DU_Epinephrine

-Maxon5

Alpha Beta Pruning:

```
int alphaBeta(int depth, int mask, int player, int alpha, int beta)
{
    if(depth == n)
        return val[mask];
    int ret;
    if(player == MAXI)
    {
        ret = -inf;
        for(int i = 0; i<2; i++)
        {
            if(i == 0)
                ret = max(ret, alphaBeta(depth+1, mask, MINI,
                          alpha, beta));
            else
                ret = max(ret, alphaBeta(depth+1, setBit(mask, depth),
                          MINI, alpha, beta));
            alpha = max(alpha, ret);
            if(alpha >= beta)
                return ret;
        }
    }
    else
    {
        ret = inf;
        for(int i = 0; i<2; i++)
        {
            if(i == 0)
                ret = min(ret, alphaBeta(depth+1, mask, MAXI,
                          alpha, beta));
            else
                ret = min(ret, alphaBeta(depth+1, setBit(mask, depth),
                          MAXI, alpha, beta));
            beta = min(beta, ret);
            if(alpha >= beta)
                return ret;
        }
    }
    return ret;
}
```

Blue Red Stalk Hackenbush Calculating Pile Value:

/// sample string s = "BWBBBWW"

```
// box.xx has the nim value, box.yy has the length of the pile.
pll process(char *s)
{
    pll box;
    box.yy = strlen(s);
    if(s[0] == 'W')
        box.xx = V;
    else
        box.xx = -V;
    LL tmp = box.xx;
    int st = -1;
    for(int i = 1; i<box.yy; i++)
    {
        if(s[i] == s[0])
            box.xx += tmp;
        else
        {
            st = i;
            break;
        }
    }
    if(st == -1)
        return box;
    LL nw = 2;
    for(int i = st; i<box.yy; i++)
    {
        if(s[i] == 'W')
            box.xx += (V/nw);
        else
            box.xx -= (V/nw);
        nw <<= 1;
    }
    return box;
}
```

Burnside Lemma:

/// Number of different coloring: Number of self loops / number of total movements

```
LL pk[1010], inv[1010];
const LL MOD = 1000000007;

LL expo(int a, int n)
{
    if(n == 0)
        return 1;
    if(n&1) return ( a * expo(a, n-1))%MOD;
    LL ret = expo(a, n/2);
    return (ret * ret)%MOD;
}

int main()
{
//    freopen("maxon.txt","r",stdin);
    //freopen("out.txt","w",stdout);
    int n, k, t;
    inv[0] = 1;
    for(int i = 1; i<=1000; i++)
        inv[i] = expo(i, MOD-2);
    sf(t);
    for(int cs = 1; cs<=t; cs++)
    {
        sff(n,k);
        pk[0] = 1;
        for(int i = 1; i<=n; i++)
            pk[i] = (pk[i-1] * k)%MOD;
        LL ans = 0;
        for(int i = 1; i<=n; i++)
        {
            ans += pk[__gcd(i,n)];
            if(ans >= MOD)
                ans -= MOD;
        }
        ans *= inv[n];
        ans %= MOD;
        printf("Case %d: %lld\n",cs,ans);
    }
    return 0;
}
```

