Tutorium #3

11/11/2022

There is an undirected connected tree with `n` nodes labeled from `0` to `n - 1` and `n - 1` edges.

You are given the integer `n` and the array `edges` where `edges[i] = [a_i, b_i]` indicates that there is an edge between nodes `a_i` and `b_i` in the tree.

Return an array `answer` of length `n` where `answer[i]` is the sum of the distances between the `i^th` node in the tree and all other nodes.

Is hard

Problem 1 (4 points)

1. Give a state of a *Fibonacci heaps* s.t. the next `deleteMin` operation need time $O(n)$. Also give a corresponding `deleteMin` operation.

2. Describe how the state that you described can be reached.

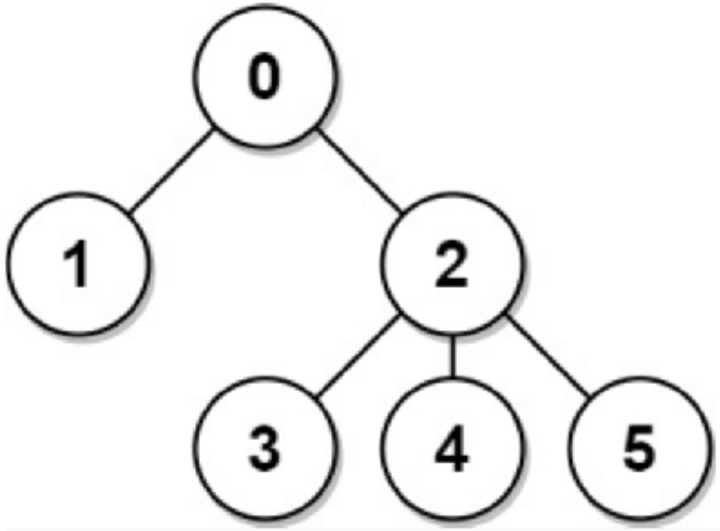
Solution:

1. If at the beginning of a `deleteMin` operation we have a forest in which each node forms its own tree, then `deleteMin` takes linear time. This is due to the linear number of `link` operations that have to be performed on the forest. In every step, the number of trees is reduced by one. Is the number of nodes a power of two, then the process is continued until only one tree is left. The number of set bits of the binary represantation of n yields the number of created trees. Since we can only have logarithmically many bits set, the `deleteMin` operations runs in time $n - log(n)$ and is hence linear.

   Note: this argument only works since we look at full trees from which no node has been removed using a `cut` operation. Hence, a Fibonacci tree constains in bucket k exactly 2^k nodes.

2. One can reach such a state by a sequence of n `insert` operations, since an `insert` only inserts into a new tree into the forest.

Example 1:




```
Input: n = 6, edges = [[0,1],[0,2],[2,3],[2,4],[2,5]]
Output: [8,12,6,10,10,10]
Explanation: The tree is shown above.
We can see that dist(0,1) + dist(0,2) + dist(0,3) + dist(0,4) + dist(0,5)
equals 1 + 1 + 2 + 2 + 2 = 8.
Hence, answer[0] = 8, and so on.
```

Example 2:

