Codechef Learn, Episode 1
Lec-1 **Square Root Decomposition on Trees**

https://youtu.be/g5g1UqSjOIQ

tanujkhattar@

Square Root Decomposition Recap

There are 4 different types of “standard” Square Root Decomposition techniques:

- Array Square Root Decomposition
- (Lookahead) Query Square Root Decomposition
- MOs Algorithm (+ updates)
- Heavy Set / Light Set based Query Decomposition

Recap: Array Square Root Decomposition

- Step-1: Divide the array of **N** elements into **N / B** blocks of size **B**.
- Step-2: Maintain some data structure / information for every block **b.**
    - The DS should support query and updates in time **sublinear(B).**
- Step-3: Support a query/update on range [L, R] by
    - Iterating over individual elements for blocks of **L** and **R** - takes **O(2 * B) ~ O(B * sublinear(B))**
    - Iterating over at-most **N / B** blocks in between **L** and **R** and use the data structure maintained + some lazy update information for query & updates - takes **O(N / B * sublinear(B))**
- Step-4: Choose appropriate B to minimize the complexity.
    - B ~ sqrt(N) in general

Recap: Array Square Root Decomposition

- Given an array A of N elements, support the following operations:
  - Query L R X : Tell number of elements >= X in [L, R]
  - Update L R V: Add V to all elements in the range [L, R]

Recap: Array Square Root Decomposition

- Given an array A of N elements, support the following operations:
  - Query L R X : Tell number of elements >= X in [L, R]
  - Update L R V: Add V to all elements in the range [L, R]
- Solution:
  - Divide the array into blocks of size B and maintain a sorted vector of elements for each block
  - Also maintain a lazy_add[b] integer for each block.
  - To process a query [L, R], X
    - For blocks of L & R, iterate over all individual elements and check A[i] + lazy_add[b] >= X.
      - O(2 * B)
    - For every block in between, binary search for # elements >= X - lazy_add[b] :
      - O(N / B * logB)
  - To process an update [L, R], V
    - Recompute the sorted vector of blocks L & R -- O(2 * B logB)
    - For blocks in between, simply update lazy_add[b] += V -- O(N / B)
  - Total complexity: O(N / B logB + B * logB) → O(N * sqrt(N) * logN) (choose B ~ sqrt(N))

Recap: Query Square Root Decomposition

- Divide the **Q** queries into **Q / B** blocks of size **B**.
- Maintain some information / data structure using which the queries can be answered efficiently.
  - **The data structure can be of the type “hard to update” but “easy to query”. Eg: Prefix Sums.**
- After every block, process all the updates which occurred in that block together via some expensive method (eg: in O(N) time).
  - Since there are Q / B blocks, this would take O(N * Q / B) time.
- To answer a query,
  - Compute an approximate answer using information maintained till end of previous block
  - **Iterate on all updates in the current block and reflect the contribution of an update on the current query quickly (~O(B) such updates).**

Recap: Query Square Root Decomposition

- Given a tree with N nodes, each node contains 0 coins. Support Q operations of the form:
  - Update L Y : Add Y coins to all nodes which are distance L from the root.
  - 2 X : Tell the sum of coins of all nodes in the subtree rooted at node X.
- https://codeforces.com/gym/100589/problem/A

Recap: Query Square Root Decomposition

- Given a tree with N nodes, each node contains 0 coins. Support Q operations of the form:
  - Update L Y : Add Y coins to all nodes which are distance L from the root.
  - 2 X : Tell the sum of coins of all nodes in the subtree rooted at node X.
- Solution
  - Do a DFS and push all nodes at level L in a sorted vector nodes[L]
  - Divide the queries in blocks of sqrt(N) and for each query,
    - Get an approx answer by a precomputed dfs storing subtree sums of each node.
    - Process all updates that have occurred before this query in the same block.
    - Reflect contribution of an update L Y on query X, find number of nodes in subtree of X at distance L from the root -- This can easily be done by binary search on nodes[L].
    - This takes O(sqrt(N) * logN) per query.
  - After each block of sqrt(N), process all updates in a single O(N) DFS.
  - Therefore, total complexity is O(N * sqrt(N) * logN).

