Algorithms Notes

Bliss Of Comprehension

April 15, 2022

Contents

preface

This book is largely concerned with algorithms helpful for solving OJ problems.

Donot forget to use both **"LATERAL THINKING"** and "Vertical thinking" while solving problems.
Proof: All the editorials you have seen consists of a combination of max 2 to 3 previously known concepts or tricks. This implies that there is a path from start(problem statement) to goal(answer) which consists of atmost 3 edges, so simply think laterally untill you find a vertex or edge in that shortest path. (whose size is guaranteed to be ≤ 3).

In some problems vertical thinking may produce lesser times than Lateral thinking, But Lateral is consistent in producing good times in all problems.

Consider two approaches of thinking.

- Think of an idea and spend time on that. If answer not found repeat.(normal thinking/ dfs type thinking)
- List down all the ideas initially(each from each step of 1st type of thinking) (ex 10 unrelated ideas). And sort the ideas by probability of reaching answer. Now spend time on each idea in sorted order.(Popularly known as "brainstorming").

Clearly 2nd way is better than 1st. And it is guarenteed to reach answer in less amount of time.

Let the algorithm space be a set of points(exact algorithms) and there is an edge between two points $\iff$ we can reach algo1 to algo2 with just a minamilistic tweak.

- All points(exact algorithms) related to a concept are really close together in the space

- An ideaset is set of all points(in which this particular idea is used).
- The exact solution(point) to a problem may really far away from our starting (point or ideaset).
- So starting with multiple points (or ideasets) may work faster in our search.
- we can also skip through multiple edges(instead of single edge each time) to reach our solution faster.

Your brain is just a bunch of neurons. Combined states of billions of neurons represent a snapshot of your thoughts(your state). Similary thought about each idea can be roughly thought as a set of action potentials of all the neurons.
When you see a question some of the neurons fire based upon the words and pictures in the question and your previous memory.(read about long term potentiation and hebbs law). (Initially neurons related to visual signal of words and pictures fire, but due to LLP(memory) some more neurons will fire) this gives rise to initial idea.
Initial idea differs by individuals previous experiences(what might be a creative and non obvious idea to you, might be someone's initial idea). And the mind goes on in a line of thought firing different neurons, but there won't be a drastic state change in the brain.
This is where creativity comes in to picture. Implement any thinker toy from thinker toys book, the state of the brain changes drastically. For example, associating a random word to our problem (neurons related to the "random word" will fire, so in conjuction with the problem neurons initially fired they combine to form a different state of the brain i.e thought of new idea).
Explicitly speaking, since you read thinker toys from the book, whenever you encounter a problem, neurons related to thinker toys book trigger, which inturn triggers a specific thinker toy essentials neurons(who might send impulses to eyes to look for a word and head to turn to the screen or book where words are present), and associates with problem neurons.

An algorithmic way of approaching codeforces problems.

- Read and understand the problem statement.
- Read input format.
- Verify sample tests with brute force to check your understanding of problem statement.

- ReRead Bold words in the problem statement.
- Check the time limit.(may be you can use sqrt decomp etc)
- Come up with algorithm.
- Verify the sample tests with your algorithm.
- check accepted/wrong answer ratio. If it's too low then the obvious algorithms are wrong. Take time and recheck your algorithm.
- Think of an implementation. Rigorously try to **IMPROVE** your implementation, instead of coding on the first thought.

It is really important to improve your speed. In a normal scenario, You have done A-D(0-1900 rated) questions.

- It can put you in rank (140- 960). For a rating of 1950 it is (+90 or -45).
- And Simply decreasing each question time by 5min increases rating a lot, and leaves +20min for the 5th question.

Methods to check why your solution is giving wrong answer.

- Go to input section, find the lowest and highest value a parameter takes and test your solution with that, for all parameters.
- May be it's not your cpp code that's wrong, but your algorithm itself. (You fucked up and wasted a lot of time by now)

Tricks to approach a problem

- If the input only contains single number, Then the answer is probable in **OEIS**.

Chapter 1

Specific Heuristics

Consult these before and during contest. These only contain heuristics that are not obvious. For example to consider Minimum spanning tree during problem is hard, but once you saw that considering prim's/kruskal's is straightforward.
Always look at General/adhoc section first.

1.1 General/Adhoc

Topics to force

- Binary Search.
- Convex Hull trick.
- Dilworth's theorem.
- Longest increasing subsequence.
- Meet in the middle. (i.e $O(2^{\frac{n}{2}})$)
- Mo's algorithm
- Nearest small element on $O(n)$
- Stack.
- Ternary search.
- Two pointers.
- Represent as graph.

- DP
- Inclusion - exclusion.
- Invariants/monovariants.
- Bruteforce with unexpected time complexity.
- Maybe no single observation. It's a case bashing problem.

Strategies

- If asked to count certain objects (pairs, subsequences, subarrays etc). Write down given required condition they should satisfy and also try to infer more conditions from them.
- Solve for the objects **not** satisfying the condition and subtract them from total.
- Inclusion-exclusion.
- Invariants/monovariants.
- For optimization problems one strategy is to construct the object first(like guess) and prove that object is optimal.
- Brute force with unexpected time complexity.
- operations can be converted in to equivalent forms. Any set of given operations can be interchanged with some set of our operations and vice-versa.
- In some problems only few variables are independent rest all are dependent on the former due to the condition given. We can use this to decrease our complexity.
- Bitsets for complexity reduction.

1.2 Bitwise

Topics to force

- Trie.
- Lexicographical ordering.

- SOS DP.

Strategies

- Consider each bit separately.
- Divide and conquer on sorted array(since bits are ordered).

1.3 Number Theory

Topics to force

- Binet's formula
- Euler's totient function
- Mobius function
- Fibonacci numbers.
- GCD.
- MEX.
- Prime decomposition.