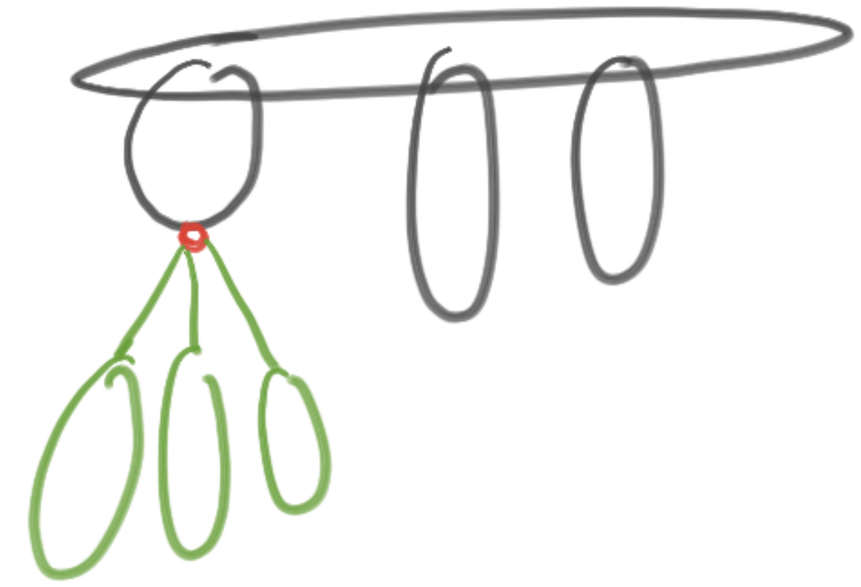

```
Input: n = 1, edges = []
Output: [0]
```

Problem 2 (4 points)

1. Extend the data structure *pairing heap* with an operation `increaseKey(h: Handle, k: Key)`. Your operation should need time $O(\log n)$ amortized. Give pseudocode. What would you do if the starting point would be a *binary heap*?

2. Design a data structure that provides the operations `insert` in $O(\log n)$, Median in $O(1)$ and remove Median in $O(\log n)$. A description with words is sufficient.

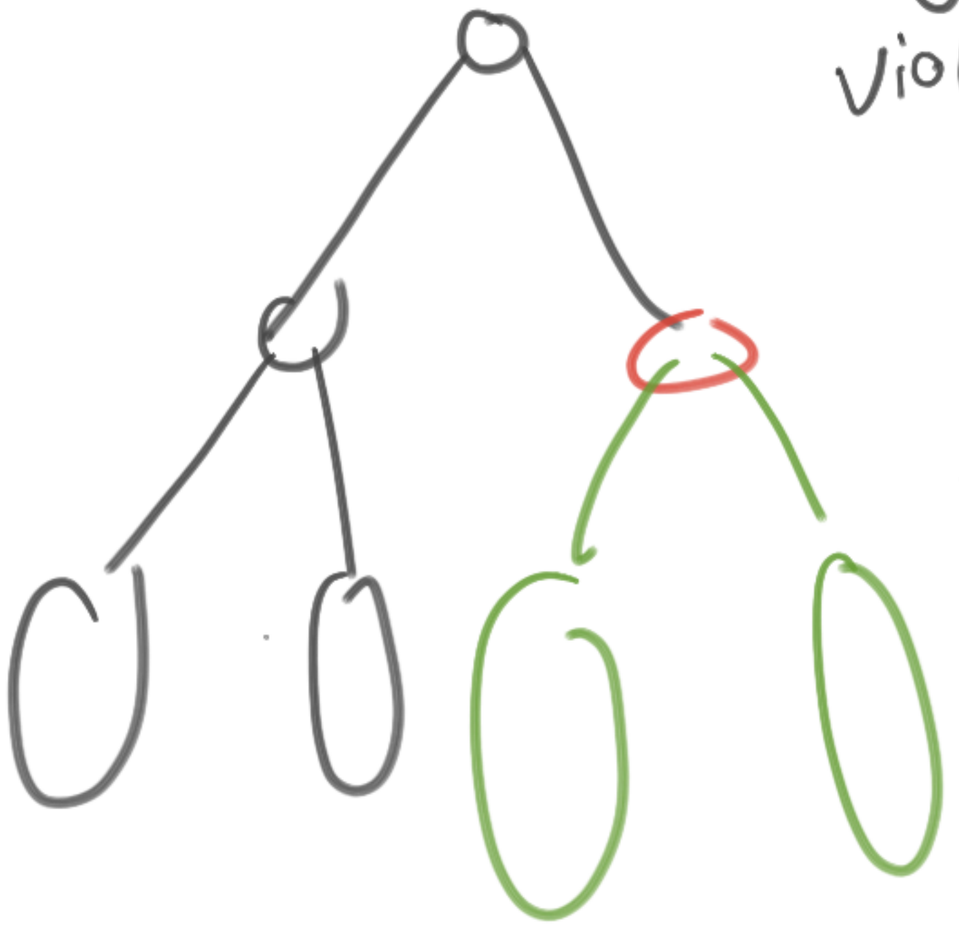
Pairing heap



the heap property might be violated towards the children

→ cut all children and perform union-ops.