Circle Polygon Intersection Area:

```
struct Point{
    double x, y;
    Point() {
    }
    Point (double tx,double ty){
        x = tx;
        y = ty;
    }
    Point operator - (const Point &t) const{
        Point res;
        res.x = x - t.x;
        res.y = y - t.y;
        return res;
    }
};

int n;
Point p[5050];

double Dot_Product(Point &a, Point &b, Point &c){
    return (b.x-a.x)*(c.x-a.x) + (b.y-a.y)*(c.y-a.y);
}

double Cross_Product(Point &a, Point &b, Point &c){
    return (b.x-a.x)*(c.y-a.y) - (c.x-a.x)*(b.y-a.y);
}

/// Cross product of two points:
double cp(Point &a, Point &b){
    return a.x*b.y - b.x*a.y;
}

/// The angle between two vectors:
double angle(Point &a, Point &b){
    double ans = fabs((atan2(a.y, a.x) - atan2(b.y, b.x)));
    return ans > PI+eps ? 2*PI-ans : ans;
```

```
}

/// Straight line and circle intersection:
void Circle_Line_Intersection(Point center, double r, Point l1, Point l2,
Point& p1, Point& p2){
    double a1 = l2.x - l1.x;
    double a2 = l2.y - l1.y;
    double b1 = l1.x - center.x;
    double b2 = l1.y - center.y;

    /// Ax + By + C = 0:
    double A = a1*a1 + a2*a2;
    double B = (a1*b1 + a2*b2)*2;
    double C = b1*b1 + b2*b2 - r*r;

    /// t1,t2 are the value of x:
    double t1 = (-B - sqrt(B*B - 4.0*A*C))/2.0/A;
    double t2 = (-B + sqrt(B*B - 4.0*A*C))/2.0/A;

    p1.x = l1.x + a1*t1;
    p1.y = l1.y + a2*t1;
    p2.x = l1.x + a1*t2;
    p2.y = l1.y + a2*t2;
}

/// Intersected Area of a circle and a simple polygon:
double Circle_Polygon_Intersection(Point center, double R, Point *p, int n){
    Point o(0,0), a, b, t1, t2;
    double sum = 0, res = 0;
    p[n] = p[0];

    for (int i = 0; i < n; i++) {

        a = p[i]-center;
        b = p[i+1]-center;
        double sign = cp(a,b) > 0 ? 1 : -1;
        double d1 = a.x*a.x + a.y*a.y;
        double d2 = b.x*b.x + b.y*b.y;
        double d3 = sqrt((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));

        if (d1 < R*R+eps && d2 < R*R+eps) {   /// two points are inside the
circle
```

```
            res = fabs(cp(a, b));

        } else if (d1 < R*R-eps || d2 < R*R-eps) {   /// only a point inside
a circle
            Circle_Line_Intersection(o, R, a, b, t1, t2);
            if ((a.x-t2.x)*(b.x-t2.x) < eps && (a.y-t2.y)*(b.y-t2.y) < eps) {
                t1 = t2;
            }
            if (d1 < d2) {
                res = fabs(cp(a, t1)) + R*R*angle(b, t1);
            } else {
                res = fabs(cp(b, t1)) + R*R*angle(a, t1);
            }

        } else if (fabs(cp(a, b))/d3 > R-eps) {   /// two points in the
garden outside, and only a segment of a circle as much as the intersection
            res = R*R*angle(a, b);

        } else {    /// line and circle there are two intersections
            Circle_Line_Intersection(o, R, a, b, t1, t2);
            if (Dot_Product(t1, a, b) > eps || Dot_Product(t2, a, b) > eps) {
                res = R*R*angle(a, b);
            } else {
                res = fabs(cp(t1, t2));
                if (Cross_Product(t1, t2, a) < eps){
                    res += R*R*(angle(a, t1) + angle(b, t2));
                } else {
                    res += R*R*(angle(a, t2) + angle(b, t1));
                }
            }
        }

        sum += res * sign;
    }

    return fabs(sum)/2.0;
}
```

