Greedy algorithms

CMPSC 465 - Yana Safonova

Disjoint sets or union-find sets or merge-find sets

Optimized disjoint set - implementation

```
#finding root of an element
def root(Arr, i):
    while Arr[ i ] != i:
        i = Arr[ i ]
    return i

def union(Arr, A, B):
    root_A = root(Arr, A)
    root_B = root(Arr, B)
    Arr[ root_A ] = root_B


def find(Arr, A, B):
    return root(Arr, A)==root(Arr, B)
```

Optimized disjoint set - only a half-way through!

A loop is hidden here

```
#finding root of an element
def root(Arr, i):
    while Arr[ i ] != i:
        i = Arr[ i ]
    return i
```

We don't have a loop here anymore

```
def union(Arr, A, B):
    root_A = root(Arr, A)
    root_B = root(Arr, B)
    Arr[ root_A ] = root_B
```

```
def find(Arr, A, B):
    return root(Arr, A)==root(Arr, B)
```

Optimized disjoint set - the worst case scenario

0	1	2	3	4	5
0	1	2	3	4	5

Optimized disjoint set - the worst case scenario

0	1	2	3	4	5
0	0	2	3	4	5

Union(1, 0)

Optimized disjoint set - the worst case scenario

0	1	2	3	4	5
2	0	2	3	4	5

Union(1, 0)
Union(0, 2)

Optimized disjoint set - the worst case scenario

0	1	2	3	4	5
2	0	3	3	4	5

Union(1, 0)
Union(0, 2)
Union(2, 3)

Optimized disjoint set - the worst case scenario

0	1	2	3	4	5
2	0	3	4	4	5

Union(1, 0)
Union(0, 2)
Union(2, 3)
Union(3, 4)

Optimized disjoint set - the worst case scenario

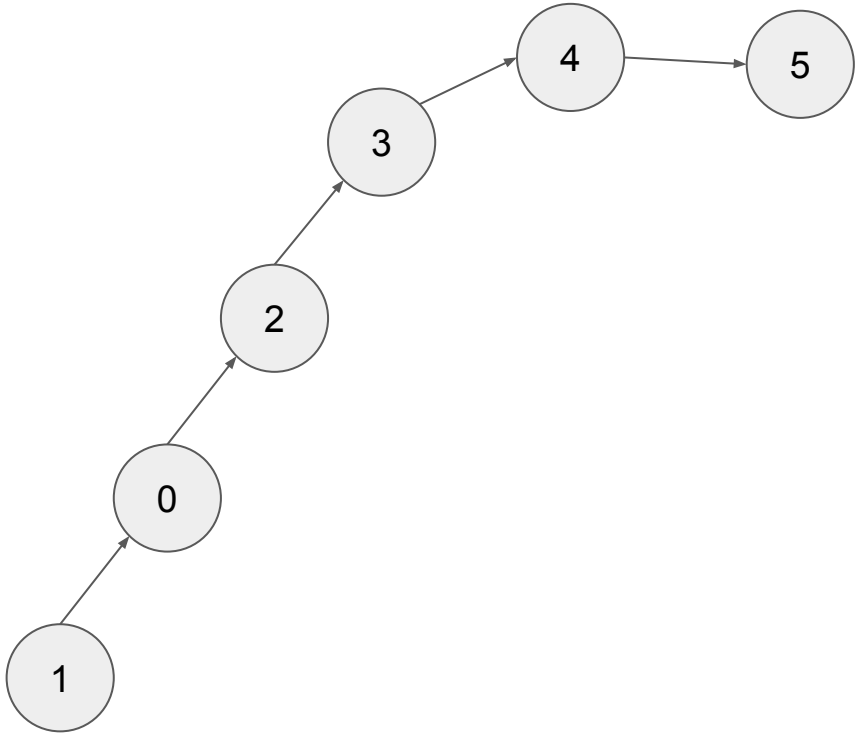
0	1	2	3	4	5
2	0	3	4	5	5

Union(1, 0)
Union(0, 2)
Union(2, 3)
Union(3, 4)
Union(4, 5)

Optimized disjoint set - the worst case scenario

0	1	2	3	4	5
2	0	3	4	5	5

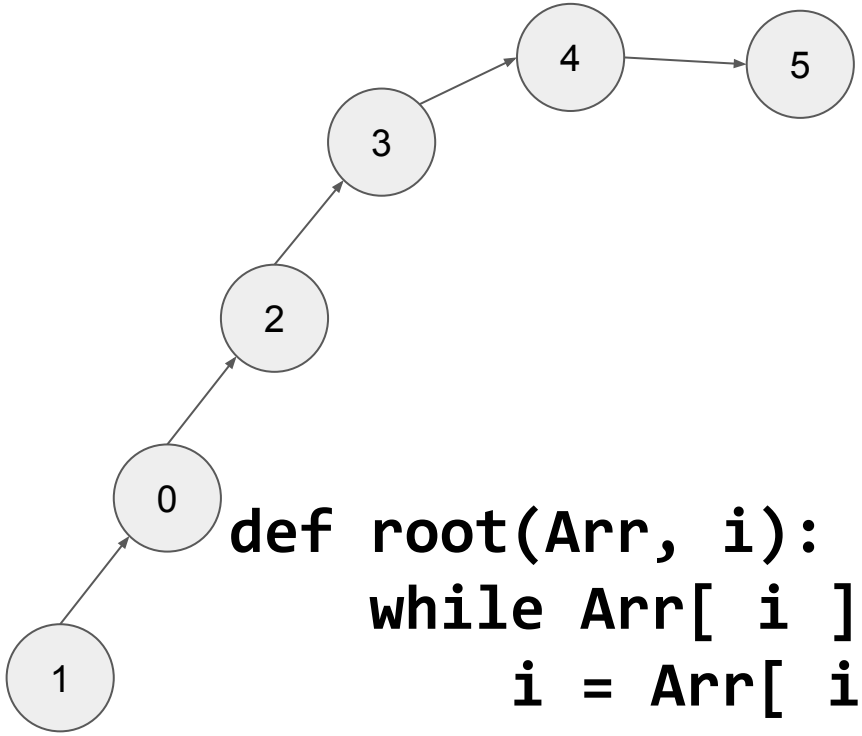
Union(1, 0)
Union(0, 2)
Union(2, 3)
Union(3, 4)
Union(4, 5)



Optimized disjoint set - the worst case scenario

0	1	2	3	4	5
2	0	3	4	5	5

Union(1, 0)
Union(0, 2)
Union(2, 3)
Union(3, 4)
Union(4, 5)




```
def root(Arr, i):
    while Arr[ i ] != i:
        i = Arr[ i ]
    return i
```

Still O(N)!

