 file 7
5.3 pile 7
5.4 treeDiameter 8

6 Tri **8**
6.1 Tri$_Fusion$ 8
6.2 Tri$_Insertion$ 9
6.3 Tri$_Tas$ 9

7 X - Misc **9**
7.1 equations 9
7.2 Trigonometry 9
7.3 Triangles 10
7.4 Quadrilaterals 10
7.5 Spherical coordinates 10
7.6 Derivatives/Integrals 10
7.7 Sums 10
7.8 Series 10
7.9 Geometric series 10

1 Dynamic Programming

1.1 dice proba

```
#include <stdio.h>
#include <stdlib.h>

#define MOD 1000000007

// Tableau pour la mmosation
long long memo[1000001];

// Fonction rcursive avec  mmosation
long long countWays(int n) {
    // Si la valeur a dj   t   calcule , la retourner
    if (memo[n] != -1) {
        return memo[n];
    }
    // Initialiser le compteur pour n
    long long count = 0;
    // Pour chaque valeur possible du d (1  6),
    // ajouter le nombre de faons de former la somme n - i
    for (int i = 1; i <= 6; i++) {
        if (n - i >= 0) {
            count = (count + countWays(n - i)) % MOD;
        }
    }
    // Stocker le rsultat dans le tableau de mmosation et
     le retourner
    return memo[n] = count;
}

int main() {
    int n;
    scanf("%d", &n);
    // Initialiser le tableau de mmosation avec des
     valeurs de -1
    for (int i = 0; i <= n; i++) {
        memo[i] = -1;
    }
    // Cas de base
    memo[0] = 1;
    // Calculer et afficher le rsultat
    printf("%lld\n", countWays(n));
    return 0;
}
```

2 Geometry

2.1 all

```
double INF = 1e100;
double EPS = 1e-12;
struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y)    {}
  PT operator + (const PT &p)  const { return PT(x+p.x,
     y+p.y); }
  PT operator - (const PT &p)  const { return PT(x-p.x,
     y-p.y); }
  PT operator * (double c)     const { return PT(x*c,
     y*c  ); }
  PT operator / (double c)     const { return PT(x/c,
     y/c  ); }
};
double dot(PT p, PT q)     { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q)   { return dot(p-q,p-q); }
double cross(PT p, PT q)   { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
```

```
  return os << "(" << p.x << "," << p.y << ")";
}
// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p)   { return PT(-p.y,p.x); }
PT RotateCW90(PT p)    { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}
// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}
// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b-a,b-a);
  if (fabs(r) < EPS) return a;
  r = dot(c-a, b-a)/r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}
// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}
// compute distance between point (x,y,z) and plane
     ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
    double a, double b, double c, double d) {
  return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}
// determine if lines from a to b and c to d are parallel
     or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b-a, c-d)) < EPS;
}
bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
    && fabs(cross(a-b, a-c)) < EPS
    && fabs(cross(c-d, c-a)) < EPS;
}
// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
        dist2(b, c) < EPS || dist2(b, d) < EPS) return
     true;
    if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b,
     d-b) > 0)
      return false;
    return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
  return true;
}
// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b*cross(c, d)/cross(b, d);
}
// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c,
     c+RotateCW90(a-c));
}
// determine if point is in a possibly non-convex polygon
     (by William
// Randolph Franklin); returns 1 for strictly interior
     points, 0 for
// strictly exterior points, and 0 or 1 for the remaining
     points.
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
        p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y)
    / (p[j].y - p[i].y))
     c = !c;
  }
  return c;
}
// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()],
     q), q) < EPS)
     return true;
  return false;
}
// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double
     r) {
  vector<PT> ret;
  b = b-a;
  a = a-c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}
// compute intersection of circle centered at a with
     radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r,
     double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0)
    ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}
// This code computes the area or centroid of a (possibly
     nonconvex)
// polygon, assuming that the coordinates are listed in a
     clockwise or
// counterclockwise fashion.  Note that the centroid is
     often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area / 2.0;
}
double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}
PT ComputeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
    c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
  }
  return c / scale;
}
// tests whether or not a given polygon (in CW or CCW
     order) is simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}
```

2.2 center 2 points + radius

```
vector<point> find_center(point a, point b, long double r)
     {
  point d = (a - b) * 0.5;
  if (d.dot(d) > r * r) {
    return vector<point> ();
  }
```

```
  point e = b + d;
  long double fac = sqrt(r * r - d.dot(d));
  vector<point> ans;
  point x =  point(-d.y, d.x);
  long double l = sqrt(x.dot(x));
  x = x * (fac / l);
  ans.push_back(e + x);
  x =  point(d.y, -d.x);
  x = x * (fac / l);
  ans.push_back(e + x);
  return ans;
}
```