1.4 Graph Theory

Topics to force

- Bipartite Graph.
- Centroid Decomposition.
- Graph coloring.
- Halls theorem.
- Heavy-Light decomposition.
- LCA.
- Matching
- Flow

- Spanning tree.
- Min cut.
- Strongly connected component.
- Eulerian path.

1.5 DP

Strategy

- If we doing a bool dp, and it is monotonic over one dimension then we can remove that dimension and put it in the dp-value itself reducing complexity.(For example minimum index to make dp = 1).

1.6 Coding

- Unordered map.

1.7 Strings

Topics to force

- Hashing
- Z-algorithm.
- Suffix array.

Chapter 2

Stack

Stack is the suprising Data Structure that comes in handy in unexpected of times.

we can have a modified stack which gives minimum among the elements in $O(1)$ with rest other stack operations time remaining same.blog.

we can have a standard queue with same operations too blog.

2.1 Balanced Parantheses

A bracket sequence is balanced if either of the below are true. As all of them are equivalent.

- Every element of prefix sum array is ≥ 0 and last element is 0.
- When open bracket is encountered push it to the stack and pop the top open bracket otherwise. If there is no open bracket to pop or the stack is not empty at the end, the sequence is not balanced.
- Base Case : Empty sequence is balanced.
  Constructor Case : If s,t (sequences) are balanced then s(t) is balanced.
  Now check if our sequence can be produced this way using recursion.

Overlapping two balanced sequences produces a balanced sequence. Prove it using the 1st definition.

C_i (catalan number) counts the number of sequences containing n pairs of parentheses which are correctly matched. Using this we can calculate in O(n)

unlike other DP solutions.

Let s[0..n-1] be a balanced sequence

- s[0..i] is balanced iff s[i+1..n-1] is balanced.

When sequence is balanced, we can pairup open and closed brackets. This is done during the 2nd definition. When we are popping out the open bracket from stack(when closed br is encountered) we pair those two.

The pairing can also be done like this. For open bracket at i, find the minimum index j S.t sum[i..j]=0. (i,j) are paired

Let ith and jth index (i<j) are paired.

- If $i < l < j$ then the partner of l (named as r), $i < r < j$.
  Proof: We know $sum[i..l-1] > 0 => sum[l..j] < 0$ since $sum[l..l] = 1$ there exists $r < j$ S.t $sum[l..r] = 0$. We used the property that $sum[i..j]$ is continous over j.
- s[i..j] is balanced.
- Number of subarrays ending at 'j' and are balanced = $dp[i-1] + 1$.

For any bracket sequence you can create a forest of trees where each node represents a balanced subarray and children are balanced subarrays contained in the parent's range.

Chapter 3

XOR

Competitive problem setters love XOR (never understood the physical significance of it).

Different techniques to solve problems on XOR

- Bit Trie.
- Solving for each individual bit and combining them at the end.
- Using Lexicographic property. If the most significant bit in binary form a number is 1 and the other number has 0 then first number is greater (no matter what the other bits are).
- In problems concerning XOR over subarrays converting the array into prefix array can be useful.
- In tougher problems, divide and conquer is also used (i.e solve the problem for the numbers with 0 at considered bit, and solve for numbers with 1 at considered bit and combine them for the answer).
  Problem link
- Gaussian elimination is used to find subset with maximum XOR.
- XOR of two numbers is just bitwise addition (mod 2).

Some tricks in solving XOR problems

- $4k \oplus (4k+1) \oplus (4k+2) \oplus (4k+3) = 0$ (used in finding XOR of numbers from 1..1e18)
- If $a \oplus b = 0$ and $b \neq c \implies a \oplus c \neq 0$. This trick is mostly used in number theory to prove theorems on nim.

- If the array is sorted, all the elements with the same first x bits will be contigous $\forall$x.
- $a + b = a \oplus b + 2 * (a \ \& \ b)$
- $a + b = a \mid b + a \ \& \ b$

3.1 Game Theory

Different techniques for solving these problems.

- Any game resembling NIM with small removals will have a small period of MEX. This is because Mex is calculated using previous size's mexes so at some point the set of previous size's mexes reappear(pigeon hole principle). For this trick to work the moves(no.of stones removed) should be really small otherwise the periodicity is not guaranteed(think about it).
- One other trick to find periodicity (for ex in case of big move sizes) is to write a brute force algorithm and print out the mexes and find the patter from output.

There are different kinds of differentiation used in games.

- Simultaneous move game and extensive form game.
- Finite games and infinite games (both are extensive form games)
- perfect and imperfect information games.
- partial and impartial games.(In Extensive form games, if state of game is given but moves differ for two players then its partial game).
- zero sum and non zero sum

We deal with games which are two player, extensive, finite, perfect info, impartial and zero sum(not always) in nature(in OJ's).
Game Tree : A tree where nodes are all the states and edges are possible moves with root as starting state.
Strategy : For a player, A strategy is a complete algorithm for playing the game, telling a player what to do for every possible situation(state) throughout the game. (i.e for every state possible, a single move is associated with it).

payoff (reward) : In the game tree, we define payoff's for each players at every terminal node.(for example it may simply state who won i.e $payoff_a = inf, payoff_b = -inf$ if player a wins at that terminal node).
In zerosum games $payoff_b = -payoff_a$ at every terminal node.
Payoff function: It takes strategies of players as input and outputs payoff's for both the players. It just outputs the payoffs of the terminal node at which game ends when actually played.(Since strategies are known we know how game will play out).
When optimal play is stated in the problem. They imply what's the payoff when backward induction strategy profile is used?
When player tries to maximize some quantity is given in the problem. Then that quantity is payoff.
Proposition: ***The set of subgame perfect equilibria of a finite horizon extensive game with perfect information is equal to the set of strategy profiles isolated by the procedure of backward induction.*** from "osborne introduction to game theory".