Disjoint sets

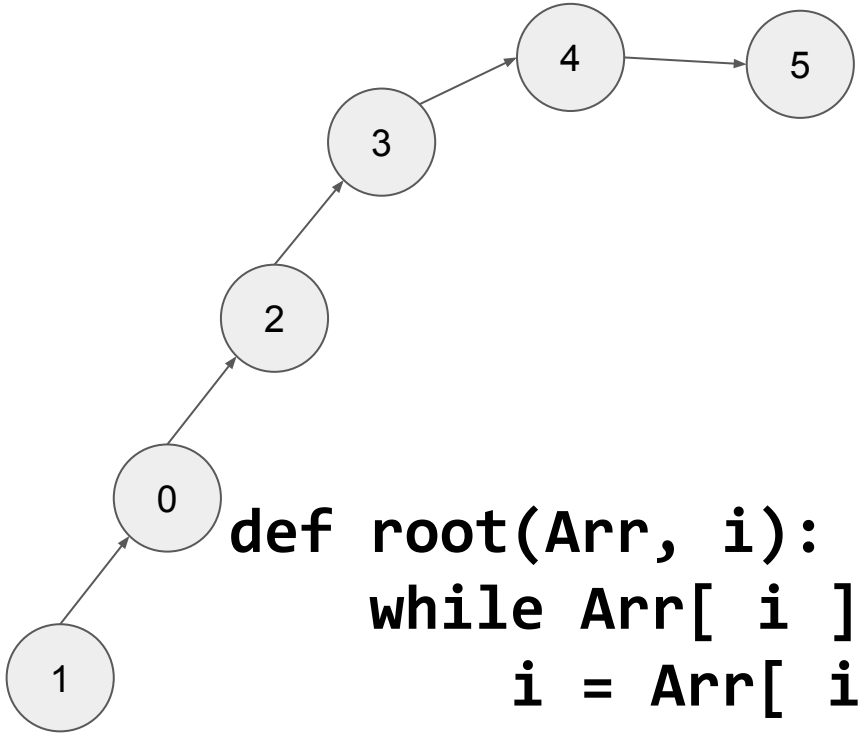
- Stores non-overlapping sets
- Union(a, b) = merges sets where a and b are located
- Find(a, b) = tells whether or not a and b are located in the same set

	Naive implementation	Semi-optimized implementation
Find	O(1)	O(N) - uses root
Union	O(N)	O(N) - uses root
Union for all elements	$O(N^2)$	$O(N^2)$
Root	-	O(N)

From semi-optimal to optimized implementation

Consecutive merging elements from left to right (or vice versa) is the worst case scenario for the semi-optimized implementation

The tree representing all unions looks like a single branch in this case



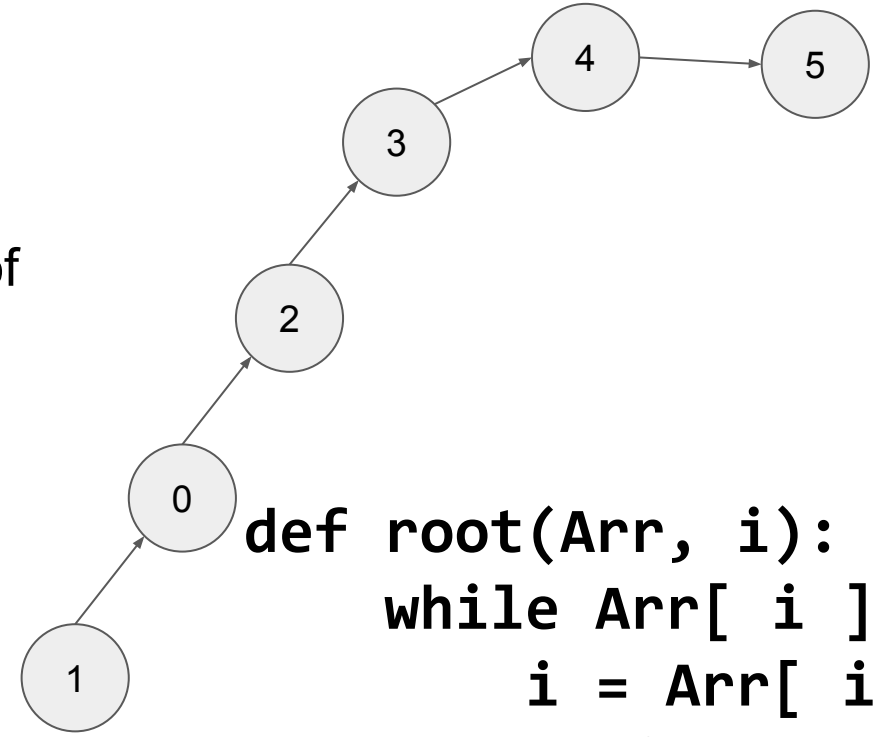

```
def root(Arr, i):
    while Arr[ i ] != i:
        i = Arr[ i ]
    return i
```

From semi-optimal to optimized implementation

Consecutive merging elements from left to right (or vice versa) is the worst case scenario for the semi-optimized implementation

The tree representing all unions looks like a single branch in this case

Do we really care about the shape of the tree representing elements of the same set?




```
def root(Arr, i):
    while Arr[ i ] != i:
        i = Arr[ i ]
    return i
```

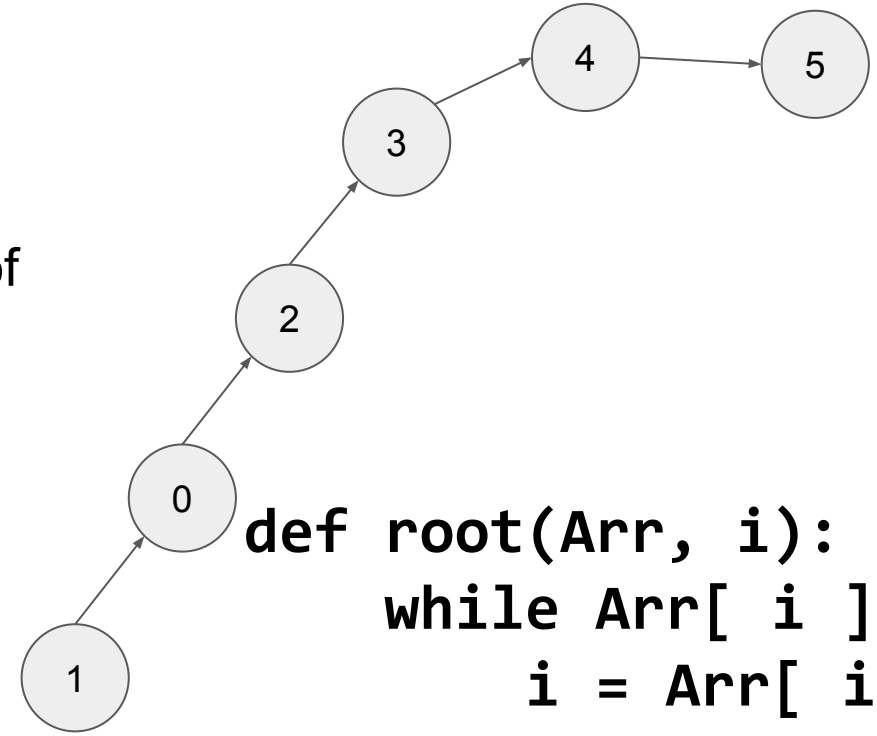
From semi-optimal to optimized implementation