Binary Heap



again, maybe violated towards children

⇒ 1 siftdown

1. Delete the element from the data structure ($O(\log n)$) and insert it with the changed key ($O(1)$):

```
1: function INCREASEKEY(h : Handle, k : Key)
2:     remove(h)
3:     key(h) := k
4:     insert(h)
5: end function
```

In a *binary heap* one would first change the key and then call a `siftDown` operation.

2. Construction of data structure:

   - store current median v.
   - use a maximum *priority queue* (MaxPQ) for elements smaller than v and a minimum *priority queue* (MinPQ) for elements larger than v.

   Find median:
   return v directly: $O(1)$.

   Delete median:
   Exchange current median v by the top element of the larger *priority queue* (if both have the same size, use MinPQ): `deleteMin` in $O(\log n)$, for example using *Fibonacci heap*.

   Insert element:
   Insert new element –depending on v– in one of the *priority queues*: `insert` in $O(1)$. Insert v in the smaller one (if both have the same size, take MaxPQ): `insert` in $O(1)$. Replace v by the top element of the larger *priority queue* (if both have the same size use MinPQ): `deleteMin` in $O(\log n)$, for example using *Fibonacci Heap*.

Problem 3 (4 points)

1. Let $\text{pot}(\cdot)$ be a feasible potential function for the search of a t in graph $G = (V,\ E)$. Verify if

$$\text{pot}^c = \text{pot} + c,\ \ c = const.$$

is also a feasible potential function.

Solution:

1. We have to check if

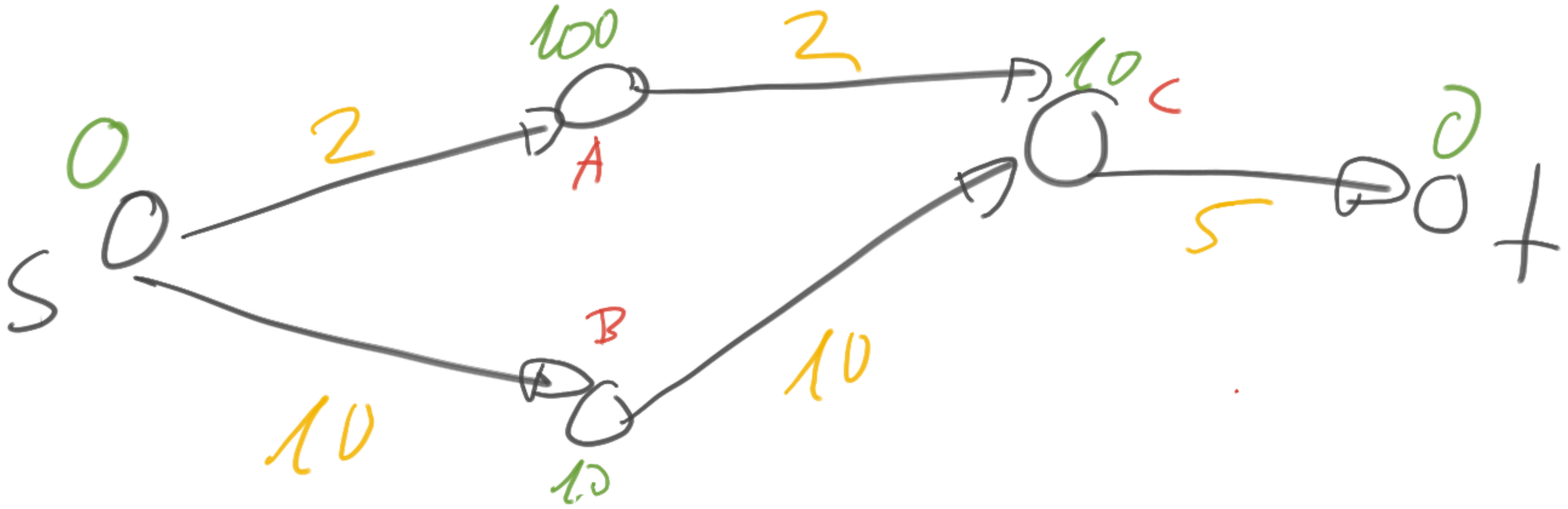
$$\begin{aligned} c(u,v) + \text{pot}^c(v) - \text{pot}^c(u) &\geq 0 && (1)\\ \text{pot}^c(u) &\leq \mu(u,t) && (2) \end{aligned}$$