Circulation Flow:

```
// Circulation
// 0 based indexing
// Given a directed weighted graph, computes the minimum cost to run the
maximum
// amount of circulation flow through the graph.
//
// Configure: MAXV
// Configure: MAXE (at least 2 * calls_to_edge)
//
// Functions:
// - init(n) initializes the algorithm with the given number of nodes
// - edge(x, y, c, w) adds an edge x->y with capacity c and weight w
// - run() runs the algorithm and returns total cost
//
// Time complexity: No idea, but it should be fast enough to solve any
problem
// where V and E are up to around 1000.
//
// Constants to configure:
// - MAXV is the maximum number of vertices
// - MAXE is the maximum number of edges (i.e. twice the calls to function
edge)

#define FOR(i, a, b) for (int i = (a); i < (b); ++i)
#define REP(i, n) FOR(i, 0, n)

typedef long long llint;

LL inf = 1e12;
namespace Circu {
  const int MAXV = 1000100;
  const int MAXE = 1000100;

  int V, E;
  int how[MAXV], good[MAXV], bio[MAXV], cookie = 1; llint dist[MAXV];
  int from[MAXE], to[MAXE]; llint cap[MAXE], cost[MAXE];

  void init(int n) { V = n; E = 0; }

  void edge(int x, int y, llint c, llint w) {
    from[E] = x; to[E] = y; cap[E] = c; cost[E] = +w; ++E;
    from[E] = y; to[E] = x; cap[E] = 0; cost[E] = -w; ++E;
  }
```

```
void reset() {
  REP(i, V) dist[i] = 0;
  REP(i, V) how[i] = -1;
}

bool relax() {
  bool ret = false;
  REP(e, E) if (cap[e]) {
    int x = from[e];
    int y = to[e];

    if (dist[x] + cost[e] < dist[y]) {
      dist[y] = dist[x] + cost[e];
      how[y] = e;
      ret = true;
    }
  }
  return ret;
}

llint cycle(int s, bool flip = false) {
  int x = s;
  llint c = cap[how[x]];
  do {
    int e = how[x];
    c = min(c, cap[e]);
    x = from[e];
  } while (x != s);

  llint sum = 0;
  do {
    int e = how[x];
    if (flip) {
      cap[e] -= c;
      cap[e^1] += c;
    }
    sum += cost[e] * c;
    x = from[e];
  } while (x != s);
  return sum;
}
```

```
  llint push(int x) {
    for (++cookie; bio[x] != cookie; x = from[how[x]]) {
      if (!good[x] || how[x] == -1 || cap[how[x]] == 0) return 0;
      bio[x] = cookie;
      good[x] = false;
    }
    return cycle(x) >= 0 ? 0 : cycle(x, true);
  }

  llint run() {
    reset();
    llint ret = 0;
    REP(step, 2*V) {
      if (step == V) reset();
      if (!relax()) continue;

      REP(i, V) good[i] = true;
      REP(i, V) if (llint w = push(i)) ret += w, step = 0;
    }
    return ret;
  }
}
```

Hilbert Curve Everything:

```
LL p2[70];

inline bool ok(LL x, LL y, int k)
{
    return (0 <= x && x < p2[k]) && (0 <= y && y < p2[k]);
}

int getQuadrant(LL x, LL y, int k)
{
    if(x < p2[k-1])
    {
        if(y < p2[k-1])
            return 0;
        else
```

```
            return 1;
    }
    else
    {
        if(y < p2[k-1])
            return 3;
        else
            return 2;
    }
}

LL getRank(LL x, LL y, int k)
{
    if(x == 0 && y == 0)
        return 1;
//    DDD(x, y, k);
    if(!(x < p2[k] && y < p2[k]))
    {
        DDD(x, y, k);
        D(p2[k]);
        assert(false);
    }

    int quad = getQuadrant(x, y, k);
    LL ret = quad * p2[2*k-2];
    if(quad == 0)
    {
        pll st = {0, 0};
        LL dx = x - st.xx;
        LL dy = y - st.yy;
        x = dy;
        y = dx;
        ret += getRank(x, y, k-1);
    }
    else if(quad == 1)
    {
        pll st = {0, p2[k-1]};
        LL dx = x - st.xx;
        LL dy = y - st.yy;
        x = dx;
        y = dy;
        ret += getRank(x, y, k-1);
    }
    else if(quad == 2)
    {
        pll st = {p2[k-1], p2[k-1]};
        LL dx = x - st.xx;
        LL dy = y - st.yy;
        x = dx;
        y = dy;
        ret += getRank(x, y, k-1);
    }
    else
    {
        pll st = {p2[k]-1, p2[k-1]-1};
        LL dx = x - st.xx;
        LL dy = y - st.yy;
        x = -dy;
        y = -dx;
        ret += getRank(x, y, k-1);
    }
    return ret;
}

pll findMth(int k, LL m)
{
    if(m == 1)
        return {0, 0};
    assert((p2[k] * p2[k]) >= m);
    LL part = p2[2*k-2];
    for(int i = 0; i<4; i++)
    {
        if(part >= m)
        {
            pll ret = findMth(k-1, m);

            if(i == 0)
                swap(ret.xx, ret.yy);
            else if(i == 1)
                ret.yy += p2[k-1];
            else if(i == 2)
                ret.xx += p2[k-1], ret.yy += p2[k-1];
            else
            {
                swap(ret.xx, ret.yy);
                ret.xx *= -1;
                ret.yy *= -1;
                ret.xx += p2[k]-1;
                ret.yy += p2[k-1]-1;
            }
            return ret;
        }
        else
            m -= part;
    }
    assert(false);
}
```

Hill Climbing:

/*
Solution to Lightoj Speed Zone. Problem statement in Original Library in Lagrange Multiplier.
parameter p of solve should be very close to 1 for other problems.
*/

```
int n, d, speed[110];

inline double go(double x, double s)
{
    double total = sqrt(x*x+100*100);
    return total/s;
}
double have[110];

double solve(double p, double EPS)
{
    double x = (double)d/n;
    double ans = 0;
    for(int i = 1; i<=n; i++)
        ans += go(x,speed[i]), have[i] = x;
    double change = x;
    int cnt = 0;
    while(change > 1e-9 )
    {
        bool done = false;
        cnt++;
        for(int i = 1; i<=n; i++)
        {
            if(change > have[i])
```

```
                continue;
            for(int j = 1; j<=n; j++)
            {
                if(i == j)
                    continue;
                double tmp = ans;
                tmp -= go(have[i], speed[i]);
                tmp -= go(have[j], speed[j]);
                tmp += go(have[i] - change, speed[i]);
                tmp += go(have[j] + change, speed[j]);
                if(tmp + 1e-9 < ans )
                {
                    done = true;
                    have[i] -= change;
                    have[j] += change;
                    ans = tmp;
                }
            }
        }
        if(!done || cnt > 7)
            change *= p, cnt = 0;
        else
            cnt++;
    }
    return ans;
}

int main()
{
    int t;
    sf(t);
    for(int cs = 1; cs<=t; cs++)
    {
        sff(n, d);
        for(int i = 1; i<=n; i++)
            sf(speed[i]);
        printf("Case %d: %.9f\n",cs,solve(0.5, 1e-2));
    }
    return 0;
}
```

Inverse Phi Backtrack:

```
/*
P <= 1e9
Long long integers will be needed for this range of phi
runs in 4.3 seconds for random 1e5 cases.
runs in 2.7 seconds for the worst 100 cases picked from randomly generated 1e6 cases.
Takes 1.621 seconds for 10 instances of the largest highly composite number under 1e9.
Completely unrelated: 1e7 calls to clock() takes around 2.7 seconds on RM's PC which can calculate
1e9 additions in 1.1 seconds
*/
#define MAX        100000
vector<LL>prime;
bool notPrime[MAX+10];
void sieve()
{
   prime.pb(2);
   for(int i = 3; i<=MAX; i+=2)
   {
      if(notPrime[i] == 0)
      {
         prime.pb(i);
         LL st = (LL)i*i;
         if(st > MAX)
            continue;
         for(int j = st; j<=MAX; j+=(i+i))
            notPrime[j] = true;
      }
   }
}

LL P;

vector<pll> factor;

void factorize(LL n)
{
   factor.clear();
   for(auto p: prime)
   {
      if(p*p > n)
         break;
```

```
        int cnt = 0;
        while(n%p == 0)
        {
            n/=p;
            cnt++;
        }
        if(cnt)
            factor.pb({p, cnt});
    }
    if(n != 1)
        factor.pb({n, 1});
}

bool isPrime(LL n)
{
    for(auto p: prime)
    {
        if(p*p > n)
            return true;
        if(n%p == 0)
            return false;
    }
    return true;
}

vector<LL>sltn;
void go(int pos, LL made, LL total, LL n, LL last)
{
    if(pos == 0 && total == 1)
    {
        sltn.pb(n);
        return;
    }
    if(pos == factor.size())
    {
        if(made <= last)
            return;
        if(isPrime(made+1))
        {
            n *= (made+1);
            LL d = made;
            go(0, 1, total/made, n, d);
            for(int i = 0; i<factor.size(); i++)
            {
                if(factor[i].xx == made+1)
                {
                    int cnt = factor[i].yy;
                    for(int j = 1; j<=cnt; j++)
                    {
                        made *= factor[i].xx;
                        factor[i].yy--;
                        n *= factor[i].xx;
                        assert(total%made == 0);
                        go(0, 1, total/made, n, d);
                    }
                    factor[i].yy = cnt;
                    break;
                }
            }
        }
        return;
    }

    go(pos+1, made, total, n, last);
    int cnt = factor[pos].yy;

    for(int i = 0; i<cnt; i++)
    {
        made *= factor[pos].xx;
        factor[pos].yy--;
        go(pos+1, made, total, n, last);
    }
    factor[pos].yy = cnt;
}
int main()
{
//  freopen("maxon.txt", "r", stdin);
//  freopen("out.txt", "w", stdout);
    sieve();
    while(sl(P) == 1)
    {
        sltn.clear();
        if(P == 1)
        {
            puts("1 2");
            continue;
        }
        factorize(P);

        go(0, 1, P, 1, false);

        if(sltn.size() == 0)
        {
            puts("No solution.");
            continue;
        }
        bool space = false;
        sort(all(sltn));
        for(auto v: sltn)
        {
            if(space)
                printf(" ");
            space = true;
            printf("%lld",v);
        }
        puts("");
    }
}
```