Recap: MOs Algorithm

- Find a way to quickly "add" and "remove" an element to a range.
  - Given some DS and an answer for the range [L, R], we should be able to quickly "add"/"remove" an element s.t. we have updated DS and updated answer for range [L, R+1] / [L, R - 1].
- Notice that it takes $|L_1 - L_2| + |R_1 - R_2|$ operations to go from $[L_1, R_1]$ to $[L_2, R_2]$.
  - Here an "operation" refers to the add or remove operation.
- Sort the queries offline such that $\sum (|L_i - L_{i+1}| + |R_i - R_{i+1}|)$ is minimized.
  - Reduces to TSP - NP Hard.
  - Can sort the queries smartly such that this summation is O((N + Q) * Sqrt(Q))
  - bool cmp(Query a, Query b) {
    - return (a.lb < b.lb) || (a.lb == b.lb && a.r < b.r);
  - }

Recap: MOs Algorithm

- Given a tree with N nodes, Support Q operations of the form:
  - Q u v : Tell the number of distinct elements on the path from u to v
- https://www.spoj.com/problems/COT2/

Recap: MOs Algorithm

- Given a tree with N nodes, Support Q operations of the form:
  - Q u v : Tell the number of distinct elements on the path from u to v
- Solution
  - Linearize the tree using ETT Way-2 such that each node is pushed in the ETT array twice upon entry and exit.
  - Range [en[u], st[v]] contains single occurrence of all nodes on the path from u to v (except LCA) and double occurrence of all nodes not on the path from u to v.
  - Sort the queries via MOs algorithm and maintain count of each node + frequency of each distinct element seen the range.
  - To add/remove an element in the range
    - If count of node is event, add the element or else remove the element.
    - To add/remove an element, increase/decrease counts in frequency array.
    - Keep a track of number of distinct elements seen so far by observing $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions in the frequency array.

Recap: Heavy Set / Light Set based Sqrt Decomp.

- Identify some property in the problem statement which can divided into Heavy(Large) Set and Light (Small) Set
  - SizeOf(HeavySet) > B
  - SizeOf(LightSet) < B
- We process the Heavy Set and Light Sets independently in order to obtain amortized O(N * Sqrt(N))
  - Heavy Set
    - Since SizeOf(HeavySet) > B, Therefore at-most N / B such Heavy Sets possible.
    - Process each Heavy Set in O(g(N)) like O(N) time -- O(N * N / B).
  - Light Set
    - Process each Light Set in O(f(CNT)) like O(CNT^2) etc. where CNT = SizeOf(LightSet)
    - ∑ (CNT^2) <= O(N * B)

Recap: Heavy Set / Light Set based Sqrt Decomp.

- Given a weighted tree with N nodes and Q queries of the form:
  - K n1, n2, n3 … nk : Find the sum of distances between each pair of nodes.
- https://www.hackerrank.com/contests/worldcupsemifinals/challenges/wccity

Recap: Heavy Set / Light Set based Sqrt Decomp.

- Given a weighted tree with N nodes and Q queries of the form:
  - K n1, n2, n3 … nk : Find the sum of distances between each pair of nodes.
- Solution:
  - Since sum of K over all queries is bounded (~O(N)).
  - For K <= sqrt(N) : Do an O(K ** 2) approach by iterative over each pair.
  - For K >= sqrt(N) : Do an O(N) approach by DFS on the whole tree.
  - Final complexity: O(N * root(N)).

Recap: Square Root Decomposition

- Array Square Root Decomposition
  - Divide the array into blocks of Size B ~ Sqrt(N) & maintain some information for each block.
  - Divide a query/update into two parts
    - Individual elements in blocks of L & R
    - Complete Blocks which lie between L & R
  - Perform Range Updates Lazily
- Query Square Root Decomposition
  - Divide Q queries into blocks of size B ~Sqrt(Q)
  - Process all updates at the end of each block and maintain a “hard-to-update” DS for queries.
  - Reflect contribution of B updates on a query “quickly”.
- MOs Algorithm
  - Find a way to quickly “add”/”remove” elements in a range (eg: [L, R] → [L, R + 1] / [L, R - 1])
  - Process the queries in a sorted order s.t. ∑ (|Li - Li+1| + |Ri - Ri+1|) is minimized / bounded.
- Heavy Set / Light Set based Query Decomposition
  - Identify a property whose sum(count) is bounded and can divided into heavy and light sets.
  - Process the heavy and light sets separately - Eg: O(N * #HeavySets) + O(SizeOf(LightSet)^2).

