Advanced Algorithms for Programming Contests
Lecture 10. Lowest common ancestor

Bakhodir Ashirmatov,
Thilo Stier,
Philip Schär

University of Göttingen
Institute of Computer Science

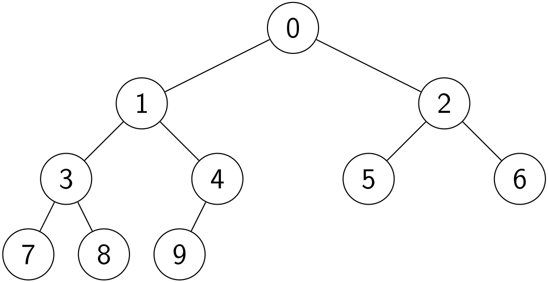
26.06.2019

Overview

Lowest common ancestor
LCA binary lifting
LCA interval tree
Union find
LCA union find (Tarjan)

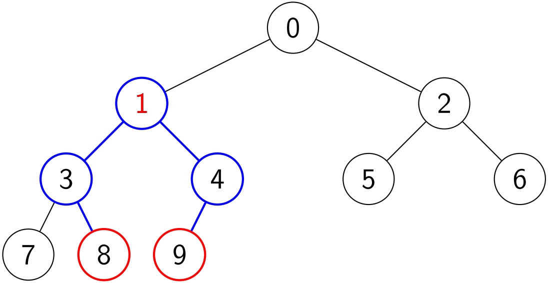
Lowest common ancestor

- Lowest common ancestor of two nodes v and w in a (rooted) tree is the node furthest away from the root having both v and w as a decendent.



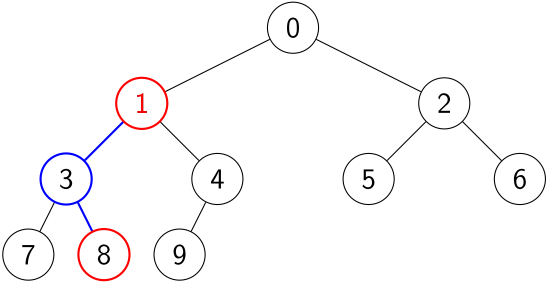
Lowest common ancestor

- Lowest common ancestor of two nodes v and w in a (rooted) tree is the node furthest away from the root having both v and w as a decendent.



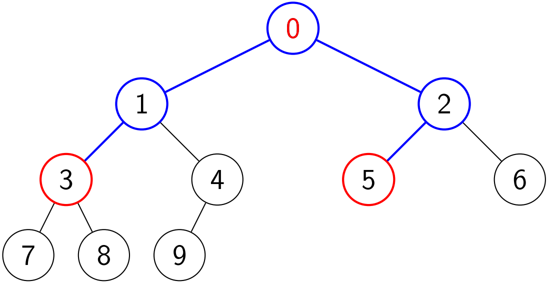
Lowest common ancestor

- Lowest common ancestor of two nodes v and w in a (rooted) tree is the node furthest away from the root having both v and w as a decendent.



Lowest common ancestor

- Lowest common ancestor of two nodes v and w in a (rooted) tree is the node furthest away from the root having both v and w as a decendent.



LCA binary lifting

Problem

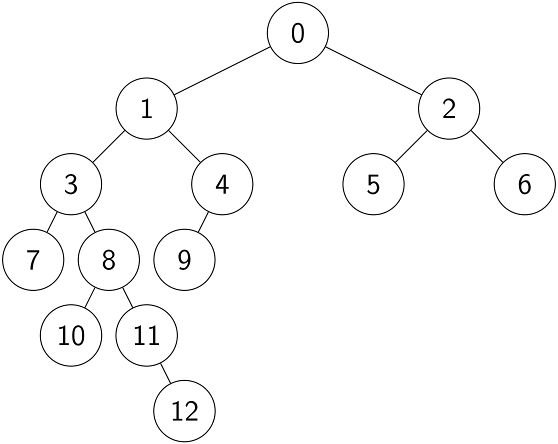
- Given a rooted tree, find LCAs of pairs of nodes.