2.3 closest pair

```
struct point {
  double x, y;
  int id;
  point() {}
  point (double a, double b) : x(a), y(b) {}
};
double dist(const point &o, const point &p) {
  double a = p.x - o.x, b = p.y - o.y;
  return sqrt(a * a + b * b);
}
double cp(vector<point> &p, vector<point> &x,
     vector<point> &y) {
  if (p.size() < 4) {
    double best = 1e100;
    for (int i = 0; i < p.size(); ++i)
      for (int j = i + 1; j < p.size(); ++j)
        best = min(best, dist(p[i], p[j]));
    return best;
  }
  int ls = (p.size() + 1) >> 1;
  double l = (p[ls - 1].x + p[ls].x) * 0.5;
  vector<point> xl(ls), xr(p.size() - ls);
  unordered_set<int> left;
  for (int i = 0; i < ls; ++i) {
    xl[i] = x[i];
    left.insert(x[i].id);
  }
  for (int i = ls; i < p.size(); ++i) {
    xr[i - ls] = x[i];
  }
  vector<point> yl, yr;
  vector<point> pl, pr;
  yl.reserve(ls); yr.reserve(p.size() - ls);
  pl.reserve(ls); pr.reserve(p.size() - ls);
  for (int i = 0; i < p.size(); ++i) {
    if (left.count(y[i].id)) yl.push_back(y[i]);
    else yr.push_back(y[i]);

    if (left.count(p[i].id)) pl.push_back(p[i]);
    else pr.push_back(p[i]);
  }
  double dl = cp(pl, xl, yl);
  double dr = cp(pr, xr, yr);
  double d = min(dl, dr);
  vector<point> yp; yp.reserve(p.size());
  for (int i = 0; i < p.size(); ++i) {
    if (fabs(y[i].x - l) < d)
      yp.push_back(y[i]);
  }
  for (int i = 0; i < yp.size(); ++i) {
    for (int j = i + 1; j < yp.size() && j < i + 7; ++j) {
      d = min(d, dist(yp[i], yp[j]));
    }
  }
  return d;
}
double closest_pair(vector<point> &p) {
  vector<point> x(p.begin(), p.end());
  sort(x.begin(), x.end(), [](const point &a, const point
     &b) {
    return a.x < b.x;
  });
  vector<point> y(p.begin(), p.end());
  sort(y.begin(), y.end(), [](const point &a, const point
     &b) {
    return a.y < b.y;
  });
  return cp(p, x, y);
}
```

2.4 convex hull

```
#define REMOVE_REDUNDANT
typedef double T;
const T EPS = 1e-7;
struct PT {
  T x, y;
  PT() {}
  PT(T x, T y) : x(x), y(y) {}
  bool operator<(const PT &rhs) const { return
     make_pair(y,x) < make_pair(rhs.y,rhs.x); }
  bool operator==(const PT &rhs) const { return
     make_pair(y,x) == make_pair(rhs.y,rhs.x); }
};
T cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
T area2(PT a, PT b, PT c) { return cross(a,b) + cross(b,c)
     + cross(c,a); }
#ifdef REMOVE_REDUNDANT
bool between(const PT &a, const PT &b, const PT &c) {
  return (fabs(area2(a,b,c)) < EPS && (a.x-b.x)*(c.x-b.x)
     <= 0 && (a.y-b.y)*(c.y-b.y) <= 0);
}
#endif
void ConvexHull(vector<PT> &pts) {
  sort(pts.begin(), pts.end());
  pts.erase(unique(pts.begin(), pts.end()), pts.end());
  vector<PT> up, dn;
  for (int i = 0; i < pts.size(); i++) {
    while (up.size() > 1 && area2(up[up.size()-2],
     up.back(), pts[i]) >= 0) up.pop_back();
    while (dn.size() > 1 && area2(dn[dn.size()-2],
     dn.back(), pts[i]) <= 0) dn.pop_back();
    up.push_back(pts[i]);
    dn.push_back(pts[i]);
  }
  pts = dn;
  for (int i = (int) up.size() - 2; i >= 1; i--)
     pts.push_back(up[i]);
#ifdef REMOVE_REDUNDANT
  if (pts.size() <= 2) return;
  dn.clear();
  dn.push_back(pts[0]);
  dn.push_back(pts[1]);
  for (int i = 2; i < pts.size(); i++) {
    if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i]))
     dn.pop_back();
    dn.push_back(pts[i]);
  }
  if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
    dn[0] = dn.back();
    dn.pop_back();
  }
  pts = dn;
#endif
}
```

2.5 rotating calipers

```
typedef long double gtype;
const gtype pi = M_PI;
typedef complex<gtype> point;
typedef complex<gtype> point;
#define x real()
#define y imag()
#define polar(r, t) polar((gtype) (r), (t))
// vector
#define rot(v, t) ( (v) * polar(1, t) )
#define crs(a, b) ( (conj(a) * (b)).y )
#define dot(a, b) ( (conj(a) * (b)).x )
#define pntLinDist(a, b, p) ( abs(crs((b)-(a), (p)-(a)) /
     abs((b)-(a))) )
bool cmp_point(point const& p1, point const& p2) {
    return p1.x == p2.x ? (p1.y < p2.y) : (p1.x < p2.x);
}
// O(n) - rotating calipers (works on a ccw closed convex
     hull)
gtype rotatingCalipers(vector<point> &ps) {
    int aI = 0, bI = 0;
    for (size_t i = 1; i < ps.size(); ++i)
        aI = (ps[i].y < ps[aI].y ? i : aI), bI = (ps[i].y
     > ps[bI].y ? i : bI);
    gtype minWidth = ps[bI].y - ps[aI].y, aAng, bAng;
    point aV = point(1, 0), bV = point(-1, 0);
    for (gtype ang = 0; ang < pi; ang += min(aAng, bAng)) {
        aAng = acos(dot(ps[aI + 1] - ps[aI], aV)
            / abs(aV) / abs(ps[aI + 1] - ps[aI]));
        bAng = acos(dot(ps[bI + 1] - ps[bI], bV)
            / abs(bV) / abs(ps[bI + 1] - ps[bI]));
        aV = rot(aV, min(aAng, bAng)), bV = rot(bV,
     min(aAng, bAng));
        if (aAng < bAng)
```

```
        minWidth = min(minWidth, pntLinDist(ps[aI],
    ps[aI] + aV, ps[bI]))
        , aI = (aI + 1) % (ps.size() - 1);
      else
        minWidth = min(minWidth, pntLinDist(ps[bI],
    ps[bI] + bV, ps[aI]))
        , bI = (bI + 1) % (ps.size() - 1);
    }
    return minWidth;
}
```

