Algorithms & Advanced Programming
ICPC SWERC Training

Jill-Jênn Vie   Hang Zhou

First class

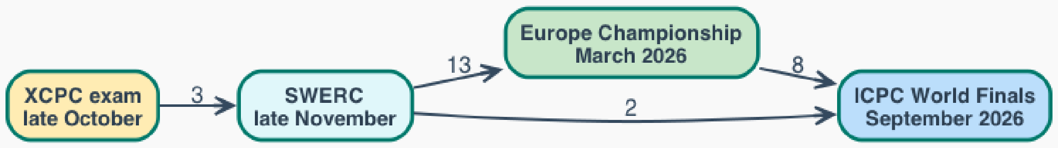
This course is about algorithmic problem solving

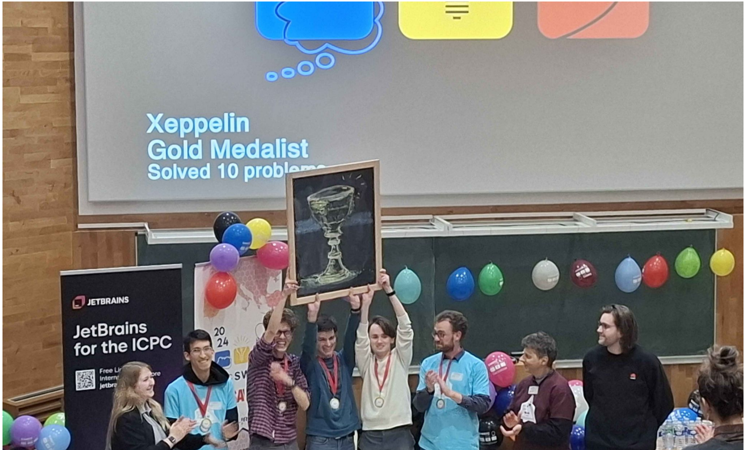
- You don't know an algorithm unless you've implemented it (without any bugs).
- Combining simple techniques to solve bigger problems
- Learn to use the right data structures & methods

Judge answers: AC (accepted), WA (wrong answer), TLE (time limit exceeded), etc.

2023-11-10 01:13:15	Pong Tournament	Accepted	0.55 s	C++	45/45
2023-11-10 01:01:54	Pong Tournament	Wrong Answer	0.00 s	C++	10/45

International Collegiate Programming Contest (since 1977!)





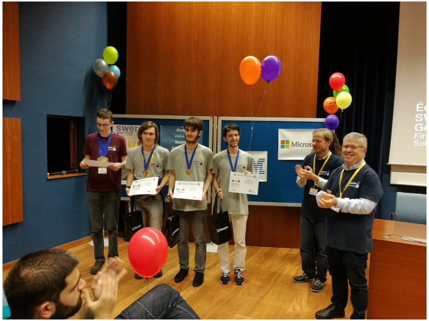
XCPC: this course exam

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

- 6 problems, 4 hours
- Teams of 2 or 3 on (multiple?) computers
- Solving 1 problem validates the MODAL (A, 20/20)
- If you cannot come to the exam: we monitor your performance on online judges (SPOJ, Kattis, DOMjudge)

Tentative date: Friday October 17

ICPC SWERC, November 21–23, Lyon, Lisbon, Pisa



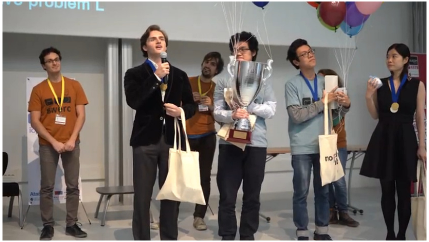
- 13 problems
- 5 hours
- 3 people
- 1 keyboard

`swerc.eu`

icpc International Collegiate Programming Contest

icpc.foundation

advancing the art and sport of competitive programming



3 teams per university/school

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 10:00-12:00 on Mondays and 10:30-18:00 on Fridays
- October: Team selection XCPC (Oct 17?)
- SWERC registration deadline (team names): Oct 27
- End of November: 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
- Set up an account on https://open.kattis.com and tell me your username
- Hang will use SPOJ https://www.spoj.com

Contest of the day: https://open.kattis.com/contests/wbuqao

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 $< k$

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