Algorithms & Advanced Programming
ICPC SWERC and ICAIC Training

Michael Anoprenko Hadrien Barral Nour Cheour
Pascal Vanier Jill-Jênn Vie Hang Zhou

First class, again

ICPC International Collegiate Programming Contest

The 2026 ICPC World Finals Dubai

// hosted by **Dubai Chamber of Digital Economy**

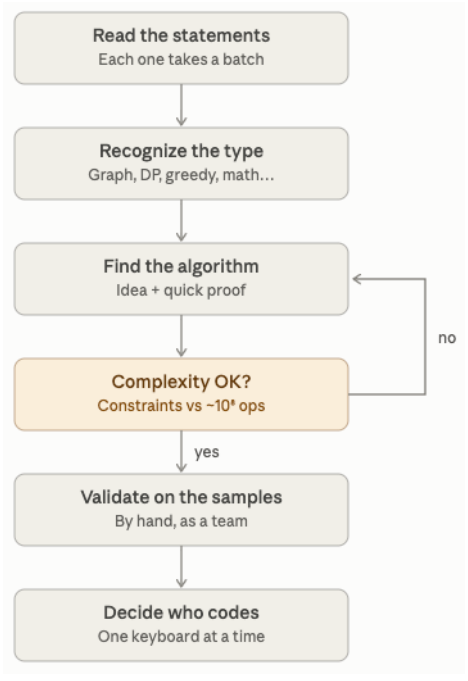
icpc.foundation



- 13 problems
- 5 hours
- 3 people on 1 keyboard



3 teams of 3 contestants per university


Read the statements
Each one takes a batch
Recognize the type
Graph, DP, greedy, math…
Find the algorithm
Idea + quick proof
Complexity OK?
Constraints vs ~10⁸ ops
no
yes
Validate on the samples
By hand, as a team
Decide who codes
One keyboard at a time

Schedule of trainings

- 10:15 AM: important information (plenary)
- 10:30 AM: you code, as a team
- 12:00 PM: problem analysis, "editorial"

Afternoon until 6 PM: engineers in computer rooms

By today

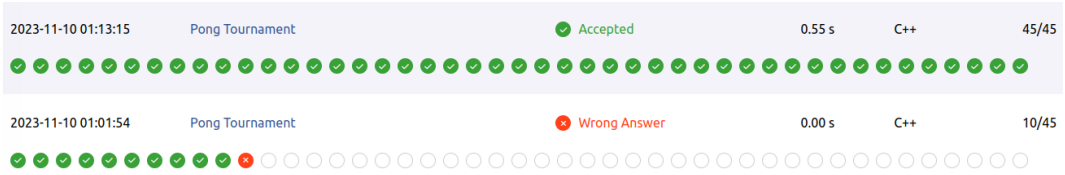
- Identify and solve an easy problem among the 13 problems of the day
- If not solved yet, attempt Frosh Week

For next week

- Solve at least Frosh Week and Longest path in a DAG

In competitive programming

- Find the right data structure (one that allows a complexity that fits within 1 sec)
- Recognize a problem and model it (reduce it to another problem) in order to give it to a solver already written in your 25-page notebook

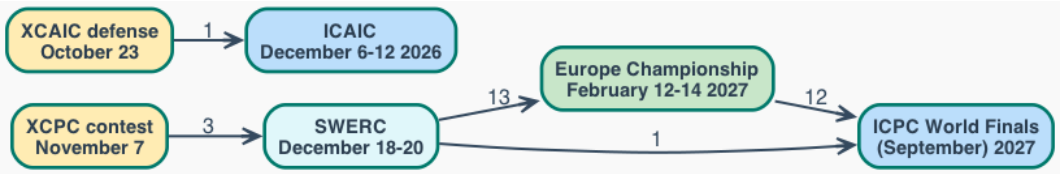


DOMjudge answers: CORRECT, WRONG-ANSWER, TIMELIMIT, RUN-ERROR, etc.

↓ last week (runtime error: memory limit exceeded, segfault, etc.)



Important dates



XCPC: local selection contest

Only C++ & documentation allowed, Internet & LLMs forbidden

- 12 problems, 5 hours, just like SWERC
- Teams of 3 on a single computer
- Optional: if you don't attend XCPC, please attempt a few problems every week on DOMjudge.

