But as the graph is connected, there must be some path from v to v' (marked in green).
By step 1 of algorithm, we include all the edges in the graph. So we must have included the edges between v to v'.
This contradicts our claim and we can conclude that T has to be connected (there exists a path from v to v' in T.)

So T is connected (and contains all vertices of G).
So T is a spanning tree of G.

2. T has the maximum weight among all spanning trees.

Subclaim : at any time of execution of the algorithm, T is entirely contained in some maximum spanning tree.

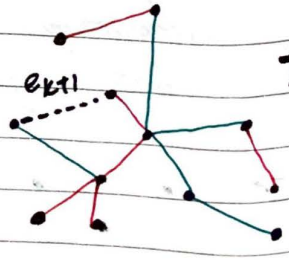
We prove this by induction on the number of times (k) the for loop in step 3 is executed.

When $k = 0$
$T = \phi$ i.e., T =

So the subclaim is correct (as there are no edges).

Suppose the claim is true for $k \geq 0$.
Let us consider the case for $k+1$.

— : in tree T
— : in $T_{max} - T$

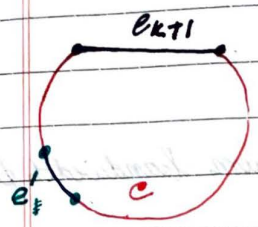


If e_{k+1} is not added to T, then subclaim holds true.

If e_{k+1} is added to T,

→ if e_{k+1} is in T_{max}, subclaim holds true.
→ if e_{k+1} is not in T_{max},

adding e_{k+1} to the tree makes a cycle C with T_{max}.

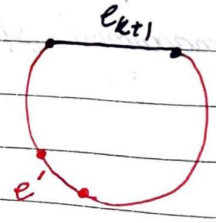


In this cycle, there must be some green edges. Otherwise, all the vertices are connected and e_{k+1} would not be added (that leads to a cycle).

We have arranged edges in decreasing order.

$$wt(e_{k+1}) > wt(e')$$

So adding e_{k+1} to T gives rise to a spanning tree which is larger than maximum spanning tree. → a contradiction



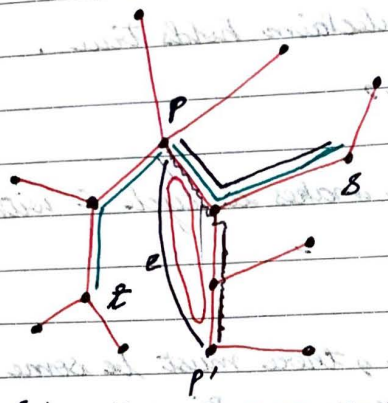
Another possibility is $wt(e_{k+1}) = wt(e')$
So we can trade e_{k+1} for e'. (as they have same weights).
We get another maximum spanning tree (different edges, same weight).

→ This algo gives a maximum spanning tree (which might not be fixed).

Once we construct a maximum spanning tree, same for MBP for any two vertices.

PROVE:— Let T be a maximum spanning tree of G. For any two vertices u and v, the path in T connecting u and v is a maximum bandwidth path.

T:



— : Spanning tree
— : maximum bandwidth path
— : maximum bandwidth path (claimed)

We claim that the green path is the maximum bandwidth path.

If we add e in the MBP, e+T makes a cycle C.
We claim that all edges in C have bandwidth not smaller than e.

Let us say we take path p-p' which does not follow maximum bandwidth path {p-p': black path}

Since green path is maximum bandwidth spanning tree, black path p-p' has lower bandwidth than maximum spanning tree.
e+T makes a cycle C

We can claim that all edges in C have bandwidth not smaller than e.
So, e can be replaced by some tree edge such that the bandwidth is higher (black dotted path).

This is true for any edge not in maximum spanning tree.
(because edges not in previous MBP just decreases the bandwidth).

NOTE:— [The first two largest edges must always be in a maximum spanning tree] → PROVE : these edges can never form a cycle.