Johnson:

Johnson's algorithm consists of the following steps:[1][2]

1. First, a new node q is added to the graph, connected by zero-weight edges to each of the other nodes.
2. Second, the Bellman–Ford algorithm is used, starting from the new vertex q, to find for each vertex v the minimum weight $h(v)$ of a path from q to v. If this step detects a negative cycle, the algorithm is terminated.
3. Next the edges of the original graph are reweighted using the values computed by the Bellman–Ford algorithm: an edge from u to v, having length $w(u,v)$, is given the new length $w(u,v) + h(u) - h(v)$.
4. Finally, q is removed, and Dijkstra's algorithm is used to find the shortest paths from each node s to every other vertex in the reweighted graph.
   $dist(u,v)$ in original graph $= dist(u,v)$ in reweighted graph - $h(u) + h(v)$

O(VE + V * dijkstra())

Prims:

```
#define MAX 200000
vector<pii>E[MAX+10];
struct edgeData{
    int u, v, c;
};
bool vis[MAX+10];
bool operator < (edgeData a, edgeData b)
{
    return a.c > b.c;
}

priority_queue<edgeData>Q;

void add(int u)
{
    vis[u] = true;
    for(auto v: E[u])
    {
        if(!vis[v.xx])
            Q.push({u, v.xx, v.yy});
    }
}

void fix()
{
    while(!Q.empty())
    {
        if(vis[Q.top().u] && vis[Q.top().v])
            Q.pop();
        else
            break;
    }
}

vector<edgeData>sltn;
int prims()
{
    while(!Q.empty())
```

```
        Q.pop();
    mem(vis, 0);
    sltn.clear();

    int ret = 0;
    add(1);
    while(!Q.empty())
    {
        fix();
        if(Q.empty())
            break;
        int u = Q.top().u, v = Q.top().v, c = Q.top().c;
        sltn.pb(Q.top());
        Q.pop();
        ret += c;
        add(v);
    }
    return ret;
}
```

Radial Sort:

```
pll firstPoint;
inline int triArea2(pll a, pll b, pll c) // includes sign.. +ve -> ccw, -ve -> cw;
{
        LL ret = 0;
    ret += (LL) a.xx*b.yy + (LL) b.xx*c.yy + (LL) c.xx*a.yy - (LL) a.xx*c.yy - (LL) c.xx*b.yy - (LL)
b.xx*a.yy;
    if(ret < 0)
        return -1;
    else if(ret == 0)
        return 0;
    else
        return 1;
}

inline int getQuadrant(pll a)
{
    a.xx -= firstPoint.xx;
    a.yy -= firstPoint.yy;
    if(a.xx >= 0 && a.yy >= 0)
```

```
        return 0;
    if(a.xx  < 0 && a.yy > 0)
        return 1;
    if(a.xx <= 0 && a.yy <= 0)
        return 2;
    return 3;

}

inline bool cmp(point a, point b)
{
    int qa = getQuadrant(a.co), qb = getQuadrant(b.co);
    if(qa == qb)
    {
        int ret = triArea2(firstPoint, a.co, b.co);
        if(ret <= 0)
            return false;
        else
            return true;
    }
    else
        return qa < qb;
}
```

Simulated Annealing:

Problem:
A simplified version of the UNIX terminal is described below. You start with the empty terminal, with no history of previous commands executed. There are two ways to execute a command at any moment.
1. Type the command character by character, finally followed by a new line (press enter
) to execute the command. This requires |command| + 1 (for the enter) keystrokes in total.
2. Press the up arrow any number of times to jump to any of the previously executed commands. If you press the up arrow once, the last command you executed will automatically appear in the shell. Press enter and it shall be executed. If you press the up arrow again, the second to last command will fill up the terminal. If there is no previous command, the terminal shell will remain unchanged. So for example, if you are back at the first command executed, or if you haven't executed any commands before, pressing the up arrow will have no

effect. When you go back to any previous command, you can also add new characters to the end of it or use the backspace key any number of times to remove the last character of the text in your terminal.
Whenever you press enter, the content of the current line in your shell will be treated as a command and be executed. Also it will be added to the list of executed commands in the terminal history. Every time you execute a command by pressing enter, the text will disappear and you will start with an empty cursor in the beginning of the next line. Note that the cursor always stays at the end of the current line and you can only add or remove characters from the end. Also, you can never go back to any previous line once you press enter, pressing the backspace key when the current line is empty simple has no effect.