Consecutive merging elements from left to right (or vice versa) is the worst case scenario for the semi-optimized implementation

The tree representing all unions looks like a single branch in this case

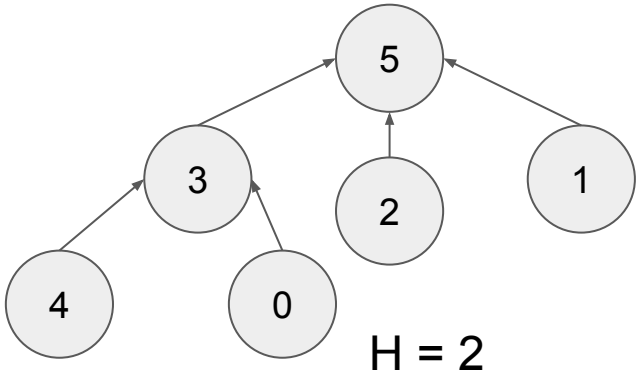
Do we really care about the shape of the tree representing elements of the same set?

No!

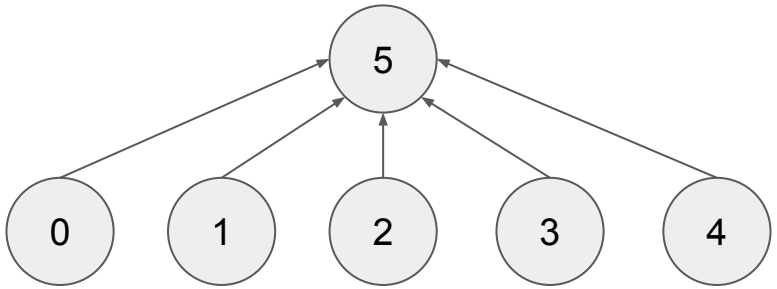



```
def root(Arr, i):
    while Arr[ i ] != i:
        i = Arr[ i ]
    return i
```

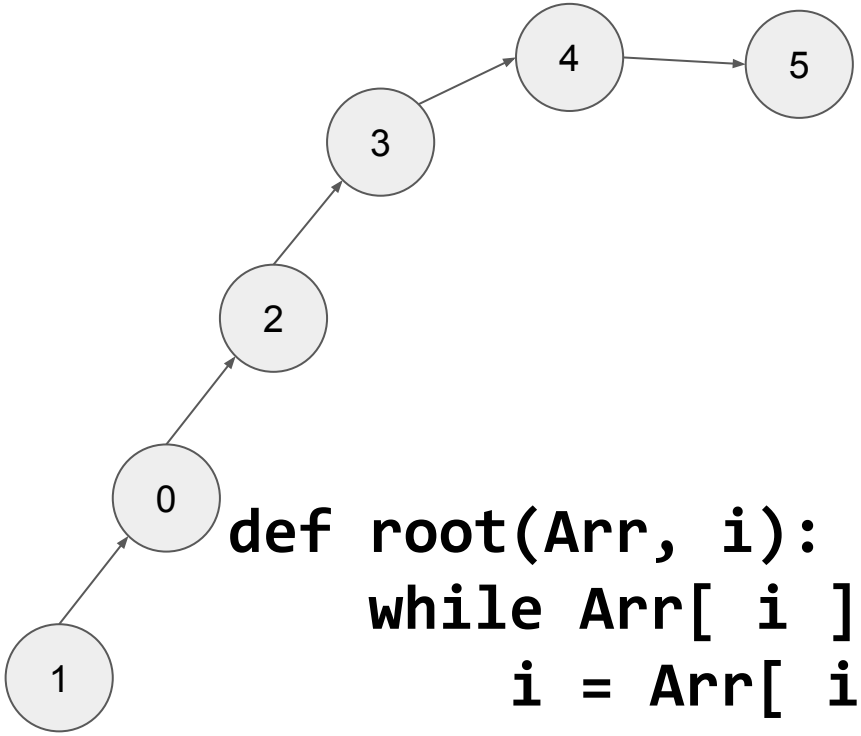
The running time of the “root” procedure is defined as the tree height