There are two type of operations involved in game theory of impartial extensive(second player plays after first player finishes the move) games.

- MEX
- XOR

Sprague-Grundy Theorem :

If we know the MEX function of different games (G1, G2..) then the MEX function of combined games is XOR of all the functions.

- In a nim game with custom moves (examp. we can remove 1,2,5 stones at once) there will always be a periodicity of mexes with number of stones.

Chapter 4

Counting

- Given a binary string, find number of 3-tuple (indices $i, j, k \ni i < j < k$) and exactly one of $s[i], s[j], s[k]$ is 1.
- There are r boxes in a row and n identical balls. Find number of different ways we can distribute balls among boxes.
  ${}^{n+r-1}C_{r-1}$
- In problems, to convert A to B using minimum operations. Looking at the final operation may help.
- When trying to solve non-linear recurrences using matrix expo. $f_i = f_{i-1} + 2i^2 + 5$. The left column vectors should be $\begin{pmatrix} f_i \\ i^2 \\ i \\ 1 \end{pmatrix}$ and $\begin{pmatrix} f_{i-1} \\ (i-1)^2 \\ i-1 \\ 1 \end{pmatrix}$ otherwise the matrices wouldn't be same and we cannot combine the equations.
- let $x_1 + x_2 + x_3 + \ldots + x_r = n$ and $x_i \leq \sqrt{n}$ then $x_1^2 + x_2^2 + x_3^2 + \ldots + x_r^2 \leq n\sqrt{n}$.
- **Lucas Theorem** :

$$\binom{m}{n} \equiv \prod_{i=0}^{k} \binom{m_i}{n_i} \pmod{p}$$

  where $m = m_k p^k + m_{k-1} p^{k-1} + \cdots + m_1 p + m_0$, and
  $n = n_k p^k + n_{k-1} p^{k-1} + \cdots + n_1 p + n_0$ are base p representations of m and n respectively. Note that $n_k < i_k \implies \binom{n_k}{i_k} = 0 \implies \binom{n}{i} \equiv 0 \pmod{p}$. This is equivalent to saying that there is no x^i term in the expansion.

- The only best way to calculate ncr other than lucas is to have prefix facts and prefix inverse facts, then answer ncr in O(1).
- When n is really big and r is small, We can answer the query in O(r), without any preprocessing.
- Even though bitset dp has two loops it's time complexity is not $O(N * 2^N)$.

```
for(int i=0;i<n;i++){
    for(int j=0;j<(1<<i);j++){
        dp[i+1][j]+= dp[i][j];
        dp[i+1][j|(1<<i)] += dp[i][j];
    }
}
```

  Because j being only iterated untill $(1 << i)$ complexity is $O(2^{n+1})$. Similarly for three loops it will be $O(N * 2^{n+1})$.

4.1 Probability

4.1.1 Randomized algorithms

Before we use randomized algorithm, we already know answer for the given input(we can find that out using bruteforce if we have a really fast computer for bruteforcing).
So to find out what the answer really is we take help of probabilities.
Since our classical computer cannot generate ideal randomness. Our algorithm cannot give a random answer(we can determine even before we run it).But it gives answer with bruteforce almost all the time (we can find test cases that doesn't though).

4.1.2 Expected Value

When we want to use dp for expected value, try to define dp[i] as "what is the expected value if we start from i and stop the experiment whenever we reach the final state n" because consider this definition "what is the expected value if we start from 0 and stop the experiment whenever we reach i",in some cases there is a mistake in definition, there may be some samples where i is never reached. Consider the problem "Find expected no of moves before 'bitwise or' becomes $2^k - 1$".

4.2 Inclusion-Exclusion principle

Consider any sets $A_1, A_2, \ldots A_n$ (discrete mathematical objects).

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{i=1}^{n}|A_i| + \sum_{i<j}|A_i \cap A_j| \ldots + (-1)^n|A_1 \cap \ldots \cap A_n|$$

. As every element will contribute to both L.H.S and R.H.S by +1, Q.E.D. Now in any problem, define the sets(A_i's) appropriately and apply above formula to get an useful result.

Important problems using inclusion-exclusion are:

- finding number of subsets with certain gcd in an array.
- Using Mobius function is just same as using inclusion-exclusion internally.
- Number of solutions to $x_1 + x_2 + x_3 \ldots + x_r = n, 0 \leq x_i \leq m \ \forall i$.

4.3 Dilworth's Theorem

- LIS can be calculated in $O(nlogn)$, using link. It can be done using optimized DP anyway though. $(dp[i] = 1 + max(dp[j]) \forall j \ni a[j] < a[i])$. Where $dp[i]$ is LIS ending at i.
- There is another dp approach. $dp[i][j] = min(dp[i-1][j], a[i] \iff a[i] > dp[i-1][j-1])$. where $dp[i][j]$ is, using $[0 \ldots i]$ elements, the minimum ending element among all the j length "increasing sequences".
  - By induction on i we can prove that $dp[i]$ is sorted.
  - $dp[i]$ differs from $dp[i-1]$ at atmost one position. That is the index j_i where $j_i = max\{j | dp[i-1][j-1] < a[i]\}$.
  - We can update $dp[i-1]$ in-place to produce $dp[i]$.
- To solve this problem in general. Create a bipartite graph with each element x of poset on both sides (u_x and v_x), add an edge $u_x \to v_y \iff x \leq y$ for$(x \neq y)$. Find the maximum matching m, then the minimum partition (width of poset) is $n - m$;
- Most of the problems can be solved by converting to LIS or LDS.