Given a list of distinct commands that needs to be executed, you want to complete all of them, in any order, using the minimum number of keystrokes. Please refer to the sample input/output and explanation for more clarity if required.

1 ≤ T ≤ 13

1 ≤ N ≤ 13

1 ≤ |command| ≤ 1300

Solution:
```
int n, len[15], dist[15][15];
char command[15][1310];

int getDist(int a, int b)
{
    int matched = 0;
    for(int i = 0; i<min(len[a], len[b]); i++)
    {
        if(command[a][i] != command[b][i])
            break;
        matched++;
    }
    return len[b] - matched + len[a] - matched;
}

inline int eval(vector<int>V)
{
    int ret = 0;
    for(int i = 0; i<V.size(); i++)
    {
        int tmp = len[V[i]];
        for(int j = 0; j<i; j++)
            tmp = min(tmp, dist[V[i]][V[j]] + (i-j));
        ret += tmp;
    }
    return ret;
}

inline double getProbability(int e1, int e2, bool minimize, double temperature)
{
    if(!minimize)
        e1 *= -1, e2 *= -1;
    if(e2 < e1)
        return 1;
    return exp(-(e2-e1)/temperature);
}

int solve(int iteration, double multiplier)
{
    vector<int>V;
    for(int i = 0; i<n; i++)
        V.pb(i);
    int e1 = eval(V);
    /// for better performances, temperature should be at least equal to the
highest possible answer initially.
    double temperature = 1300*13*10;
    int ret = e1;
    for(int it = 0; it < iteration; it++)
    {
        int i = rand()%n, j = rand()%n;
        swap(V[i], V[j]);
        int e2 = eval(V);
        ret = min(ret, e2);
        double p = getProbability(e1, e2, true, temperature);
        double r = (double)rand()/RAND_MAX;
        if(r <= p)
            e1 = e2;
        else
            swap(V[i], V[j]);
        if(temperature > 1e-20) temperature *= multiplier;
    }
    return ret+n;
}
```

```
//int bb = -1;
int brute()
{
//    if(bb != -1)
//        return bb;
    vector<int>V;
    for(int i = 0; i<n; i++)
        V.pb(i);
    int ret = 1e9;
    do{
        ret = min(ret, eval(V));
    }while(next_permutation(all(V)));
    return ret + V.size();
}

int main()
{
    srand(time(NULL));
    int t;
    sf(t);
    for(int cs = 1; cs<=t; cs++)
    {
        sf(n);
        getchar();
        for(int i = 0; i<n; i++)
            gets(command[i]), len[i] = strlen(command[i]);
        for(int i = 0; i<n; i++)
        {
            for(int j = 0; j<n; j++)
                dist[i][j] = getDist(i, j);
        }
        printf("Case %d: %d\n",cs,solve(2.8e6,0.99997));
    }
    return 0;
}
```

Strassen:

Divide: Whereas before we divided integers of size n bits into sub-integers of $\frac{n}{2}$ bits, with matrices we subdivide the matrices into 4 quadrants of submatrices, each of size $(\frac{n}{2} \times \frac{n}{2})$.

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Treating A and B as two 2×2 matrices, we can multiply them and write down the result:

$$\mathbf{C} = \begin{pmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{22} \\ A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{22} \end{pmatrix} \tag{2.4}$$

It is easy to verify that this is still valid even if A and B are submatrices rather than single values. We have divided the original $n \times n$ problem into eight $(\frac{n}{2} \times \frac{n}{2})$ problems. The input size has been divided by 4, but here n is the dimension of the matrix, so each problem is of size $\frac{n}{2}$ rather than $\frac{n}{4}$. The runtime is:

This naive divide-conquer algorithm did not improve our runtime, and again the problem is the number of recursive subproblems, which in this case is 8. However, like we did in the warm-up problem of multiplying two n-bit numbers, we can reduce the number of multiplications by defining the following 7 sub-multiplications:

$$\begin{aligned} P_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_2 &= (A_{21} + A_{22}) \times B_{11} \\ P_3 &= A_{11} \times (B_{12} - B_{22}) \\ P_4 &= A_{22} \times (-B_{11} + B_{21}) \\ P_5 &= (A_{11} + A_{12}) \times B_{22} \\ P_6 &= (-A_{11} + A_{21}) \times (B_{11} + B_{12}) \\ P_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \end{aligned} \tag{2.5}$$

Each of the above sub-multiplications is a product of $(\frac{n}{2} \times \frac{n}{2})$ matrices. Hence the recombination will be $\Theta(n^2)$. We claim we can compute the product $A \times B$ with these combinations:

$$\mathbf{A} \times \mathbf{B} = \begin{pmatrix} P_1 + P_4 - P_5 + P_7 & P_3 + P_5 \\ P_2 + P_4 & P_1 - P_2 + P_3 + P_6 \end{pmatrix} \tag{2.6}$$

Sublinear Sieve:

```
const int MAX = 320000000;
int smallest[MAX + 1];
vector<int> prime;

void sieve()
{
   for (int i = 2; i <= MAX; ++i)
   {
      if (smallest[i] == 0)
      {
         smallest[i] = i;
         prime.push_back(i);
      }
      for (int j = 0; j < (int)prime.size() && prime[j] <= smallest[i] && i * prime[j] <= MAX; ++j)
      {
//           assert(smallest[i*prime[j]] == 0);
         smallest[i * prime[j]] = prime[j];
      }
   }
}
```

Obscure Lemmas:

1. Number of non-isomorphic graphs on n unlabeled nodes:
   For general case, there are 2^($^n{}_2$) non-isomorphic graphs on n vertices where ($^n{}_2$) is binomial coefficient "n above 2".
2. BST(n) = catalan(n)
3. Number of rooted binary trees (n) = (2n)! / (n+1)!
4. Number of labeled forests with n nodes = number of labeled trees with n+1 nodes. (n+1)^(n-1)
5. Let $T_{n,k}$ be the number of labelled forests on n vertices, such that vertices 1, 2, ..., k all belong to different trees. Then $T_{n,k} = k\, n^{n-k-1}$.

6. Car Ci prefers space ai. If ai is occupied, then Ci takes the next available space. We call (a1; : : : ; an) a parking function (of length n) if all cars can park.

   Let f (n) be the number of parking functions of length n.
   Then f (n) = (n+1)^(n-1).
7. An ear of a polygon is defined as a vertex v such that the line segment between the two neighbors of v lies entirely in the interior of the polygon. The two ears theorem states that every simple polygon has at least two ears.
   In geometry, the **two ears theorem** states that every simple polygon with more than three vertices has at least two ears, vertices that can be removed from the polygon without introducing any crossings. The two ears theorem is equivalent to the existence of polygon triangulations.

8. The **Chicken McNugget Theorem** (or **Postage Stamp Problem** or **Frobenius Coin Problem**) states that for any two relatively prime positive integers m, n, the greatest integer that cannot be written in the form $am + bn$ for nonnegative integers a, b is $mn - m - n$.

   A consequence of the theorem is that there are exactly $\dfrac{(m-1)(n-1)}{2}$ positive integers which cannot be expressed in the form $am + bn$. The proof is based on the fact that in each pair of the form $(k, (m-1)(n-1) - k + 1)$, exactly one element is expressible.

   If m and n are not relatively prime, then we can simply rearrange $am + bn$ into the form

$$\gcd(m,n)\left(a\frac{m}{\gcd(m,n)} + b\frac{n}{\gcd(m,n)}\right)$$

$\dfrac{m}{\gcd(m,n)}$ and $\dfrac{n}{\gcd(m,n)}$ are relatively prime, so we apply Chicken McNugget to find a bound

$$\frac{mn}{\gcd(m,n)^2} - \frac{m}{\gcd(m,n)} - \frac{n}{\gcd(m,n)}$$

We can simply multiply $\gcd(m,n)$ back into the bound to get

$$\frac{mn}{\gcd(m,n)} - m - n = \operatorname{lcm}(m,n) - m - n$$

Therefore, all multiples of $\gcd(m,n)$ greater than $\operatorname{lcm}(m,n) - m - n$ are representable in the form $am + bn$ for some positive integers a, b.