XCAIC: defense

- Kaggle and CodaLab competitions
- Defense (short meeting) on October 23

ICPC SWERC, December 18–20, Lyon, Bologna, Barcelona

cppreference.com accessible during the contest



You can bring to the contest 25 pages of code, cf. KTH's KACTL


```
    splice(e->r(), b);
    auto side = s[0].cross(s[1], s[2]);
    Q t = side ? connect(b, a) : 0;
    return (side ? 0 ? o->r() : e, side < 0 ? t : b->r() };
  }

#define H(e) e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
  Q A, B, ra, rb;
  int half = sz(s) / 2;
  tie(ra, A) = rec({all(s) - half});
  tie(rb, B) = rec({s[sz(s) - half + all(s)});
  while ({B->p.cross(H(A)) < 0 && (A = A->next())} ||
         {A->p.cross(H(B)) > 0 && (B = B->r()->o)});
  Q base = connect(B->r(), A);
  if (A->p == ra->p) ra = base->r();
  if (B->p == rb->p) rb = base;

#define DEL(e, init, dir) Q e = init->dir; if (valid(e)) \
    while (circle(e->dir->F(), H(base), e->F())) { \
      Q t = e->dir; \
      splice(e, e->prev()); \
      splice(e->r(), e->r()->prev()); \
      e->o = H; H = e; e = t; \
    }
  for (;;) {
    DEL(LC, base->r(), o); DEL(RC, base, prev());
    if (!valid(LC) && !valid(RC)) break;
    if (!valid(LC) || (valid(RC) && circ(H(RC), H(LC)))
      base = connect(RC, base->r());
    else
      base = connect(base->r(), LC->r());
  }
  return { ra, rb };
}

vector<P> trianglulate(vector<P> pts) {
  sort(all(pts)); assert(unique(all(pts)) == pts.end());
  if (sz(pts) < 2) return {};
  Q e = rec(pts).first;
  vector<Q> q = {e};
  int qi = 0;
  while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
#define ADD { Q c = e; do { c->mark = 1; pts.push_back(c->p); \
  q.push_back(c->r()); c = c->next(); } while (c != e); }
  ADD; pts.clear();
  while (qi < sz(q)) if (!(e = q[qi++])->mark) ADD;
  return pts;
}
```

8.5 3D

PolyhedronVolume.h

Description: Magic formula for the volume of a polyhedron. Faces should point outwards.

3084c3, 6 lines

```
template<class V, class L>
double signedPolyVolume(const V& p, const L& trilist) {
  double v = 0;
  for (auto i : trilist) v += p[i.a].cross(p[i.b]).dot(p[i.c]);
  return v / 6;
}
```

Point3D.h

Description: Class to handle points in 3D space. T can be e.g. double or long long.

```
template<class T> struct Point3D {
  typedef Point3D P;
  typedef const P& R;
  T x, y, z;
  explicit Point3D(T x=0, T y=0, T z=0) : x(x), y(y), z(z) {}
  bool operator<(R p) const {
    return tie(x, y, z) < tie(p.x, p.y, p.z); }
  bool operator==(R p) const {
    return tie(x, y, z) == tie(p.x, p.y, p.z); }
  P operator+(R p) const { return P(x+p.x, y+p.y, z+p.z); }
  P operator-(R p) const { return P(x-p.x, y-p.y, z-p.z); }
  P operator*(T d) const { return P(x*d, y*d, z*d); }
  P operator/(T d) const { return P(x/d, y/d, z/d); }
  T dot(R p) const { return x*p.x + y*p.y + z*p.z; }
  P cross(R p) const {
    return P(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x);
  }
  T dist2() const { return x*x + y*y + z*z; }
  double dist() const { return sqrt((double)dist2()); }
  //Azimuthal angle (longitude) to x-axis in interval [-pi, pi]
  double phi() const { return atan2(y, x); }
  //Zenith angle (latitude) to the z-axis in interval [0, pi]
  double theta() const { return atan2(sqrt(x*x+y*y),z); }
  P unit() const { return *this/(T)dist(); } //makes dist()=1
  //returns unit vector normal to *this and p
  P normal(R p) const { return cross(p).unit(); }
  //returns point rotated 'angle' radians ccw around axis
  P rotate(double angle, P axis) const {
    double s = sin(angle), c = cos(angle); P u = axis.unit();
    return u.dot(u)*(1-c) + (*this)*c - cross(u)*s;
  }
};
```

3dHull.h

Description: Computes all faces of the 3-dimension hull of a point set. *No four points must be coplanar*, or else random results will be returned. All faces will point outwards.
Time: $O(n^2)$