H = 2



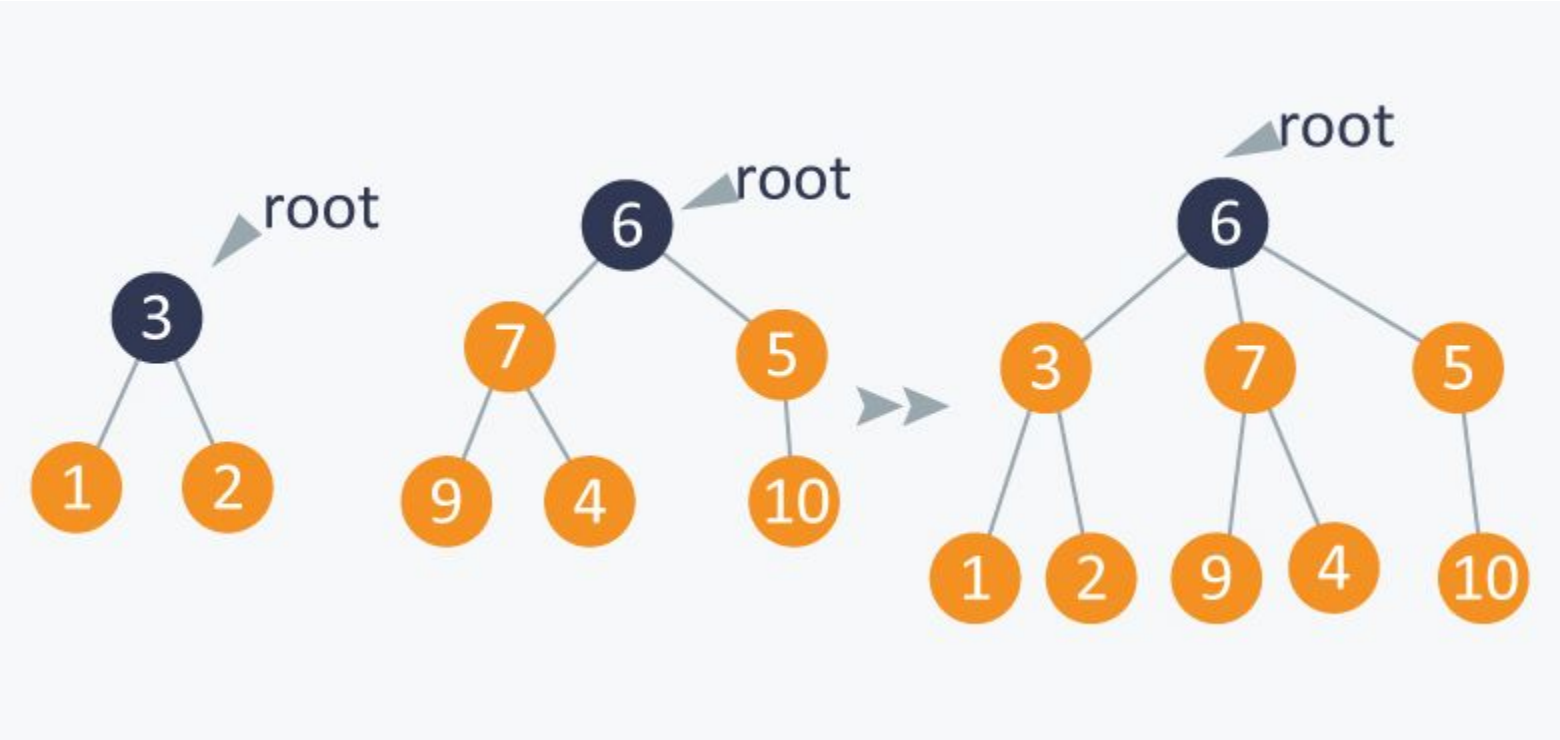
H = 1



H = 5

```
def root(Arr, i):
    while Arr[ i ] != i:
        i = Arr[ i ]
    return i
```

Optimized disjoint set: weighted-union operation

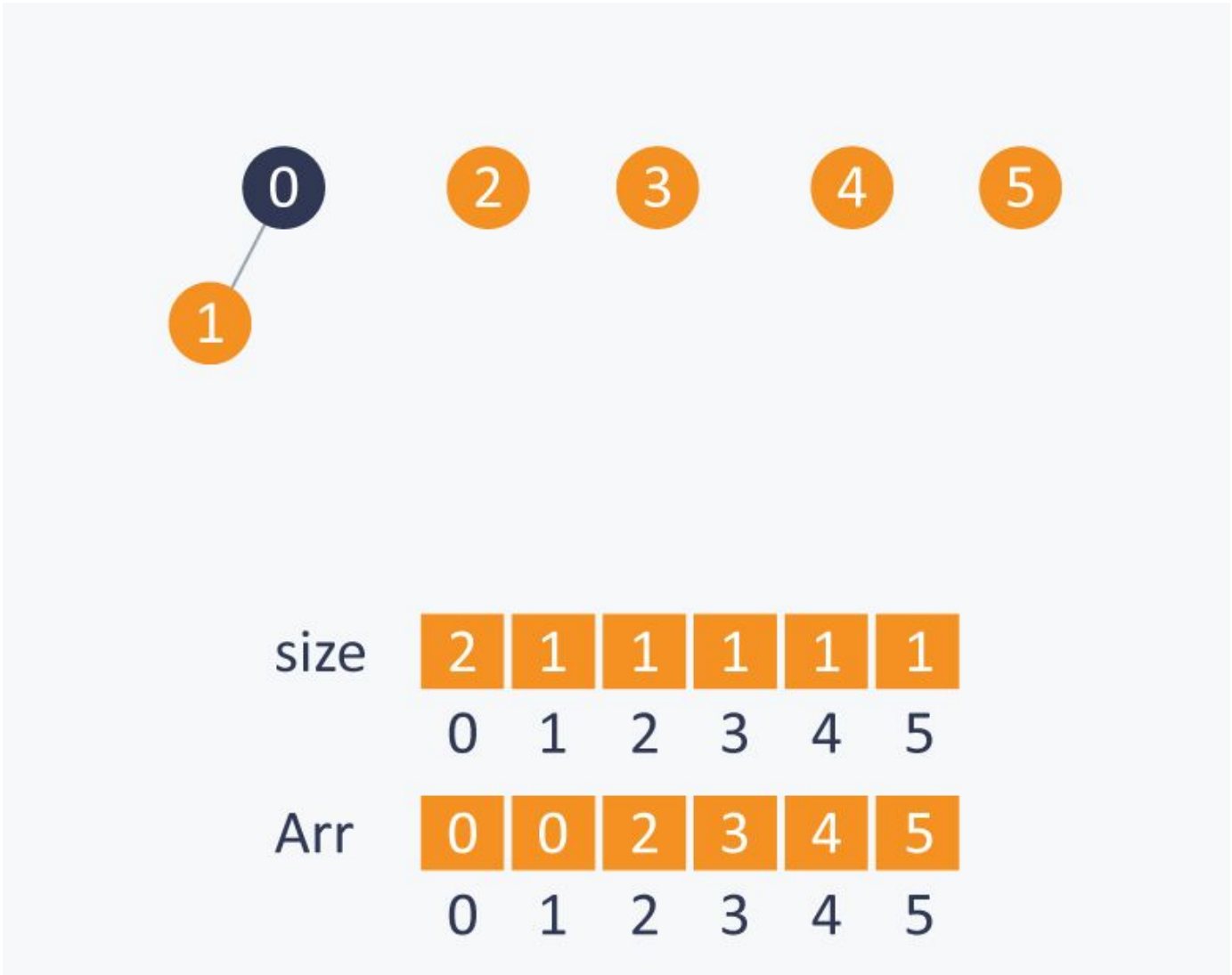


Union(A, B): instead of always merging the left set with the right set, let's merge the smallest set with the biggest one