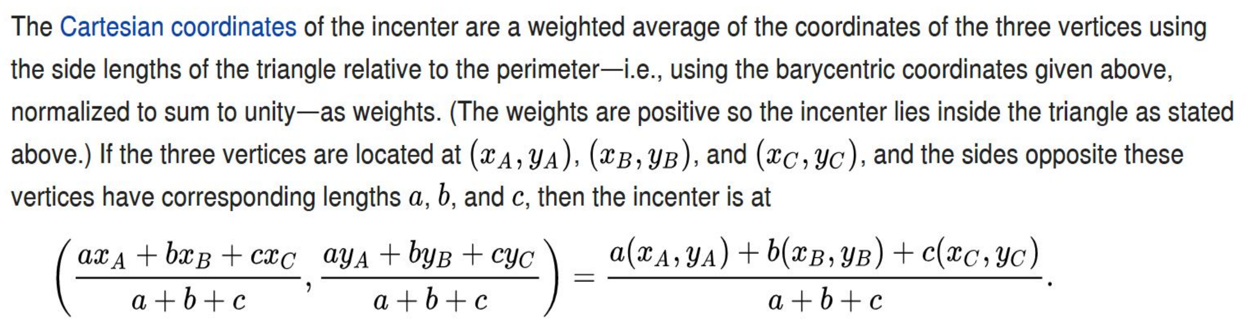
9. Vandermonde's Identity states that $\sum_{k=0}^{r} \binom{m}{k}\binom{n}{r-k} = \binom{m+n}{r}$, which can be proven combinatorially by noting that any combination of r objects from a group of $m+n$ objects must have some $0 \le k \le r$ objects from group m and the remaining from group n.
10. It states that for any prime integer $p > 2$, it can be written as $p = a$^2 $+ b$^2, where both a and b are positive integers, if and only if p mod $4 = 1$.
11. In an election where candidate A receives *p* votes and candidate B receives *q* votes with *p* > *q*, what is the probability that A will be strictly ahead of B throughout the count?" The answer is
(p-q)/(p+q)
12. Fibonacci nim is played by two players, who alternate removing coins or other counters from a pile of coins. On the first move, a player is not allowed to take all of the coins, and on each subsequent move, the number of coins removed can be any number that is at most twice the previous move. According to the normal play convention, the player who takes the last coin wins.[2] Or according to the Misère game, the player who takes the last coin loses.

    The optimal strategy in Fibonacci nim can be described in terms of the "quota" q (the maximum number of coins that can currently be removed: all but one on the first move, and up to twice the previous move after that) and the Zeckendorf representation of the current number of coins as a sum of non-consecutive Fibonacci numbers. A given position is a losing position (for the player who is about to move) when q is less than the smallest Fibonacci number in this representation, and a winning position otherwise. In a winning position, it is always a winning move to remove all the coins (if this is allowed) or otherwise to remove a number of coins equal to the smallest Fibonacci number in the Zeckendorf representation. When this is possible, the opposing player will necessarily be faced with a losing position, because the new quota will be smaller than the smallest Fibonacci number in the Zeckendorf representation of the remaining number of coins. From a losing position, any move will lead to a winning position.[1]

In particular, when there is a Fibonacci number of coins in the starting pile, the position is losing for the first player (and winning for the second player). However, when the starting number of coins is not a Fibonacci number, the first player can always win with optimal play.[2]

For the misere game of this game, if there are initially n coins, then the first player can remove n−1 coins and leave 1 coin to win.

13.



The Cartesian coordinates of the incenter are a weighted average of the coordinates of the three vertices using the side lengths of the triangle relative to the perimeter—i.e., using the barycentric coordinates given above, normalized to sum to unity—as weights. (The weights are positive so the incenter lies inside the triangle as stated above.) If the three vertices are located at (x_A, y_A), (x_B, y_B), and (x_C, y_C), and the sides opposite these vertices have corresponding lengths a, b, and c, then the incenter is at

$$\left(\frac{ax_A + bx_B + cx_C}{a+b+c}, \frac{ay_A + by_B + cy_C}{a+b+c}\right) = \frac{a(x_A, y_A) + b(x_B, y_B) + c(x_C, y_C)}{a+b+c}.$$

13.