2.6 split convex polygon

```
typedef long double Double;
typedef vector <Point> Polygon;
// This is not standard intersection because it returns
    false
// when the intersection point is exactly the t=1 endpoint
    of
// the segment. This is OK for this algorithm but not for
    general
// use.
bool segment_line_intersection(Double x0, Double y0,
    Double x1, Double y1, Double x2, Double y2,
    Double x3, Double y3, Double &x, Double &y){
    Double t0 = (y3-y2)*(x0-x2) - (x3-x2)*(y0-y2);
    Double t1 = (x1-x0)*(y2-y0) - (y1-y0)*(x2-x0);
    Double det =(y1-y0)*(x3-x2) - (y3-y2)*(x1-x0);
    if (fabs(det) < EPS){ //Paralelas
        return false;
    }else{
        t0 /= det;
        t1 /= det;
        if (cmp(0, t0) <= 0 and cmp(t0, 1) < 0){
            x = x0 + t0 * (x1-x0);
            y = y0 + t0 * (y1-y0);
            return true;
        }
        return false;
    }
}
// Returns the polygons that result of cutting the CONVEX
// polygon p by the infinite line that passes through (x0,
    y0)
// and (x1, y1).
// The returned value has either 1 element if this line
// doesn't cut the polygon at all (or barely touches it)
// or 2 elements if the line does split the polygon.
vector<Polygon> split(const Polygon &p, Double x0, Double
    y0,
                      Double x1, Double y1) {
    int hits = 0, side = 0;
    Double x, y;
    vector<Polygon> ans(2);
    for (int i = 0; i < p.size(); ++i) {
        int j = (i + 1) % p.size();
        if (segment_line_intersection(p[i].x, p[i].y,
            p[j].x, p[j].y,   x0, y0, x1, y1,   x, y)) {
            hits++;
            ans[side].push_back(p[i]);
            if (cmp(p[i].x, x) != 0 or cmp(p[i].y, y) !=
    0) {
                ans[side].push_back(Point(x, y));
            }
            side ^= 1;
            ans[side].push_back(Point(x, y));
        } else {
            ans[side].push_back(p[i]);
        }
    }
    return hits < 2 ? vector<Polygon>(1, p) : ans;
}
```

2.7 triangles

Let a, b, c be length of the three sides of a triangle.

$$p = (a + b + c) * 0.5$$

The inradius is defined by:

$$iR = \sqrt{\frac{(p-a)(p-b)(p-c)}{p}}$$

The radius of its circumcircle is given by the formula:

$$cR = \frac{abc}{\sqrt{(a+b+c)(a+b-c)(a+c-b)(b+c-a)}}$$

3 Graphs

3.1 Dijkstra

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
#define INF 1e9

typedef struct {
    int V; // Nombre de sommets
    int adj[MAX][MAX]; // Matrice d'adjacence (pour les
    poids)
} Graph;

void initGraph(Graph *g, int V) {
    g->V = V;
    for(int i = 0; i < V; i++) {
        for(int j = 0; j < V; j++) {
            g->adj[i][j] = 0; // ou INF si on prfre
    indiquer qu'il n'y a pas de lien
        }
    }
}

void addEdge(Graph *g, int src, int dest, int weight) {
    g->adj[src][dest] = weight;
    // g->adj[dest][src] = weight; // Si le graphe est non
    dirig
}

void dijkstra(Graph *g, int src) {
    int dist[MAX];
    int visited[MAX] = {0};

    for(int i = 0; i < g->V; i++) {
        dist[i] = INF;
    }
    dist[src] = 0;

    for(int i = 0; i < g->V - 1; i++) {
        int u = -1;

        // Trouver le sommet avec la distance minimale,
    parmi les sommets non traits.
        for(int j = 0; j < g->V; j++) {
            if(!visited[j] && (u == -1 || dist[j] <
    dist[u])) {
                u = j;
            }
        }

        visited[u] = 1;
        //
        for(int v = 0; v < g->V; v++) {
            if(!visited[v] && g->adj[u][v] && dist[u] +
    g->adj[u][v] < dist[v]) {
                dist[v] = dist[u] + g->adj[u][v];
            }
        }
    }