Chapter 5

Graphs

In OJ's most of the problems concerning MST's can be solved with Kruskal's algorithm.(others are Prim's and Boruvka's).

There are certain kind of problems where number of edges are huge. They are not given in the input but told to us using certain properties of vertices. (Example. There are n nodes and each has a value a_i. There is an edge between i and j iff $a[i] \oplus a[j] == k$).

- Sometimes these problems can be solved by adding some more nodes to the graph(may be nodes spanning all p-bit numbers or each node representing a bit $O(lgn)$ nodes in total) and connecting the real vertices with new vertices. The property for the graph(ex.number of components) asked in the question may remain same for the new graph and now edges may be less.

Let $d(u, v)$ = shortest path b/w u, v. Then $d(u, v, i)$ s.p passing through i is $d(u, i) + d(v, i)$. similary s.p passing through edges (i, j) is $min(d(u, i) + d(j, v), d(u, j) + d(i, v)) + w(i, j)$.

A graph is bipartite $\iff$ there is no odd cycle in the graph.

Given a graph with both directed and undirected edges. If there are no cycles using only directed edges, we can always direct all undirected edges without forming a cycle.proof.
Normally problems like above involve eulerian cycle and direct edges according to that.

5.1 DFS

There are only tree edges and back edges in dfs-tree of undirected graph. Unlike in dag's can have cross edges and forward edges extra.

5.1.1 Un-Directed Graph

In an undirected graph an edge($v \to to$ where (v,to) is tree edge in dfs-tree) is a bridge $\iff$ there is no backedges from one of to's descendants to any v's ancestor node. This can be checked by having a value $high_u$ at every node, corresponding to highest node which is adjacent to u in original graph. And by checking if the minimum among descendant nodes is higher than our bridge. Or we can use intime of dfs traversal to check instead of depth as criteria.

5.1.2 Directed Graph

DiGraph has a cycle $\iff$ it has a back-edge. And also every cycle in the Digraph should contain atleast one backedge.
Complete theory of directed graphs Link

5.2 bfs

In a bfs tree of undirected graph, for any cross edge $u \to v$. $abs(depth[u] - depth[v]) \leq 1$

Girth is the shortest cycle in undirected unweighted graph. It can be calculated in $O(VE)$.reference.

For 0-1 bfs you should visit next node if it's current distance is greater than $curdist + 0/1$(so each node maybe visited multiple times). But in normal bfs you can simply not visit new node if it's already visited even for once.

5.3 Trees

These are the different ways we approach problems based on trees.

- Tree dp.
- Normal dfs (greedy version of dp approach).

- Process nodes from lowest level to top.(leaves to root).

```
void dfs(int u, int par){
    for(auto v:g[u]){
        if(v!=par){
            dfs(v,u);    //recursively solve for all the
                subtrees
        }
    }
    //When the function reaches this step all the children
        will be proceseed.Now use the children's
        information to solve for u.
    /*
        actual code.
    */
    return;
}
```

- start time, end time on nodes.
- binary lifting.
- Using Lowest common ancestor (this can be done using binary lifting).
- sqrt decomposition on trees.
- rerooting.

While performing tree dp we can keep a map (of values in the subtree) at each node. This is a very generic trick and seems to have high potential.
we will traverse the tree from the down most level (level wise).

- Have maps at each of the lowest level nodes.(each map has only one entry)
- Now when we go up to produce map of node 'u' in this level.
  Take the largest map among children of u and add remaining children maps in to this big map. Now add the value corresponding to 'u' to the map. Now this map is the final map for 'u'.
- Since when each node moves from one map m1 to other m2, $sizeof(m2) \geq sizeof(m1)$. Final size of m2 after addition will be $\geq 2 * sizeof(m1)$. So any node will change maps only $lg(n)$ times. So the time complexity is $O(nlg(n))$.
- The memory is $O(n)$ throught out the procedure. Since we are inserting smaller maps in to the big map(which is already present).

- This can be implemented using dfs.

```
int operations = 0;
for(int v : vertices)
   for(int child1 : children[v])
         for(int child2 : children[v])
            if(child1 != child2)
               operations += size_of_subtree[child1] *
                  size_of_subtree[child2];
cout << operations;
```

The number of operation in the above code is $O(n^2)$. This fact can be used while calculating dp on trees including subtree sizes.It can be proved by induction or using the fact that any pair of nodes will contribute to operations at only their LCA.

In DSU, while performing dsu one of the three types of techniques is used.

- Path Compression

```
Find(x)
   if x.parent != x
      x.parent := Find(x.parent)
return x.parent
```

- Path Halving

```
Find(x)
   while x.parent != x
   x.parent := x.parent.parent
   x := x.parent
return x
```

- Path Splitting.

In the problems where we need a pointer to an ancestor in a tree (or similarly in the array) and update operations makes you add more links then Path compression is useful. Example

Given a tree with each node having a value a_i. There are two kind of operations
Update : update a_i with c where $c < a_i$.
Query : Given u. Find the deepest node with value >= k which is ancestor of u.

In similar problems (having pointer in tree or array) we can use binary lifting on pointer's sometimes.

- When tree is flattened(euler path) then for any node the component consisting of it's parent will be two segments in the flattened array. This will be easy to use when rerooting technique is not possible.

