Problem Tutorial: "Anti-Plagiarism"

The problem is to check if the first tree has a subgraph isomorphic to the second tree.

Let's try to solve the problem with any asymptotics, and then remove everything unnecessary from the solution.

Let's make the first tree rooted, and solve the problem using dynamic programming on its subtrees. Let $dp[u][x \to v]$, where u is the vertex of the first tree, and $x \to v$ is the oriented edge of the second tree - be $true$ or $false$ - is it possible to match the vertices u and v, while matching a subtree of u to a subtree of v, if the second tree had a root x. In this way we are matching the edges $p(u) \to u$ and $x \to v$.

To calculate such dynamic, we need the values of all $dp[u'][v \to v']$, where u' is the son of u, and v' is the son of v if the tree is rooted in x. In this case, the calculation itself is a check whether there is a matching in the bipartite graph that covers the left part.

Such dynamic works very slowly, and does not give an answer to the problem. Let's fix both of this problems at once.

Let's look at the calculation of all states of the dynamic of the form $dp[u][x \to v]$, for different x. To calculate them, we are looking for a matching for graphs with the same right part, and the left part, which differs in one vertex.

Instead, let's immediately run a matching on a graph with all vertices, that is, only once for a pair of vertices u, v. If we have found a matching that covers the left part, then the answer to the whole problem is "Yes", if we could not cover at least 2 vertices, then the value of all $dp[u][x \to v]$ - is $false$, if we were able to cover all vertices except one, then we will run one dfs from it, which goes to the right along any edges, and to the left only along those taken in the matching, so we will find all the vertices x, such that $dp[u][x \to v] = true$.

The solution ends here, it remains only to estimate its running time - we run one matching algorithm for each pair of vertices u, v. Let's calculate the maximum possible number of edges in all obtained graphs, i.e. $\sum_{u,v} deg(u) \cdot deg(v) = \sum_u deg(u) \cdot 2 \cdot m = 4 \cdot n \cdot m$.

If we honestly evaluate Kuhn's algorithm, then we get the asymptotics of the entire solution $O(n \cdot m^2)$, however, on specific graphs, Kuhn's algorithm actually runs a lot faster, so such a solution is already good enough. If you really want to write an asymptotically faster solution, write Dinitz algorithm ot get a solution in $O(n \cdot m \cdot \sqrt{m})$.

Problem Tutorial: "Bit Component"

Consider $n > 3$ (small cases are obvious).

Let $2^k \le n < 2^{k+1}$. We will prove that for $2^k \le n \le 2^k + 2^{k-1}$ there is no good permutation and for $2^k + 2^{k-1} + 1 \le n < 2^{k+1}$ such permutation exists.

It is quite clear that there isn't a good permutation for $2^k \le n \le 2^k + 2^{k-1} - 1$ because the leftmost ones can't be connected to any other ones.

If $n = 2^k + 2^{k-1}$ then there is only one number which might connect the leftmost ones to others - n itself. But then all the numbers with leftmost ones must be located on one side of n (otherwise it wouldn't connect the leftmost ones to the other ones). Then their right ones can't be connected to their leftmost ones (because in these numbers there aren't ones in the position $k-1$ and in n there aren't ones in the positions from 0 to $k-2$).

We will construct the good permutation for $2^k + 2^{k-1} + 1 \le n < 2^{k+1}$ by induction. Also, we will require that for $n = 2^{k+1} - 1$ the permutation will start with n and end with 1.

Base: $n = 7$, permutation $7, 5, 4, 6, 2, 3, 1$.

Now $k \ge 3$. The permutation for $2^k + 2^{k-1} + 1$:

- $2^k + 2^{k-1}$

- 2^k
- Permutation for $2^{k-1}-1$, but 2^k added to all numbers
- $2^k + 2^{k-1} + 1$
- Permutation for $2^k - 1$

The ones are connected because the permutation for $2^{k-1}-1$ ends with 1, so all the ones are connected to the rightmost one in $2^k + 2^{k-1} + 1$, all the leftmost ones are connected to the left ones in it. And then the permutation for $2^k - 1$ starts with the number $2^k - 1$, so it connects the ones in positions from 0 to $k-1$.

To construct a permutation for $n > 2^k + 2^{k-1} + 1$ we will take a permutation for $2^k + 2^{k-1} + 1$ and insert the number $2^k + 2^{k-1} + x$ $(1 \le x < 2^{k-2})$ to the position adjacent to the position of the number $2^k + x$. Since the one in the position $k-1$ is connected to the one in the position k and all the other ones are same as in the number $2^k + x$, the component remains connected.