    // Affichage des distances
    for(int i = 0; i < g->V; i++) {
        printf("Distance du sommet %d au sommet %d =
    %d\n", src, i, dist[i]);
    }
}
int main() {
    Graph g;
    initGraph(&g, 6); // Cration d'un graphe avec 5
    sommets (0,1,2,3,4)

    addEdge(&g, 0, 1, 5);
    addEdge(&g, 0, 2, 3);
    addEdge(&g, 1, 3, 6);
    addEdge(&g, 1, 2, 2);
    addEdge(&g, 2, 4, 4);
    addEdge(&g, 2, 5, 2);
    addEdge(&g, 2, 3, 7);
    addEdge(&g, 3, 4, -1);
    addEdge(&g, 4, 5, -2);

    dijkstra(&g, 0);
    return 0;
```

```
}
```

3.2 Recherche en Largeur

```
#define MAX 100

typedef struct {
    int V; // Nombre de sommets
    int adj[MAX][MAX]; // Matrice d'adjacence
} Graph;

void initGraph(Graph *g, int V) {
    g->V = V;
    for(int i = 0; i < V; i++) {
        for(int j = 0; j < V; j++) {
            g->adj[i][j] = 0;
        }
    }
}

void addEdge(Graph *g, int src, int dest) {
    g->adj[src][dest] = 1;
    g->adj[dest][src] = 1; // Si le graphe est non dirig
}

void BFS(Graph *g, int start) {
    int visited[MAX] = {0};
    int queue[MAX], front = -1, rear = -1;

    void enqueue(int v) {
        if(rear == MAX-1) return;
        if(front == -1) front = 0;
        queue[++rear] = v;
    }

    int dequeue() {
        if(front == -1) return -1;
        int v = queue[front];
        if(front == rear) front = rear = -1;
        else front++;
        return v;
    }

    printf("%d ", start);
    visited[start] = 1;
    enqueue(start);
    while(front != -1) {
        int curr = dequeue();
        for(int i = 0; i < g->V; i++) {
            if(g->adj[curr][i] == 1 && !visited[i]) {
                printf("%d ", i);
                visited[i] = 1;
                enqueue(i);
            }
        }
    }
}

int main() {
    Graph g;
    initGraph(&g, 6); // Cration d'un graphe avec 5
     sommets (0,1,2,3,4)

    addEdge(&g, 0, 1);
    addEdge(&g, 0, 2);
    addEdge(&g, 1, 3);
    addEdge(&g, 5, 5);
    addEdge(&g, 3, 4);

    BFS(&g, 3);

    return 0;
}
/////////////////////////////////////////////////////////////////////


#define MAX 1000

typedef struct {
    int x, y;
    char dir;
} Node;

Node queue[MAX * MAX];
int front = 0, rear = 0;

char labyrinth[MAX][MAX];
Node prev[MAX][MAX];
bool visited[MAX][MAX];
int n, m;

int dx[] = {-1, 1, 0, 0};
int dy[] = {0, 0, -1, 1};
char directions[] = {'U', 'D', 'L', 'R'};

void bfs(int startX, int startY) {
    front = rear = 0;
    queue[rear++] = (Node){startX, startY, 0};
    visited[startX][startY] = true;

    while (front < rear) {
        Node current = queue[front++];
        for (int d = 0; d < 4; d++) {
            int nx = current.x + dx[d];
            int ny = current.y + dy[d];
            if (nx >= 0 && nx < n && ny >= 0 && ny < m &&
     !visited[nx][ny] && labyrinth[nx][ny] != '#') {
                visited[nx][ny] = true;
                prev[nx][ny] = current;
                prev[nx][ny].dir = directions[d];
                queue[rear++] = (Node){nx, ny, 0};
            }
        }
    }
}

void reconstruct_path(int startX, int startY, int endX,
     int endY) {
    char path[MAX * MAX];
    int length = 0;
    while (endX != startX || endY != startY) {
        path[length++] = prev[endX][endY].dir;
        Node temp = prev[endX][endY];
        endX = temp.x;
        endY = temp.y;
    }

    printf("YES\n%d\n", length);
    for (int i = length - 1; i >= 0; i--) {
        putchar(path[i]);
    }
    putchar('\n');
}

int main() {
    scanf("%d %d", &n, &m);
    int startX, startY, endX, endY;

    for (int i = 0; i < n; i++) {
        scanf("%s", labyrinth[i]);
        for (int j = 0; j < m; j++) {
            if (labyrinth[i][j] == 'A') {
                startX = i;
                startY = j;
            }
            if (labyrinth[i][j] == 'B') {
                endX = i;
                endY = j;
            }
        }
    }

    bfs(startX, startY);

    if (visited[endX][endY]) {
        reconstruct_path(startX, startY, endX, endY);
    } else {
        printf("NO\n");
    }

    return 0;
}
```

3.3 Rechernche en profondeur

```
#include <stdbool.h>
#include <stdio.h>

#define MAX_NODES 1000

bool visited[MAX_NODES];
int graph[MAX_NODES][MAX_NODES];

void dfs(int node, int n) {
    visited[node] = true;
    printf("Visited node: %d\n", node);

    for (int i = 0; i < n; i++) {
        if (graph[node][i] && !visited[i]) {
```

```
            dfs(i, n);
        }
    }
}
//Exemple implementation

#include <stdbool.h>
#include <stdio.h>

#define MAX_SIZE 1000

char map[MAX_SIZE][MAX_SIZE];
bool visited[MAX_SIZE][MAX_SIZE];
int n, m;

void dfs(int x, int y) {
    if (x < 0 || x >= n || y < 0 || y >= m) return; //
     Vrifie les limites
    if (map[x][y] == '#' || visited[x][y]) return; //
     Vrifie les murs et les zones visites

    visited[x][y] = true;