5.3.1 Binary Lifting

Consider the path to root $u- > v_1- > v_2...- > root$ .Given the query create a binary function that satisfies (from $u- > root(1, 1, 1...1, 0..0, 0, 0)$). Then we can use the following code to find the vertex with last 1. This can be used to calculate complex problems like find the last node v(from below) such that $gcd[u, .., v] > 1$. In this case to calculate function we need one more 2D array $val[u][i]$ which gives $gcd[u, ...(v_{1<<i})]$ and also an integer that has $gcd[u..cur]$. Remember to calculate $func(p[cur][i])$ we use $gcd(pre, val[cur][i], a[p[cur][i]])$.

```
let p[u][0] = par[u];
p[u][i] = p[p[u][i-1]][i-1];

if(func(u)){
   return u;
}
int cur = u;
for(int i=20;i>=0;i--){
   if(func(p[cur][i])){
      cur = p[cur][i];
   }
}
Now cur will be the vertex with last 1 from bottom.
```

5.3.2 Flows

- There are situations where we need to think about Min-Cut to derive a bijection between cuts and the problem. So sometimes mincut concept is also useful to think about not only Max-Flow. Link.

Chapter 6

C++ tricks

- In a normal set finding the *ith* element is not possible in o(lgn) and it is also not possible to know number of elements less than a given value.
- As the above operations are theoretically possible for a balanced BST, we use policy data structure to support those two additional operations along with normal set operations. Blog link
- But for using the above structure as a multiset, there is no easy way to do it. We will use a shortcut by inserting pair (value, time of insertion) to distinguish equal elements. And do operations around that.
- `__gcd(x,0)` doesn't work in g++, even though its mathematically defined.
- While using two pointers having the second pointer ('j') pointing to the minimum index such that [i,j) subarray satisfies the condition eases the code most of the times. (instead of [i,j]).
- Use mt19937 instead of rand().(due to low max value, and periodicity).

```
mt19937 rng(chrono::steady_clock::now().
    time_since_epoch().count());
int RandomOutput = rng(); \\ generate a random number.
shuffle(a.begin(),a.end(),rng); \\ shuffle array a.
```

- when generating random trees to test your solution. Donot generate bigger size trees because edge cases will be missed(Probability that the tree will be a line is almost zero for bigger size).Similarly for the arrays too.
- when you write code to return while in the input loop. Beware this problem doesnot contain different test cases.

List of good to remember time complexities.

- map $1e7$ random insertions 10sec.
- unorderedMap $1e7$ random insertions 5sec.
- unorderedMap with custom hash $1e7$ random insertions 5sec.
- set/multiset $1e7$ random insertions 10sec.
- $1e7$ Random vector lowerBounds on array of size $2e5$ 327ms.
- SegTree $1e7$ random operations(which includes both updates and queries) 1.8sec.
- Lazy SegTree $1e7$ random operations(which includes both updates and queries) 12sec.

6.1 STL

- List has a huge constant factor, this is due to allocation of memory on heap when an insert operation is used, the memory is not contiguous to travel easily etc. And mostly due to cache non-friendly behaviour (as the memory is not contiguous).

Chapter 7

Arrays

- When we are asked to count number of pairs that satisfy a specific condition in an array. You should first try to rearrange the equation to get all the i^{th} index terms to left side and j^{th} index terms to right side. This will help us reduce $O(n^2)$ to $O(n)$.
- Let p be an array of distinct elements and $next_i$ be the smallest $j > i$ s.t $p[j] > p[i]$. Then when we draw edges(curves) from $i- > next_i$ on top of the array then no two edges intersect. (i.e the condition $i < j < next_i < next_j$ is never satisfied for any pair of indices i,j).
- **Langford pairing** : It is possible $\iff n \equiv 0 or 3 (mod 4)$.
- If we are asked to find $\sum_1^{n-k} |f(i, i+k)|$. We can partition the array in to chains, each chain has indices which are pairwise congruent modulo k and in the same order as in array. Now we can solve seperately for each chain and combine them to produce overall answer. The above is generalized version of seperating even and odd indices and solving them seperately(example in the transforming array problems(ex. Xoring adjacent elements)).
- There are 3 types(tentative) of sqrt decomp I know.
  - MO's algorithm(offline algo) : When there is no query of update form then we can take all the queries and sort them in a certain way to make naive computation faster.
  - (online algo)Divide the array in to blocks each of size ($\sqrt{n}$) and process both kind of queries in O($\sqrt{n}$) each.
    When there are updates with addition or deletion of elements.Simply add the element in to the corresponding block, which increases it's

size by 1. After each $\sqrt{q}$ updates, preprocess the array again(As the new blocks doesn't maintain the $\sqrt{n}$ size anymore).

– (online algo) Consider $\sqrt{q}$ operations at a time. Since there can be atmost $O(\sqrt{q})$ update operations. The array will be changed at atmost $\sqrt{q}$ indices hence if we find a way to answer the query operations (in this $\sqrt{q}$ block) with some preprocessing each in $O(\sqrt{q})$.
(Example. Find answer for this query without considering any updates in $O(1\, or\, lgn)$ and update this answer according to changed indices in this range(which are only $\sqrt{q}$ in number). The time complexity of each block will be $O(q)$ hence total complexity will be $O(q\sqrt{q})$.(PreProcessing is done for each block seperately just before considering the block). Example Problem Link.

- Some problems involving subarrays, we can approach this problem through prefix sums,(donot forget that dp also works, even in max subarray sum).
- Finding min element in subarray can be done using sparse table($O(nlgn)$ preprocess and $O(1)$ query). For 2d array ($O(nmlgnlgm)$ preprocess and $O(1)$ query).Where Seg tree takes $O(nm)$ preprocess and $O(lgnlgm)$ for query. (queries can be up to n^2m^2 in number).
- Queries of rectangles with fixed sizes(axb) can be done in $O(mn)$ pre and $O(1)$ query. Find $rowmin[i][j]$ = minimum among $mat[i][j] \dots mat[i][j+b-1] \forall i, j$ in $O(mn)$ using sliding window queue technique. Now compute $ans[i][j]$ = minimum among $rowmin[i][j] \dots rowmin[i+a-1][j]$ in $O(mn)$ using the same technique. This techinque can be extended to multi-dimension, instead of matrices.
- Even with a normal swap(not adjacent ones) parity of inversions changes every time.
- The order in which swaps are executed matters and results in a completely different sequence.