Page :
Date : 01 / 10 / 2019

- **Kruskal Algorithm** :→ (undirected, weighted graph)

1. Sort the edges in non increasing order:
   $e_1, e_2, \ldots, e_m$;
2. $T = \phi$ // or all vertices of G (no edges)
3. for $(i=1; i \leq m, i++)$
   $e_i = [u_i, v_i]$
   if $(T + e_i$ does not contain a cycle)
   $T = T + e_i$;
4. output (T)

1. Kruskal constructs a maximum spanning tree T.
2. For any s and t in G, the s-t path in T is a maximum bandwidth path.

Time complexity analysis :—

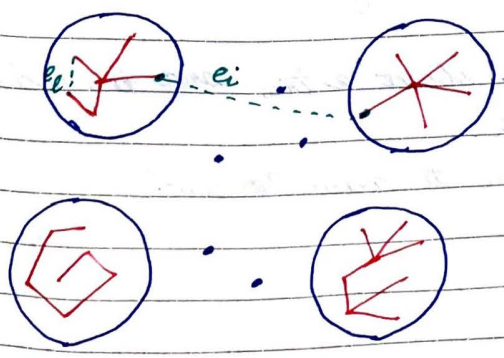
1. Sorting can be done in $O(m \log m)$ time using heap sort.

$$m \leq \binom{n}{2} \leq n^2.$$

$$\therefore \log m \leq \log n^2$$

$$\therefore \log m < 2 \log n.$$

$\therefore$ Sorting complexity $= O(m \log n)$

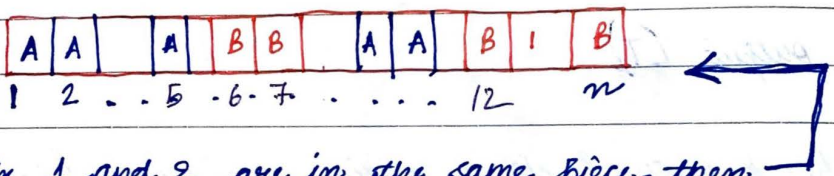


If we add an edge like e_i, (between different pieces), it can never form a cycle.

However, an edge like e_l (between same pieces) may form a cycle.

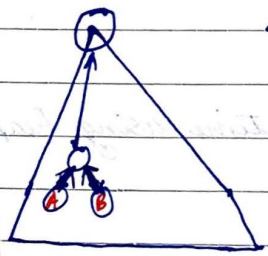
How to check if e_i forms a cycle with T?
If (u_i and v_i) belong to different pieces of T (there is no cycle)
merge the 2 pieces into one.
⇓
simplified operation to make cycle detection easy.

1. We want to put all vertices in a piece of T in the same place.
2. We should be able to merge vertices in two pieces.

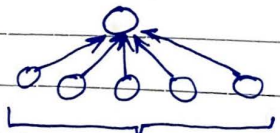


If vertex 1 and 2 are in the same piece, then
- vertex 1, 2, 5 → belong to disjoint set A.
- vertex 6, 7, 12 → belong to disjoint set B

interpretable but merging two pieces is computationally expensive.



⇓ least.



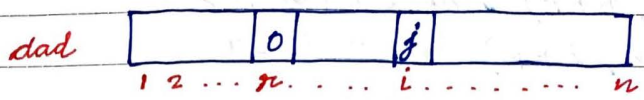
nodes.

So instead we use a tree to ~~store~~ model this.
A tree with reversed arrows pointing from leaves to children.

Vertices belonging to same piece point to same root.

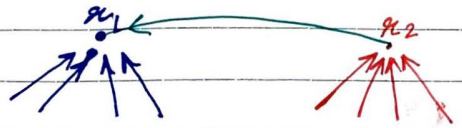
For each piece, we construct a tree:
The nodes pointing to same root are in same

For each piece, we construct a tree and the nodes of the t
are vertices of graph.
This causes the tree to grow too much.



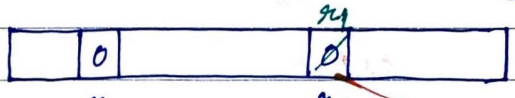
j is the parent of i.