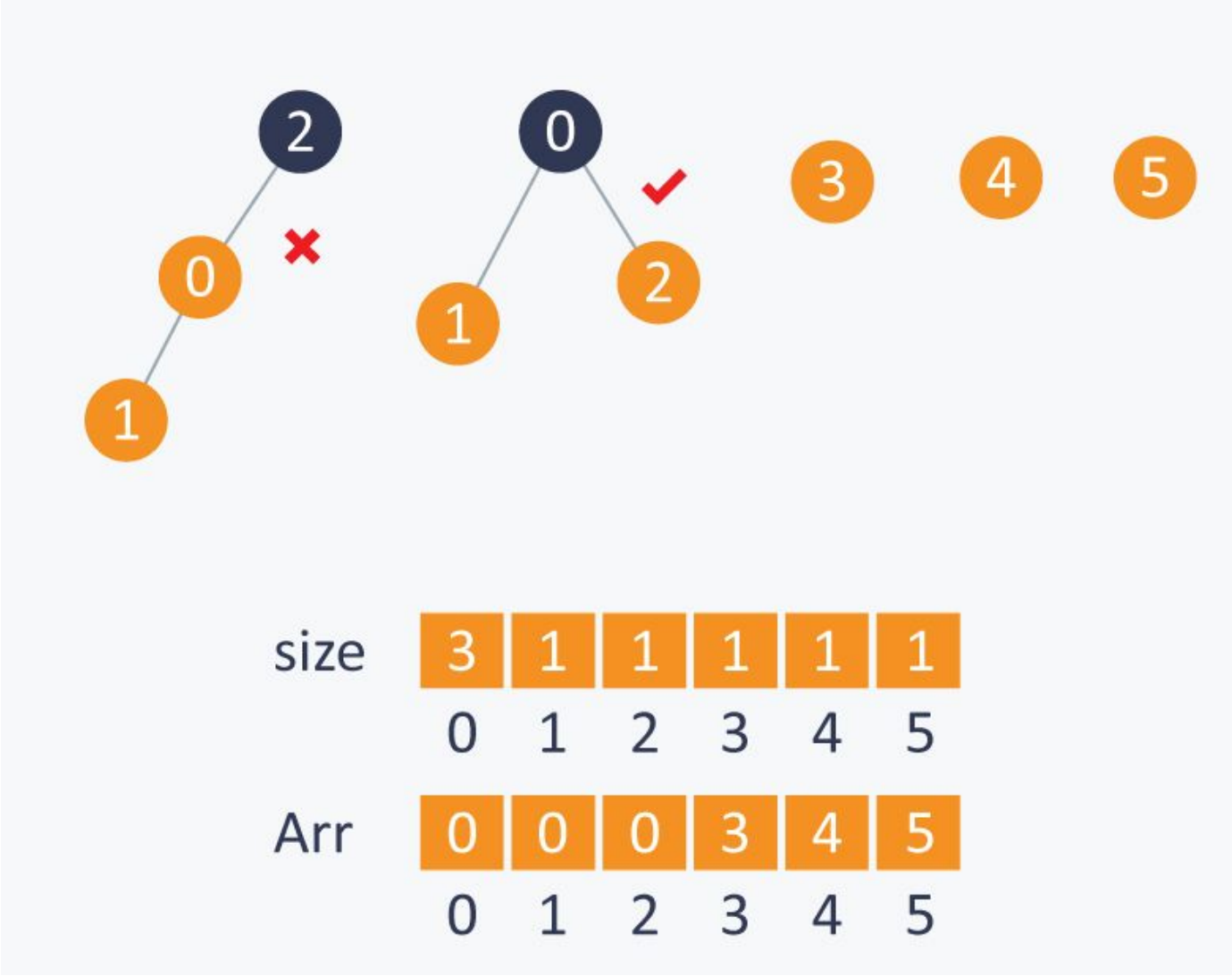
In terms of trees representing sets, it would mean rerooting the smallest tree

Weighted-union operation

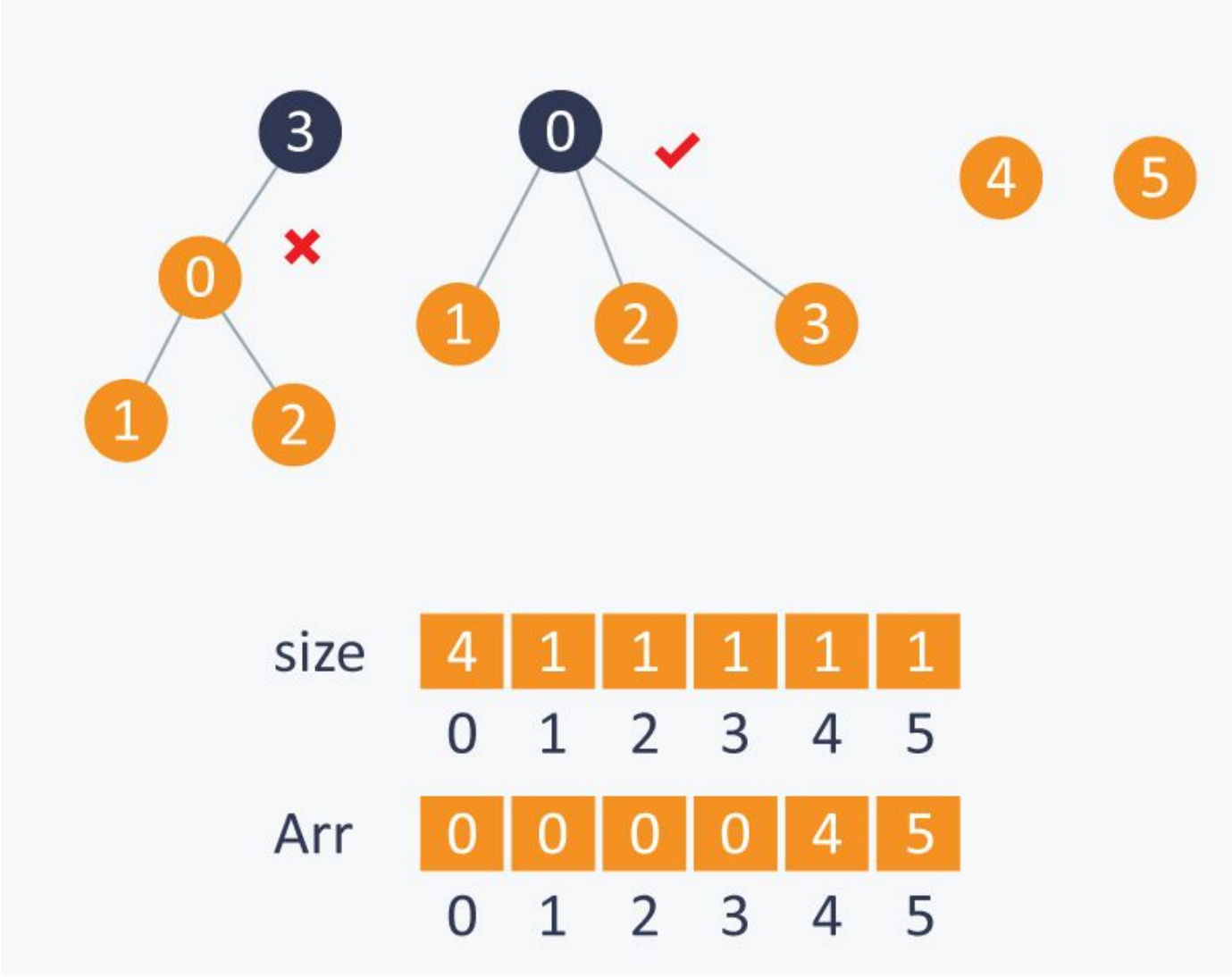
Weighted-union operation - union(0, 1)



Weighted-union operation - union(1, 2)



Weighted-union operation - union(3, 2)



Weighted-union operation

```
def initialize(Arr, size, N):
    for i in range(N):
        Arr[ i ] = i
        size[ i ] = 1

def weighted_union(Arr, size, A, B):
    root_A = root(Arr, A)
    root_B = root(Arr, B)
    if size[root_A] < size[root_B]:
        Arr[root_A] = Arr[root_B]
        size[root_B] += size[root_A]
    else:
        Arr[root_B] = Arr[root_A]
        size[root_A] += size[root_B]
```

Running time of the “root” procedure

- If we merge sets represented by trees of different heights, then the height of the resulting tree (tree tree after union) doesn’t change

- ?: If we merge sets of the same height H, what would be the height of the resulting tree?