Naive algorithm

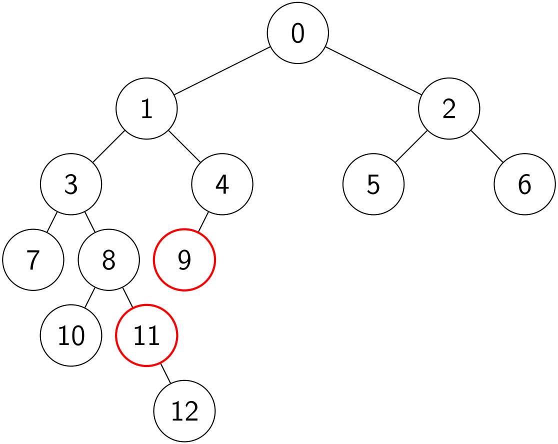
- Given v, w to find LCA.
- Check if v is ancestor of w
  - Yes $\rightarrow$ return v.
  - No $\rightarrow$ find LCA of ancestor of v and w.

Complexity: $O(N)$

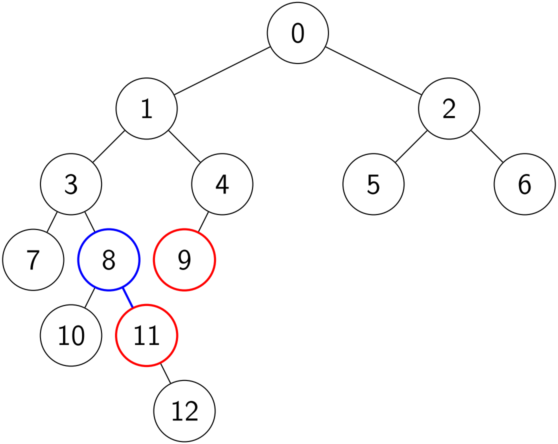
LCA binary lifting



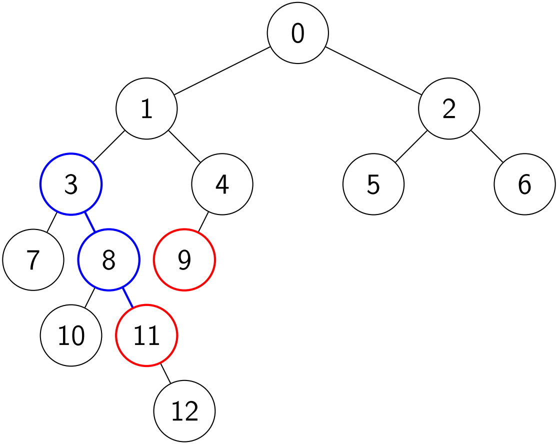
LCA binary lifting



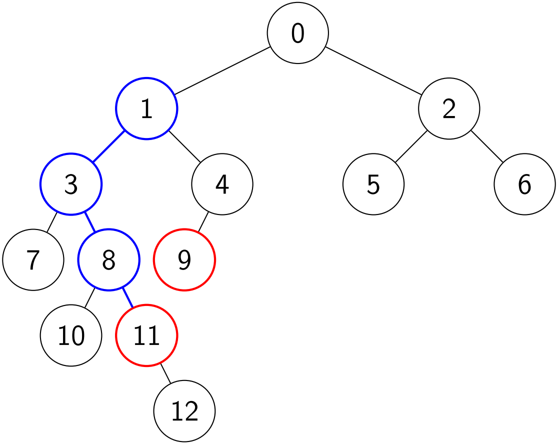
LCA binary lifting



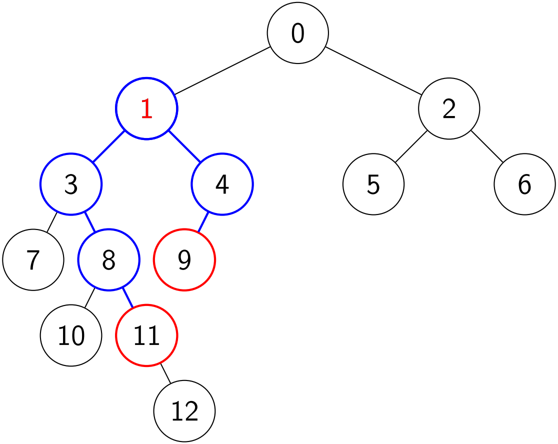
LCA binary lifting



LCA binary lifting



LCA binary lifting



LCA binary lifting

Idea

- Precalculate 2^k-th ancestor for any node.
- For every query find LCA by going up these ancestor links.

Algorithm

- $up[l][i] = (2^l)$-th ancestor of i (or root if it isn't that deep)
- Recursive: $up[l][i] = up[l-1][up[l-1][v]]$.
- Find $lca(a, b)$:
  - If a is ancestor of b or b of a $\Rightarrow$ trivial.
  - Find $\max\{l \mid up[l][a] \text{ not anc. of } b\}$.
  - Find $lca(up[l][a], b)$ recursively.

Complexity

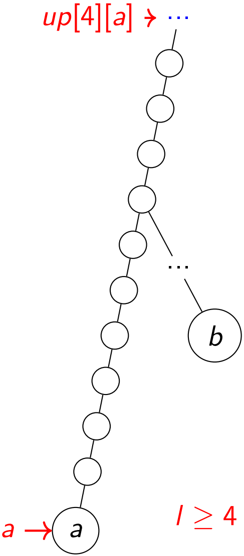
- Precalc, Memory $O(N \log(N))$
- Query $O(\log(N))$

LCA binary lifting

