Competitive Programming Notes

Only for Personal Use

dorian

Contents

1 DP

1.a SOS

As outlined in https://codeforces.com/blog/entry/45223. You want `dp[i,mask]` to represent the sum of `A[submask]` such that `submask` only differs from `mask` in the first i bits. Below shows the code that does this implicitly.

Also see Section 5.g for more bit manips.

```
for (int i = 0; i < (1 << N); i++)
    dp[i] = A[i];
for (int i = 0; i < N; i++)
    for (int mask = 0; mask < (1 << N); mask++)
        if (mask & (1 << i))
            dp[mask] += dp[mask ^ (1 << i)];
```

1.b Knuth's and DNC

If some properties of a dp recurrence is satisfied, then we apply useful optimisations. Things to watch out for are:

- Monoticity
- Convexity
- Geometric shapes (for example if the reccurence is linear we can apply CHT)

See Section 9.h for strats to prove such properties.

1.c DP on Trees

We can DP on trees by storing the state and node that we are on and progressing downwards. We can also construct a tree by DP by joining roots.

2 Segtrees

2.a Simple point update range query

As outlined in https://codeforces.com/blog/entry/18051. Just do this when you're lazy. Performance is comparatively better than a full functional segtree and is on par with Fenwick trees.

```
int sg[2 * N + 5];
// build
for (int i = 1; i <= N; i++)
    sg[N + i] = A[i];
for (int i = N - 1; i; i--)
    sg[i] = f(sg[2 * i], sg[2 * i + 1]);
// set at position i with value t
for (sg[i += N] = t; i > 1; i /= 2)
    sg[i / 2] = f(sg[i], sg[i ^ 1]);
// query [l, r]
int val = 0; // default value
for (l += N, r += N; l <= r; l /= 2, r /= 2)
{
    if (l & 1)
        val = f(val, sg[l++]);
    if (not(r & 1))
        val = f(val, sg[r--]);
}
```

2.b Annoying range update set + add

When we push down from a node, we want to:

1. Update self values according to lazy set
2. If lazy set exists, update children's lazy set and set their lazy add to 0
3. Update self values according to lazy add
4. Update children's lazy add with self lazy add
5. Reset lazy variables

Updates are relatively trivial.

2.c Beats

Segtree beats just refers to time complexity analysis on complex segtrees. For example, to set $x_i = \min(x_i, h)$ in a given range, we can simply recurse the segtree and only update our node if the update range covers the entire node range **and** $h > \text{mx}_2$ where mx_2 is the second max. This actually works out to not have a bad time complexity!

3 Maths

3.a Extended GCD

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

3.b Modular Inverses

To find a large prime, just run `openssl prime -generate -bits <nbits>`.

Note that finding the inverse of $a \bmod m$ is possible with Extended GCD:

$$\begin{aligned} ax + my &= 1 \\ \Rightarrow ax &= 1 \bmod m \end{aligned}$$

If m is prime, then $a \cdot a^{m-2} = 1$ by fermat's little theorem. Otherwise, we can do:

```
int inv(int i)
{
    return i <= 1 ? i : m - (ll)(m / i) * inv(m % i) % m;
}
```

as

$$\begin{aligned} m &= \left\lfloor \frac{m}{i} \right\rfloor i + m\%i \\ \Rightarrow m\%i &= -\left\lfloor \frac{m}{i} \right\rfloor i && \pmod m \\ \Rightarrow (m\%i)i^{-1} &= -\left\lfloor \frac{m}{i} \right\rfloor && \pmod m \\ \Rightarrow i^{-1} &= -\left\lfloor \frac{m}{i} \right\rfloor (m\%i)^{-1} && \pmod m \end{aligned}$$

3.c Generating Functions

They are simply functions whose coefficients represent something. For example, the coefficient of x^k in $(1 + x^a)(1 + x^b)(1 + x^c)$ describes the number of ways a, b, c can add up to k. They can also be infinite, for example:

$$\begin{aligned} f(x) &= 1 + x + 2x^2 + 3x^3 + 5x^4 + 8x^5 + \ldots \\ \Rightarrow f(x) - f(x)x - f(x)x^2 &= 1 \\ \Rightarrow f(x) &= \frac{1}{1 - x - x^2} \end{aligned}$$

And from there, you can split $f(x)$ into partial fractions and derive a closed formula for the i^{th} coefficient of $f(x)$.

3.d FFT

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

3.e Extra polynomial operations

3.f Lagrange Interpolation

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

4 Graphs

4.a Top sort

This can be done trivially in one dfs.

4.b Centroid decomp

Actually very useful. Every path from a to b can be thought of as a to a centroid then to b. It is also useful for considering paths that cross the centroid for example, among other techniques.