To construct a permutation for $n = 2^{k+1} - 1$ we will take a permutation for $n = 2^{k+1} - 2$ and add number n in the beginning. So it will start with n and end with 1.

Construction of the described permutation can be done in $O(n)$ time.

Problem Tutorial: "Crossing the Border"

First, let's solve the problem in $O(3^n)$:

Let's sort the items in descending order by cost, now the cost of the knapsack depends on the leftmost item in it. Also for each mask we will calculate the total weight of the items in it.

Let $dp[mask]$ be the minimum cost of transporting all items in $mask$, as well as the number of ways to achieve it.

To find the value of $dp[mask]$, it is enough to iterate over the submasks containing the most significant bit (to calculate each method once) as items in the next knapsack.

To speed up the solution, let's iterate over the "intermediate" mask. Let's represent it as a concatenation of two masks L and R, of lengths $\frac{n}{2}$. The idea is to relax all the values of $dp[LR']$ using all the values of $dp[L'R]$, where L' is a submask of L containing the most significant bit of L and R' - supermask of R.

To do this, in advance for each mask L we find all its submasks L' that do not contain it's most significant bit, and sort them in descending order of their weight, that is, in ascending order of the weight of the difference between L and L'. Similarly, for all masks R let's iterate over all their supermasks R' and sort them in descending order of weight, that is, by descending weight of the difference between R' and R. This step takes at most $O(3^{\frac{n}{2}} \cdot n)$.

Now, having the intermediate LR mask, we will iterate over the LR' masks in the calculated order, in which case their values will need to be relaxed through the prefix of $L'R$ masks (again, in the calculated order). This can be done using two pointers.

Also, we should not forget about masks of the form $0R$, for which there will be no most significant bit in the left half, but their values can be separately calculated in $O(3^{\frac{n}{2}})$.

Let us estimate the asymptotics of the solution. We can assume that for each mask R we iterate through all the pairs L, L', and also, independently, for each mask L we iterate through all the pairs R, R'. Thus the asymptotics of the solution is $O(2^{\frac{n}{2}} \cdot 3^{\frac{n}{2}})$ or $O(\sqrt{6}^n)$

Problem Tutorial: "Dinosaur Bones Digging"

We have to find maximum $max_{l \le i \le r} a_i \cdot |\{l \le j \le r : a_j > a_i\}|$ among all the query segments. It means that if we add all the subsegments of segments to the queries the answer won't change.

Then let's find R_j - the largest right border of the segment starting in j among the segments in the extended set of segments. We will consider only segments $[j, R_j]$ from now on. The answer remains the same.

The number on the position i can appear in the segments which satisfy $j \le i \le R_j$. Since R_j form a non-decreasing array, number on the position i appears in segments $[j, R_j]$ for $l \le j \le i$ for some l (or it doesn't appear in any segment at all).

To find the answer for the problem we will find for each element a_i the maximum number of elements strictly larger than a_i in one segment.

If we scan through all the numbers in decreasing order, we can maintain the amount of numbers larger than x in all the segments in some data structure (segment tree). On each step for number a_i we will relax the answer with a_i multiplied by the maximum in the subsegment of that data structure and then add one to that subsegment.

This solution works in $O(q + n \cdot logn)$.

Problem Tutorial: "Egg Drop Challenge"

We will calculate $dp[i]$ - minimum time required for the i-th person to catch an egg. It is clear that in the optimal $dp[i]$ if the i-th person catches an egg that was thrown by j-th person, than either j-th person should be throwing the egg with maximum possible speed (v_j), or i-th person should be catching the egg with maximum possible speed (u_i), because otherwise j-th person could've thrown it faster and the time spent would be less.

We can distinguish this cases with inequality $u_j^2 + 2 \cdot h_j \le v_i^2 + 2 \cdot h_i$

- If the condition is satisfied, we should relax $dp[i]$ with $dp[j] - u_j + \sqrt{u_j^2 + 2 \cdot (h_j - h_i)}$
- Otherwise, we should relax $dp[i]$ with $dp[j] - \sqrt{v_i^2 + 2 \cdot (h_i - h_j)} + v_i$ (if the value under the root is non-negative)

To support the relaxations of the second type, we will make a structure similar to Li Chao Tree. We can see that we should find minimum value among the functions which can be represented as $f_j(x) = a_j - \sqrt{b_j + x}$ in the point $x = v_i^2 + 2 \cdot h_i$, where $a_j = dp[j]$, $b_j = -2 \cdot h_j$, the domain of $f_j(x)$ is $[2 \cdot h_j, \infty)$. Also, there is a special condition - we should relax $dp[i]$ throw $f_j(x)$ only if $u_j^2 + 2 \cdot h_j > v_i^2 + 2 \cdot h_i$, this inequality actually meaning that $x < u_j^2 + 2 \cdot h_j$. That's why we will define $f_j(x)$ on $[2 \cdot h_j, u_j^2 + 2 \cdot h_j)$. Now we just have to find minimum value among $f_j(x)$ to relax $dp[i]$ with it $+v_i$.