```
const int MAXN, MAXLOG; // 1 << (MAXLOG-1) >= MAXN
int n, root;
vector<int> edges[MAXN];
int tin[MAXN], tout[MAXN], timer = 1;
int up[MAXLOG][MAXN];

void dfs(int v = root, int p = root) {
  tin[v] = timer++;
  up[0][v] = p;
  for (int l = 1; l < MAXLOG; l++)
    up[l][v] = up[l-1][up[l-1][v]];
  for (int w : edges[v]) {
    if (w != p)
      dfs(w, v);
  }
  tout[v] = timer++;
}
```

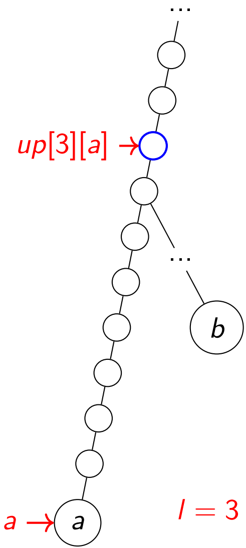
LCA binary lifting

```
bool upper(int a, int b) {
  return tin[a] <= tin[b] && tout[a] >= tout[b];
}

int lca(int a, int b) {
  if (upper(a, b)) return a;
  if (upper(b, a)) return b;
  for (int l = MAXLOG - 1; l >= 0; l--)
    if (!upper(up[l][a], b))
      a = up[l][a];
  return up[0][a];
}
```



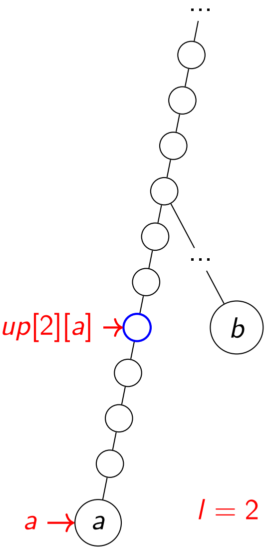
LCA binary lifting

```
bool upper(int a, int b) {
  return tin[a] <= tin[b] && tout[a] >= tout[b];
}

int lca(int a, int b) {
  if (upper(a, b)) return a;
  if (upper(b, a)) return b;
  for (int l = MAXLOG - 1; l >= 0; l--)
    if (!upper(up[l][a], b))
      a = up[l][a];
  return up[0][a];
}
```



