adamant's blog

General ideas

By adamant, 5 years ago, translation,

// Finally translated!

Hi everyone!

Do you like ad hoc problems? I do hate them! That's why I decided to make a list of ideas and tricks which can be useful in mane cases. Enjoy and add more if I missed something. :)

1. Merging many sets in $O(n \log n)$ amortized. If you have some sets and you often need to merge some of theme, you can do it in nai way but in such manner that you always move elements from the smaller one to the larger. Thus every element will be moved only $O(\log$ times since its new set always will be at least twice as large as the old one. Some versions of DSU are based on this trick. Also you can us this trick when you merge sets of vertices in subtrees while having dfs.

2. Tricks in statements, part 1. As you may know, authors can try to hide some special properties of input to make problem less obvious. Once I saw constraints like $1 \leq a \leq b \leq 10^5, \dots, ab \leq 10^5$. Ha-ha, nice joke. It is actually $a \leq 400$.

3. $\gcd$ on subsegments. Assume you have set of numbers in which you add elements one by one and on each step calculate $\gcd$ of all numbers from set. Then we will have no more than $\log a_i$ different values of gcd. Thus you can keep compressed info about all $\gcd$ on subsegments of a_i:

code

```
int a[n];
...
map<int, int> sub_gcd[n];
/*
Key is gcd,
Value is the largest length such that gcd(a[i - len], ..., a[i]) equals to key.
*/
sub_gcd[0][a[0]] = 0;
for(int i = 1; i < n; i++)
{
    sub_gcd[i][a[i]] = 0;
    for(auto it: sub_gcd[i - 1])
    {
        int new_gcd = __gcd(it.first, a[i]);
        sub_gcd[i][new_gcd] = max(sub_gcd[i][new_gcd], it.second + 1);
    }
}
```

4. From static set to expandable via $O(\log n)$. Assume you have some static set and you can calculate some function f of the whole se such that $f(x_1, \dots, x_n) = g(f(x_1, \dots, x_{k-1}), f(x_k, \dots, x_n))$, where g is some function which can be calculated fast. For examp $f(\cdot)$ as the number of elements less than k and $g(a, b) = a + b$. Or $f(S)$ as the number of occurences of strings from S into T and g a sum again.

With additional $\log n$ factor you can also insert elements into your set. For this let's keep $\log n$ disjoint sets such that their union is the wh set. Let the size of k^{th} be either 0 or 2^k depending on binary presentation of the whole set size. Now when inserting element you should ac it to 0^{th} set and rebuild every set keeping said constraint. Thus k^{th} set will tale $F(2^k)$ operations each 2^k steps where $F(n)$ is the cost of building set over n elements from scratch which is usually something about n. I learned about this optimization from **Burunduk1**.

5. $\oplus$-subsets. Assume you have set of numbers and you have to calculate something considering xors of its subsets. Then you can assum numbers to be vectors in k-dimensional space over field Z_2 of residues modulo 2. This interpretation useful because ordinary methods of linear algebra work here. For example, here you can see how using gaussian elimination to keep basis in such space and answer queries c k^{th} largest subset xor: link. (PrinceOfPersia's problem from Hunger Games)

6. Cycles in graph as linear space. Assume every set of cycles in graph to be vector in E-dimensional space over Z_2 having one if corresponding edge is taken into set or zero otherwise. One can consider combination of such sets of cycles as sum of vectors in such space. Then you can see that basis of such space will be included in the set of cycles which you can get by adding to the tree of depth first search exactly one edge. You can consider combination of cycles as the one whole cycle which goes through 1-edges odd number of times and even number of times through 0-edges. Thus you can represent any cycle as combination of simple cycles and any path as combinatio as one simple path and set of simple cycles. It could be useful if we consider pathes in such a way that going through some edge twice annihilates its contribution into some final value. Example of the problem: 724G - Xor-matic Number of the Graph. Another example: find pa from vertex v to u with minimum xor-sum.

7. Mo's algorithm. Variant of *sqrt*-decomposition. Basic idea is that if you can do non-amortized insert of element in the set (i.e. having opportunity to revert it), then you can split array in *sqrt* blocks and consider queries such that their left ends lie in the same block. Then for each block you can add elements from its end to the end of the array. If you found some right end of query in that block you can add elements from the end of block to left end of query, answer the query since all elements are in the set and revert those changes then.

8. Dinic's algorithm in $O(VE\log U)$. This algorithm in $O(EV^2)$ is very fast on the majority of testcases. But you can makes its asymptotic better by very few new lines of code. For this you should add scaling idea to your algorithm, i.e. you can iterate powers of 2 fron to 0 and while it is possible to consider only edges having capacity at least 2^k. This optimization gives you $O(VE\log U)$ complexity.

9. From expandable set to dynamic via $O(\log n)$. Assume for some set we can make non-amortized insert and calculate some queries. Then with additional $O(\log n)$ factor we can handle erase queries. Let's for each element *x* find the moment when it's erased from set. Th for each element we will wind segment of time $[a; b]$ such that element is present in the set during this whole segment. Now we can come u with recursive procedure which handles $[l; r]$ time segment considering that all elements such that $[l; r] \subset [a, b]$ are already included into the set. Now, keeping this invariant we recursively go into $[l; m]$ and $[m, r]$ subsegments. Finally when we come into segment of length 1 can handle the query having static set. I learned this idea from **Burunduk1**, and there is a separate entry about it (on dynamic connectivity)

10. Linear combinations and matrices. Often, especially in dynamic programming we have to calculate the value wich is itself linear combination of values from previous steps. Something like $D_{m,i} = \sum\limits_{j=1}^{n} a_{i,j} D_{m-1,j}$. In such cases we can write $a_{i,j}$ into the $n \times n$ matr and use binary exponentiation. Thus we get $O(n^3 \log m)$ time instead of $O(n^2 m)$.