4.c HLD

Don't do this... Not recommended unless you're very familiar with your implementation of HLD.

4.d DSU

DSU can be really usefull outside of finding connected components.

We can do offline LCA with DSU by doing a single dfs. If we are at node i and exit a child j, we merge node i and node j such that j's parent points to i. Then, as we are about to exit node i, the answer to all queries of the form $\text{lca}(s, i)$ where s is seen before is simply $\text{get\_parent}(s)$.

We can also do a bunch of other useful stuff with DSU.. just be creative!

4.e Euler Tours

- For every edge delete it and recurse on it
- When the function exits add V to the answer

4.f Implicit tree

If there are only k important nodes on a tree, we can make an implicit tree by consider the adjacent (by preorder) lcas of those k nodes.

4.g König's Theorem

It claims that the max matching is equal to the min vertex cover in case of a bipartite graph. Additionally, the maximum independent set is just the opposite of the min vertex cover.

$$I(G) = |G| - M(G)$$

Additionally, after a maximum matching is found, one can simply run a dfs from all the unmatched nodes on the LHS. The min vertex cover will simply be all the unvisited nodes on the LHS and the visited nodes on the RHS. The max independent set will be the complement of the min vertex cover. By careful when running the dfs: from right to left **only** walk the **matched** edges.

4.h Euler's Formula

For a planar graph, $V - E + F = 2 = 1 + \text{num components}$. The formula goes both ways.

If G has no bridges, then every face is bounded by a cycle (each edge included exactly once).

Every cycle in a bipartite graph is of even length.

$K_{3,3}$ and K_5 are the building blocks of all non-planar graphs.

4.i Max flow

BFS a path and augment it. Add the minimum possible flow along that path. Make sure to keep track of both the flow and the capacity. There is also Dinic's, but it might be hard to implement.

4.j 2-Sat

Obviously we want to state explode x_i into x_i and $\neg x_i$ first. Then we run SCC. For a solution to exist, it is **necessary and sufficient** for x and $\neg x$ to be in different components. We can then simply do a top sort of the DAG. If x comes before $\neg x$, then $x = \text{false}$, otherwise $x = \text{true}$.

4.k Maximum matching

Just trivially do it. Go through every possible potential partner, if not matched, then match, otherwise, try to free up.

4.l SCCs (Tarjan's)

Within a single dfs:

- Keep a counter and a stack
- Whenever we enter a node, add it to the stack, increment counter and assign $L_i = \text{counter}$
- Then, $L_i = \min L_j$ for all children j **that is on the stack**
- If L_i is the same before and after, then we can generate a connected component by popping off nodes from the stack until we pop off node i.

5 Ad Hoc

5.a Josephus

Simple recursion (1-indexed):

$$J_{n,k} = \big((J_{n-1,k} + k - 1) \bmod n\big) + 1$$

This arises when we consider what happens when we remove the first number: we delete that number and arrive at the same problem just everyone's labels are shifted. As outlined in https://cp-algorithms.com/others/josephus_problem.html#modeling-a-ok-log-n-solution, we can model a $O(k \log n)$ solution too.

```
// 0-indexed
int josephus(int n, int k)
{
    if (n == 1)
        return 0;
    if (k == 1)
        return n - 1;
    if (k > n)
        return (josephus(n - 1, k) + k) % n;
    int cnt = n / k;
    int res = josephus(n - cnt, k);
    res -= n % k;
    if (res < 0)
        res += n;
    else
        res += res / (k - 1);
    return res;
}
```

5.b SBBSTs (treaps)

Can be difficult to implement and prune to bugs. As always, remembering code is dangerous and you should always aim to understand the inner workings of a DS.

clean(x) maintains all values in node x with children's values (don't forget size!)

split$(x, p) \to (l, r)$ splits node x into 2 nodes (l, r) with some property p; for example if this is an implicit treap, p might be the size of l

merge$(l, r) \to x$ merges 2 nodes into 1 by comparing priorities

5.c Inclusion exclusion principle

Pretty self explanatory. When used in conjunction with bitmasks, `__builtin_parity(x)` may be desired.

5.d Precompiled headers

To find where the header are located, run:

```
g++ -H dummy.cc 2>&1 | head
```

Then, simply make the precompiled header:

```
sudo g++ <your flags> <header file>
```

Note that sudo privileges may or may not be required. In case that you do not have enough permissions, simply make a copy of the header file and put it in a location where you do have permissions.