    dfs(x + 1, y);
    dfs(x - 1, y);
    dfs(x, y + 1);
    dfs(x, y - 1);
}

int main() {
    scanf("%d %d", &n, &m);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            scanf(" %c", &map[i][j]);
            visited[i][j] = false;
        }
    }

    int rooms = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (!visited[i][j] && map[i][j] == '.') {
                dfs(i, j);
                rooms++;
            }
        }
    }

    printf("%d\n", rooms);

    return 0;
}
```

3.4 euler formula

Euler's formula states that if a finite, connected, planar graph is drawn in the plane without any edge intersections, and v is the number of vertices, e is the number of edges and f is the number of faces (regions bounded by edges, including the outer, infinitely large region), then:

$$f + v = e + 2$$

It can be extended to non connected planar graphs with c connected components:

$$f + v = e + c + 1$$

4 Math

4.1 Proba

```
#include <stdio.h>
double Tab[600][100];
double prob(int n, int s, int a, int b) //s = 6n
{
    if(Tab[s][n] != -1)
    {
        return Tab[s][n];
    }
        if(n==0)
        {
            if( a<=s && s<=b)
                return 1;
            else
                return 0;
        }else
        {
             return Tab[s][n] = 1./6. * prob(n-1, s, a, b)
  + 1./6. * prob(n-1, s-1, a, b) + 1./6. * prob(n-1,
  s-2, a, b) +
                     1./6. * prob(n-1, s-3, a, b) + 1./6. *
  prob(n-1, s-4, a, b) + 1./6. * prob(n-1, s-5, a, b);
        }
}

int main()
{
    int a, b, n;
    scanf("%d %d %d", &n, &a, &b);
    for (int i = 0; i<=6*n; i++)
        for (int y = 0; y<=n; y++)
            Tab[i][y] = -1;
    printf("%lf\n", prob(n, 6*n, a, b));
}
```

4.2 fibonacci

Let A, B and n be integer numbers.

$$k = A - B \tag{1}$$

$$F_A F_B = F_{k+1} F_A^2 + F_k F_A F_{A-1} \tag{2}$$

$$\sum_{i=0}^{n} F_i^2 = F_{n+1} F_n \tag{3}$$

$ev(n) =$ returns 1 if n is even.

$$\sum_{i=0}^{n} F_i F_{i+1} = F_{n+1}^2 - ev(n) \tag{4}$$

$$\sum_{i=0}^{n} F_i F_{i-1} = \sum_{i=0}^{n-1} F_i F_{i+1} \tag{5}$$

4.3 lucas

For non-negative integers m and n and a prime p, the following congruence relation holds: :

$$\binom{m}{n} \equiv \prod_{i=0}^{k} \binom{m_i}{n_i} \pmod p,$$

where :

$$m = m_k p^k + m_{k-1} p^{k-1} + \cdots + m_1 p + m_0,$$

and :

$$n = n_k p^k + n_{k-1} p^{k-1} + \cdots + n_1 p + n_0$$

are the base p expansions of m and n respectively. This uses the convention that $\binom{m}{n} = 0$ if $m \leq n$.

5 Structure

5.1 arbre

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_CHILDREN 10

struct node
{
    int nb, color, numChildren;
    struct node* children[MAX_CHILDREN];
};

typedef struct node arbre_t;

arbre_t* find_node(arbre_t* node, int target_nb)
{
```

```c
    if (node == NULL)
        return NULL;

    if (node->nb == target_nb)
        return node;

    for (int i = 0; i < node->numChildren; i++)
    {
        arbre_t* result = find_node(node->children[i],
     target_nb);
        if (result != NULL)
            return result;
    }

    return NULL;
}

void add_child_to_node(arbre_t* parent_node, int child_nb,
     int child_color)
{
    if (parent_node->numChildren < MAX_CHILDREN)
    {
        arbre_t* new_child =
     (arbre_t*)malloc(sizeof(arbre_t));
        new_child->nb = child_nb;
        new_child->color = child_color;
        new_child->numChildren = 0;

        parent_node->children[parent_node->numChildren] =
     new_child;
        parent_node->numChildren++;
    }
    else
    {
        printf("Nombre maximal d'enfants atteint pour le
     nud %d\n", parent_node->nb);
    }
}

void print_tree(arbre_t* node)
{
    if (node == NULL)
        return;

    // Afficher le nud actuel
    printf("Nud %d (Couleur : %d)\n", node->nb,
     node->color);

    // Afficher les enfants rcursivement
    for (int i = 0; i < node->numChildren; i++)
    {
        printf("Enfant de %d : ", node->nb);
        print_tree(node->children[i]);
    }
}

void free_tree(arbre_t* node)
{
    if (node == NULL)
        return;

    for (int i = 0; i < node->numChildren; i++)
    {
        free_tree(node->children[i]);
    }

    free(node);
}

int main(int argc, char const* argv[])
{
    int n;
    scanf("%d", &n);
    int tabColor[n];

    for (int i = 0; i < n; ++i)
    {
        scanf("%d", &tabColor[i]);
    }

    int parent, child;
    scanf("%d %d", &parent, &child);

    arbre_t* root = (arbre_t*)malloc(sizeof(arbre_t));
    root->nb = parent;
    root->color = tabColor[parent - 1];
    root->numChildren = 0;

    add_child_to_node(root, child, tabColor[child - 1]);

    for (int i = 0; i < n - 2; ++i)
    {
        scanf("%d %d", &parent, &child);
        arbre_t* parent_node = find_node(root, parent);

        if (parent_node != NULL)
        {
            add_child_to_node(parent_node, child,
     tabColor[child - 1]);
        }
        else
        {
            printf("Nud parent non trouv : %d\n", parent);
        }
    }

    // Afficher l'arbre
    printf("Arbre :\n");
    print_tree(root);

    // Librer la mmoire de l'arbre
    free_tree(root);

    return 0;
}
```

