Competitive Programming Algorithms

David del Val

September 17, 2021

Contents

1 Range queries **3**
- 1.1 Types of range queries . . . 3
  - 1.1.1 LCA using RMQ . . . 4
- 1.2 Sparse table . . . 5
  - 1.2.1 Example . . . 6
  - 1.2.2 Sparse Table (sort of) with $\mathcal{O}(n)$ memory . . . 8
- 1.3 Fenwick tree (BIT) . . . 12
- 1.4 Segment Trees . . . 14
  - 1.4.1 Lazy propagation . . . 15
  - 1.4.2 Range update and query . . . 16
  - 1.4.3 Persistent segment trees . . . 19

2 Graphs **25**
- 2.1 Dijkstra's . . . 25
  - 2.1.1 Dijkstra's with cost and distance . . . 27
- 2.2 Bellman Ford's . . . 28
- 2.3 Warshall's . . . 29
- 2.4 DFS . . . 31
  - 2.4.1 Articulation points and bridges . . . 31
  - 2.4.2 Trajan's algorithm for strongly connected components . . . 34
  - 2.4.3 Find all cycles in a directed graph . . . 37
- 2.5 Kosaraju's . . . 38
- 2.6 Kruskal's . . . 43
  - 2.6.1 UFDS . . . 43
  - 2.6.2 Kruskal's . . . 44
- 2.7 Flows and cuts . . . 45
  - 2.7.1 Edmonds-Karp's (Max Flow) . . . 45
- 2.8 Directed Acyclic Graphs (DAG) . . . 47
  - 2.8.1 Topological sort . . . 47
- 2.9 Bipartite Graphs . . . 50
  - 2.9.1 Algorithms to find MCBM . . . 52
  - 2.9.2 Assignment problem . . . 55

3 Mathematics **62**
3.1 General calculations . . . 62
3.1.1 Binary exponentiation . . . 62
3.2 Modular arithmetic . . . 63
3.2.1 Inverses . . . 63
3.2.2 Iterative extended euclidean algorithm . . . 63
3.2.3 Linear Diophantine Equations . . . 65
3.3 Catalan numbers . . . 68
3.4 Permutations . . . 70
3.4.1 Converting one permutation into another . . . 70
3.4.2 Converting one permutation into another using adjacent swaps . . . 71

4 Strings **68**
4.1 Definitions . . . 68
4.2 Prefix function . . . 68
4.2.1 Remarks . . . 68
4.2.2 Calculation . . . 69
4.2.3 KMP . . . 70
4.2.4 Other applications . . . 71
4.3 Suffix Array . . . 73
4.3.1 Construction . . . 73
4.3.2 Longest common prefix (LCP) . . . 77
4.3.3 Applications . . . 79

5 Geometry **82**
5.1 Area of polygon . . . 82

6 Ad-hoc **84**

Chapter 3

Mathematics

3.1 General calculations

3.1.1 Binary exponentiation

Binary exponentiation calculates a power in logarithmic time. Furthermore, it can be used for modular arithmetic:

```
20 ll power(ll base, ll exp) {
21     if (exp == 0) {
22         return 1;
23     }
24     ll res = power(base, exp / 2);
25     res = (res * res) % mod;
26     if (exp % 2) {
27         res *= base;
28     }
29     return res % mod;
30 }
```

Running time: $\mathcal{O}(\log(\text{exp}))$

3.2 Modular arithmetic

3.2.1 Inverses

To calculate a modular inverse, we will use Fermat's little theorem:

$$a^{p-1} \equiv 1 \bmod p \implies a^{p-2} \equiv a^{-1} \mod p$$

If we combine this fact with binary exponentiation, we can obtain the modular inverse in logarithmic time:

```
ll inverse(ll num) {
    return power(num, mod - 2);
}
```

Running time: $\mathcal{O}$(log(mod))

3.2.2 Iterative extended euclidean algorithm

Given two integers a, b, this algorithm calculates their gcd and two coefficients x, y such that:

$$a \cdot x + b \cdot y = g \tag{3.1}$$

```
ll gcdEx(ll a, ll b, ll &x, ll &y) {
    ll x1 = 0, y1 = 1;
    ll a1 = a, b1 = b;
    x = 1;
    y = 0;

    while (b1) {
        ll q = a1 / b1;
        tie(x, x1) = pii(x1, x - q * x1);
        tie(y, y1) = pii(y1, y - q * y1);
        tie(a1, b1) = pii(b1, a1 - q * b1);
    }
    return a1;
}
```

Running time: $\mathcal{O}$(log(min(n, m)))

Explanation

The correctness of the calculation of the gcd follows from the correctness of the traditional implementation since, in each iteration, line 55 is equivalent to:

$$\begin{aligned} a &\leftarrow b \\ b &\leftarrow a \bmod b \end{aligned}$$

and we know that $\gcd(a, b) = \gcd(b, \gcd(a, b))$.