The structure will support two operations:

- Add a function $a - \sqrt{b + x}$ on $[l, r]$
- For x find minimum $f_j(x)$ among functions in the structure

All the x from the queries are known in advance: $v_i^2 + 2 \cdot h_i$ for all i. We will build a segment tree on these points. Just like in ordinary Li Chao Tree we will store a function in each segment tree node, and for each point x an optimal function will be somewhere among the nodes on the path from root to leaf x. Since two functions can intersect only in one point, this structure works just like ordinary Li Chao Tree with operation of adding a function on subsegment in $O(\log^2 n)$.

The relaxations of first type are more complicated. Similarly, we have to find minimum value among the functions which can be represented as $f_j(x) = a_j + \sqrt{b_j + x}$ in the point $x = -2 \cdot h_i$, where $a_j = dp[j] - u_j$, $b_j = u_j^2 + 2 \cdot h_j$ and the domain of $f_j(x)$ is $[-u_j^2 - 2 \cdot h_j, \infty)$, but here we also have to support an extra condition $u_j^2 + 2 \cdot h_j \le v_i^2 + 2 \cdot h_i$.

That's why we will sort the people by value $u_j^2 + 2 \cdot h_j$, so the query is finding minimum among the functions with their indices j on the prefix of array. For it we will maintain a Fenwick Tree storing Li

Chao Trees. We will add function in the position of j (which actually causes $O(\log n)$ additions to Li Chao Tree), and find minimum among up to $O(\log n)$ Li Chao Trees to find real minimum. So, the adding operations work in $O(\log^3 n)$, operations to find minimum work in $O(\log^2 n)$.

There is a way to speed up the adding operations. We can see that all the queries we make are done in the points $-2 \cdot h_i$, while calculating $dp[i]$ for i from n to 1, meaning that these x_i form an increasing sequence. Also, each next add operation is adding a function on subsegment $[-u_j^2 - 2 \cdot h_j, \infty)$, while all the actual queries will be only on $[-2 \cdot h_j, \infty)$. That's why the Li Chao Trees we are using have an extra parameter - T, so that at any moment of time all the added functions are defined on $[T, \infty)$ and all the queries are in $[T, \infty)$ and T is increasing over time. That's why in the operations inside the Li Chao Tree we will just consider that instead of the original points x_i all the points are $max(T, x_i)$. Now, instead of adding a function on a subsegment we will add a function on the whole structure. The add operation works in $O(\log n)$, with all the additions from Fenwick Tree it works in $O(\log^2 n)$.

So, the total time is $O(n \log^2 n)$.

Problem Tutorial: "Fortune Wheel"

In this problem there are K deterministic turns and one random. It is obvious that a strategy which does some deterministic turns and then the random one is not optimal. It means that the optimal strategy is either consisting of only deterministic steps or does some random turns and then the deterministic ones.

For each position x, d_x - the minimum number of deterministic turns required to reach 0 can be calculated with bfs in $O(n \cdot K)$ time.

If the optimal strategy doesn't have random turns at all, the problem is solved. Otherwise, there is some set S_k of such positions, that if we got into one of them after k-th random turn we will start making deterministic turns to reach 0. Let sum_k be the sum of the d_x among S_k, sz_k be the size of S_k. Then the expected value of number of moves is $1 + \frac{sum_1}{n} + \frac{n-sz_1}{n} \cdot (...)$, where the $(...)$ is the same sum for 2, 3 and so on. But we can see that the rest of the sum is defined the same as the whole sum. It means that the optimal S_1 is same as optimal S_2 and so on.

And then we just have to solve an equation $X = 1 + \frac{sum_1}{n} + \frac{n-sz_1}{n} \cdot X$. $X = \frac{n+sum_1}{sz_1}$. It means that the optimal S_1 consists of sz_1 positions with minimum d_x. We can just iterate over all the possible sz and relax the answer with $\frac{n+sum}{sz}$ where sum is the sum of sz minimum d_x.