5.2 file

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 100 // Vous pouvez ajuster cette valeur
     selon vos besoins

typedef struct {
    int front, rear;
    int items[MAX_SIZE];
} Queue;

// Fonction pour initialiser une file vide
void initialize(Queue* q) {
    q->front = -1;
    q->rear = -1;
}

// Fonction pour vrifier si la file est vide
int isEmpty(Queue* q) {
    return (q->front == -1 && q->rear == -1);
}

// Fonction pour vrifier si la file est pleine
int isFull(Queue* q) {
    return (q->rear + 1) % MAX_SIZE == q->front;
}

// Fonction pour ajouter un lment   la file (  l'arrire)
void enqueue(Queue* q, int item) {
    if (isFull(q)) {
        printf("La file est pleine, impossible d'ajouter
     un lment.\n");
    } else if (isEmpty(q)) {
        q->front = q->rear = 0;
        q->items[q->rear] = item;
    } else {
        q->rear = (q->rear + 1) % MAX_SIZE;
        q->items[q->rear] = item;
    }
}

// Fonction pour supprimer un lment de la file (du front)
void dequeue(Queue* q) {
    if (isEmpty(q)) {
        printf("La file est vide, impossible de supprimer
     un lment.\n");
    } else if (q->front == q->rear) {
        q->front = q->rear = -1;
    } else {
        q->front = (q->front + 1) % MAX_SIZE;
    }
}

// Fonction pour obtenir l'lment   l'avant de la file
     sans le supprimer
int front(Queue* q) {
    if (isEmpty(q)) {
        printf("La file est vide, pas d'lment en
     avant.\n");
        return -1;
    }
    return q->items[q->front];
}

int main() {
```

```
    Queue myQueue;
    initialize(&myQueue);

    enqueue(&myQueue, 10);
    enqueue(&myQueue, 20);
    enqueue(&myQueue, 30);

    printf("Front of the queue: %d\n", front(&myQueue));

    dequeue(&myQueue);
    printf("Front of the queue after dequeue: %d\n",
     front(&myQueue));

    return 0;
}
```

5.3 pile

```
#include <stdio.h>
#include <stdlib.h>

struct pile {
    int sommet, taille;
    int *reste;
};

typedef struct pile pile;

int estVide(pile *p) {
    if (p->taille == 0) return 1;
    else return 0;
}

void empiler(pile *p, int element) {
    p->reste[p->taille] = element;
    p->taille++;
    p->sommet = element;
}

int depiler(pile *p) {
    if (estVide(p)) {
        printf("Impossible de dpiler pile vide\n");
        return -1;
    }
    else {
        int sommet = p->sommet;
        p->reste[p->taille - 1] = -1;
        p->taille--;
        if (p->taille > 0) {
            p->sommet = p->reste[p->taille - 1];
        } else {
            p->sommet = -1;
        }
        return sommet;
    }
}

int sommet(pile * p){
  if(estVide(p)){
    printf("pile vide\n");
        return -1;
  }
  else return p->sommet;
}

void ecrirePile(pile *p) {
    for (int i = p->taille - 1; i >= 0; i--) {
        printf("%d ", p->reste[i]);
    }
    printf("\n");
}

int main(int argc, char const *argv[]) {
    pile p;
    p.taille = 0;
    p.sommet = -1;
    p.reste = (int *)malloc(sizeof(int) * 100); // Allouez
     de l'espace pour 100 lments, par exemple

    empiler(&p, 1);
    empiler(&p, 5);
    empiler(&p, 7);
    empiler(&p, 8);
    ecrirePile(&p);
    printf("sommet depiler : %d\n", depiler(&p));
    ecrirePile(&p);

    free(p.reste); // Librez la mmoire  alloue  pour le
     tableau

    return 0;
}
```

5.4 treeDiameter

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_N 200000

// Structure to represent an edge
struct Edge {
    int to;
    struct Edge* next;
};

struct Edge* graph[MAX_N + 1];  // Adjacency list
     representation of the tree
int maxDepth = 0;

// Depth-First Search to find the diameter
int dfs(int node, int parent) {
    int maxDepth1 = 0, maxDepth2 = 0;
    struct Edge* edge = graph[node];

    while (edge != NULL) {
        int neighbor = edge->to;
        if (neighbor != parent) {
            int depth = dfs(neighbor, node);
            if (depth > maxDepth1) {
                maxDepth2 = maxDepth1;
                maxDepth1 = depth;
            } else if (depth > maxDepth2) {
                maxDepth2 = depth;
            }
        }
        edge = edge->next;
    }

    // Update the diameter
    maxDepth = (maxDepth > maxDepth1 + maxDepth2) ?
     maxDepth : maxDepth1 + maxDepth2;

    // Return the maximum depth rooted at this node
    return maxDepth1 + 1;
}

int main() {
    int n;
    scanf("%d", &n);

    // Initialize the graph
    for (int i = 1; i <= n; i++) {
        graph[i] = NULL;
    }

    // Build the tree
    for (int i = 0; i < n - 1; i++) {
        int a, b;
        scanf("%d %d", &a, &b);

        struct Edge* edge1 = (struct
     Edge*)malloc(sizeof(struct Edge));
        edge1->to = b;
        edge1->next = graph[a];
        graph[a] = edge1;

        struct Edge* edge2 = (struct
     Edge*)malloc(sizeof(struct Edge));
        edge2->to = a;
        edge2->next = graph[b];
        graph[b] = edge2;
    }

    dfs(1, 0);  // Start the DFS from node 1 as the root

    printf("%d\n", maxDepth);

    return 0;
}
```