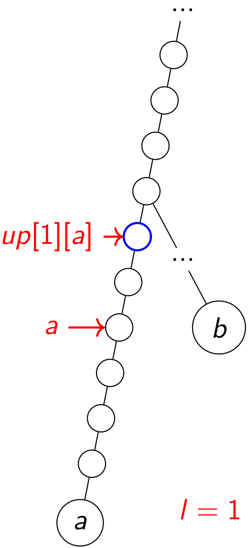
LCA binary lifting

```
bool upper(int a, int b) {
  return tin[a] <= tin[b] && tout[a] >= tout[b];
}

int lca(int a, int b) {
  if (upper(a, b)) return a;
  if (upper(b, a)) return b;
  for (int l = MAXLOG - 1; l >= 0; l--)
    if (!upper(up[l][a], b))
      a = up[l][a];
  return up[0][a];
}
```



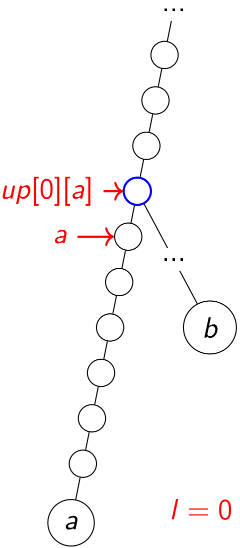
LCA binary lifting

```
bool upper(int a, int b) {
  return tin[a] <= tin[b] && tout[a] >= tout[b];
}

int lca(int a, int b) {
  if (upper(a, b)) return a;
  if (upper(b, a)) return b;
  for (int l = MAXLOG - 1; l >= 0; l--)
    if (!upper(up[l][a], b))
      a = up[l][a];
  return up[0][a];
}
```



LCA binary lifting

```
bool upper(int a, int b) {
  return tin[a] <= tin[b] && tout[a] >= tout[b];
}

int lca(int a, int b) {
  if (upper(a, b)) return a;
  if (upper(b, a)) return b;
  for (int l = MAXLOG - 1; l >= 0; l--)
    if (!upper(up[l][a], b))
      a = up[l][a];
  return up[0][a];
}
```



LCA interval tree

Problem

- Calculate LCA efficiently in linear memory.

Idea

- The $lca(a, b)$ is the vertex closest to root of those visited by dfs between first entering a and first entering b.
- Use interval tree to find it in $O(\log(N))$.

Algorithm

- Run dfs for visiting order and *depth* calculation.
- Build interval tree for $\text{minarg}_v\, depth(v)$ on visit vector.
- Answer queries $lca(a, b) = \text{minarg}([firstVisit[a], firstVisit[b]])$.

LCA interval tree

```
int n, root;
vector<int> edges[MAXN];

vector<int> visit;
int firstVisit[MAXN];
int depth[MAXN];

void lca_dfs(int v = root, int d = 1) {
  firstVisit[v] = visit.size();
  visit.push_back(v);
  depth[v] = d;
  for (int w : edges[v]) {
    if (depth[w] != 0) continue;
    lca_dfs(w, d+1);
    visit.push_back(v);
  }
}
```

LCA interval tree

```
int tree[8*MAXN];
void lca_build_tree(int v = 1, int tl = 0, int tr = visit.size()-1) {
  if (tl == tr)
    tree[v] = visit[tl];
  else {
    int tm = (tl + tr) / 2;
    lca_build_tree(2*v, tl, tm);
    lca_build_tree(2*v+1, tm+1, tr);
    if (depth[tree[2*v]] < depth[tree[2*v+1]])
      tree[v] = tree[2*v];
    else
      tree[v] = tree[2*v+1];
  }
}

void lca_prepare() {
  lca_dfs();
  lca_build_tree();
}
```