Point3D.h 2b4f6c, 49 lines

```
typedef Point3D<double> P3;

struct PR {
  void ins(int x) { (a == -1 ? a : b) = x; }
  void rem(int x) { (a == x ? a : b) = -1; }
  int cnt() { return (a != -1) + (b != -1); }
  int a, b;
};

struct F { P3 q; int a, b, c; };

vector<F> hull3d(const vector<P3>& A) {
  assert(sz(A) >= 4);
  vector<vector<PR>> E(sz(A), vector<PR>(sz(A), {-1, -1}));
#define E(x,y) E[f.x][f.y]
  vector<F> FS;
  auto mf = [&](int i, int j, int k, int l) {
    P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
    if (q.dot(A[l]) > q.dot(A[i]))
      q = q * -1;
    F f{q, i, j, k};
    E(a,b).ins(k); E(a,c).ins(j); E(b,c).ins(i);
    FS.push_back(f);
  };
  rep(i,0,4) rep(j,i+1,4) rep(k,j+1,4)
    mf(i, j, k, 6 - i - j - k);

  rep(i,4,sz(A)) {
    rep(j,0,sz(FS)) {
      F f = FS[j];
      if (A[i].dot(f.q) > A[f.a].dot(f.q)) { // it sees face
        E(a,b).rem(f.a);
        E(a,c).rem(f.b);
        E(b,c).rem(f.c);
        swap(FS[j--], FS.back());
        FS.pop_back();
      }
    }
    int nw = sz(FS);
    rep(j,0,nw) {
      F f = FS[j];
#define C(a, b, c) if (E(a,b).cnt() != 2) mf(f.a, f.b, i, f.c);
      C(a, b, c); C(a, c, b); C(b, c, a);
    }
  }
  for (F& it : FS) if ((A[it.b] - A[it.a]).cross(
      A[it.c] - A[it.a]).dot(it.q) <= 0) swap(it.c, it.b);
  return FS;
};
```

sphericalDistance.h

Description: Returns the shortest distance on the sphere with radius *radius* between the points with azimuthal angles (longitude) $f1$ (ϕ_1) and $f2$ (ϕ_2) from x axis and zenith angles (latitude) $t1$ (θ_1) and $t2$ (θ_2) from z axis (0 = north pole). All angles measured in radians. The algorithm starts by converting the spherical coordinates to cartesian coordinates so if that is what you have you can use only the two last rows. dz*radius is then the difference between the two points in the x direction and d*radius is the total distance between the points.

011f07, 8 lines

```
double sphericalDistance(double f1, double t1,
    double f2, double t2, double radius) {
  double dx = sin(t2)*cos(f2) - sin(t1)*cos(f1);
  double dy = sin(t2)*sin(f2) - sin(t1)*sin(f1);
  double dz = cos(t2) - cos(t1);
  double d = sqrt(dx*dx + dy*dy + dz*dz);
  return radius*2*asin(d/2);
}
```

Strings (9)

KMP.h

Description: p[x] computes the length of the longest prefix of s that ends at x, other than s[0..x] itself (abacaba -> 0010123). Can be used to find all occurrences of a string.
Time: $O(n)$

d1377c, 16 lines

```
vi pi(const string& s) {
  vi p(sz(s));
  rep(i,1,sz(s)) {
    int g = p[i-1];
    while (g && s[i] != s[g]) g = p[g-1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}

vi match(const string& s, const string& pat) {
  vi p = pi(pat + '\0' + s), res;
  rep(i,sz(p)-sz(s),sz(p))
    if (p[i] == sz(pat)) res.push_back(i - 2 * sz(pat));
  return res;
}
```

Zfunc.h

Description: z[i] computes the length of the longest common prefix of s[i:] and s, except z[0] = 0. (abacaba -> 0010301)
```
```

Strings (9)

KMP.h

Description: pi[x] computes the length of the longest prefix of s that ends at x, other than s[0...x] itself (abacaba -> 0010123). Can be used to find all occurrences of a string.

Time: $\mathcal{O}(n)$

d4375c, 16 lines

```
vi pi(const string& s) {
  vi p(sz(s));
  rep(i,1,sz(s)) {
    int g = p[i-1];
    while (g && s[i] != s[g]) g = p[g-1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}

vi match(const string& s, const string& pat) {
  vi p = pi(pat + '\0' + s), res;
  rep(i,sz(p)-sz(s),sz(p))
    if (p[i] == sz(pat)) res.push_back(i - 2 * sz(pat));
  return res;
}
```

Computing a big number modulo some prime, Bézout or extended Euclid

```
// g++ function __gcd(a, b)
// gcd(a0, a1) = a0 u0 + a1 v0
int euclid(int a0, int a1, int& u0, int& v0) {
        u0 = 1, v0 = 0;
        int u1 = 0, v1 = 1;
        while (a1 != 0)
        {
                int q = a0 / a1;
                a0 -= q * a1; swap(a0, a1);
                u0 -= q * u1; swap(u0, u1);
                v0 -= q * v1; swap(v0, v1);
        }
        return a0;
}
```