We can have pointers to the nearest equal, less or greater elements (either to left or right). This can help in certain problems

- Find number of unique elements in range. More generally count number of elements in range, but if freq of an element is greater than 'k' take it only k.
We can do it by having pointer to kth equal element to left. Then in range[l,r] find number of elements with $ptr[i] < l$, which is the answer.

- Number of distinct subsequences in an array. For 'i' have the nearest left equal element 'j'. Then $dp[i] = 2 * dp[i-1] - dp[j]$ (where dp[i]=number of distinct subsequences in $[0 \dots i]$)
- In problem given an operation similar to compressing subarray then we can imagine the final array and each element in final array represent contiguous segment in original array.

Consider a divide and conquer algorithm with time complexity recurrence. $T(n) = T(e_1) + T(e_2) + e_1(e1 \leq e2 \wedge e1 + e2 = n)$.
The closed form is $T(n) = O(nlgn)$.
This can be proved using induction or realising the fact the every element is included in the e_1 term atmost lgn times.

Given two increasing sequences a,b and m pairs of indices which are bad. Find a two elements one from each array with minimum sum excluding the bad pairs. This can be solved in $O(m)$. For each bad pair add (x+1,y) and (x,y+1) to an array of potentially useful pairs and also add (0,0). Now we only need to consider pairs from that array for minimum.

7.1 Segment Tree

You are given an array of elements(may be null initially) and queries over subarrays.
There are two properties for a segment tree.

- SegProp : required property from a segment.
- update type.

For the Segment tree to be valid it needs to satisfy

- req 1 : We need to be able to find segprop from segprop's of any contigous partition subsegments of segment.

For the lazy segment tree to be valid it needs to satisfy this below additional properties.

- req 2 : If there are two or more updates on the same segment, we need to be able to combine them in to a single update.
- req 3 : If there is an update for a segment(whole segment). We need to be able to update segprop for the segment in $O(1)$.

Lazy segment tree update template

```
void update(int si ,int ss, int se, int l, int r, int val){
   If there is lazy update pending:
      update seg prop for this node with lazy update; \\req 3
      make lazy update for this node null;
      combine the lazy updates of children with this update;
            \\req 2

   If current node segment is out of update range return;

   If the current node segment is completely in update range:
      update segprop of this node;  \\req 3
      combine the lazy updates of children with this update;
         \\req 2
      return;
   recursively call update for both children;
   update this node segprop with childrens seg prop. \\req 1
}
```

Store the segprop's and lazy updates in 4*N sized array.

We can even store entire subsegment's in each node. The memory is only $O(nlgn)$.

If there are updates of type "change an element to x". Then to find that element now, we only need to consider last update(of above type).

Given an 0-1 array we can find the index which has kth one from left, with updates. This is done by traversing the segment tree down.

- When updates of delete an element is given and queries(l,r) on updated array is given. we can simulate a process, instead of deleting the elements we just make thier value NULL at that position. And just answer the query.But as the query(l,r) indices are given for the deletion based array, we need to convert $(l, r) \rightarrow (l', r')$ indices on NULL based array (this is done by above technique). Now from (l',r') we can answer by querying on NULL based array. (This requires two segment trees).

7.2 Permutation

- If we have a directed graph of n nodes, and connect ith to jth iff $p[i] = j$; then we get a bunch of disjoint cycles.

- You can find number of inversions in an array using policy data structure (balance BST). Infact we can do many variations of inversion problem. (Example. Find number of inversions when equal elements are present, Which is not possible using merge sort.)

- **Factorial number system** is used to number permutations. Finding the k^{th} lexicographic permutation, will be easy if k is represented in this number system. And given a permutation finding its lexicographic order in this number system is straight forward. (After that we may need to convert this in to decimal). Search in wikipedia for process.

- Let $dp[i][j]$ is number of permutations of length i with j inversions.

$$dp[i][j] = \sum_{k=0}^{i-1} dp[i-1][j-k]$$

  . (Iterate over the last element of permutation).

- The minimum number of swaps(not adjacent) required to make a permutation $a -> b$ is (n-no.of components(which are cycles btw)) in the graph where edges are $a_i -> b_i$

- Let the inversion number be the number of inversions in a permutation(also minimum number of swaps), then with a swap the parity of inversion number always changes.

Chapter 8

Numbers

- $\lfloor a/(bc) \rfloor = \lfloor \lfloor a/b \rfloor / c \rfloor$
- Number of perfect squares in $[l, r], l \leq r$ is $\lfloor \sqrt{r} \rfloor - \lceil \sqrt{l} \rceil + 1$
- $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + ...$ is divergent. Sum of first n elements $\ln(n+1) \leq \sum_{i=1}^{n} \frac{1}{i} \leq \ln(n) + 1$ Proof link
- the factors of any number are quite smaller compared to the number. There doesn't exist a pair of distinct factors whose sum is $\geq$ the actual number.
- 'a' is multiple of n_1 and 'a' is multiple of n_2 iff 'a' is multiple of $LCM(n_1, n_2)$.
- similarly 'd' is divisor of n_1 and 'd' is divisor of n_2 iff 'd' is divisor of $GCD(n_1, n_2)$.
- The set of all integer combinations of a and b is precisely the set of all integer multiples of the GCD of a and b. proof
- The above statement is generalized version of Bezout's identity.
- $dp[i][j]$, number of subsets of length i whose gcd is j, then $dp[i][j] = \binom{cnt_j}{i} - \sum_{k=2}^{\infty} dp[i][k * j]$, where cnt_j is number of elements divisible by j. This is a generalized version of simply finding number of subsets of gcd exactly == j(which is solved similarly but with 1-dim dp).
- $GCD(a_l, a_{l+1}, ..a_r) = GCD(a_l, a_{l+1} - a_l, a_{l+2} - a_{l+1}, .., a_r - a_{r-1})$

