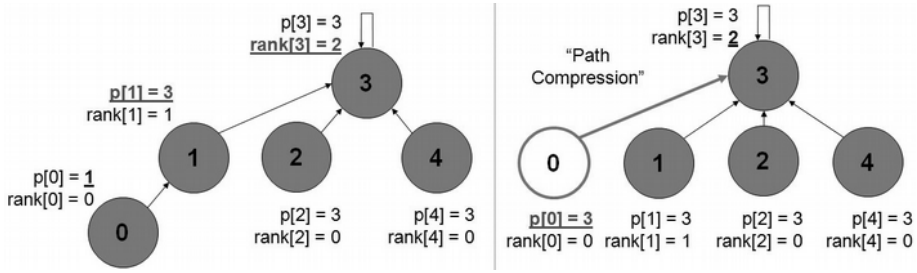
// Union Find Disjoint Set

```
vector<int>u_list, u_set;                    // u_list[x] contains the size of a set x   u_set[x] contains the root of x

int unionRoot(int n) {                       // Finds the root of a point/element
   if(u_set[n] == n)                         // If u_set[n] == n, then n is the root of set
      return n;
   else                                      // Else keep searching for root
      return u_set[n] = unionRoot(u_set[n]);
}

int makeUnion(int a, int b) {                // Takes two points in the same set, and return the root
   int x = unionRoot(a);                     // Find the root of x and y
   int y = unionRoot(b);

   if(x == y)                                // If root of both points are same, then nothing to do
      return x;
   else if(u_list[x] > u_list[y]) {          // If the size of x set is larger than set y (Path Compression)
      u_set[y] = x;                          // Make set y the subset of x
      u_list[x] += u_list[y];                // Increase the size of set x
      return x;
   }
   else {
      u_set[x] = y;
      u_list[y] += u_list[x];
      return y;
   } }
```




```
void unionInit(int len) {                    // Union Disjoint Set Initialization
   u_list.resize(len+5, 1);                  // Space allocation
   u_set.resize(len+5);
   for(int i = 0; i <= len; i++)             // At first, root of all points is the point itself
      u_set[i] = i;
}

bool isSameSet(int a, int b) {               // Check if two points are in same set
   if(unionRoot(a) == unionRoot(b))          // If the root of both point are same, then they are same
      return 1;
   return 0;
}
```

// Segment Tree

```
int arr[N], tree[4*N];                       // Always take the tree size 4x
                                             // arr[] contains values starting from index 1
void segment_build(int pos, int L, int R) {  // Builts the segment tree call : segment_build(1, 1, len_of_arr)
   tree[pos] = 0;                            // Initialization of tree position
   if(L==R) {                                // Mid point reached
      tree[pos] = arr[L];
```

```
      return;  }
   int mid = (L+R)/2;
   segment_build(pos*2, L, mid);
   segment_build(pos*2+1, mid+1, R);
   tree[pos] = tree[pos*2] + tree[pos*2+1];                    // Depends on usage (This is the main point to tweak)
}

void segment_update(int pos, int L, int R, int i, int val) {    // Val contains the value to update SINGLE UPDATE
   if(L==R) {                                                   // Call: segment_update(1, 1, len_of_arr, pos_in_arr new_value)
      tree[pos] = val;                                          // If L==R then this is the midpoint that contains the single value
      return;  }
   int mid = (L+R)/2;
   if(i <= mid)  segment_update(pos*2, L, mid, i, val);         // Go for update_position
   else  segment_update(pos*2+1, mid+1, R, i, val);
   tree[pos] = tree[pos*2] + tree[pos*2+1];                     // Depends on usage (here summation)
}

// Query in range l-r
int segment_query(int pos, int L, int R, int l, int r) {        // Finds value in l-r segment of arr[]
   if(R < l || r < L)                                           // Out of range l-r
      return 0;
   if(l <= L && R <= r)                                         // In range l-r
      return tree[pos];
   int mid = (L+R)/2;
   int x = segment_query(pos*2, L, mid, l, r);
   int y = segment_query(pos*2+1, mid+1, R, l, r);
   return x+y;                                                  // Return the total value
}

// Toggle bit in range of [l, r]                  call : update(1, 1, length_of_input, l, r)      EX:  Update lights in range l-r
void update(int pos, int L, int R, int l, int r) {              // Lazy without propagation
   if(l <= L && R <= r) {                                       // If segment is in range l-r
      arr[pos] ^= 1;                                            // Tweak according to problem
      return; }
   if(r < L || R < l)                                           // Out of range l-r
      return;
   int mid = (L+R)/2;
   update(pos*2, L, mid, l, r);
   update((pos*2) + 1, mid+1, R, l, r);
}

// Single position query
bool query(int at, int L, int R, int pos) {                     // Query in arr position 'pos'
   if(pos < L || R < pos)    return 0;                          // Range is out of pos
   if(L <= pos && pos <= R)    return arr[at];                  // pos is in range
   int mid = (L+R)/2;
   if(pos < = mid)
      return query(at*2, L, mid, pos) ^ arr[at];
   else
      return query(at*2+1, mid+1, R, pos) ^ arr[at];
}
```