6 Tri

6.1 Tri$_{Fusion}$

```
#include <stdio.h>
```

```
#include <stdlib.h>

void fusion(int tableau[], int gauche, int milieu, int
     droite) { //O(n*log(n)) rapide mais utilise

          // beaucoup de mmoire
    int i, j, k;
    int n1 = milieu - gauche + 1;
    int n2 = droite - milieu;

    // Crer des tableaux temporaires
    int L[n1], R[n2];

    // Copier les donnes dans les tableaux temporaires L[]
     et R[]
    for (i = 0; i < n1; i++)
        L[i] = tableau[gauche + i];
    for (j = 0; j < n2; j++)
        R[j] = tableau[milieu + 1 + j];

    // Fusionner les tableaux temporaires de nouveau dans
     le tableau[gauche..droite]
    i = 0;
    j = 0;
    k = gauche;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            tableau[k] = L[i];
            i++;
        } else {
            tableau[k] = R[j];
            j++;
        }
        k++;
    }

    // Copier les  lments  restants de L[], s'il y en a
    while (i < n1) {
        tableau[k] = L[i];
        i++;
        k++;
    }

    // Copier les  lments  restants de R[], s'il y en a
    while (j < n2) {
        tableau[k] = R[j];
        j++;
        k++;
    }
}

void triFusion(int tableau[], int gauche, int droite) {
    if (gauche < droite) {
        // Trouver le point milieu du tableau
        int milieu = gauche + (droite - gauche) / 2;

        // Trier la premire et la deuxime moiti
        triFusion(tableau, gauche, milieu);
        triFusion(tableau, milieu + 1, droite);

        // Fusionner les deux moitis tries
        fusion(tableau, gauche, milieu, droite);
    }
}

int main() {
    int tableau[] = {12, 11, 13, 5, 6, 7, 5, 60, 2, 1, 8};
    int taille = sizeof(tableau) / sizeof(tableau[0]);
    triFusion(tableau, 0, taille - 1);
    printf("Tableau tri par tri fusion : \n");
    for (int i = 0; i < taille; i++) {
        printf("%d ", tableau[i]);
    }
    return 0;
}
```

6.2 Tri$_{Insertion}$

```
#include <stdio.h>

void triInsertion(int tableau[], int taille) { //O(n) bien
     pour les tri courts
    int i, j, cle;
    for (i = 1; i < taille; i++) {
        cle = tableau[i];
        j = i - 1;

        // Dplace les  lments  du tableau[0..i-1] qui sont
     plus grands que la cl
        while (j >= 0 && tableau[j] > cle) {
            tableau[j + 1] = tableau[j];
            j = j - 1;
        }
        tableau[j + 1] = cle;
    }
}

int main() {
    int tableau[] = {12, 11, 13, 5, 6};
    int taille = sizeof(tableau) / sizeof(tableau[0]);
    triInsertion(tableau, taille);
    printf("Tableau tri par insertion : \n");
    for (int i = 0; i < taille; i++) {
        printf("%d ", tableau[i]);
    }
    return 0;
}
```