If r is the root, $dad[r] = 0$.



Now to add an edge between the two pieces, we simply have to make r_1 the parent of r_2.

The modified dad array becomes :-



'0' updated to r_1 to indicate the parent of r_2.

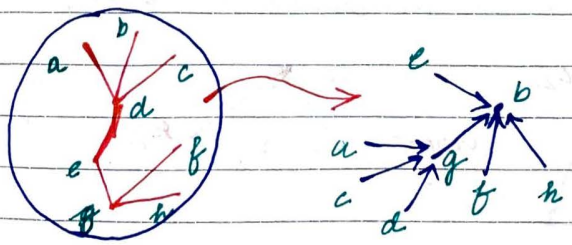
(BY RANK)

O(logn) time ← **Find (v)** // find the root of the tree containing v

```
1.  w = v;
2.  while ( dad[v] != 0)
        w = dad[v];
```

O(1) time ← **Union (r_1, r_2)** // merge the trees rooted at r_1 and r_2

$dad[r_2] = r_1$;



O(1) time ← **MakeSet (v)** // make a single node tree for v.

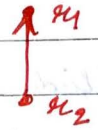
```
dad[v] = 0;
rank[v] = 0;
```

So modified Kruskal becomes :—

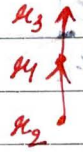
1. Sorting → $m \log n$
2. for ($v = 1$, $v \leq n$, $v++$) Make Set (v) . → n
3. for ($i = 1$, $i \leq m$, $i++$) → m times
   - $e_i = [u_i, v_i]$
   - $x_1 = find(u_i)$, $x_2 = find(v_i)$ → $\log n$
   - if ($x_1 != x_2$)
     - union (x_1, x_2).

   → $m \log n$
4. output (T) .

Union (x_1, x_2)

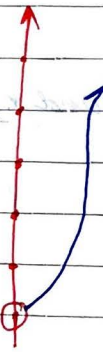


Union (x_3, x_4)



Union (x_4, x_3)

If we continue doing these operations, we end up with a tree like



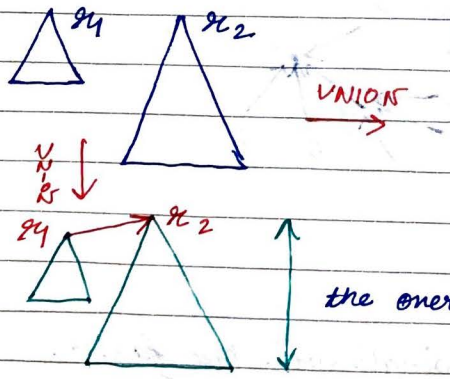
If we wish to compute the dad of the circled vertex, it becomes computationally expensive.

This is not a good representation.

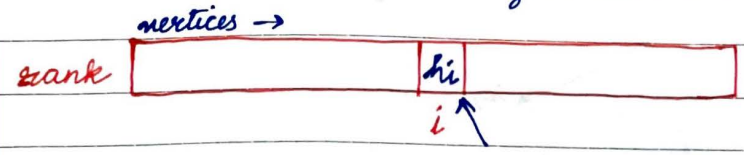
So target is not to increase height of the tree.

TARGET:— attach the smaller tree to the larger one.

Union (x_1, x_2)



So now we need to record height of the tree as well, for which we maintain another array rank.

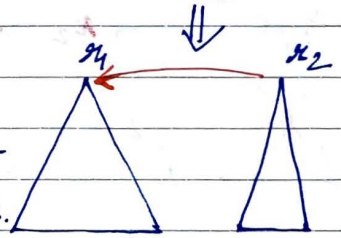


is the height of the tree rooted at i (if i is a root).

So modified union (x_1, x_2) becomes.

Union (x_1, x_2)

if (rank $[x_1]$ > rank $[x_2]$)
    dad $[x_2] = x_1$;
else if (rank $[x_1]$ < rank $[x_2]$)
    dad $[x_1] = x_2$;
else dad $[x_2] = x_1$; ----> case when both the trees have
    rank $[x_1]$ = rank $[x_1]+1$; same height



we attach x_2 to x_1. So update rank of x_1. As x_2 is not a root, we donot update its rank.