11. Matrix exponentiation optimization. Assume we have $n \times n$ matrix A and we have to compute $b = A^m x$ several times for differen *m*. Naive solution would consume $O(qn^3 \log n)$ time. But we can precalculate binary powers of A and use $O(\log n)$ multiplications of matrix and vector instead of matrix and matrix. Then the solution will be $O((n^3 + qn^2) \log n)$, which may be significant. I saw this idea i one of **AlexanderBolshakov**'s comments.

12. Euler tour magic. Consider following problem: you have a tree and there are lots of queries of kind add number on subtree of some vertex or calculate sum on the path between some vertices. *HLD?* Damn, no! Let's consider two euler tours: in first we write the vertex whe we enter it, in second we write it when we exit from it. We can see that difference between prefixes including subtree of *v* from first and second tours will exactly form vertices from *v* to the root. Thus problem is reduced to adding number on segment and calculating sum on prefixes. **Kostroma** told me about this idea. Woth mentioning that there are alternative approach which is to keep in each vertex linear function from its height and update such function in all *v*'s children, but it is hard to make this approach more general.

13. Tricks in statements, part 2. If k sets are given you should note that the amount of different set sizes is $O(\sqrt{s})$ where *s* is total size o those sets. There is even stronger statement: no more than $\sqrt{s}$ sets have size greater than $\sqrt{s}$. Obvious example is when we are given several strings with total length *s*. Less obvious example: in cycle presentation of permutation there are at most $\sqrt{n}$ distinct lengthes of cycles. This idea also can be used in some number theory problems. For example we want calculat $\sum\limits_{n=1}^{k} \left\lfloor \frac{k}{n} \right\rfloor$. Consider two groups: numbers less than $\sqrt{k}$ we can bruteforce and for others we can bruteforce the result of $\frac{k}{n}$ And calculate how many numbers will have such result of division.
Another interesting application is that in Aho-Corasick algorithm we can consider pathes to the root in suffix link tree using only terminal vertices and every such path will have at most $O(\sqrt{s})$ vertices.

14. Convex hull trick. Assume we have dp of kind $D_i = \max(D_j + i \cdot k_j)$, then we can maintain convex hull of linear functions which we have here and find the maximum with ternary search.

15. xor-, and-, or-convolutions. Consider ring of polynomials in which $x^a x^b = x^{a \oplus b}$ or $x^a x^b = x^{a \& b}$ or $x^a x^b = x^{a|b}$. Just like in usua case $x^a x^b = x^{a+b}$ we can multiply such polynomials of size $n = 2^k$ in $n \log n$. Let's interpret it as polynomial from k variables such tha each variable has power ≤ 1 and the set of variables with quotient a_k is determined by binary presentation of k. For example, instead of $P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$ we will consider the polynomial $P(x_1, x_2) = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_1 x_2$. Now note that if w consider values of this polynomial in the vertices of cube $[-1, 1]^k$ then due to $x_i^2 = 1$, we can see that product of such polynomials will us exactly *xor* rule in powers. *or*-convolution can be done in the same way considering vertices of $[0, 1]^k$ and having $x_i^2 = x_i$. *and*-convolution you can find yourself as an excercise.

xor-convolution

```
void transform(int *from, int *to)
{
    if(to - from == 1)
        return;
    int *mid = from + (to - from) / 2;
    transform(from, mid);
    transform(mid, to);
    for(int i = 0; i < mid - from; i++)
    {
        int a = *(from + i);
        int b = *(mid + i);
        *(from + i) = a + b;
        *(mid + i) = a - b;
    }
}
```

or-convolution

```
void transform(int *from, int *to)
{
    if(to - from == 1)
        return;
    int *mid = from + (to - from) / 2;
    transform(from, mid);
    transform(mid, to);
    for(int i = 0; i < mid - from; i++)
        *(mid + i) += *(from + i);
}

void inverse(int *from, int *to)
{
    if(to - from == 1)
        return;
    int *mid = from + (to - from) / 2;
    inverse(from, mid);
    inverse(mid, to);
    for(int i = 0; i < mid - from; i++)
        *(mid + i) -= *(from + i);
}
```

Finally I may note that or-convolution is exactly sum over all submasks and that inverse transform for xor-convolution is the same with initi one, except for we have to divide everything by n in the end. Thanks to **Endagorion** for explaining me such interpretation of Walsh-Hadamard transform.

16. FFT for two polynomials simultaneously. Let $A(x), B(x)$ be the polynomials with real quotients. Consider $P(x) = A(x) + iB(x)$. Note that $\overline{P(\overline{x})} = A(x) - i\overline{B(x)}$, thus $A(w_k) = \dfrac{P(w_k) + \overline{P(w_{n-k})}}{2}$, $B(w_k) = \dfrac{P(w_k) - \overline{P(w_{n-k})}}{2i}$.

Now backwards. Assume we know values of A, B and know they have real quotients. Calculate inverse FFT for $P = A + iB$. Quotient: for A will be real part and quotients for B will be imaginary part.

17. Modulo product of two polynomials with real-valued FFT. If mod is huge we can lack accuracy. To avoid this consider $A = A_1 + 2^{16}A_2$, $B = B_1 + 2^{16}B_2$ and calculate $AB = A_1B_1 + 2^{16}(A_1B_2 + A_2B_1) + 2^{32}A_2B_2$. Using the previous point can be done in total of two forward and two backward FFT.