The solution works in $O(n \cdot K)$

Problem Tutorial: "Growing Sequences"

Let $dp[n][k]$ be the number of arrays of length n satisfying the condition and starting with k. For $1 \le k \le c$, $dp[1][k] = 1$.

Then we can calculate $dp[n][k] = \sum_{i=2 \cdot k}^{c} dp[n-1][i]$

It can be proved by induction that $dp[n][k]$ is actually some polynomial $P_n(k)$ defined on numbers from 1 to $\lfloor \frac{c}{2^{n-1}} \rfloor$. (and for numbers greater than $\lfloor \frac{c}{2^{n-1}} \rfloor$ $dp[n][k] = 0$)

It is true for $dp[1][k]$, $P_1(k) = 1$.

For n,

$$dp[n][k] = \sum_{i=2 \cdot k}^{c} dp[n-1][i] = \sum_{i=2 \cdot k}^{\lfloor \frac{c}{2^{n-2}} \rfloor} dp[n-1][i]$$

This sum is not zero only for $2 \cdot k \le \lfloor \frac{c}{2^{n-2}} \rfloor$, $k \le \lfloor \frac{c}{2^{n-1}} \rfloor$. Let $c' = \lfloor \frac{c}{2^{n-2}} \rfloor$

Let $P_{n-1}(k) = \sum_{i=0}^{m} a_i k^i$.

Let $Q_n(k) = \sum_{i=1}^{k} i^n$. It is a polynomial of degree $n+1$.

$$dp[n][k] = \sum_{i=2\cdot k}^{c'} P_{n-1}(i) = \sum_{i=2\cdot k}^{c'} \sum_{j=1}^{m} a_j i^j = \sum_{j=1}^{m} a_j \sum_{i=2\cdot k}^{c'} i^j = \sum_{j=1}^{m} a_j (Q_j(c') - Q_j(2\cdot k - 1))$$

This sum is a polynomial $P_n(k)$ of degree $m+1$.

The answer is $\sum_{i=1}^{\lfloor \frac{c}{2^{n-1}} \rfloor} P_n(i)$ which can be found with the usage of $Q_m(k)$ too.

We can precalculate the polynomials Q_m for $0 \le m \le n$. For example, with Lagrange interpolating polynomial in $O(n^3)$ time.

Then calculation of $P_n(k)$ from $P_{n-1}(k)$ works in $O(m^3)$ where m is the degree of $P_{n-1}(k)$. Then the calculation of all the polynomials $P_n(k)$ works in $O(n^4)$.

Problem Tutorial: "Hierarchies of Judges"

The task was to calculate the number of labeled rooted trees, where all vertices are either reliable or unreliable, relative order of unreliable children matters and for every vertex the total number of reliable vertices constitute at least half of all vertices amongst itself and its children.

Let's say that reliable vertex has a reliable children and b unreliable children. Then $b \leqslant a+1$, so if r_n is the number of aforementioned trees with a reliable root and u_n is the number of trees with an unreliable root then

$$r_n = \sum_{a=0}^{\infty} \sum_{b=0}^{a+1} \sum_{p_1+\cdots+p_a+q_1+\cdots+q_b=n-1} r_{p_1} \dots r_{p_a} u_{q_1} \dots u_{q_b} \frac{1}{a!} \frac{n!}{p_1! \dots p_a! q_1! \dots q_b!}$$

(we need to divide by $a!$ because order of reliable vertices doesn't matter)

Then for EGFs of R and U of the sequences $\{r_n\}_{n=0}^{\infty}$ and $\{u_n\}_{n=0}^{\infty}$ we get

$$R = x \sum_{a=0}^{\infty} \frac{R^a}{a!} \sum_{b=0}^{a+1} U^b = x \sum_{a=0}^{\infty} \frac{R^a}{a!} \frac{1-U^{a+2}}{1-U} = \frac{x}{1-U}(e^R - U^2 e^{RU})$$

The only difference for an unreliable vertex is that if such vertex has a reliable and b unreliable children that $b \leqslant a-1$, so for U we get

$$U = x \sum_{a=0}^{\infty} \frac{R^a}{a!} \sum_{b=0}^{a-1} U^b = \frac{x}{1-U}(e^R - e^{RU})$$

Now we have two closed-form equation on two generating functions, if we solve them then we solve the problem. The rest of the editorial will be about solving this system.