Whenever a node is not a root, it can never become root again.

So height of the new tree can not be very much, Find(v) would take less tim

Claim :- Now the height of a tree is $\leq \log_2 n$.

PROOF:- [we prove this by induction]
So we have to prove:—
A tree of l nodes has height $\leq \log_2 l$.

Using induction on l, we get:-

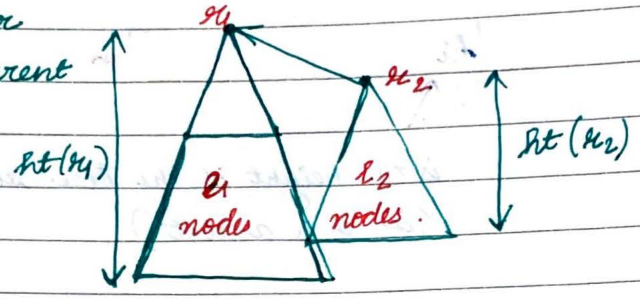
$l = 1$ • [single node]
height = 0,
$0 = \log_2 1 \Rightarrow$ correct

whenever my # vertices > 1, that tree is made of union function

Now consider $l > 1$,

Suppose that the tree is made by Union (r_1, r_2).

Suppose we consider two trees of different heights.



Suppose $ht(r_1) > ht(r_2) \Rightarrow$ so merging two trees cannot increase height here.

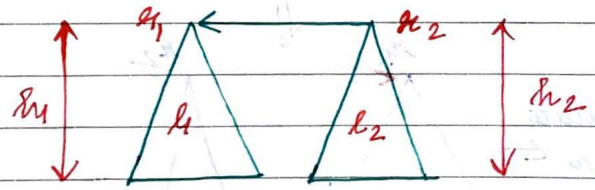
So height of tree rooted at r_1, $ht(r_1) \leq \log l_1$

$l_1 \leq l \Rightarrow \log_2 l_1 \leq \log_2 l$

we have $(l_1 + l_2)$ nodes in new tree.

ht (tree with $l_1 + l_2$ nodes) $\leq \log_2 l_1 \leq \log_2 l$.

Now we consider two trees with same height



$h_1 = h_2$

$h_1 \leq \log_2 l_1$

$h_2 \leq \log_2 l_2$

The resulting tree has height $(h_1 + 1)$

$(h_1 + 1) \leq \log_2 (l_1 + l_2)$ ← we want to prove

$l_1 \leq l_2$

$\log_2 l = \log_2 (l_1 + l_2)$

$\geq \log_2 (2 l_1)$

$= \log_2 (l_1) + 1 \geq ht + 1$

$\geq h_1 + 1$.

So total time complexity of Kruskal : $O(m \log n)$

Page :
Date : 03/ 10 /2019

Makeset (v)

dad [v] = 0 ;
rank [v] = 0 ;

Union (x_1, x_2)

cases

1. (rank [x_1] > rank [x_2])
   dad [x_2] = x_1 ;
2. (rank [x_1] < rank [x_2]).
   dad [x_1] = x_2 ;
3. (rank [x_1] == rank [x_2])
   dad [x_1] = x_2 ; rank [x_2]++;

Find (v) → Takes time at most O(log n) as the height of the tree is bounded by log n.

1. w = v;
2. while (dad[w] != 0)
   w = dad [w];
3. output : return (w)

→ Makeset and Union take constant time.

Kruskal

1. Sort edges: $e_1, e_2, \ldots, e_m$;
2. $T = \phi$; for ($v=1$; $v \leq n$, v++) Makeset(v);
3. for ($i=1$; $i \leq m$, i++) — m times
   $e_i = [u_i, v_i)$;
   x_1 = Find(u_i), x_2 = Find(v_i)
   if ($x_1 != x_2$)
   Union(x_1, x_2)
   $T = T + e_i$
   → $m \log n$
4. return T.

