```
                minLen[table[v][i - 1]][i - 1]);
      //maxLen[v][i] = max(maxLen[v][i - 1],
                maxLen[table[v][i - 1]][i - 1]);
    }
  }
  for (auto &child : g[v])
  {
    if (child == par) continue;
    dfs(child, v, dep + 1);
    //dfs(child.first, v, dep + 1, child.second,
            child.second); //for max & min
  }
  tout[v] = ++Time;
}
void lca_build() //=> O(n*logn)
{
  dfs(1);
}
int lca_query(int a, int b) //=> O(logn)
{
  if (level[a] < level[b]) swap(a, b);
  // int dis = level[a] - level[b];
  // while (dis)  //a and b come to the same level
  // {
  //    int i = log2(dis);
  //    a = table[a][i], dis -= (1 <<i);
  // }
  for (int i = lg; i>= 0; i--)
  //a and b come to the same level
  {
    if (table[a][i] != -1 && level[table[a][i]] >=
level[b])
      a = table[a][i];
  }
  if (a == b) return a;
  for (int i = lg; i>= 0; i--)
  {
    if (table[a][i] != -1 && table[a][i] != table[b][i])
      a = table[a][i], b = table[b][i];
  }
  return table[a][0];
}
int dist(int u, int v)
// distance between two node
{
  int l = lca_query(u, v);
  return level[u] + level[v] - (level[l] << 1);
//level[l]*2
}
Int kth(int u, int k)
{
  for (int i = 0; i<= lg; i++)
      if (k & (1 <<i)) u = table[u][i];
  return u;
}
int findKth(int u, int v, int k)
// kth node from u to v, 0th node is u
{
  int l = lca_query(u, v);
  int d = level[u] + level[v] - (level[l] << 1);
  if (level[l] + k <= level[u]) return kth(u, k);
  k -= level[u] - level[l];
  return kth(v, level[v] - level[l] - k);
}
bool is_ancestor(int u, int v) //u is an ancestor of v
{
  return tin[u] <= tin[v] && tout[u] >= tout[v];
}

void reset() {
  for (int i = 0; i<= n; i++) {
    g[i].clear();
    level[i] = 0;
    for (int j = 0; j <= lg; j++) table[i][j] = -1;
  }
}
int main() {
  cin>> n;
  lg = log2(n) + 1;
  reset();
  // Input .... Query ...
} // endl
```

Trie: //=> O(length)

```
struct Trie {
  static const int rangeSize = 26; // for lower_case
letter ('a' <= 'z')

  struct node {
    node *next[rangeSize];
    bool completedWord;
    int cnt;
    node() {
      completedWord = false;
      cnt = 0;
      for (int i = 0; i < rangeSize; i++)
        next[i] = nullptr;
    }
  } *root;

  Trie() {
    root = new node();
  }

  void trieInsert(const string &s) {
    node *cur = root;
    for (char ch : s) {
      int x = ch - 'a'; // for lowercase letter
      if (cur->next[x] == nullptr) {
        cur->next[x] = new node();
```