Running time of the “root” procedure

- If we merge sets represented by trees of different heights, then the height of the resulting tree (tree tree after union) doesn’t change

- ?: If we merge sets of the same height H, what would be the height of the resulting tree? (H + 1)

Running time of the “root” procedure

- If we merge sets represented by trees of different heights, then the height of the resulting tree (tree tree after union) doesn’t change

- ?: If we merge sets of the same height H, what would be the height of the resulting tree? (H + 1)

Running time of the “root” procedure

- If we merge sets represented by trees of different heights, then the height of the resulting tree (tree tree after union) doesn’t change

- ?: If we merge sets of the same height H, what would be the height of the resulting tree? (H + 1)

Running time of the “root” procedure

- If we merge sets represented by trees of different heights, then the height of the resulting tree (tree tree after union) doesn’t change

- ?: If we merge sets of the same height H, what would be the height of the resulting tree? (H + 1)

By merging two elements, we increased the maximum height

Running time of the “root” procedure

- If we merge sets represented by trees of different heights, then the height of the resulting tree (tree tree after union) doesn’t change

- ?: If we merge sets of the same height H, what would be the height of the resulting tree? (H + 1)

To increase the max height further, we need to merge the red set with another set of the same height

Running time of the “root” procedure

- If we merge sets represented by trees of different heights, then the height of the resulting tree (tree tree after union) doesn’t change

- ?: If we merge sets of the same height H, what would be the height of the resulting tree? (H + 1)

To increase the max height further, we need to merge the red set with another set of the same height > we need to construct it!

Running time of the “root” procedure

- If we merge sets represented by trees of different heights, then the height of the resulting tree (tree tree after union) doesn’t change

- ?: If we merge sets of the same height H, what would be the height of the resulting tree? (H + 1)

To increase the max height further, we need to merge the red set with another set of the same height > we need to construct it > union of red and green sets increases the height

Running time of the “root” procedure

- If we merge sets represented by trees of different heights, then the height of the resulting tree (tree tree after union) doesn’t change

- ?: If we merge sets of the same height H, what would be the height of the resulting tree? (H + 1)

To increase the max height further, we need to merge the red set with another set of the same height > we need to construct it > union of red and green sets increases the height

Running time of the “root” procedure

- If we merge sets represented by trees of different heights, then the height of the resulting tree (tree tree after union) doesn’t change

- ?: If we merge sets of the same height H, what would be the height of the resulting tree? (H + 1)

To increase the max height further, we need to merge the red set with another set of the same height > we need to construct it > union of red and green sets increases the height

Running time of the “root” procedure

- If we merge sets represented by trees of different heights, then the height of the resulting tree (tree tree after union) doesn’t change

- ?: If we merge sets of the same height H, what would be the height of the resulting tree? (H + 1)

To increase the max height further, we need to merge the red set with another set of the same height > we need to construct it > union of red and green sets increases the height

Running time of the “root” procedure

- If we merge sets represented by trees of different heights, then the height of the resulting tree (tree tree after union) doesn’t change

- ?: If we merge sets of the same height H, what would be the height of the resulting tree? (H + 1)

To increase the max height further, we need to merge the red set with another set of the same height > we need to construct it > union of red and green sets increases the height

Running time of the “root” procedure

The maximum height is achieved when we merge “similar” sets: represented by trees of the same heights and in the worst case having the same numbers of elements

Running time of the “root” procedure

The maximum height is achieved when we merge “similar” sets: represented by trees of the same heights and in the worst case having the same numbers of elements

?: what is the maximum number of unions of sets of the same sizes?

Running time of the “root” procedure

The maximum height is achieved when we merge “similar” sets: represented by trees of the same heights and in the worst case having the same numbers of elements

?: what is the maximum number of unions of sets of the same sizes? O(log N)

Disjoint sets

- Stores non-overlapping sets
- Union(a, b) = merges sets where a and b are located
- Find(a, b) = tells whether or not a and b are located in the same set

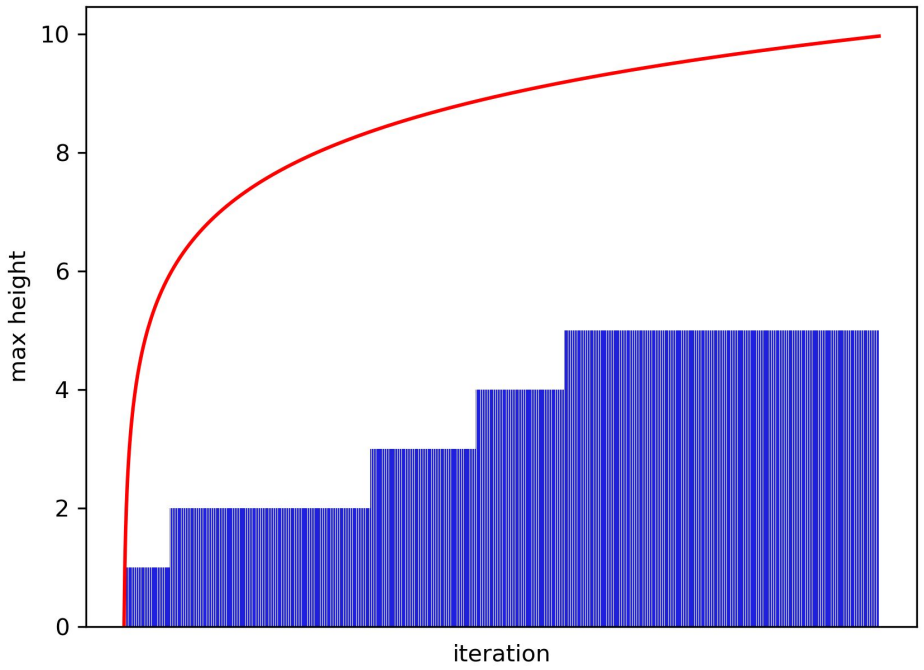
	Naive implementation	Semi-optimized implementation	Weighted-union implementation
Find	O(1)	O(N)	O(logN)
Union	O(N)	O(N)	O(logN)
Union for all elements	$O(N^2)$	$O(N^2)$	O(NlogN)
Root	-	O(N)	O(logN)