Complexity:— $O(m \log n)$

Can Kruskal Algorithm be implemented in time $< m \log n$?

Ans:— No.

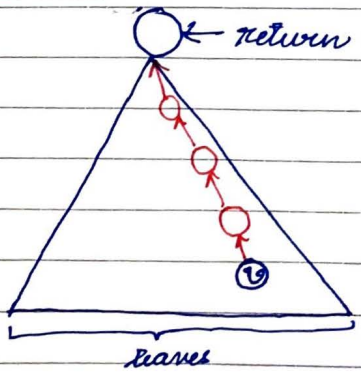
This algorithm involves sorting which can not be implemented in time lower than $m \log n$.

(In practical scenarios, sorting takes much less time than the $O(m \log n)$ time complexity of step 3. So we aim to improve complexity of step # 3).

PROBLEM:—

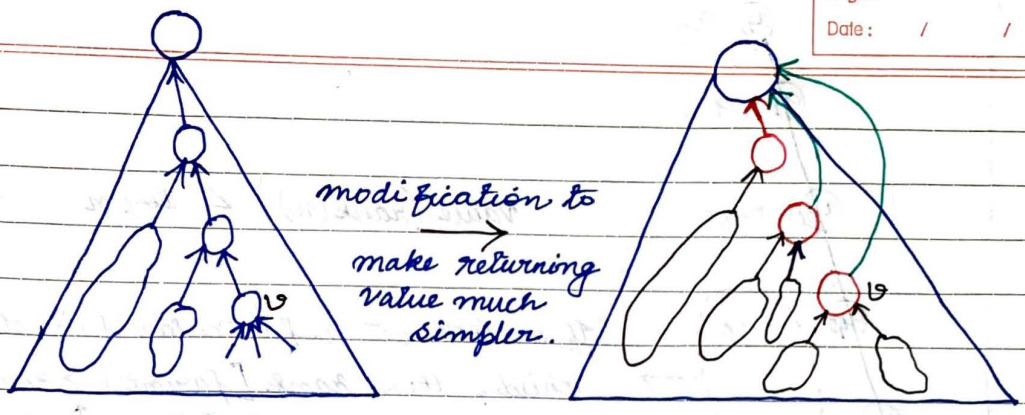
Given m, M-U-F operations on a set $\{a_1, a_2, \ldots, a_n\}$:—
What is the time this sequence of operations takes?

Find (v)



Every node has a parent.
Time to compute parent this operation bounded by $\log n$.

Every node is an element.



This modification makes the tree much smaller and flatter.

BY PATH COMPRESSION ← Find (v) → [This function actually does not improve complexity of this operation. However it has the potential to improve future operations].

```
1. w = v;  S ← φ;
2. while (dad[w] != 0)
       S ← w;  w = dad[w];
3. while (S != φ)
       v ← S
       dad[v] = w;
4. return w;
```

→ Rank now does not define height of tree. It, however bounds the height of the tree.

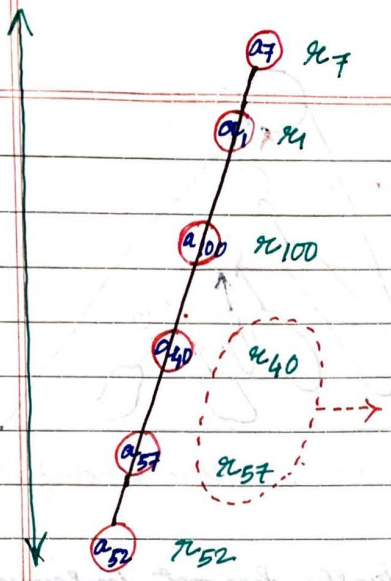
Let us see an example:-

M M U F U F F M F F F U F M F U F

Unions and Makesets take constant time. So we need to find total time taken by all find operations.
For each find, we have different path lengths.

