# Binary Lifting

Binary lifting is a pretty beautiful technique, which has good design stratigies and wide applications. Perhaps mostly popular, it is used to find LCS(**lowest common ancestor**) on trees, but it can do more. I think it can help **query paths** on trees.

By DFS, we could maintain the following information:

(1) The level of each node: $L[u]$

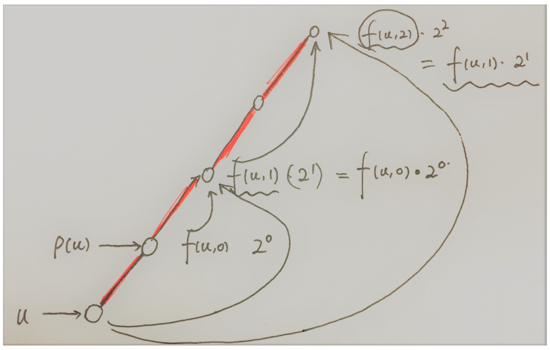
$$L[u] = \begin{cases} 0, & \text{if } u = root \\ \\ L[p(u)] + 1, & p(u) \text{ is the parent node of } u. \end{cases}$$

Obviously, we could know the parent of $u$ during the same dfs process.

(2) The $2^i$-th ancestor of $u$: $f[u, i]$. This is the most wonderful part of binary lifting technique. The matrix $f$ has a dimension: $n \times \log n$, so it is both time and memory efficient. We do it by a DP and divide-and-conquer strategy.

$$f[u, i] = \begin{cases} p(u), & \text{if } i = 0 \\ \\ f\left[f[u, i-1], i-1\right], & \text{otherwise} \end{cases}$$

Below is a simple example:




So now it is clear this is true because:

$$2^i = 2^{i-1} + 2^{i-1}$$

Now let's see the code:

```
L[0]=-1;L[1] = 0; f[1,0]=1; // Initialize: root=1 with level 0
dfs(1,0);
void dfs( int root, int level ){
        L[root] = level; int tl = level;
        nodes[ root ].visited = true;

        for( int i = 1; (tl >> i) ; i ++ )
                f[ root ][i] = f[ f[root][i-1] ][ i-1 ]; // Visit after its ancestors have been visited.

        for( int i = 0; i < mst[root].size() ; i ++ ){
                        int next = tree[root][i];
                        if( !nodes[next].visited ){
                                f[ next ][0] = root; // Initialize: direct ancestor
                                dfs( next, level+1 ); // With level+1 compared with its parent
                        }
        }
        return;
}
```

Many things could be done by the $f$ matrix. Let's give some examples:

(1). Longest path along two nodes $x, y$ to their LCA:

```
1  int lca( int x, int y ){
2          if( L[y] > L[x] )
3                  swap(x,y);  // Make sure x is lower
4  
5          int res = 0;
6          for( int i = 19; i >= 0; i -- ) // 19 > log(n), pre-defined
7                  if( L[ f[x][i] ] >= L[y] ){ // Still lower than y
8                          res = max( res,dis[x][i] ); // Information along the path
9                          x = f[x][i]; // Notice in the next iteration, i->i-1
10                         // No break here!
11                 }
12 
13         if( x == y ) return res;
14 
15         for( int i = 19; i >= 0; i -- ){
16                 if( f[x][i] != f[y][i] ){
17                         res = max( res,max( dis[x][i],dis[y][i] ) );
18                         x = f[x][i];
19                         y = f[y][i];
20                 }
21         }
22 
23         return max( res, max( dis[x][0],dis[y][0] ) );
24 }
```

Some explainations:

- $res$ matrix, this can be done by the same way of $f$:

$$res[u, i] = \begin{cases} w(u, p(u)), & \text{if } i = 0 \\ \\ \max(res[u, i-1], res[f[u, i-1], i-1]), & \text{otherwise} \end{cases}$$

- row 6-11: this is the key. To make it clear,
  - $L[f[x, 19]]$ is surely less than $L[y]$.
  - The very first $i$ to make $L[f[x, i]] \geq L[y]$ has two meanings: (1). this node is below than $y$; (2). $f[f[x, i], i]$ is upper than $y$. But we don't know $f[f[x, i], i-1]$. This is what the codes done next, to check $L[f[x, i], i-1]$.
  - Finally, when $i = 0$, we'll get a node having the same level of $y$, cause $i = 1$ check the nodes two levels above, $i = 0$ check the nodes above.

- The remain things are simple. Row 15-21, $x, y$ always have the same level.

(2). Longest edge along the path to root. Just let $y = 1$,...

(3). Almost everything could be done for answering queries about the dfs path on trees.

## Some problems

- 609E. Minimum spanning tree for each edge
- 733F. Drivers dissatisfaction
- 739B. Alyona and a tree

## Tutorial of 739B

There's two key parts of this problem:

- Binary lifting to document and query the path
- Partial sum on trees for answering questions

$$dist[u, v] = depth[v] - depth[u]$$

During binary lifting or binary search, a most important care of the bound defintion, for example *lower_bound* or *upper_bound*.In this problem, *upper_bound* may be appropriate. We document the first ancestor which can not control one node.

Then for one node, the answer can be right:

$$ans[r] = \sum_{child_i} (ans[child_i] + 1) - no[r]$$

where $no[r]$ documents the number of nodes ending on $r$.

**Reference:** **`http://codeforces.com/blog/entry/22325`**