5.e GCC Policy DS

Mentioned in https://codeforces.com/blog/entry/11080 and https://codeforces.com/blog/entry/60737.

```
#include <bits/extc++.h>
// a faster version of unordered_map with less features
template <typename T, typename V> using hmp = __gnu_pbds::gp_hash_table<T, V>;
using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag,
             tree_order_statistics_node_update>
    OrderedSet;
ordered_set.insert(x); // inserts x
*ordered_set.find_by_order(i); // gets x[i] (in sorted order)
ordered_set.order_of_key(val); // gets num elems < val
```

5.f Sorted stack and jump pointers

5.g Bits

Some useful and common bit manips.

```
// least significant bit
x & -x
// loop through every submask (except 0)
for (int submask = mask; submask; submask = (submask - 1) & mask)
// mask with only ith bit set; 2^i
1 << i
// mask with least k bits set
(1 << k) - 1
// number of most significant zeros in the bit representation
__builtin_clz(x)
// number of least significant zeros in the bit representation
__builtin_ctz(x)
// popcount
__builtin_popcount(x)
// parity of popcount
__builtin_parity(x)
// print in binary form
cout << bitset<32>(x)
```

5.h Gray code

As outlined in https://cp-algorithms.com/algebra/gray-code.html.

This is a binary system where sucessive values differ by exactly one bit.

```
int g(int n)
{
    return n ^ (n >> 1);
}
int g_inv(int n)
{
    int n = 0;
    for (; g; g >>= 1)
        n ^= g;
    return n;
}
```

5.i XOR basis

This procedure is very similar in how one would compute a basis in an actual k-dimensional vector space.

```
int basis[k];
int sz = 0; // size of basis

void insert(int mask)
{
    for (int i = 0; i < k; i++)
    {
        if (not ((mask >> i) & 1)) continue;
        if (not basis[i])
        {
            basis[i] = mask;
            sz++;
            return;
        }
        mask ^= basis[i];
    }
}
```

5.j Sqrt

Sqrt is very useful.

5.k Meet in the Middle

This is probably not *that* useful..

5.l Burnside/Pólya

5.m Heuristics

Hill Climb Pretty much the best there is in the context of informatics

Ant colony simulate a bunch of ants moving on a graph. If a path is good, then they deposit pheromones on all edges on the path so that it will be taken more in the future.

Simulated annealing bunch of random state transitions with regard to a decreasing temperature which controls the probability of each transition.

6 Strings

6.a Tries

Self explanatory.

6.b Suffix arrays

We can sort the suffixes of a string/list of items in their lexicographical order. To do that, we first sort the indices by the 1 sized substrings they represent. Then using the sorted ranks, we can sort the indices by the 2 sized substrings they represent by sorting a pair (a, b) where a represents the rank of the first half of the substring and b represents the second half. Note that intermediate ranks are also valuable in many computations. For instance, one can trivially apply a sparse table query over the ranks.

6.c Hashing

We can hash a list of items with a polynomial:

$$\text{hash}(x_1, x_2, x_3, ..., x_N) = \left(x_1 + x_2 k + x_3 k^2 + ... + x_N k^{N-1}\right) \bmod M$$

Normally we just grab a large prime for M (and a smaller prime for k) as we can calculate modular inverses quickly with a prime mod.

6.d Z

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

6.e Palindrome tree

7 Comp Geom

Everything in comp geom is pretty much grade 5 maths. Just consider everything in vectors and spam dot products and you're good to go.

7.a Upper Convex Hull

Sort points by x, then loop from left to right maintaining decreasing gradients.

7.b Convex Hull (Graham's scan)

Get an extreme point p_e in one dimension, tiebreak by other dimension. Then, sort all other points by the angle they make with p_e. Loop through them in order, and maintain monotone adjacent cross products (so that the points only ever turn in one direction).

7.c Shoelace

This finds the area of a polygon given the vertices.

$$\left|\frac{\sum_{i,j}^{\text{cyclic}}(x_j - x_i)\left(y_i + y_j\right)}{2}\right|$$

7.d Closest pair

Just DNC and you're done. Make sure to solve left and right first to get answer d so that you can use d for $O(n)$ merging.

7.e Minimum enclosing circle (Welzl's)

```
fn f(points, contain) -> Circle {
    if points is empty or |contain| >= 3 {
        return circle formed by contain
    }
    let p = random point from points
    let circ = f(points - p, contain)
    if p in circ {
        return circ
    }
    return f(points - p, contain + p)
}
```

Note that extending to higher dimensions is difficult as the constant factors increase exponentially.

8 Game Theory

Most of game theory is just minimax.

8.a Nim

$p = x_1 \oplus x_2 \oplus x_3 ... \oplus x_N$

A game state is losing iff $p = 0$, as on the next turn, whichever move you made, your opponent can make $p = 0$ again.

9 Tricks

9.a Alien's Trick

Outlined in https://mamnoonsiam.github.io/posts/attack-on-aliens.html. In essence, you have $f(x)$ which is convex/concave. Then, you can find binary search on the slope of $f(x)$ as it is monotonic.