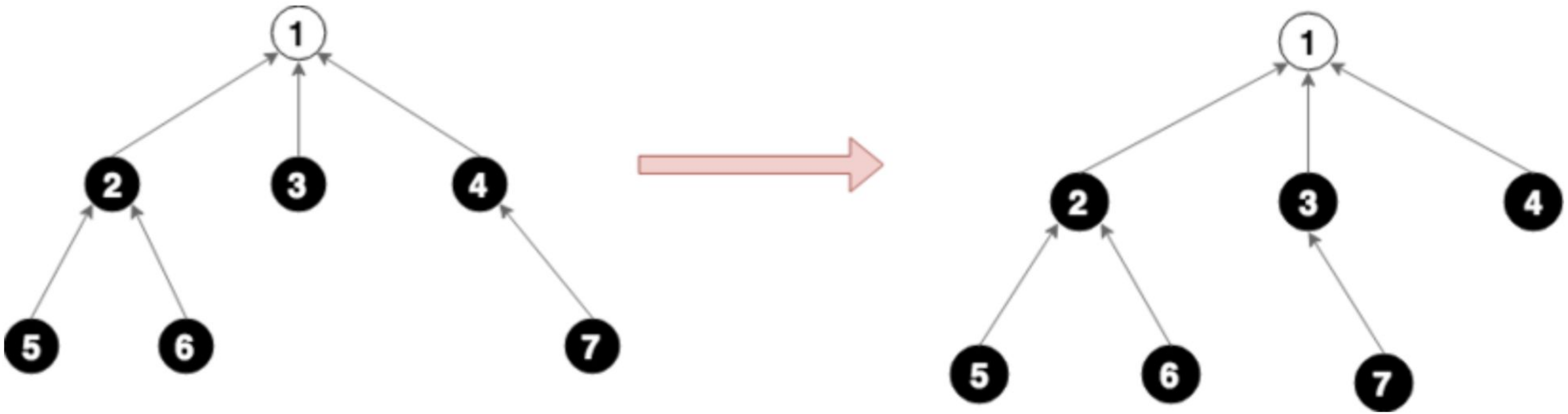
Dynamic Trees - Hackerrank

- Given a tree with N nodes where p_i (< i) represents the parent of node i and each node is either black or white. Support the following queries
  - T x : Toggle the color of node x.
  - C u v : Set p_u = v i.e. remove the edge u → p_u and add the edge u → v (v < u to ensure that graph remains a tree after the update).
  - K u v k : Output k'th black node on the path from u to v.

- Practice Problem
  - https://www.hackerrank.com/contests/world-codesprint-13/challenges/dynamic-trees

Dynamic Trees - Hackerrank

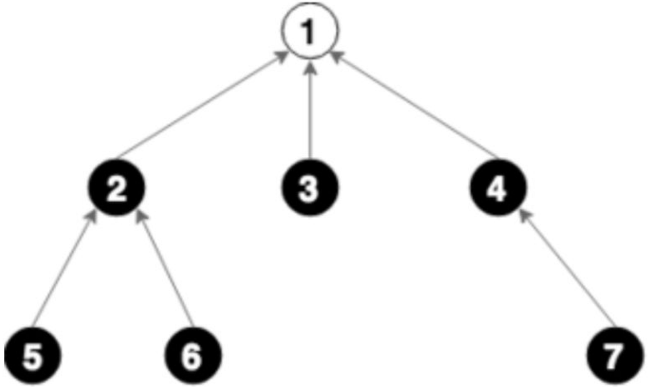
- `K 5 3 2` - The query gives result 2, since the $2nd$ black node on the simple path from 5 to 3 is 2
- `T 1` - The query results in node 1 being toggled from black to white
- `C 7 3` - The parent node of node 7 is set to 3



Any Ideas?

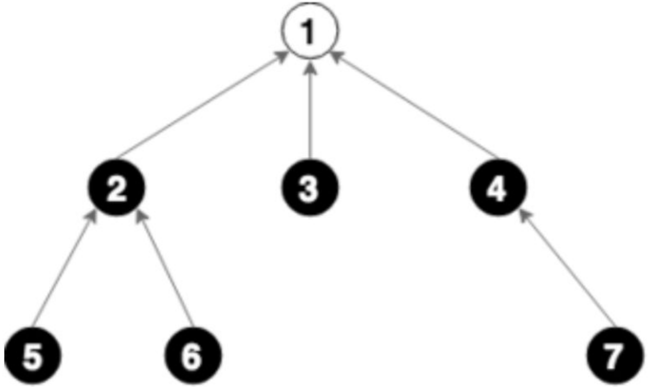
Solution Idea

- Notice the invariant in the problem that $p_i < i$ always.
- What happens if I divide the array [1, 2, …, N] into blocks of size B and add edges between $P[i] \to i$?