The total time taken by all find operations = $O(\sum_{p:\text{F-path}} \text{length}(p))$

Let us take a sample path to find an element:-



bounded by $\log n$

Value (rank(n)) $\leq \log n$

If we look at a particular pair of parent & child, then. rank [parent] > rank [child]

↑ strictly larger

Compression changes structure but not no. of nodes in tree.

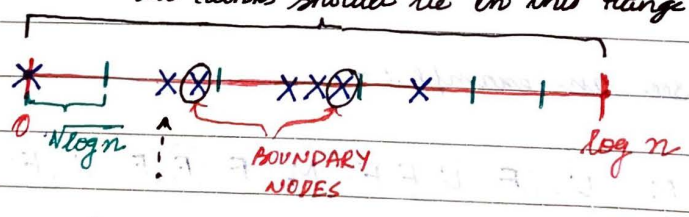
1. FACT 1: rank [parent] > rank [child]

Whenever we introduce a new parent-child relation, this holds.

(We can look into the Union (and Find) operation steps to understand the correctness of fact 1).

2. Rank is bounded by $\log n$.

We draw a line and mark it with integer values from 0 to $\log n$.



Every time we visit a node, we mark its rank here.

We subdivide the entire line into intervals of length $\sqrt{\log n}$.

Boundary Node :- the last node (mark) in each interval.

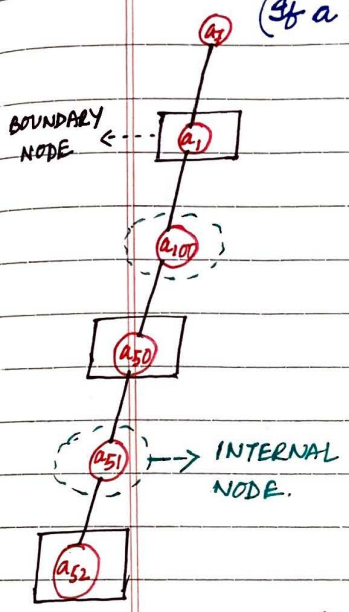
Internal Node: nodes that are not boundary nodes.

Child of the root is always a boundary node.

As there are at most $\sqrt{\log n}$ subintervals, thus each path can have at most $\sqrt{\log n}$ boundary nodes.

∴ The total no. of boundary nodes along all find paths $\leq m \log n$

(If a node is not a root, we do not change its rank).



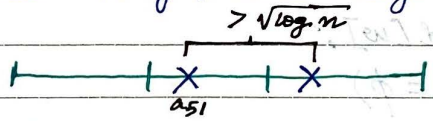
How many times can an internal node appear on all the find paths? → $\sqrt{\log n}$ times.

(After that, the internal node becomes a boundary node & can never be an internal node again).

After each find, each internal node on the path gets a new father.

Rank of the new father is strictly larger than the old father.

Thus if we perform this operation $\sqrt{\log n}$ times, then the internal node (say, a_{51}) gets a new father which is at least ($\sqrt{\log n}$) times larger than the original rank.



The # of internal nodes on all find paths is bounded by $n\sqrt{\log n}$.

total # of b-nodes along all find paths $\leq m\sqrt{\log n}$

total # of i-nodes along all find paths $\leq n\sqrt{\log n}$

Thus, total time taken by all F-operations $= O\left(\sum_{P:\ F.Path} length(P)\right)$

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

Why do we choose $\sqrt{\log n}$ as length of subinterval?

If we take length of length of subinterval as $\sqrt{\log n}$, then we can say that # of subintervals $= \sqrt{\log n}$. So a node can be a boundary node at most $\sqrt{\log n}$ times (we can bound this). This would not happen if we choose $\sqrt[3]{\log n}$ as length of subinterval.