We use the approach similar to Newthon's method. Finding the first two coeffitients of R and U is trivial. Then we assume that we know both R and S modulo x^n so $R = R_0 + x^n R_1$ and $U = U_0 + x^n U_1$ where R_0 and U_0 are known and we want to find R_1 and U_1 modulo x^n. We want to preserve the form of $\alpha + x^n \beta R_1 + x^n \gamma U_1$ with some known α, β and γ through all of our multiplications, inversions and exponents, so we look at our equations modulo x^{2n}. The form naturally preserves through multiplications because all the non-linear in respect to R_1 and U_1 ($R_1 U_1$ also counts) are also divisible by x^{2n}, so they cancel out. And it preserves through any analytic (such as e^x of $\frac{1}{1+x}$) function, since, like in Newthon's method, for any analytic function $H(x)$, $H(f(x) + x^n g(x)) = H(f(x)) + x^n g(x) H'(f(x)) \pmod{x^{2n}}$. In our case f is known and g is linear in respect to R_1 and U_1 with known coefficients.

After we remove all the brackets in all the functions in our equations, we are left with a system of linear equations on R_1 and U_1 modulo x^n (after we divide by x^n both the module and the equations). If we explicitly write them, we will see that the determinant is not divisible by x, so this system is easily solvable since we can divide formal power serieses. Note that this modification of the Newthon's

method isn't problem-specific and can solve any solvable system of functional equations with calculatable functions.

Like in Newthon's method the time complexity is $\mathcal{O}(n \log n)$ but with a very large constant.

Also, to reduce both running time of the program and complexity of the formulas that we write on paper, we can replace our formulas with $R-U$ and $R-U^3$ to cancel one of the exponents in each equation. This way we get

$$R - U = x(1+U)e^{RU}$$

and

$$R - U^3 = x(1+U)e^{R}$$

which are much easier to deal with.

Problem Tutorial: "Institute"

Let's leave only the vertices that are reachable from the first one along any edges, that is, those vertices that can be reached at all.

After that, let's remove all the edges for which a pass is needed. That is, leave only those that you can walk along after losing the pass.

All that's left is check whether the graph has a pair of vertices u, v such that v is reachable from u but u is not reachable from v. That is, is it true that all connected components (assuming all edges are undirected) are strongly connected.

Problem Tutorial: "Jumping Lights"

This is one of the harder problems in the contest so we will describe the solution first and then prove it and then find it's time complexity.

0.1 Algorithm

We will maintain two special structures for the tree. We will call them even tree and odd tree. One of the trees will be representing the current state of the tree and after the operation 2 (we will call it switch operation) we will switch the trees and do some additional changes so that all the states of vertices are correct.

Tree structure has:

- Structure storing the state of all non-leaf vertices (marked or unmarked)
- In each vertex v (which is not a leaf) there is a special structure for its children which are leaves. This structure will store states of all the leaves and supports operations: get state of vertex, mark/unmark a vertex, mark/unmark all the vertices, get the number of marked vertices in $O(1)$
- A number - counter of marked vertices
- In each vertex v (which is not a leaf) there is a special structure storing its unmarked children (which aren't leaves). (this can be done with set or a hash-set to support changes in $O(\log n)$) or $O(1)$)
- A bag - any structure allowing storing vertices to iterate over them and then clear the structure

When some vertex's (not a leaf) state is changed, in this tree structure some extra things will be done:

- If the vertex is marked, it should be added to the bag
- If the vertex has a parent, it should be added to it's parents structure storing unmarked children, or removed from it

- Update the counter of marked vertices

To complete the first two operations (mark or unmark a vertex v) we will do the following:

- If the vertex is a leaf, we will update it's state using the special leaves structure and fix the counter of marked vertices in the current tree. Then we will add its parent to the bags in both trees.
- If the vertex is not a leaf, we will update it's state as described above. Then we will also add the vertex to the bags in both trees, add its parent to the bag in the current tree.

To complete the switch operation we will iterate over all the vertices in the bag. For each vertex v we will do the following:

- Calculate the state it should have in the second tree, and update the state as described above (if necessary)
- Recalculate the states of its children leaves in the second tree (they all will be either marked or unmarked)
- If v was marked in the current tree, we will iterate over its unmarked children in the second tree and mark them all (as described above) and also mark its parent if it was unmarked.

After it we will switch the trees - second becomes the current one and the current one becomes the second.

0.2 Correctness

We will prove the following statements with the induction on time:

- The bag in a tree contains all the vertices (non-leaves) which were unmarked two switch operations ago and are marked now
- All the maintained states of all the vertices in the current tree are correct

It is obvious that after any operation of first two types (mark or unmark a vertex) the statements remain true.