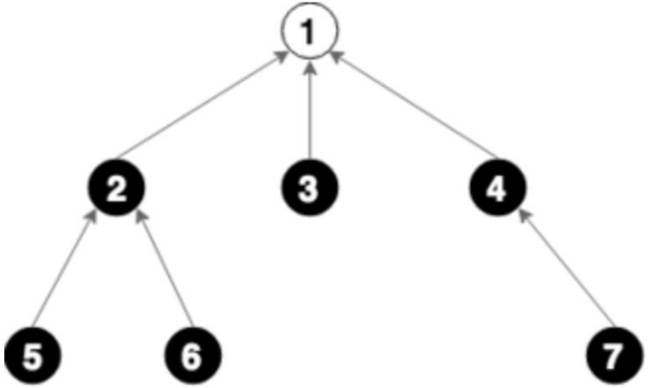
Solution Idea

- Notice the invariant in the problem that p_i < i always.
- What happens if I divide the array [1, 2, …, N] into blocks of size B and add edges between P[i] → i ?
- We have divided the tree into a forest of trees s.t.
  - All nodes in any tree lie in a particular block. Hence no. of nodes in a tree <= O(sqrtN).



Solution Idea

- Notice the invariant in the problem that p_i < i always.
- What happens if I divide the array [1, 2, …, N] into blocks of size B and add edges between P[i] → i ?
- We have divided the tree into a forest of trees s.t.
  - All nodes in any tree lie in a particular block. Hence no. of nodes in a tree <= O(sqrtN).
  - Every time you traverse an edge from x → P[x] s.t. root[x] != root[P[x]], you jump to a block on the left.
    - Because P[x] < x and hence BLOCK[P[x]] < BLOCK[x].



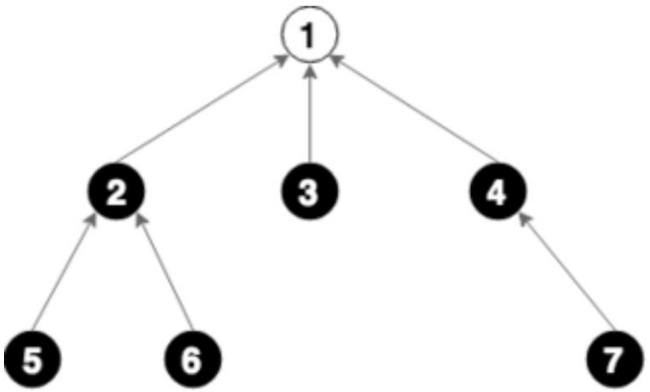
Decomposing the Tree

```
void dfs(int u) {
  for (auto w : g[u]) {
    // P[w] = u & BLOCK[w] == BLOCK[u];
    level[w] = level[u] + 1;
    ans[w] = ans[u] + A[w];
    root[w] = root[u];
    dfs(w);
  }
}
```

```
void process(int t) {
  // Build forest for each square root block
  int st = max(1, t * SQRT), en = min(n + 1, (t + 1) * SQRT);
  for (int i = st; i < en; i++)
    if (BLOCK[P[i]] != t || P[i] == i)
      level[i] = 0, root[i] = i, ans[i] = A[i], dfs(i);
  // done :)
}
```

Solution Idea

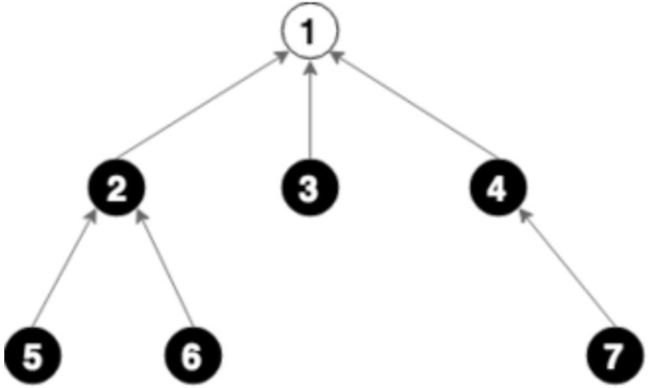
- How do you find LCA of two nodes x & y using this square root structure?