6.3 Tri$_{Tas}$

```
#include <stdio.h>

// Fonction pour changer deux  lments  dans un tableau
void swap(int* a, int* b) { //O(n*log(n)) efficace lorsque
     que contraintes
                           //mmoire
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Fonction pour rorganiser le tas pour maintenir la
     proprit de tas max
void maxHeapify(int arr[], int n, int i) {
    int largest = i;  // Initialisation de la racine comme
     le plus grand lment
    int left = 2 * i + 1; // Indice du fils gauche
    int right = 2 * i + 2; // Indice du fils droit

    // Si le fils gauche est plus grand que la racine
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // Si le fils droit est plus grand que le plus grand
     lment jusqu' prsent
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // Si le plus grand  lment n'est pas la racine
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        // Rorganiser rcursivement le sous-arbre affect
        maxHeapify(arr, n, largest);
    }
}

// Fonction pour trier un tableau en utilisant le tri par
     tas
void heapSort(int arr[], int n) {
    // Construire le tas ( partir du bas)
    for (int i = n / 2 - 1; i >= 0; i--)
        maxHeapify(arr, n, i);

    // Extraire les  lments  un par un depuis le tas
    for (int i = n - 1; i > 0; i--) {
        // Dplacer la racine actuelle  la fin
        swap(&arr[0], &arr[i]);
        // Appel rcursif pour rduire la taille du tas
        maxHeapify(arr, i, 0);
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Tableau non tri : \n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    heapSort(arr, n);

    printf("\nTableau tri : \n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

7 X - Misc

7.1 equations

$$ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The extremum is given by $x = -b/2a$.

$$\begin{array}{l} ax + by = e \\ cx + dy = f \end{array} \Rightarrow \begin{array}{l} x = \dfrac{ed - bf}{ad - bc} \\ y = \dfrac{af - ec}{ad - bc} \end{array}$$

In general, given an equation $Ax = b$, the solution to a variable x_i is given by

$$x_i = \frac{\det A'_i}{\det A}$$

where A'_i is A with the i'th column replaced by b.

7.2 Trigonometry

$$\sin(v + w) = \sin v \cos w + \cos v \sin w$$
$$cos(v + w) = \cos v \cos w - \sin v \sin w$$

$$\tan(v + w) = \frac{\tan v + \tan w}{1 - \tan v \tan w}$$
$$\sin v + \sin w = 2 \sin \frac{v + w}{2} \cos \frac{v - w}{2}$$
$$\cos v + \cos w = 2 \cos \frac{v + w}{2} \cos \frac{v - w}{2}$$

$$(V + W) \tan(v - w)/2 = (V - W) \tan(v + w)/2$$

where V, W are lengths of sides opposite angles v, w.

$$a \cos x + b \sin x = r \cos(x - \phi)$$
$$a \sin x + b \cos x = r \sin(x + \phi)$$

where $r = \sqrt{a^2 + b^2}, \phi = \text{atan2}(b, a)$.

7.3 Triangles

Side lengths: a, b, c

Semiperimeter: $p = \dfrac{a + b + c}{2}$

Area: $A = \sqrt{p(p - a)(p - b)(p - c)}$

Circumradius: $R = \dfrac{abc}{4A}$

Inradius: $r = \dfrac{A}{p}$

Length of median (divides triangle into two equal-area triangles): $m_a = \frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}$

Length of bisector (divides angles in two):

$$s_a = \sqrt{bc \left[1 - \left(\frac{a}{b + c}\right)^2\right]}$$

Law of sines: $\dfrac{\sin \alpha}{a} = \dfrac{\sin \beta}{b} = \dfrac{\sin \gamma}{c} = \dfrac{1}{2R}$

Law of cosines: $a^2 = b^2 + c^2 - 2bc \cos \alpha$

Law of tangents: $\dfrac{a + b}{a - b} = \dfrac{\tan \frac{\alpha + \beta}{2}}{\tan \frac{\alpha - \beta}{2}}$

7.4 Quadrilaterals

With side lengths a, b, c, d, diagonals e, f, diagonals angle θ, area A and magic flux $F = b^2 + d^2 - a^2 - c^2$:

$$4A = 2ef \cdot \sin \theta = F \tan \theta = \sqrt{4e^2 f^2 - F^2}$$

For cyclic quadrilaterals the sum of opposite angles is $180°$, $ef = ac + bd$, and $A = \sqrt{(p - a)(p - b)(p - c)(p - d)}$.

7.5 Spherical coordinates

$$\begin{array}{cc} x = r \sin \theta \cos \phi & r = \sqrt{x^2 + y^2 + z^2} \\ y = r \sin \theta \sin \phi & \theta = \text{acos}(z/\sqrt{x^2 + y^2 + z^2}) \\ z = r \cos \theta & \phi = \text{atan2}(y, x) \end{array}$$

7.6 Derivatives/Integrals

$$\frac{d}{dx} \arcsin x = \frac{1}{\sqrt{1 - x^2}} \qquad \frac{d}{dx} \arccos x = -\frac{1}{\sqrt{1 - x^2}}$$
$$\frac{d}{dx} \tan x = 1 + \tan^2 x \qquad \frac{d}{dx} \arctan x = \frac{1}{1 + x^2}$$
$$\int \tan ax = -\frac{\ln |\cos ax|}{a} \qquad \int x \sin ax = \frac{\sin ax - ax \cos ax}{a^2}$$
$$\int e^{-x^2} = \frac{\sqrt{\pi}}{2} \text{erf}(x) \qquad \int xe^{ax} dx = \frac{e^{ax}}{a^2}(ax - 1)$$

Integration by parts:

$$\int_a^b f(x)g(x)dx = [F(x)g(x)]_a^b - \int_a^b F(x)g'(x)dx$$

7.7 Sums

$$c^a + c^{a+1} + \cdots + c^b = \frac{c^{b+1} - c^a}{c - 1}, c \neq 1$$

$$1 + 2 + 3 + \cdots + n = \frac{n(n + 1)}{2}$$
$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(2n + 1)(n + 1)}{6}$$
$$1^3 + 2^3 + 3^3 + \cdots + n^3 = \frac{n^2(n + 1)^2}{4}$$
$$1^4 + 2^4 + 3^4 + \cdots + n^4 = \frac{n(n + 1)(2n + 1)(3n^2 + 3n - 1)}{30}$$

7.8 Series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots, (-\infty < x < \infty)$$

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots, (-1 < x \leq 1)$$

$$\sqrt{1 + x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \ldots, (-1 \leq x \leq 1)$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots, (-\infty < x < \infty)$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots, (-\infty < x < \infty)$$

7.9 Geometric series

$$r \neq 1$$

$$a + ar + ar^2 + ar^3 + \cdots + ar^{n-1} = \sum_{k=0}^{n-1} ar^k = a\left(\frac{1 - r^n}{1 - r}\right)$$