

# Kcits970 Problem Solving Handbook

When I learn algorithms/techniques, I often forget their correctness proofs. This handbook contains all algorithms that I have learned, but forgotten the proofs for. Please note that the proofs included here are not intended for complete rigor, but to provide a convincing, memorable argument.

## 1 Longest Increasing Subsequence

Reference: [https://cp-algorithms.com/dynamic\\_programming/longest\\_increasing\\_subsequence.html](https://cp-algorithms.com/dynamic_programming/longest_increasing_subsequence.html)

Implementation: `./code/arr/lis.cpp`

1. Let  $A$  be the given sequence of length  $n$ .
2. Let  $f = (A_1, \underbrace{\infty, \infty, \dots, \infty}_{n-1})$ .
3. For each  $i$  from 2 to  $n$ , binary search the smallest  $l$  such that  $a_i \leq f_l$  and update  $f_l$  to  $A_i$ .
4. The largest  $l$  such that  $f_l < \infty$  is the length of the longest increasing subsequence of  $A$ .

Define  $g_{i,l}$  as the smallest terminal element across all increasing subsequences of length  $l$  within the  $i$ th prefix of  $A$ . We show that in the above algorithm,  $f_l$  is equal to  $g_{i,l}$  after the  $i$ th iteration. (Then the rest of the correctness should be trivial.) Notice the following recurrence relation of  $g$ .

$$g_{i,l} = \begin{cases} A_i & (g_{i-1,l-1} < A_i \leq g_{i-1,l}) \\ g_{i-1,l} & (\text{otherwise}) \end{cases}$$

Additionally, for all  $i$ ,  $g_{i,l}$  must be strictly increasing in respect to  $l$ . This is because if  $g_{i,l-1} \geq g_{i,l}$ , we can obtain an increasing subsequence of length  $l-1$  with a terminal element smaller than  $g_{i,l-1}$ . Therefore, for all  $A_i$ , there exists exactly one  $l$  such that  $g_{i-1,l-1} < A_i \leq g_{i-1,l}$ , and the recurrence relation of  $g$  matches exactly the operations being done on  $f$ . Since the initial state of  $f$  is equal to  $g_{1,l}$  (trivially), the principle of induction tells us that  $f$  after the  $i$ th iteration equals  $g_{i,l}$ .

## 2 Strongly Connected Component

Reference: <https://cp-algorithms.com/graph/strongly-connected-components.html>

Implementation: `./code/graph/scc.cpp`

1. Given a directed graph  $G$ , construct its transpose  $H$ .
2. Sort the vertices of  $G$  in the order of dfs exit time.
3. Find an *unvisited* vertex  $u$  with the greatest exit time, and collect all *unvisited* vertices reachable from  $u$  in  $H$ .
4. Repeat the above until all vertices are *visited*.

Consider two different strongly connected components  $S$  and  $T$  in  $G$ . If there exists an edge from  $S$  to  $T$  in the condensation graph  $C$ , the dfs exit time of  $S$  must be greater than that of

$T$ . This can be proven by splitting the dfs entry order into two cases. (The rest is kind of trivial, so it is omitted here.)

$$\begin{cases} 1. S \text{ is visited first.} \dots \\ 2. T \text{ is visited first.} \dots \end{cases}$$

Therefore, the strongly connected component containing  $u$  (the vertex with the greatest exit time) must not have any incoming edges from other strongly connected components in  $G$ . Hence, by collecting all vertices reachable from  $u$  in the transpose, we end up with a strongly connected component. (This is valid because the set of all strongly connected components of  $G$  is equal to that of  $H$ .) Thus, repeating until exhaustion must correctly find all strongly connected components.

### 3 Cross Product of 2D Vectors

1. Let  $\mathbf{u} = (a, b)$ ,  $\mathbf{v} = (c, d)$ , and  $\theta$  be the angle from  $\mathbf{u}$  to  $\mathbf{v}$  measured in counterclockwise direction. ( $0 \leq \theta < 2\pi$ )
2. The signed area of the parallelogram formed by  $\mathbf{u}$  and  $\mathbf{v}$  is equal to  $ad - bc$ , and is notated as  $\mathbf{u} \times \mathbf{v}$ .