holds.

Condition (1) is always satisfied. After substitution we get $c(u,v) + \text{pot}(v) - \text{pot}(u) \geq 0$. Since $\text{pot}(\cdot)$ is a feasible potential function, this is always true.

Condition (2) is only true if $c \leq \mu(u,t) - \text{pot}(u)$ f.a. $u \in V$.

Hence, $\text{pot}^c(\cdot)$ is only for proper choices of c a feasible potential function.



$(s,0) \rightarrow (B,20), (A,102) \rightarrow (C,30), (A,102)$

$\rightarrow (t,25), (A,102)$ STOP

http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

The heuristic can be used to control A*'s behavior.

- At one extreme, if `h(n)` is 0, then only `g(n)` plays a role, and A* turns into Dijkstra's Algorithm, which is guaranteed to find a shortest path.
- If `h(n)` is always lower than (or equal to) the cost of moving from `n` to the goal, then A* is guaranteed to find a shortest path. The lower `h(n)` is, the more node A* expands, making it slower.
- If `h(n)` is exactly equal to the cost of moving from `n` to the goal, then A* will only follow the best path and never expand anything else, making it very fast. Although you can't make this happen in all cases, you can make it exact in some special cases. It's nice to know that given perfect information, A* will behave perfectly.
- If `h(n)` is sometimes greater than the cost of moving from `n` to the goal, then A* is not guaranteed to find a shortest path, but it can run faster.
- At the other extreme, if `h(n)` is very high relative to `g(n)`, then only `h(n)` plays a role, and A* turns into Greedy Best-First-Search.

Problem 4 (4 points)

Given is a directed graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, as well as a weight function $c : E \to \mathbb{R}_0^+$.

1. Proof the claim of the lecture that for $m = \Omega(n \log n \log\log n)$ Dijkstras algorithm with *binary heap* has a average running time of $O(m)$.

2. A special *priority queue* has the following running time properties:

   - `insert`: $O(\log n)$
   - `decreaseKey`: $O(1)$
   - `deleteMin`: $O(\sqrt{m})$

   (TBH we don't if such a data structure exists, but this is a different question and for the sake of the excercise, we just assume it :))

   Give the smallest upper bound for the running time of Dijkstras algorithm if this priority queue is used. Under which condition of the relationship of n and m does the running time become linear in the input size?

3. Give a family of graphs such that the number of `deleteMin` operations from an arbitrary node to all reachable nodes depends linearly on the path length $\mu(s, \cdot)$, given that $m = \Omega(n)$.