LCA interval tree

```
int lca_get_tree(int l, int r, int v=1, int tl=0, int tr=visit.size()-1) {
  if (l == tl && r == tr)
    return tree[v];
  int tm = (tl + tr) / 2;
  if (r <= tm)
    return lca_get_tree(l, r, 2*v, tl, tm);
  if (l > tm)
    return lca_get_tree(l, r, 2*v+1, tm+1, tr);
  int lmin = lca_get_tree(l, tm, 2*v, tl, tm);
  int rmin = lca_get_tree(tm+1, r, 2*v+1, tm+1, tr);
  return depth[lmin] < depth[rmin] ? lmin : rmin;
}

int lca(int a, int b) {
  int l = min(firstVisit[a], firstVisit[b]);
  int r = max(firstVisit[a], firstVisit[b]);
  return lca_get_tree(l, r);
}
```

Union find

Problem

- Starting with n different elements in n sets $\{i\}$.
- Perform operations:
  - $union(a, b)$ - merge the sets containing a and b.
  - $find(a)$ - return a unique representative of the set containing a.

Idea

- Keep a tree for every set.
- Implement $find(a)$ as the root of tree containing a.
- Implement $union(a, b)$ as setting the parent of $find(a)$ to $find(b)$ if they are not the same.

Naive approach

```
void make_set(int v) {
  parent[v] = v;
}

int find_set(int v) {
  if (v == parent[v])
    return v;
  return find_set(parent[v]);
}

void union_sets(int a, int b) {
  a = find_set(a);
  b = find_set(b);
  if (a != b)
    parent[b] = a;
}
```

Path compression

Idea

- Use lazy dynamics on *find* to optimize.

```
int find_set(int v) {
  if (v == parent[v])
    return v;
  return parent[v] = find_set(parent[v]);
}
```

Complexity on average $O(\log(n))$

Union by rank

Idea

- Consider *rank* of trees to always keep them balanced.

```
void make_set(int v) {
  parent[v] = v;
  rank[v] = 0;
}

void union_sets(int a, int b) {
  a = find_set(a);
  b = find_set(b);
  if (a != b) {
    if (rank[a] < rank[b])
      swap(a, b);
    parent[b] = a;
    if (rank[a] == rank[b])
      ++rank[a];
  }
}
```

Complexity on average $O(\log(n))$

Final realisation

```
void make_set(int v) {
  parent[v] = v; rank[v] = 0;
}

int find_set(int v) {
  return (v == parent[v]) ? v : parent[v] = find_set (parent[v]);
}

void union_sets(int a, int b) {
  a = find_set(a); b = find_set(b);
  if (a != b) {
    if (rank[a] < rank[b])
      swap(a, b);
    parent[b] = a;
    if (rank[a] == rank[b])
      ++rank[a];
  }
}
```

Complexity on average $O(\log^*(n))$

LCA union find (Tarjan)

Problem

- Find LCAs of a given set of pairs of vertices.

Idea

- The LCA of two vertices a, b is the lowest vertex, that has a and b in different subtrees.
- Use dfs, merge completed subtrees with their direct ancestor using union-find.
- Remember highest vertex of merged subtree.
- LCA of current vertex v and vertex w visited before is ancestor of completed subtree containing w.

LCA union find (Tarjan)

```cpp
const int MAXN = 100000;
vector<int> edges[MAXN], requests[MAXN];
int dsu[MAXN], ancestor[MAXN];
bool used[MAXN];

int find_set(int v) {
  return (v == dsu[v]) ? v : (dsu[v] = find_set(dsu[v]));
}

void union_sets(int a, int b, int new_ancestor) {
  a = find_set(a), b = find_set(b);
  if (rand() & 1)
    swap(a, b);
  dsu[a] = b;
  ancestor[b] = new_ancestor;
}
```

LCA union find (Tarjan)

```
void dfs (int v) {
  dsu[v] = v;
  ancestor[v] = v;
  used[v] = true;
  for (int w : edges[v])
    if (!used[w]) {
      dfs(w);
      union_sets(v, w, v);
    }
  for (int w : requests[v])
    if (used[w])
      printf("%d %d -> %d\n", v, w, ancestor[find_set(w)]);
}
```

Do your homework!