- Range queries for GCD with a single update is straight forward. But for range update, we keep two segment trees, one to find a single element with the range update, one over difference between adjacent elements to get GCD over range of difference array. As for difference array only two elements change with range update(update is addition). We answer in $O(lgn)$ using above formula.
- if d is gcd of a list of numbers. Then d divides thier sum.
- In gcd problems, iterating over the prime divisors(and enumerating all the elements which are multiples of this) almost always works, Instead of iteration over all divisors.The first method has less complexity as every element has atmost lgn prime divisors but $n^{\frac{1}{3}}$ factors hence getting enumerated only lgn times.
- $n!(modm)$ is zero when $n \geq m$.
- $$\sum_{i=0}^{n} i.i! = (n+1)! - 1$$
  .
- There is a number system called **Factorial number system**. From the above fact any number can be uniquely represented in this system.
- For $m, n \geq 1$ $\gcd(f_m, f_n) = f_{\gcd(m,n)}$. Where f is fibonacci. proof
- The sum of number of factors from $[1 \dots n]$ is $O(nlgn)$. This can be proved by observing, the answer is equal to sum of number of multiples $\leq n$ from $[1 \dots n]$.
- $ab \leq c \iff a \leq \lfloor c/b \rfloor$
  $ab < c \iff a \leq \lfloor (c-1)/b \rfloor$
  $ab \geq c \iff a \geq \lfloor (c+b-1)/b \rfloor$
  $ab > c \iff a \geq \lfloor (c+b)/b \rfloor$
- $1 + 3 + 5.. + (2n-1) = n^2$
- Let $p = 2^i$ then if we keep on removing powers of 2 from p in non increasing order then p will become 0 before getting negative.
- Estimation of number of prime numbers is $n/lg(n)$ not $n/4$.
- $\lfloor n/i \rfloor = p$ iff $i \in (\lfloor n/(p+1) \rfloor, \lfloor n/p \rfloor]$

8.1 Modulo

- If you want to calculate a^b modulo p(prime) where b is very large ($b = 100000^{100000}$) we can use Fermat's **little** theorem. As $a^{p-1} \equiv 1(modp)$, $a^b \equiv a^{bmod(p-1)}(modp)$.
- For prime p, and any number a relatively prime to p, a,2a,3a..p*a are all distinct modulo p.
- But $a^1, a^2, \ldots a^{p-1}$ are not distinct modulo p. (a=6, p=5)
- $ab \bmod ac = a(b \bmod c)$.
- Consider a number n, if you keep on modulo it by different numbers x_i, it only changes $logn$ times. Since every time it changes it becomes $\leq \frac{n}{2}$.

8.2 CRT

Find the solution set of $x \equiv a_1(modn_1)$ and $x \equiv a_2(modn_2)$.

Let $gcd(n_1, n_2) = d$ and find x' and y' s.t $n_1x' + n_2y' =$ d. (this is bezout's identity. Find it using Extended euclidean algorithm).
Note: d should divide $a_2 - a_1$, otherwise there is no solution for the given two equations.
One of our solution is $x_o = a_1 + x' \frac{a_2 - a_1}{d} n_1$. And our solution set is
$x_o + u * lcm(n_1, n_2), u \in Z$. This obtained solution set is equal to solution set of $x \equiv x_o(mod\ lcm(n1, n2))$.

For solving multiple equations combine first two equations in to one using above technique. Then again combine the newly obtained equation with the 3^{rd} one and so on.

8.3 Euler's phi function

- Euler's phi function is multiplicative. proof.
- using above we can calculate $\varphi(n) = n \cdot \prod_{p|n} \left(1 - \frac{1}{p}\right)$. where p's are primes.
- Let any number k coprime to n then (k mod n) $\in \varphi(n)$.

- A more generalized form $\varphi(mn) = \varphi(m)\varphi(n) \cdot \frac{d}{\varphi(d)}$. This is when m and n aren't necessarily co-prime.
- if d\n, the number of integers $k <= n$ and $GCD(k, n) = d$ are $\varphi(n/d)$.
- a more general euler's theorem (which is inturn more general fermat's little theorem) $x^n \equiv x^{\phi(m)+[n \bmod \phi(m)]} \mod m$. This is when x and m aren't necessarily coprime.

Chapter 9

DP

Write recurrence relation for a problem. When we draw a directed graph from the recurrence(i.e draw edge from state s_i to all the subproblems it requires to find dp[s_i](i.e the right side of recurrence states)). Then in order to solve the problem using dp the graph should be acyclic. (If not we cannot come to the answer with this dp alone, even though the recurrences are correct. You can even check if recurrences are correct between states after finding out answers with some other algorithm).
On the above graph there are three ways to solve.

- Topdown dp.
- bottom up dp.
- forward dp.

Some Points to note

- Counting problems either involves combinatorics or dp and nothing else.
- There are counting DP problem's where we need to find no of ways to pair or group elements. We can have dp with one dimension representing no of groups which are not closed yet. This can be used both in arrays, trees etc.
- we proove most optimization dp's using this kind of proving techniques. proof.

- while solving optimization dp(topdown,bottomup,forward), Keep another array 'prev' which stores the state from which this state's answer is calculated.(among all the searched subproblem states). Which makes it easy to recover the complete solution instead of only value. This can be done just by having a while loop starting at final state and iterating through prev states.

- consider $dp[state]$, which gives whether state is reachable(i.e boolean). We can instead compute, in how many ways state is reachable. In most of the problems this computation does not add any additional complexity. But this information can be used to solve problems.