The correctness of the calculated coefficients x, y is slightly more complex. To understand it, we have to introduce the following loop invariant

$$x \cdot a + y \cdot b = a_1 \quad x_1 \cdot a + y_1 \cdot b = b_1 \tag{3.2}$$

Before the loop starts, these two equations are trivial since:

$$(x, y) = (1, 0) \quad (x_1, y_1) = (0, 1) \quad (a_1, b_1) = (a, b)$$

Now, we must prove that the invariant holds from one iteration to the next. To do so, we can denote the values of the variable x in the next iteration as x'. Therefore, we can assume that (3.2) holds true and we also know the values that the variables take in the next iteration:

$$\begin{aligned} x' &\leftarrow x_1 & \qquad a' &\leftarrow a \\ y' &\leftarrow y_1 & \qquad b' &\leftarrow b \\ x'_1 &\leftarrow x - q \cdot x_1 & \qquad a'_1 &\leftarrow b_1 \\ y'_1 &\leftarrow y - q \cdot y_1 & \qquad b'_1 &\leftarrow a_1 - q \cdot b_1 \end{aligned}$$

Now we can easily prove the corresponding equations:

$$x' \cdot a' + y' \cdot b' = a'_1 \iff x_1 \cdot a + y_1 \cdot b = b_1 \iff (3.2)$$

$$\begin{aligned} x'_1 \cdot a' + y'_1 \cdot b' = b'_1 &\iff (x - q \cdot x_1) \cdot a + (y - q \cdot y_1) \cdot b = a_1 - q \cdot b_1 \iff \\ &\iff x \cdot a + y \cdot b - q(x_1 \cdot a + y_1 \cdot b) = a_1 - q \cdot b_1 \\ &\overset{3.2}{\iff} a_1 - q(b_1) = a_1 - q \cdot b_1 \end{aligned}$$

□

3.2.3 Linear Diophantine Equations

A Linear Diophantine Equation is an equation of the general form:

$$a \cdot x + b \cdot y = c$$

where $a, b, c \in \mathbb{Z}$ are given constants and $x, y \in \mathbb{Z}$ are the unknowns.

Algorithm

If one of the constants is 0, finding the solutions is trivial. Therefore, in the following, we will assume that they are not 0. Furthermore, we have to keep in mind that:

$$\textit{The solution has at least one solution} \iff c \mid \gcd(a, b)$$

Then, we can use the result of the Extended Euclidean Algorithm to obtain a solution:

$$(3.1) \implies a \cdot x_g + b \cdot y_g = g \implies a \cdot x_g \cdot \frac{c}{g} + b \cdot y_g \cdot \frac{c}{g} = g \cdot \frac{c}{g}$$

Therefore, one solution is:

$$x_0 = x_g \cdot \frac{c}{g} \qquad y_0 = y_g \cdot \frac{c}{g}$$

However, that is not the only solution. It can be proven that the set of pairs (x, y) that satisfy the equations is:

$$\left\{ \left(x_0 + k \cdot \frac{b}{g}, y_0 - k \cdot \frac{a}{g} \right) : k \in \mathbb{Z} \right\}$$

One interesting problem that we will be able to solve is finding the values of k for which the values of a and b are in certain ranges. This is a relatively simple task since we can first find the range of k that yields a valid x value and then find the range of k that yields a valid y value. The intersection will be the answer.

Furthermore, since the functions that return the values of x and y for a given k are linear, the range of k that leads to values in an interval is very easy to calculate:

$$k \longrightarrow x_0 + k \cdot \frac{b}{g} \qquad k \longrightarrow y_0 + k \cdot \frac{a}{g}$$

Note that we have to be careful when an endpoint of an interval has the value `inf` to avoid precision and overflow issues.

```
pll restrictRange(pll range, ll n, ll m, pll imageRange) {
    int sign = m > 0 ? 1 : -1;
    ll low = -sign * inf, high = sign * inf;

    if (imageRange.fi != -inf) {
        low = ((imageRange.fi - n) / m);
        if (low * m + n < imageRange.fi) {
            low += sign;
        }
    }

    if (imageRange.se != inf) {
        high = ((imageRange.se - n) / m);
        if (high * m + n > imageRange.se) {
            high -= sign;
        }
    }

    // Maybe reverse the endpoints
    std::tie(low, high) = pll(min(low, high), max(low, high));

    return {max(low, range.fi), min(high, range.se)};
}
```