Actually, KACTL:

```
/**
 * Author: Simon Lindholm
 * Date: 2016-07-24
 * License: CC0
 * Source: Russian page
 * Description: Pre-computation of modular inverses.
 * Assumes LIM $\le$ mod and that mod is a prime.
 * Status: Works
 */
#pragma once

// const ll mod = 1000000007, LIM = 200000; ///include-line
ll* inv = new ll[LIM] - 1; inv[1] = 1;
rep(i,2,LIM) inv[i] = mod - (mod / i) * inv[mod % i] % mod;
```

Explanation on Codeforces

First Kaggle challenge from IOAI



A passage begins as human-written text and, at some point, silently switches to a continuation generated by a language model. Read as a whole, it looks like one coherent piece — but somewhere in the middle the author changes from a person to a machine. **Your task is to find that switch: the character index where the human part ends and the machine part begins.**

All XCAIC tasks

- Task 1: NLP task on Kaggle (public & private dataset)
- Task 2: Vision task ~~on CodaLab~~ on some homemade judge
- Task 3: Reinforcement learning task, probably MountainCar

This course is about algorithmic problem solving

- You don't know all subtleties of an algorithm unless you've implemented it (without any bugs).
- Combining simple techniques to solve bigger problems
- Learn to use the right data structures & methods

Judges

Input

```
9 10
##########
.....#...#
####.###.#
#..#.#...#
#..#.#.###
###..#.#.#
#.#.####.#
#........#
########.#
```

Output

```
##########
XXXXX#...#
####X###.#
#..#X#...#
#..#X#.###
###XX#.#X#
#X#X####X#
#XXXXXXXX#
#########X#
```

```
python laby.py < laby.in > laby.out   # Python

make laby
./laby < laby.in > laby.out   # C++
```

Schedule

- Contests are 10:30-18:00 on Fridays
- October: Team selection XCPC (Nov 7)
- SWERC registration deadline (team names): possibly Nov 9
- Early December: ICAIC
- Mid December: SWERC

Outline

- Pathfinding
- DP: Dynamic Programming
- Meta (strategies)
- Advanced graphs
- Matching & flows
- Advanced and dynamic data structures (segment trees)
- Maths: Arithmetics, Combinatorics and Linear algebra
- Geometry & sweep line
- Strings (suffix arrays)

It is a team competition

- Divide the work between your team
  - Identify asap the easy problems
  - Highlight the important points of the statement (bounds).
    Is it a DP? A graph problem?
- You should learn to sketch a solution and explain it to your teammates
  - Think about corner cases / edge cases for the rest of your team
- You should learn to debug each other's code

Only one keyboard

- Learn to solve problems on paper
- If a submission fails, print your code and debug it by hand in order to free the keyboard for someone else

Technical advice

- Avoid presentation errors (missing spaces, etc.)
- Think about extreme cases (empty graph)
- Think about out-of-bounds (sometimes it is better to allocate more memory)
  - E.g. integer bounds: you may need an `unsigned long long int (%lld)`
- Evaluate the complexity before implementing it
  - Sometimes it is good to code the naive solution just to debug a better one
- If there are several instances, make sure everything is cleared, notably global variables
- Upsolving after the competition: nothing left unsolved

Contests

- Let's configure VSCode for competitive programming helper
- You can `git clone` `https://github.com/kth-competitive-programming/kactl`

Contest of the day: https://domjudge.r2.enst.fr

As a backup we may use Kattis online judge.

Dynamic programming

> *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

Bellman Equation

Given a state s, we choose an action a that yields us a reward $R(s, a)$ and puts us in state s'. V indicates the average reward obtained if we act optimally.

$$V(s) = \max_a R(s, a) + \gamma V(s')$$

Does this ring a bell?

Dynamic programming

> *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

Bellman Equation

Given a state s, we choose an action a that yields us a reward $R(s, a)$ and puts us in state s'. V indicates the average reward obtained if we act optimally.

$$V(s) = \max_a R(s, a) + \gamma V(s')$$

Does this ring a bell?

Bellman-Ford

$$V(u) = \max_v -w_{u,v} + V(v) \quad V = -d$$

BTW, what shortest path algorithms do you know?

When should we use which?