Solution:

1. For the average running time of Dijkstras algorithm with *binary heap* we have:

$$O(m + n \log \frac{m}{n} \log n)$$

We have to show that this running time is in $O(m)$ for $m = \Omega(n \log n \log \log n)$. Use the smallest possible value of m. If the assumption hold for this value, it also holds for larger values of m. We get:

$$\begin{aligned}
& O(n \log n \log \log n + n \log \frac{n \log n \log \log n}{n} \log n) \\
\overset{\text{shortening}}{=} \quad & O(n \log n \log \log n + n \log(\log n \log \log n) \log n) \\
\overset{\log ab = \log a + \log b}{=} \quad & O(n \log n \log \log n + n \log \log n + n \log \log \log n \log n) \\
\overset{\log \log \log n = O(\log \log n)}{=} \quad & O(n \log n \log \log n) \\
= \quad & O(m)
\end{aligned}$$

Hence, the running time is in $O(m)$.

2. The general running time of Dijkstras algorithm is:

$$O(m + m \cdot T_{decreaseKey}(n) + n \cdot (T_{deleteMin}(n) + T_{insert}(n)))$$

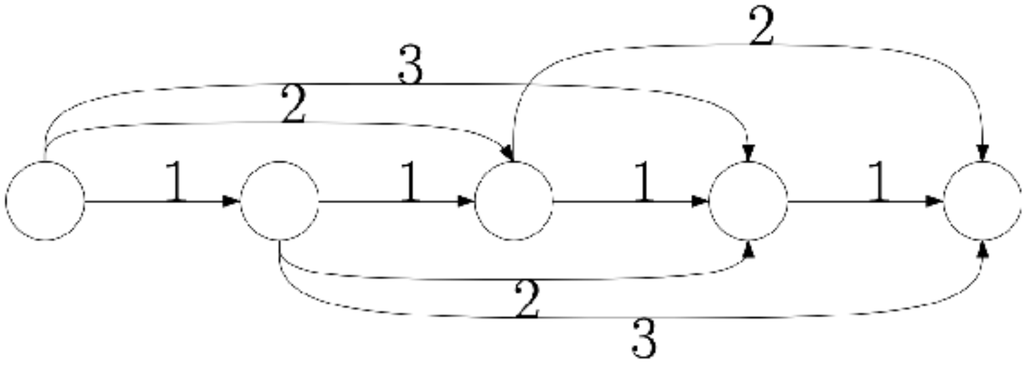
With the given running times of the operations we get:

$$O(m + n\sqrt{m} + n \log n)$$

With the condition that $n \cdot \sqrt{m} = O(m)$ and $n \log n = O(m)$, we get linear running time in m. This can be transformed to $\Omega(n) = \sqrt{m}$ and $\Omega(n \log n) = m$. Hence, $m = \Omega(n^2)$.

3. A family of graphs that satisfies these conditions can be constructed as follows:

Build a directed chain of n nodes, conneced by edges of weight 1. Moreover, insert edges from every node i to $\log n$ successors j with an edge that has weight larger or equal to the distance of i and j on the chain.