Now we will prove that after switch operation the statements remain true. We will consider the difference between the states of the vertices of the new tree (the states which should be) and the state of the vertices of the tree right before the previous switch operation (which is currently stored in the second tree structure).

We will call the current tree T_1 (right before the switch), the states of all the vertices right before the previous switch operation T_0 (just a tree without marked vertices if it's the first switch operation) and the states of all the vertices after the switch T_2. (analogically we will define the tree T_{-1} : states of vertices right before switch operation which was before the previous one)

Let v be the vertex, which states are different in T_0 and T_2.

- It is not a leaf, marked in T_0, unmarked in T_2. Then all of it's children were marked right after the first switch operation. But none of them is marked in T_1 (because otherwise v would be marked in T_2). It means that all of them were unmarked manually with the operation of type 0. Then the vertex v was added to the bag in current tree during that operation. Then its state is recalculated correctly. (also, here it is added to the bag)
- It is a leaf, marked in T_0, unmarked in T_2. Similarly to the first case, its parent was marked right after first switch operation and is unmarked in T_1, similarly, its parent was added to the bag in current tree when it was manually unmarked. Then its state is recalculated correctly.

- It is not a leaf, unmarked in T_0, marked in T_2. It means that one of its children or its parent is marked in T_1 - vertex u.
  - If u was marked in T_{-1}, v was marked right after switch operation between T_{-1} and T_0, it means that it was manually unmarked. Then u is in the current bag and v as its parent or its child will be marked. (or u is a leaf, when it was manually unmarked, its parent - v was added into both bags)
  - If u wasn't marked in T_{-1}, then according to the first statement, it should be in the current bag. Then v as its parent or its child will be marked. (or u is a leaf, since v was unmarked in T_0, u was unmarked right after the first switch operation, it means that u was manually marked, v is in the bag in current tree)
- It is a leaf, unmarked in T_0, marked in T_2. It means that its parent is marked in T_1 - vertex u.
  - If u was marked in T_{-1}, v was marked right after switch operation between T_{-1} and T_0, it means that it was manually unmarked. Then u was added to both bags and v as its child will be recalculated correctly
  - If u wasn't marked in T_{-1}, then according to the first statement, it should be in the current bag. Then v will be recalculated correctly

Now the correctness of both statements is proved. It means that the algorithm is working correctly.

0.3 Time complexity

Let sz : the sum of the sizes of both bags, cnt : the number of unmarked vertices in both trees, m : the number of manually unmarked vertices since the last switch operation. We will define a potential function $\Phi = sz + 2 \cdot cnt + 3 \cdot m$.

Then the change of the state of the vertex inside the tree structure works either in $1+1-2=0$ amortized time (1 : real time, 1 : add it to the bag, -2 : number of unmarked vertices decreases) if it is marked, or in $1+2=3$ amortized time (1 : real time, 2 : number of unmarked vertices increases) if it is unmarked.

Then the iteration over all the unmarked children (and unmarked parent if necessary) and marking them works in 0 amortized time.

Iteration over vertex v in a bag works in:

- Its state remains the same - $1-1=0$ amortized time (1 : real time, -1 : decreases the size of the bag)
- It gets marked - $1-1+0=0$ amortized time (1 : real time, -1 : decreases the size of the bag, 0 : amortized time to change its state in the structure)
- It gets unmarked - $1-1+3=3$ amortized time (1 : real time, -1 : decreases the size of the bag, 3 : amortized time to change its state in the structure)

So the switch operation works in $3 \cdot k$ amortized time, where k is the number of vertices unmarked during the iteration over the bag. But, as we could see from the proof above, if the vertex gets unmarked inside this iteration over vertices, one of its children was manually unmarked somewhere between this switch operation and the previous switch operation. So, since the potential is decreased by $3 \cdot m$ after this switch operation, it works in ≤ 0 amortized time.

The operations of types 0 and 1 work in up to $1+3+3+3=10$ amortized time (1 : real time, 3 : maximum amortized time to change its state in the structure, 3 : maximum increase of the sizes of bags, 3 : if it gets unmarked)

Since $\Phi \leq 9 \cdot n$, all the operations work amortized in up to $10 \cdot q$ time, the total time is linear $O(n+q)$

(or, if we use sets instead of hash-sets to store the unmarked children of vertices, it works in $O((n+q)\log n)$ time, which in practice works faster)