Solution Idea

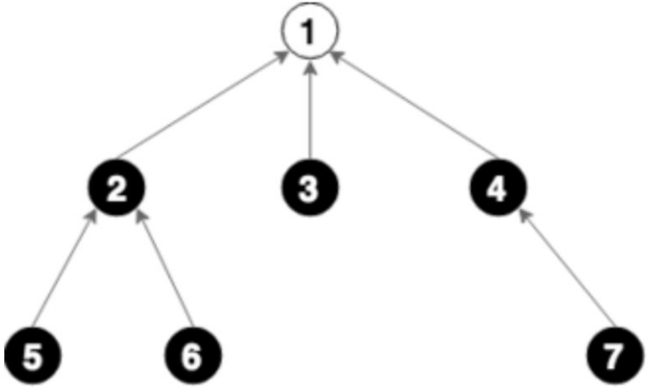
- How do you find LCA of two nodes x & y using this square root structure?

```
int lca(int x, int y){
    while(root[x] != root[y]){
        if(x > y) swap(x, y);
        y = P[root[y]];
    }
    if(level[x] > level[y]) swap(x, y);
    while(level[y] > level[x])y = P[y];
    while(x != y) x = P[x], y = P[y];
    return x;
}
```



Solution Idea

- How to answer Queries? (K’th black node from x → y)
    - For each node x in the forest, store no. of black nodes from x → root[x].
    - Use this information and traverse the blocks like LCA query to answer the query in O(rootN).



Solution Idea

- How to handle Updates?
  - **Type-1: Toggle Color of node**
    - Only cnt[x] values of all x in the same block change.
    - Recompute the block in O(rootN).
  - **Type-2: Change Parent.**
    - The structure would only change in 1 block in which node x lies.
    - Hence recompute the values of those block in O(rootN).

Implementation

- https://github.com/tanujkhattar/cp-teaching/blob/master/CWC/Ep02%20SRD%20on%20Trees/dynamic_trees.cpp

Subtree Update & Path Queries (non invertible fn)

- Given a rooted tree with N nodes and Q queries of the form:
  - A X V - Add V to all nodes in the subtree of X.
  - Q X Y - Report maximum value on the path from A to B.
- Practice Problem:
  - https://www.hackerrank.com/challenges/subtrees-and-paths/problem

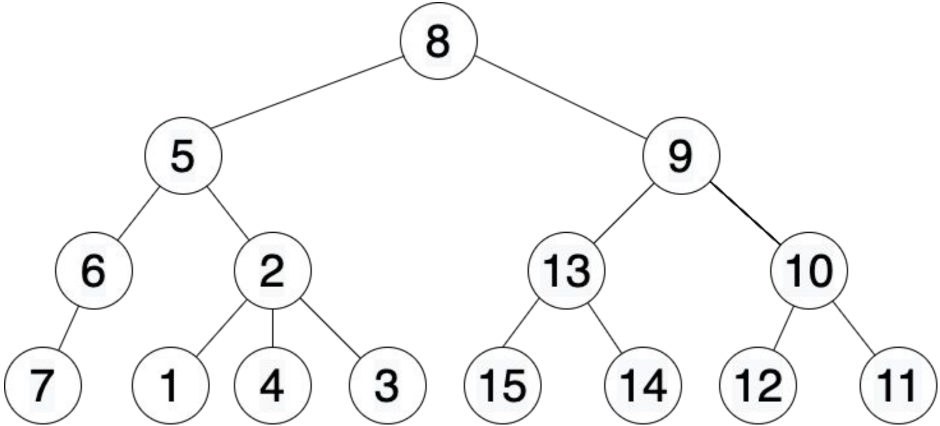
Subtree Update & Path Queries (non invertible fn)

- Linearize the tree using ETT Way-1
- Note that once the tree is linearized using ETT, the condition that st[P[x]] < st[x] is satisfied for all x.
- Hence, we can do Array Square Root Decomposition on this ETT Way-1 linearized array.

Subtree Update & Path Queries (non invertible fn)

- To perform subtree updates
  - perform an update from [L, R] on the ETT using Array Square Root Decomposition.
- To answer path queries,
  - traverse the decomposed blocks from right to left on the ETT array.
  - Use the DS + Lazy information stored for each block to answer path queries.
  - Note that path query for any block is from node to root of the processed tree.

ETT Way-1 : [8, 5, 6, 7, 2, 1, 4, 3, 9, 13, 15, 14, 10, 12, 11]



Implementation

- https://github.com/tanujkhattar/cp-teaching/blob/master/CWC/Ep02%20SRD%20on%20Trees/subtree_and_paths.cpp

Can we support Path Updates as well?

- Updates on paths would require us to maintain another special structure for each processed tree inside the ETT sqrt blocks because ETT doesn't store path information by default.
- We could do HLD (+ETT) and build a sqrt structure on top of it but then updates would take O(logN * sqrt(N) * (time of DS per update))
- We'll look at another advance technique tomorrow to support path updates!
  - Try https://www.codechef.com/problems/TRS for tomorrow!
- See Lecture-2 to learn more https://youtu.be/8VHWdNnP3h4