Page :
Date 8 / 10 /2019

Makeset (v)

dad [v] = 0;

rank [v] = 0;

Union (r_1, r_2)

cases

1. rank [r_1] > rank [r_2] : dad [r_2] = r_1;
2. rank [r_1] < rank [r_2] : dad [r_1] = r_2;
3. rank [r_1] == rank [r_2] : dad [r_1] = r_2;
   rank [r_2] ++;

Find (v)

1. w = v;
   S = ϕ;
2. while (dad [w] != 0)
   S ← w;
   w = dad [w];
3. while (S != ϕ)
   t ← S
   dad [t] = w;
4. return (w)

Given a ~~set~~ of sequence of m M-U-F operations on a set $\{a_1, a_2, \ldots, a_n\}$ of n elements, what is the total time of the sequence?

Answer: $O((m+n) \log^* n)$

Find paths are the concern.

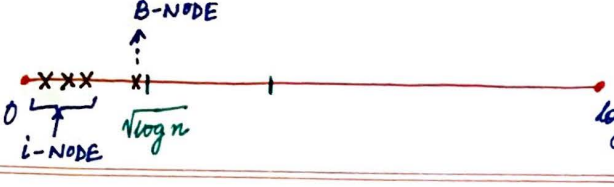
Total time $\approx$ the sum of find path lengths.

Break interval $(0, \ldots, \log n)$ ~~sub~~ into subintervals.

On each find paths.

b-nodes: the last node on each subinterval

i-nodes: otherwise



Total no. of b-nodes in a subinterval $\leq \sqrt{\log n}$

Total no. of b-nodes along all subintervals $\leq m\sqrt{\log n}$
(∵ there can be at most m find paths).

For each internal node, there must be a father (apart from root).
Once a node becomes a non root, its rank will never change.

Suppose, after 1 find operation, a_1 gets a father whose rank is larger than the former.

Final father of a_1 has $\sqrt{\log n}$ larger ranked father than the original rank. Thus this father is not in the same subinterval as a_1. as a_1 becomes the last node in its own subinterval. (b-node).

Total no. of i-nodes = $n\sqrt{\log n}$

→ Total no. of nodes on all find paths $\leq (n+m)\sqrt{\log n}$

If the subinterval is larger, then a node can stay as i-node longer.
So we make the subinterval smaller to balance the structure.

Let us assume, length of subinterval = $(\log n)^{1/3}$

$$\therefore \text{No. of subintervals} = \frac{\log n}{(\log n)^{1/3}} = \log n^{2/3}.$$

$$\therefore \#\text{ of b nodes} = (\log n)^{2/3}$$
$$\text{time} = m(\log n)^{2/3}$$

Hence time complexity increases. So we do not choose this as length of subinterval.

If the length of subinterval = $(\log n)^{2/3}$

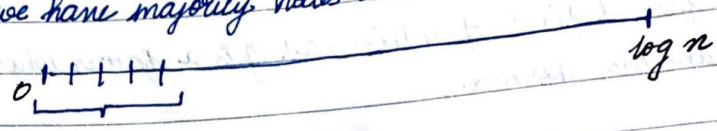
$$\#\text{ subintervals} = (\log n) / (\log n)^{2/3} = (\log n)^{1/3}$$

$$\#\text{ internal nodes} = n(\log n)^{2/3}$$
$$\#\text{ b-nodes} = n(\log n)^{1/3}$$

Again, time increases.

? We now ask whether majority nodes have lower rank or majority nodes have higher rank?

If we have majority nodes as lower rank,



dense region

↓

majority of nodes lie here

↓

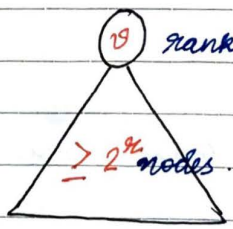
keep small subinterval so that an i-node can quickly jump from one region to other.

- **Lemma :—** There are $\leq \frac{n}{2^r}$ nodes of rank r.

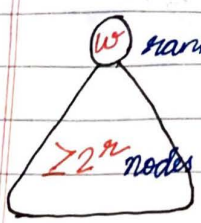
(There can be all n nodes at rank 0 when $r = 0$; we get there by executing makeset operation n times).
(As rank increases, no. of nodes decreases).

PROOF:— $r=0$; # nodes $= \frac{n}{2^0} = n \Rightarrow$ TRUE.