It suffices to show that  $ad - bc = |\mathbf{u}||\mathbf{v}|\sin\theta$ . First, we use the rotation matrix to find a different expression for  $\mathbf{v}$ .

$$\mathbf{v} = \sqrt{\frac{c^2 + d^2}{a^2 + b^2}} \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \sqrt{\frac{c^2 + d^2}{a^2 + b^2}} \begin{pmatrix} a\cos\theta - b\sin\theta \\ a\sin\theta + b\cos\theta \end{pmatrix}$$

Then we calculate  $ad - bc$  using the alternate expression.

$$\begin{aligned} ad - bc &= \sqrt{\frac{c^2 + d^2}{a^2 + b^2}} (a(a\sin\theta + b\cos\theta) - b(a\cos\theta - b\sin\theta)) \\ &= \sqrt{\frac{c^2 + d^2}{a^2 + b^2}} (a^2\sin\theta + b^2\sin\theta) \\ &= \sqrt{a^2 + b^2} \sqrt{c^2 + d^2} \sin\theta \\ &= |\mathbf{u}||\mathbf{v}|\sin\theta \end{aligned}$$

This fact implies that  $\mathbf{v}$  is oriented counterclockwise to  $\mathbf{u}$  if  $\mathbf{u} \times \mathbf{v}$  is positive, and clockwise if  $\mathbf{u} \times \mathbf{v}$  is negative. The cross product of zero indicates that the two vectors are parallel to each other.

### 4 Orientation of 3 Points

Implementation: `./code/geom/ccw.cpp`

This algorithm is also known as *CCW*.

1. Let  $A, B, C$  be arbitrary *distinct* points on a plane.
2. The orientation of  $A, B, C$  is counterclockwise if  $\overrightarrow{AB} \times \overrightarrow{BC} > 0$ , and clockwise if  $\overrightarrow{AB} \times \overrightarrow{BC} < 0$ .
3. If the cross product is zero, then the three points are on a line.

This is a direct derivation from the cross product of 2D vectors.

## 5 Line Segment Intersection Check

Implementation: `./code/geom/isect.cpp`

1. Let  $A, B, C, D$  be arbitrary *distinct* points on a plane.
2. If  $A, B, C$  and  $A, B, D$  are oriented in the same direction, then  $\overline{AB}$  and  $\overline{CD}$  do not intersect.
3. The same argument applies to the orientations of  $C, D, A$  and  $C, D, B$ .
4. If the four points are on a line, then the two segments intersect if and only if one of the following conditions hold.
  - The rightmost endpoint of  $\overline{AB}$  is to the right of the leftmost endpoint of  $\overline{CD}$ .
  - The leftmost endpoint of  $\overline{AB}$  is to the left of the rightmost endpoint of  $\overline{CD}$ .
  - The uppermost endpoint of  $\overline{AB}$  is above the lowermost endpoint of  $\overline{CD}$ .
  - The lowermost endpoint of  $\overline{AB}$  is below the uppermost endpoint of  $\overline{CD}$ .

The correctness of this algorithm can be shown visually, but I don't really want to write a dedicated TikZ diagram for it. Also, it should be noted that when implementing the algorithm, the usage of  $<, \leq, >, \geq$  may depend on how the problem sets the condition for an *intersection*.

## 6 Convex Hull

Reference: <https://cp-algorithms.com/geometry/convex-hull.html>

Implementation: `./code/geom/cvxh.cpp`

This algorithm is also called *Graham scan*.

1. Given a set of points  $S$ , let  $P_0 \in S$  be the point with the smallest  $x$ -value. (If there are multiple, select the one with the smallest  $y$ -value.)
2. Sort the points of  $S$  with respect to the polar angle from  $P_0$ , with  $P_0$  being first in order. (If multiple points have the same angle, break ties by amplitude.)
3. Maintain a stack  $H$  to keep the points of the convex hull.
4. Let  $a, b$  denote the two topmost points of  $H$ . For each point  $v$  in  $S$ , pop  $H$  while  $b, a, v$  are not oriented in counterclockwise order, then push  $v$  into  $H$ .
5.  $H$  contains the points of the convex hull in counterclockwise order.

It can be shown by induction that after the  $i$ th iteration,  $H$  contains the points of the convex hull of the first  $i$  points in counterclockwise order. To visualize the induction argument, one can imagine a ray from  $P_0$  rotating in counterclockwise direction.

## 7 Diameter of a Tree

Reference: <https://codeforces.com/blog/entry/101271>

Implementation: `./code/tree/diam.cpp`

If the weights can be negative, then this algorithm does not work in the general case.

1. Let  $T$  be a given tree with non-negative weights, and  $p$  be an arbitrary node on  $T$ .
2. Let  $a$  be any farthest node from  $p$ .
3. Let  $b$  be any farthest node from  $a$ .
4. The path from  $a$  to  $b$  is one of the diameters of  $T$ .

Let  $P$  denote the path from  $a$  to  $b$ . For each vertex  $v$ , define  $r_v$  as the first *reachable* vertex on  $P$  from  $v$ . First, we show that for every vertex  $v$ ,  $\text{dist}(r_v, v) \leq \text{dist}(r_v, b)$ . By definition of  $b$ ,  $\text{dist}(a, v) \leq \text{dist}(a, b)$ . Subtracting  $\text{dist}(a, r_v)$  on both sides immediately yields  $\text{dist}(r_v, v) \leq \text{dist}(r_v, b)$ .