You need to be able to find the optimal $f(x) - kp$ for a fixed k, then by shifting k, you can adjust the value p which optimises $f(x)$ to your liking.

9.b Slope Trick

Maintain a convex function with a constant c_0 and the x_i at which the function changes slope.

9.c CDQ

DNC on time: complete all updates on the LHS before answering queries on the RHS.

9.d CHT

Monotonic CHT is best: just sort by gradient then compare intersection points. If offline, every CHT can be converted into monotonic CHT by CDQ (while not damaging the time complexity as both adds an extra $O(\log n)$.

Otherwise, doing a li chao tree might be easier.

```
ll eval(pair<ll, ll> line, ll p) { return line.first + line.second * p; }

bool skip_middle(pair<ll, ll> a, pair<ll, ll> b, pair<ll, ll> c)
{
    return (b.first - a.first) * (c.second - b.second) >
           (c.first - b.first) * (b.second - a.second);
}
```

Note that when querying the above CHT, a binary search is required on `eval`.

9.e Butterfly

Bit iteration. Can be useful to consider each bit of a number individually, or to split a group by their i^{th} bit.

9.f Persistence

If a data structure strictly supports operations in $O(a)$, then persistence can simply be achieved by recording the operations then reverting them.

9.g Offline deletion

DNC on the time segments for which the updates take place can achieve some sort of offline deletion.

9.h Proving Monoticity or Convexity

Quadrangle Inequality $f(a,c) + f(b,d) \leq f(a,d) + f(b,c) \quad \forall a \leq b \leq c \leq d$
Monoticity $f(x) \leq f(x+1)$
Convexity $f(x) - f(x-1) \leq f(x+1) - f(x)$

10 Performance Engineering

10.a Flags

The following pragmas at the start of your code **can** speed it up. But use it sparingly, since they might slow your code down as well.

```
#pragma GCC optimize("Ofast")
#pragma GCC target("avx2")
```

10.b Cache Locality

Memory access up to magnitudes faster. In general, just make sure to have the correct loop order for bottom-up dp.

10.c DS

Some DS are a lot faster than others. In general:

- Fenwick trees are faster than segment trees
- Priority queues are faster than sets, which in turn are faster than segments trees
- Treaps are pretty slow, but not as slow as you think (alternatives are much slower!)
- Policy DS tend to be even slower (?)

11 Setup

11.a Vim bindings

```
syntax on
filetype indent on

set mouse=a
set number
set numberwidth=6

let mapleader=' '
au filetype cpp nnoremap <buffer> <leader>c :up<cr>:!g++ -std=gnu++20 -DDEBUG -O2 -
Wall -Wextra -Wshadow "%" -o "%:r"<cr>
au filetype cpp nnoremap <buffer> <leader><leader> :vsplit<cr><c-w>w:term "./%:r"<cr>
```

12 Problem Solving

Making observations

- Play around with small data.. if you're stuck you haven't done this enough!
- Take data to the extreme and see what happens. For instance, in an inequality, see where it is equal or almost equal

Things to try

- Exploit offline when possible: allows for persistent DS and many algorithms such as DNC.
- Chuck on a segtree on top of an arbitrary dimension with wild functions. For example, on a graph, build a segtree on the preorder of nodes or ID of edges.
- Popping dimensions. For example, viewing segments on a line as a point in 2D
- Taking hints from subtasks and constraints. For example, $N \leq 20$ suggests $O\left(2^N\right)$ i.e. some form of bitmask dp.
- Look for bijections.
- Look for optimal orderings.
- Consider: shortest paths, MSTs or other optimal paths as they are often used in proofs/solutions.
- If the idea does not work out nicely, still go through with it as the solution might be complicated.
- Swap perspective. For example, instead of considering every pair of vertices u and v, instead consider every node p first and find vertices u and v which has p as their LCA.
  $\sum_i \sum_j a_{i,j} = \sum_j \sum_i a_{j,i}$
- Binary search and jump pointers.
- Stuff in reverse. Processing the latest events in an offline problem first rather than the earliest ones.
- Consider euler path on trees, as well as preorder and postorders.
- Flavours of the pigeonhole principle. If $\sum a_i = K$, then there will be at most $\sqrt{K}$ unique a_i. If k points are on a line of L length, then theres one segment of under/over $\frac{L}{k}$.
- Calculate time complexity correctly. Don't overestimate the time complexity for a correct solution.
- Look for monoticity, convexity or even convexity on a slope function.
- Read the problem statement again and carefully.
- Graphs are everywhere.
- Read every question first before beginning.
- Take regular breaks

When debugging

- Make a python script along with a bash script to automate testing (go for an easier subtask so you know that it's correct)
- Maybe the order of operations is wrong (a + (b == c)) or multiplied then added in a dp etc.
- Make sure to print a lot!