Beispiel mit 5 Knoten

Obviously from all nodes

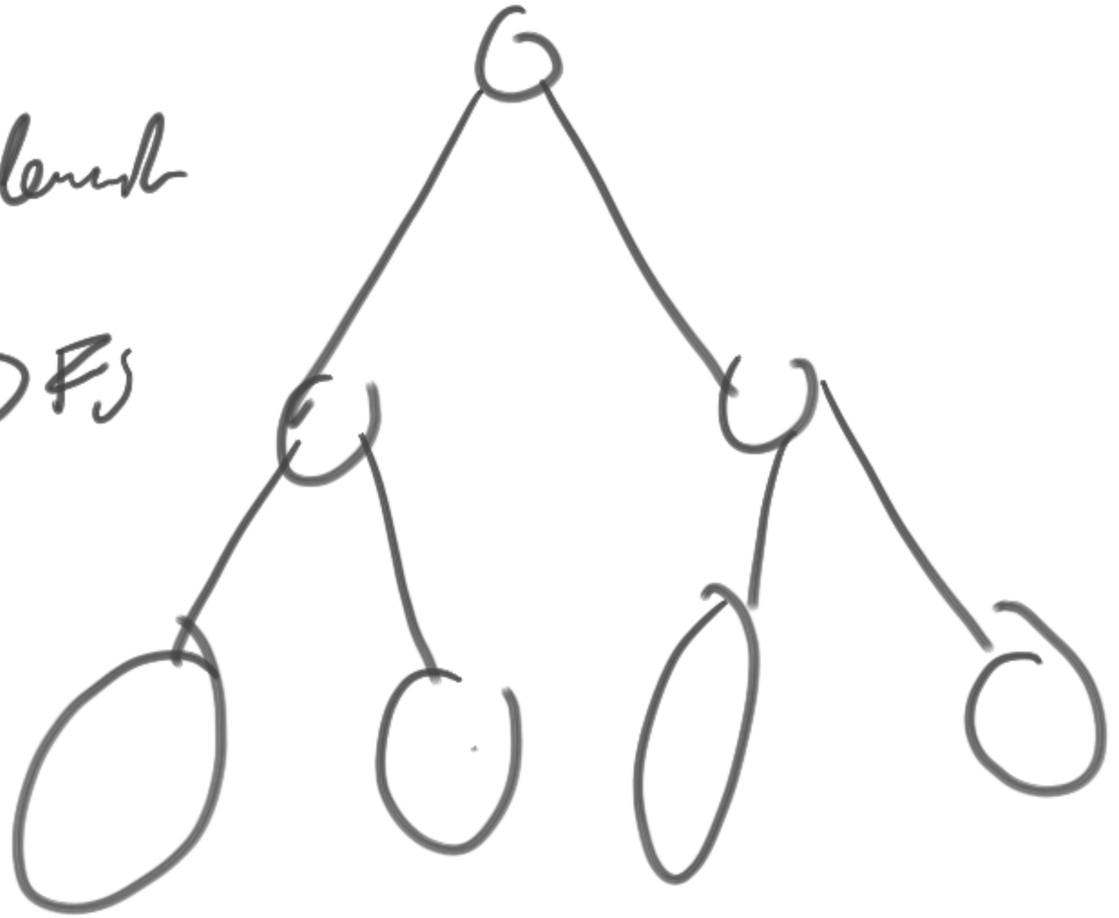
BFS

$O(n(n+m))$

BAD

Beauty of trees:

- recursive
- easy to implement
- fast BFS/DFS



Write tips on BlackBoard!

```
tree = [
    [1, 2],
    [],
    [3, 4, 5],
    [],
    [],
    []
]

sum_of_distances = [0 for _ in range(6)]
subtree_size = [1 for _ in range(6)]

def bfs(tree, node):
    current_sum = 0
    seen = [False for _ in range(len(tree))]
    queue = [node]

    bfs_depth = 1
    while len(queue) > 0:
        for _ in range(len(queue)):
            node = queue[0]
            del queue[0]

            seen[node] = True
            for child in tree[node]:
                if (not seen[child]):
                    current_sum += bfs_depth
                    queue.append(child)
        bfs_depth += 1
    return current_sumncat([res[0]])) : acc
  }
  return go(f, seed, [])
}
```

```python
def subtree(tree, node):
    seen = [False for _ in range(len(tree))]

    dfs(tree, node, seen)

def dfs(tree, node, seen):
    seen[node] = True
    for n in tree[node]:
        if (not seen[n]):
            dfs(tree, n, seen)
            subtree_size[node] += subtree_size[n]

def bfs_correct_for_all(tree, node):
    seen = [False for _ in range(len(tree))]
    queue = [node]

    while len(queue) > 0:
        for _ in range(len(queue)):
            node = queue[0]
            del queue[0]

            seen[node] = True
            for child in tree[node]:
                if (not seen[child]):
                    sum_of_distances[child] = sum_of_distances[node] - subtree_size[child]
+ (len(tree) - subtree_size[child])
                    queue.append(child)


sum_of_distances[0] = bfs(tree, 0)
subtree(tree, 0)
bfs_correct_for_all(tree, 0)

print(sum_of_distances)
```