Final implementation

To solve the Linear Diophantine Equations, we will use a class to that stores the three coefficients as well as the base solution. Then, we can query the class for the solutions of the equation in a given range using the method `findRangeOfK`

```
struct DioEq {
    // Equation of the form ax + by = c
    // where x,y are the unknowns
    ll a, b, c;
    ll x0, y0, g;
    pll rangeOfK = {0, -1};
    bool solvable = true;

    DioEq(){};
    DioEq(ll _a, ll _b, ll _c) : a(_a), b(_b), c(_c) {
        // Find a solution
        ll xg, yg;
        g = gcdEx(abs(a), abs(b), xg, yg);
        if (c % g != 0) {
            this->solvable = false;
```

```
        }
        x0 = xg * (c / g);
        y0 = yg * (c / g);
        if (a < 0) x0 = -x0;
        if (b < 0) y0 = -y0;
    }

    pll findRangeOfK(pll rangeX, pll rangeY) {
        pll rangeK(-inf, inf);
        if (!this->solvable) {
            return {0, -1};
        }
        rangeK = restrictRange(rangeK, x0, b / g, rangeX);
        rangeK = restrictRange(rangeK, y0, -a / g, rangeY);
        return rangeK;
    }
};
```

Running time: $\mathcal{O}(\log(\min(n, m)))$

For instance, if we want to get the solutions that are positive in both variables, we can write:

```
DioEq e1(a, b, d);
pll rangeK = e1.findRangeOfK({0, inf}, {0, inf});
```

Remarks

- Diophantine Linear Equations with three variables are simple to solve. For instance, to solve

  $$a \cdot x + b \cdot y + c \cdot z = d$$

  we can first solve

  $$a \cdot x_1 + b \cdot x_2 = \gcd(a, b)$$

  and then solve

  $$\gcd(a, b) \cdot x_2 + c \cdot y_2 = d$$

  And the final answers will be:

  $$x = x_1 \cdot x_2 \quad y = y_1 \cdot x_2 \quad z = y_2$$

  This process works because the linear combination of a and b can only result in numbers that are multiples of their gcd.

  However, we cannot reuse the way to calculate the number of solutions (with given ranges for x and y)

3.3 Catalan numbers

We define the nth Catalan number as:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{1}{n+1}\frac{(2n)!}{n!\,n!}$$

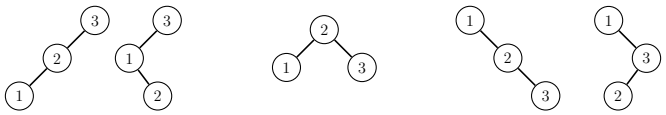
We can also define them recursively:

$$C_0 = 1 \qquad C_{n+1} = \sum_{i=0}^{n} \left(C_i\, C_{n-i}\right)$$

They can be used to solve many different problems. For instance:

- Number of different binary trees of n nodes. We can look at the specific case $n = 3$. As we can see in the figure below, $C_3 = 5$. Furthermore, we can clearly identify the recursive relationship:

$$C_3 = (3 \text{ is root}) + (2 \text{ is root}) + (1 \text{ is root}) = C_2 \cdot C_0 + C_1 \cdot C_1 + C_0 \cdot C_2$$



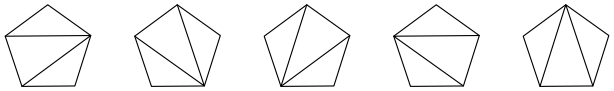
- Number of expressions that contain n pairs of parenthesis correctly paired. For instance, for $n = 3$, there only $C_3 = 5$ valid arrangements are:

$$()()() \quad ()(()) \quad (())() \quad ((())) \quad (()())$$

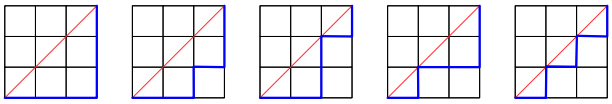
- Number of different ways of arranging $n+1$ factors in parenthesis In the case of $n = 3$, we can use the factors a, b, c, d. Then, we will have the following options:

$$(ab)(cd) \quad a(b(cd)) \quad ((ab)c)d \quad (a(bc))d \quad a((bc)d)$$

- Number of ways to split a convex polygon of $n + 2$ sides into triangles.



- Number of paths from the bottom-left corner to the top-right corner of a $n \times n$ grid moving only up or right at every point and while not reaching the diagonal.



3.4 Permutations

3.4.1 Converting one permutation into another

In this problem we are given two permutations of the same size and we are tasked with converting one of them into the other. In order to do so, we will swap the elements at two positions in the permutation zero or more times.