Example - random case

Let’s start with N elements and at each step merge two randomly chosen elements

Example - random case

Let’s start with N elements and at each step merge two randomly chosen elements

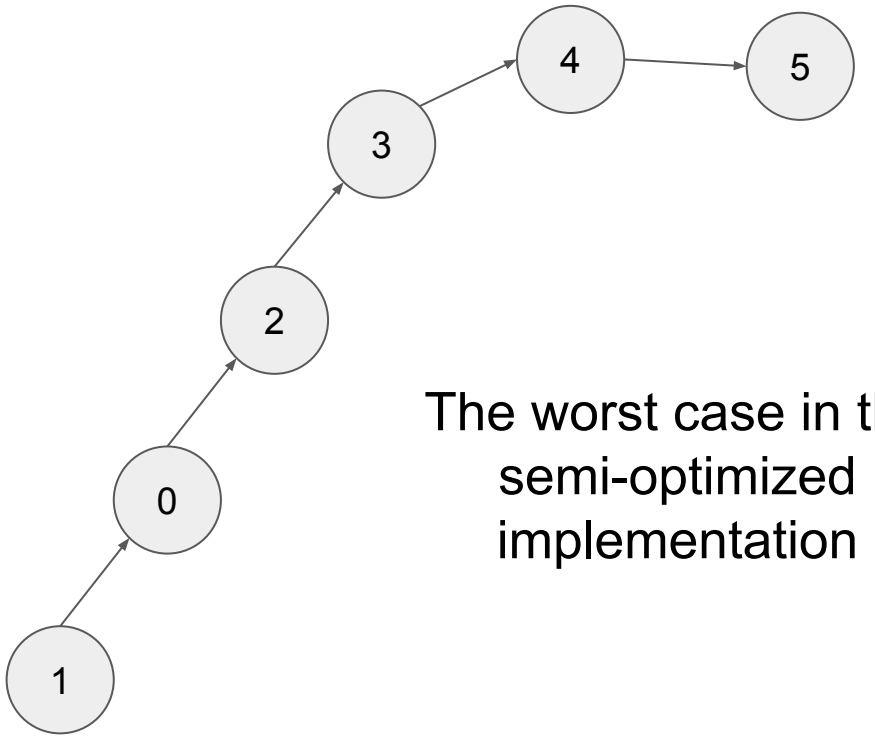


$Y = \log_2 X$

N = 1000

Example - previous worst case

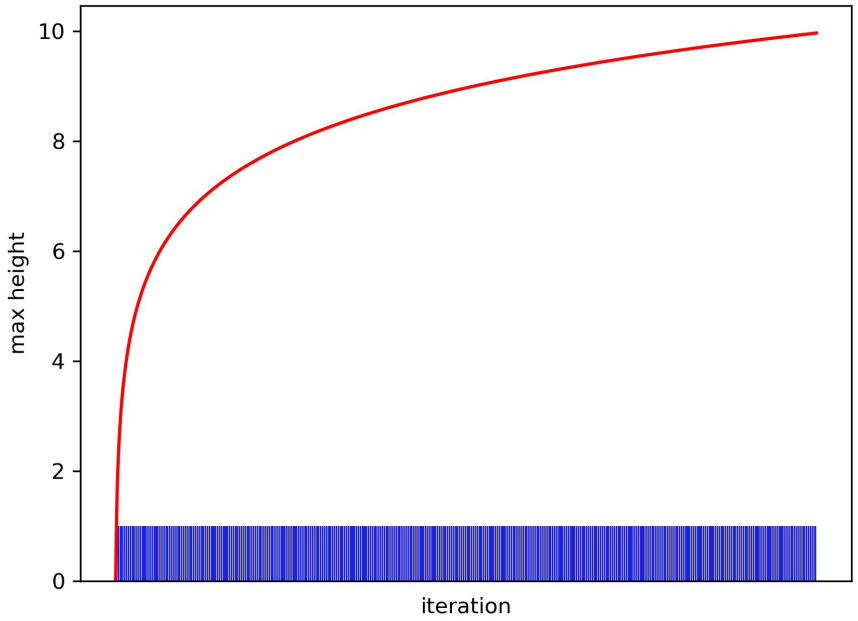
Let’s start with N elements and at each step merge two consecutive elements

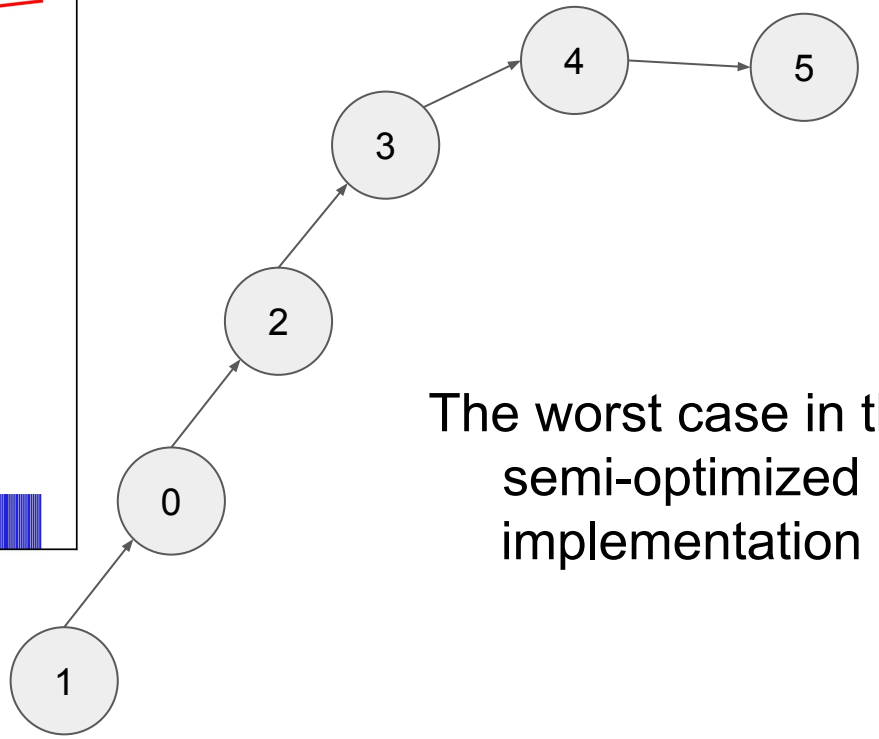


The worst case in the semi-optimized implementation

Example - previous worst case

Let’s start with N elements and at each step merge two consecutive elements





The worst case in the semi-optimized implementation

Example - previous worst case

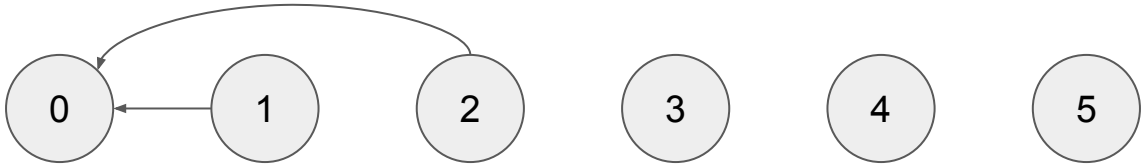
Let’s start with N elements and at each step merge two consecutive elements