- For bitset optimization of dp ex. subset sum problem $bitset < smax > dp[nmax]$. If $nmax = 10^5, smax = 10^5$ it only took 577ms in codeforces. For memory use $bitset < smax > dp[2]$.

- In subset sum problem if number of distinct elements(k) are less we can make complexity $O(k*lg(n)*S)$.Construct a new array and for each pair in $(a_1, na_1), ..(a_k, na_k)$ add $2^0 * a_i, 2^1 * a_i ...., 2^r * a_i, (na_i - (2^{r+1} - 1)) * a_i$ where 'r' is the maximum possible.

- If the dp is too hard to implement, change the whole dp by adding dimensions. For example dp in circular array instead of having dp[i] we can have $dp[i][0/1][0/1]$ 2nd dim indicates whether the index '1' is included or not and 3rd dim indicates whether index 'i' is included or not. Now in $dp[N][0/1][0/1]$ based on the second and third dimension we can update the final answer.

- Consider the problem, given an array($a_i \leq 9$) make all elements equal by doing "select all occurrences of a number and change it to another number" and also given the cost to change k elements at once. Here the intuitive way is to do dp[msk] = minimum cost to make all bits in msk equal. But transitions are **not** to take another digit and make it equal to those in msk. It's actually to take another mask and making them both equal. This gives us $O(3^n)$.

9.1 arrayDP

- Consider an array 'A' of length 2^n. Then we need to calculate

$$F[mask] = \sum_{i \subseteq mask} A[i]$$

. This can be naively be calculated in $O(3^n)$. Let

$$S(mask, i) = \{x | x \subseteq mask \wedge mask \oplus x < 2^{i+1}\}$$

. Then

$$S(mask, i) = \begin{cases} S(mask, i-1) & i^{th}\ bit\ OFF\ in\ mask \\ S(mask, i-1) \bigcup S(mask \oplus 2^i, i-1) & i^{th}\ bit\ ON\ in\ mask \end{cases}$$

(We can similarly do it for other operation than sigma).

- If asked a problem on subsets(optimization or counting), just sort the array or have a frequency array and dp on that.

9.2 treeDP

- In some problems we root the tree at random node(r). Initially we calculate a certain property $SubProp_u$ for a subtree rooted at node u $\forall u \in V(vertexSet)$ in $O(n)$ using dfs. Now we can reroot the tree to one of the children of r and recompute all the $Subprop_u$ in $O(1)$ as only two $SubProp_u$'s changes(one is r and the other is the new root(selected child of r)). Template is below
precalculate SubProp's for random node initially. Then call the below dfs from that node.

```
void dfs(int u, int par){
   \\ Now all the SubProp array is correctly
      calculated when u is the root. Use it.
   for(auto v:g[u]){
      if(v!=par){
         \\reroot the tree to v and adjust SubProp
            array correctly.
         dfs(v,u);
         \\reroot the tree to u and adjust SubProp
            array correctly.
      }
   }
   return;
}
```

adjusting "maximum depth from node" can be done by maintaining max and 2^{nd} max depths(among all maximum depth paths, each going through seperate children) at each node.

Chapter 10

Ad-hoc

Any kind of observations, tricks etc are just used to decrease the time complexity and or memory complexity of the problem compared to brute force.(Similar to any standard algo or ds you studied).

Ways to approach adhoc problem.

- solve for smaller input by making one or multiple input dimensions 0 or 1 (may gives an idea or shows a pattern).
- GUESS
- If problem asks you to tell if any solution is not feasible. Identify the case when its not, probably rest of the cases will be possible.
- For the optimization problems or counting problems(count objects of some type), put multiple **EXAMPLES** on the paper and find the pattern.
- Similarly for the problems that require some quantity (nth element in array, sum of nth row in a matrix), when given relation between elements or rows to calculate them. But cannot do it due to time constraint. There may be a pattern present, write down **EXAMPLES** or write a bruteforce code for smaller n and output it to visualize the pattern.
- Optimization problems: Given a set of objects(graphs, integers, etc) implicitly. Find min or max among the attributes of the objects.
  Example maximization problem, One solution is to prove that any object has attribute $<= p$. And construct the object with attribute p.

- Consider an empty grid, you can paint any 2x2 square with any color, is it possible to obtain a given grid. If a square is already painted it will be **repainted**. This "will be repainted" sentence should trigger us to look at operations from back to front. The solutions is, if we have the optimal sequence of operations lets say of size n, for any 'i' make union of all the squares included in operations [i+1..n] '.'s then 2x2 square of ith operation must contain only single color with maybe some '.'s. Now we can just bfs from last operation(which we know).

Given two objects(both are of same class) A and B and an operation which transforms any object **X** to some object **Y**. Find minimum number of operations to convert A to B. Some ways to approach this problem.

- Find an invariant(property of the object) that remains constant after a single operation.
- Find a monovariant(propert of the object) that changes by fixed amount after a single operation, no matter what the starting object is.

When printed all the constructions for inputs and they are huge in number, try to print only constructions satisfying some additional constraint. For example, construct a permutation with LIS = LDS then simply print only constructions with LIS = 1 or LIS = n/2 etc.

Searching google is an art too. For example

- For a pure mathematical problems like combinations and stuff, use their terminology. Use 'Sequences' instead of 'Arrays'.

10.1 specific problems

- when asked to construct a new object where elements which are adjacent in initial object are not adjacent here, try to rotate(left or right shift) some part of the object(which may produce optimal answer).
  Ex. Produce a derangement.

Chapter 11

Strings

Different ways to approach string problems.

- Every array algorithm can be applied to strings and not vice-versa. In strings the each element can take only small range of values(ex. lower case latin letters 26) unlike arrays.
- Hashing, the most potential algorithm in strings.