Bellman-Ford

Repeat at most $|V|$ times
  For each edge $(u, v) \in E$
    $d(v) = \min(d(v), d(u) + w_{uv})$

```
const int oo = 1e9;
int n, m;
int dist[N];
vector<tuple<int, int, int>> edge;

bool bellmanford (int u0) {
        fill_n (dist, n, +oo);
        dist[u0] = 0;
        bool stable = false;
        for (int t = 0; t < n && !stable; t++) {
                stable = true;
                for (auto[u, v, c] : edge) if (dist[u] < +oo && dist[u] + c < dist[v]) {
                        dist[v] = dist[u] + c;
                        stable = false;
                }
        }
        return stable;
}
```

$d_k[v]$ = shortest length from source to v using at most k edges

Floyd-Warshall: all source–destination pairs

```
void floydwarshall(vector<vector<int>>& d) {
        // d[u][v] = c(u, v) si (u, v) arc et +oo sinon
        int n = d.size();
        for (int w = 0; w < n; w++)
                for (int u = 0; u < n; u++)
                        for (int v = 0; v < n; v++)
                                d[u][v] = min(d[u][v], d[u][w] + d[w][v]);
}
```

$d_k[u][v]$ = shortest length between u and v using only nodes $< k$

$$d(u, v) = \min_w d(u, w) + d(w, v)$$

Distributivity in semirings (thanks Bellman)

Matrix multiplication	$\sum_k a_{ik} b_{kj}$	$(+, \times)$	(Binet, 1812)
Shortest path	$\min_k d_{ik} + d_{kj}$	$(\min, +)$	(Bellman, 1958)
Most probable path	$\max_k p_{ik} p_{kj}$	$(\max, \times)$	(Viterbi, 1967)

What is the difference between Dijkstra and A*?

Both A* and Dijkstra are Best-first search

Algorithm Best-first search

Put *source* in the priority queue
while queue is not empty **do**
  Extract the node u having minimal priority $f(u)$
  if u is *target* **then return** dist, prec
  for all neighbor v of node u **do**
    candidate $\leftarrow$ dist(u) + edge weight w_{uv}
    if it's a better candidate i.e. candidate $<$ dist(neighbor) **then**
      dist(neighbor v) $\leftarrow$ candidate
      prec(neighbor v) $\leftarrow$ node u
      Add v to queue with priority $f(v)$

Priority values $f(u)$

- Dijkstra: shortest distance(source, node u)
- *Greedy best-first search*: $h(u)$ distance as the crow flies to target
- A*: ascending $dist(u) + h(u)$
- Prim (minimal spanning tree): distance to the current tree

Knapsack

We are given n objects of sizes $c_1, \ldots, c_n \in \mathbb{N}$ and values $v_1, \ldots, v_n$. Given a knapsack of capacity C, what is the highest value one can obtain using objects of max total size C?

Knapsack

We are given n objects of sizes $c_1, \ldots, c_n \in \mathbb{N}$ and values $v_1, \ldots, v_n$. Given a knapsack of capacity C, what is the highest value one can obtain using objects of max total size C?

States: (i first objects, capacity c)

Let us call $maxValue[i][c]$ the highest value one can obtain with first $i \in [1, n]$ objects and capacity $c \in [0, C]$. First, initialize. Then:

For the ith object:

- Either we take it, and go back to ($i - 1$, $c - c_i$ state) if exists

$$v_i + maxValue[i-1][c-c_i]$$

- Or we don't: go to $(i - 1, c)$ state

$$maxValue[i-1][c]$$

Knapsack

We are given n objects of sizes $c_1, \ldots, c_n \in \mathbb{N}$ and values $v_1, \ldots, v_n$. Given a knapsack of capacity C, what is the highest value one can obtain using objects of max total size C?

States: (i first objects, capacity c)

Let us call $maxValue[i][c]$ the highest value one can obtain with first $i \in [1, n]$ objects and capacity $c \in [0, C]$. First, initialize. Then:

For the ith object:

- Either we take it, and go back to ($i - 1$, $c - c_i$ state) if exists

$$v_i + maxValue[i-1][c-c_i]$$

- Or we don't: go to $(i - 1, c)$ state

$$maxValue[i-1][c]$$

Variants (besides coin change)

- Taking several times the same object instead of once
- Does this work with negative values? Negative capacities?

DP method

1. Try to identify states.
2. Find the recurrence relation.
3. If memoization: use `std::map`. If DP: initialize well.

Application to shortest paths

- Bellman-Ford: $d_k[v]$ = shortest length from source to v using at most k edges

$$d(v) = \min_u d(u) + w_{uv}$$

- Floyd-Warshall: $d_k[u][v]$ = shortest length between u and v using only nodes $< k$

$$d(u, v) = \min_w d(u, w) + d(w, v)$$