	0	1	2	3	4	5	6
A =	1	2	3	4	5	7	6

	0	1	2	3	4	5	6
B =	1	3	2	4	5	7	6

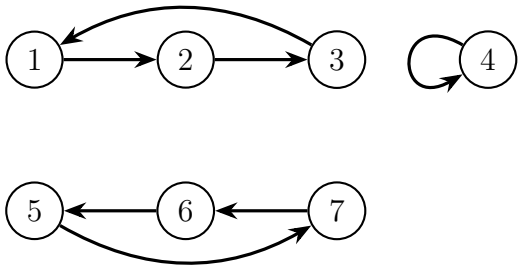
In this case, the most efficient way is to swap positions 1 and 2 of permutation B to get permutation A. From examples like this one, we can deduce the following observations:

- One swap will fix at most two numbers out of place.
- We will need at most $n-1$ swaps to change one permutation into the other. Note that we can "skip" the swap that corresponds to the last element since, at that point, by the Pigeonhole Principle, all elements must be in the correct position.

To solve the problem, we shall change the representation to a graph where the edge `a` $\rightarrow$ `b` represents that there exists a position `j` such that `A[j]=a, B[j]=b`. Since the previous example was extraordinarily simple, we will present a new one to show this process:

	0	1	2	3	4	5	6
A =	1	2	3	4	5	6	7

	0	1	2	3	4	5	6
B =	2	3	1	4	7	5	6



This graph will fulfill the following properties:

- All edges are part of a cycle (possibly of length 1). This is due to the fact that all vertices have an in-degree and out-degree of 1.
- Since all edges form cycles, connectivity and strong connectivity are equivalent. In particular, we can make the edges undirected without altering the connectivity.

The final piece of the puzzle is noticing that the elements that form a cycle of length k need exactly $k-1$ swaps to be rearranged. In order to prove this, we can use induction.

If $k \leq 2$, it is trivial. We can now assume that we have a cycle of length k. When we swap the first two elements, we can either fix one element that was out of place or two. However, if we had fixed two, that would mean that those two were interchanged, in which

case they would form a 2-cycle and they could not be part of any larger cycle. Therefore, we can only fix one element in this first swap. After this swap, the cycle has length $k-1$ and, by the inductive hypothesis, we will need $(k-1)-1$ moves to rearrange all the elements. Thus, if we add the swap that we already performed, we obtain the expected result of $k-1$ swaps.

To get the final formula, we observe that if there is a big cycle that includes all elements, we would need $n-1$ swaps and every cycle that we introduce decreases the number of swaps required by one. Therefore:

$$\#\text{ Swaps} = n - \#\text{ Components}$$

To implement this formula we can construct the graph using undirected edges and count the components applying UFDS

```
55 int swapsPermutation(vi perm1, vi perm2) {
56     int n = perm1.size();
57     UFDS::initialize(n + 1);
58     int components = n;
59     for (int i = 0; i < n; ++i) {
60         if (UFDS::find(perm1[i]) != UFDS::find(perm2[i])) {
61             components--;
62             UFDS::Union(perm1[i], perm2[i]);
63         }
64     }
65     return n - components;
66 }
```

Running time: $\mathcal{O}(n)$

3.4.2 Converting one permutation into another using adjacent swaps

This is a very similar problem; however, now we can only swap elements that are next to each other in the array. Furthermore, in this analysis, we will reduce the problem to calculating the minimum number of adjacent swaps required to sort a permutation. This is equivalent since we can use a bijective application to "redefine" the order according to how the elements are present in the first permutation.

The key observation in this case is the fact that the solution is the number of inversions in the permutation. We can define an inversion as a pair of indices (i, j) such that $i < j$ but $A_i > A_j$. We can easily notice that (choosing optimally), each adjacent swap will fix exactly one inversion since it will fix the relative order between two elements.

Furthermore, this idea is surprisingly easy to implement since we can use a modified version of Merge Sort that counts the swaps that would have been required to place an element in a specific position while maintaining a log-linear complexity.

To do so, we can just count at each step how many position we had to move the elements of the left half of the array to sort them.

```cpp
ll mergeSort(int left, int right, vi &elems) {
    if (left == right) {
        return 0;
    }
    int mid = (left + right) / 2;
    ll res = 0;
    res += mergeSort(left, mid, elems);
    res += mergeSort(mid + 1, right, elems);

    int index = 0;
    vi aux(right - left + 1, 0);
    int i1 = left, i2 = mid + 1;
    while (i1 != mid + 1 || i2 != right + 1) {
        int *picked;
        if (i1 > mid) {
            picked = &i2;
        } else if (i2 > right) {
            picked = &i1;
        } else if (elems[i1] <= elems[i2]) {
            picked = &i1;
        } else {
            picked = &i2;
            res += abs((*picked) - (left + index));
        }
        aux[index++] = elems[*picked];
        (*picked)++;
    }
    for (int i = 0; i < right - left + 1; ++i) {
        elems[i + left] = aux[i];
    }
    return res;
}

ll countAdjacentSwaps(vi &elems) {
    return mergeSort(0, int(elems.size()) - 1, elems);
}
```

Running time: $\mathcal{O}(n \log(n))$