If v has a rank of r,



Now, lets see another node w and suppose this is another root node with rank r :—



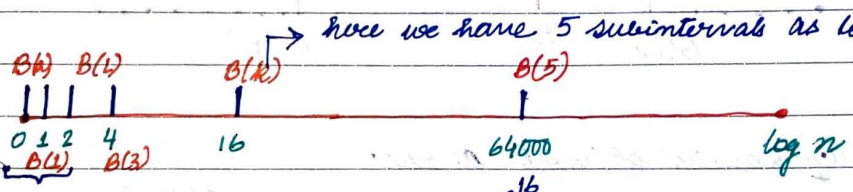
These 2 trees do not have any common descendant.
2^r nodes in both cases are different.

Total no of rank r nodes $\leq \frac{n}{2^r}$

The no. of lower ranked nodes is much higher than no. of higher ranked nodes.
We split the rank line non uniformly.

→ Goal is to :-

1. reduce no of subintervals
2. make the low ranked nodes have smaller subintervals and higher ranked nodes are spread apart.



for these subintervals
i-node is the B-node.

$$B(i+1) = 2^{B(i)}$$

# of subintervals = i (→ $\log^*(n)$) such that $B(i) \geq \log n$.

$$2^{2^{2^{\cdots}}} \Big\} \text{ } i \text{ 2's here} \geq \log n$$

$\log^* n$ → repeatedly take log n until the no. becomes ≤ 1.

$$\underbrace{\log \log \log \cdots}_{i \text{ logs here.}} n \leq_m 1$$

$$\log^* n = \min\{ i \mid 2^{2^{\cdots^{2}}} \Big\} i \geq n \}$$

# of subintervals = $\log^* n$

→ Practically, $\log^* n$ can never be larger than 8.
↓
This is considered as a linear time.

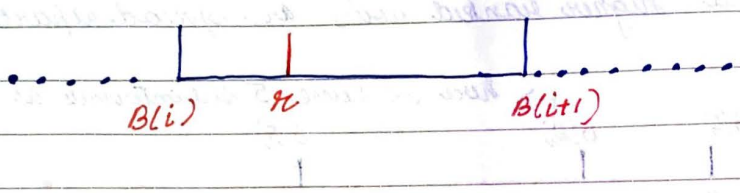
[We compute b-nodes in terms of paths, we compute # of inodes in terms of elements]

no. of b-nodes on each path $\leq \log^* n$

$\therefore$ no. of total b-nodes $\leq m \log^* n$ (for m paths)

? What about internal nodes?

Let a_i be an internal node of rank r.
Question is how long can a_i stay as an internal node.



~~a_i can be at most x now~~
a_i can be an i-node at most $B(i+1) - r$ times.

So the total no. of i-nodes of rank r that can stay as internal nodes :—

$$\frac{n\,(B(i+1) - r)}{2^r}$$

In the rank line, from rank $B(i)$ to $B(i+1)$ along each subinterval, the no. of internal nodes :—

$$\sum_{r=B(i)}^{B(i+1)-1} \frac{n\,(B(i+1) - r)}{2^r} \leq \sum_{r=B(i)}^{B(i+1)-1} \frac{B(i+1)\cdot n}{2^r}$$

does not depend on r

$$= B(i+1)\cdot n \sum_{r=B(i)}^{B(i+1)-1} \frac{1}{2^r}$$

$$\leq \frac{B(i+1)\cdot n}{2^{B(i)}} \sum_{i=0}^{+\infty} \frac{1}{2^r} = \frac{2\,B(i+1)\cdot n}{2^{B(i)}}$$

$$= 2n$$

Total no. of i-nodes along one subinterval is bounded by $2n$
Total no. of i nodes along all subintervals $\leq 2n \log^* n$

Total no. of nodes along all find paths $\leq m \log^* n + 2n \log^* n$
$\leq (m+n) \log^* n$

Total no. of union & find in Kruskal algorithm is at most $(2m+n)$.

Kruskal is probably a lot faster than Dijkstra.