Algorithms Notes

Bliss Of Comprehension

July 25, 2019

Contents

preface

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

Chapter 1

Stack

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

1.1 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.

- s[i..k] $i < k < j$ is not balanced.
- 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 $sum[l..l] = 0$ there exists $r < j$ S.t $sum[l..r] = 0$. We used the property that $sum[i..j]$ is continous over j.

Chapter 2

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 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 $b \neq c \implies c \oplus b \neq 0$. 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.
- $a + b = a \oplus b + 2 * (a \ \& \ b)$

2.1 Game Theory

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 $payoff_a = payoff_b$ 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.
Backward induction strategy profile is a Nash equilibrium strategy profile(proven theorem).

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 3

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.

Chapter 4

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 and connection 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.

4.1 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.

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.

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.

Chapter 5

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.

Chapter 6

Arrays

- 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.
  - (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.
- Let p be permutation 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).
- 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.)

Consider a divide and conquer algorithm with time complexity recurrence. $T(n) = T(e_1) + T(e_2) + e_1(e1 <= e2 \wedge e1 + e2 = n)$.
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.

6.1 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 $O(nlgn)$.

6.2 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.

Chapter 7

Numbers

- $\lfloor a/(bc) \rfloor = \lfloor \lfloor a/b \rfloor / c \rfloor$
- $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + ...$ 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 $LCM(n_1, n_2)$.
- 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.

7.1 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)

7.2 CRT

Find the solution set of $x \equiv a_1(mod n_1)$ and $x \equiv a_2(mod n_2)$.

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.

7.3 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 $\varphi(mn) = \varphi(m)\varphi(n) \cdot \frac{d}{\varphi(d)}$.
- if d\n, the number of integers $k <= n$ and $GCD(k, n) = d$ are $\varphi(n/d)$.

Chapter 8

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 the graph should be acyclic. (Then 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.

8.1 arrayDP

8.2 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;
}
```