Next, we proceed to show  $\text{dist}(r_v, v) \leq \text{dist}(r_v, a)$  for every vertex  $v$ . Given an arbitrary  $v$ , the position of  $r_v$  splits into two cases.

$$\begin{cases} r_v \text{ exists on the path from } a \text{ (inclusive) to } r_p \text{ (exclusive).} \\ r_v \text{ exists on the path from } r_p \text{ (inclusive) to } b \text{ (inclusive).} \end{cases}$$

In the former case,  $\text{dist}(r_v, v) \leq \text{dist}(r_v, a)$  must hold, because otherwise it contradicts the fact that  $a$  is farthest from  $p$ . In the latter case,  $\text{dist}(r_v, b) \leq \text{dist}(r_v, a)$ , because otherwise  $b$  is farther away from  $p$  than  $a$ . Combining this with the previously derived inequality  $\text{dist}(r_v, v) \leq \text{dist}(r_v, b)$  returns  $\text{dist}(r_v, v) \leq \text{dist}(r_v, a)$ .

Finally, we show that  $\text{dist}(u, v) \leq \text{dist}(a, b)$  for every pair of vertices  $u$  and  $v$ . Without loss of generality, assume that  $r_u$  exists on the path from  $a$  to  $r_v$ .

$$\begin{aligned} \text{dist}(u, v) &\leq \text{dist}(u, r_u) + \text{dist}(r_u, r_v) + \text{dist}(r_v, v) \\ &\leq \text{dist}(a, r_u) + \text{dist}(r_u, r_v) + \text{dist}(r_v, b) \\ &= \text{dist}(a, b) \end{aligned}$$

We have proven that  $\text{dist}(a, b)$  is greater than or equal to every  $\text{dist}(u, v)$ ; hence the algorithm correctly finds the diameter of  $T$ .

## 8 Chinese Remainder Theorem

Reference: <https://math.stackexchange.com/a/1644698>

Implementation: `./code/math/crt.py`

1. Consider the system of modular equations given as below.

$$\begin{cases} x \equiv a \pmod{m} \\ x \equiv b \pmod{n} \end{cases}$$

2. Let  $g = \gcd(m, n)$ . If  $g \nmid a - b$ , then no solutions exist.
3. Let  $u, v$  be any pair of Bézout coefficients for  $m, n$ .
4.  $x = a - \frac{a-b}{g}mu$  is the unique solution in modulo  $\text{lcm}(m, n)$ .

We first show the correctness of the gcd divisibility condition. From the above modular equations, we derive the following.

$$\begin{cases} x \equiv a \pmod{m} \\ x \equiv b \pmod{n} \end{cases} \longrightarrow \begin{cases} m \mid x - a \\ n \mid x - b \end{cases} \longrightarrow \begin{cases} g \mid x - a \\ g \mid x - b \end{cases} \longrightarrow \begin{cases} x \equiv a \pmod{g} \\ x \equiv b \pmod{g} \end{cases}$$

From here, it is trivial that no solutions for  $x$  exist if  $a$  is not congruent to  $b$  in modulo  $g$ . Hence,  $g$  must divide  $a - b$  in order for a solution to exist.

If  $g$  divides  $a - b$ , then a solution can be constructed.

$$\begin{aligned} mu + nv &= g \\ \longrightarrow \frac{a-b}{g}mu + \frac{a-b}{g}nv &= a-b \\ \longrightarrow a - \frac{a-b}{g}mu &= b + \frac{a-b}{g}nv \end{aligned}$$

Let  $a - \frac{a-b}{g}mu = b + \frac{a-b}{g}nv = x_0$ . Clearly,  $x_0$  is congruent to  $a$ ,  $b$  in modulo  $m$ ,  $n$ , respectively. Furthermore, this solution is unique in modulo  $\text{lcm}(m, n)$ . Assume the existence of another solution  $x_1$ . Then  $x_0$  must be congruent to  $x_1$  in both modulo  $m$  and  $n$ , implying modular congruence in  $\text{lcm}(m, n)$  as well. The exact derivation is shown below.

$$\begin{aligned} \begin{cases} x_0 \equiv x_1 \pmod{m} \\ x_0 \equiv x_1 \pmod{n} \end{cases} &\longrightarrow \begin{cases} m \mid x_0 - x_1 \\ n \mid x_0 - x_1 \end{cases} \\ &\longrightarrow \text{lcm}(m, n) \mid x_0 - x_1 \\ &\longrightarrow x_0 \equiv x_1 \pmod{\text{lcm}(m, n)} \end{aligned}$$