Example - previous worst case

Let’s start with N elements and at each step merge two consecutive elements

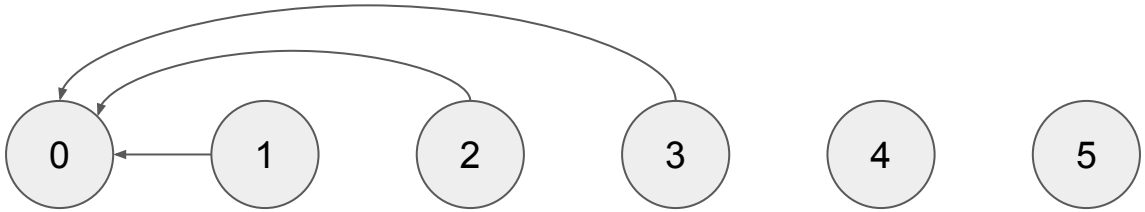
Example - previous worst case

Let’s start with N elements and at each step merge two consecutive elements



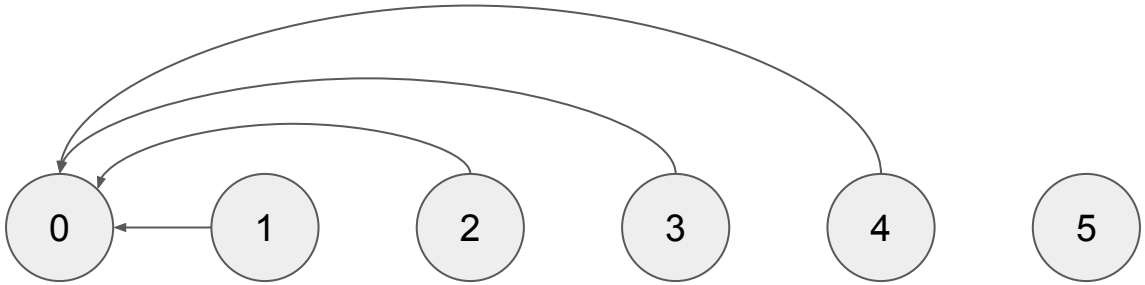
Example - previous worst case

Let’s start with N elements and at each step merge two consecutive elements



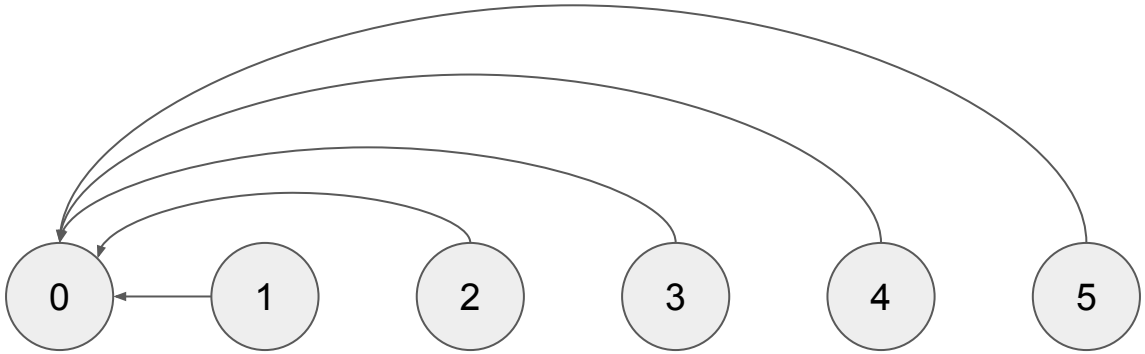
Example - previous worst case

Let’s start with N elements and at each step merge two consecutive elements



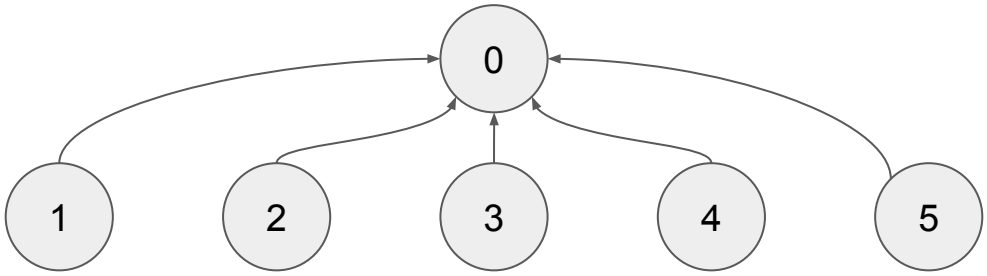
Example - previous worst case

Let’s start with N elements and at each step merge two consecutive elements



Example - previous worst case

Let’s start with N elements and at each step merge two consecutive elements



Example: how to compute the maximum tree height?

```
def root(Arr, i):
    while Arr[ i ] != i:
        i = Arr[ i ]
    return i
```

Example: how to compute the maximum tree height?

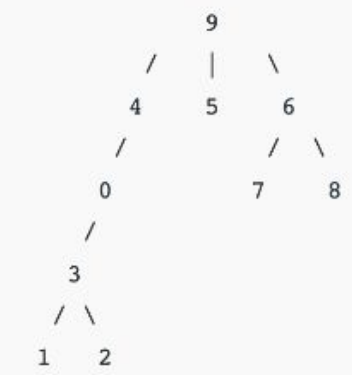
```
def root(Arr, i):
    while Arr[ i ] != i:
        i = Arr[ i ]
    return i

def height(Arr, i):
    H = 0
    while Arr[ i ] != i:
        H += 1
        i = Arr[ i ]
    return H
```

Example: how to compute the maximum tree height?

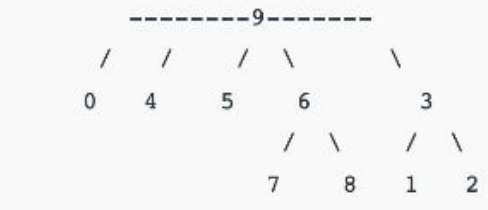
```
def height(Arr, i):
    H = 0
    while Arr[ i ] != i:
        H += 1
        i = Arr[ i ]
    return H

def max_height(Arr, N):
    return max([height(Arr, i) for i in range(N)])
```

Further optimization of disjoint set

Let the subset {0, 1, .. 9} be represented as below and find() is called for element 3.



When *find()* is called for 3, we traverse up and find 9 as representative of this subset. With path compression, we also make 3 and 0 as the child of 9 so that when find() is called next time for 0, 1, 2 or 3, the path to root is reduced.



Path compression: requires additional memory but speed up the “find” procedure

Path compression + weighted union:

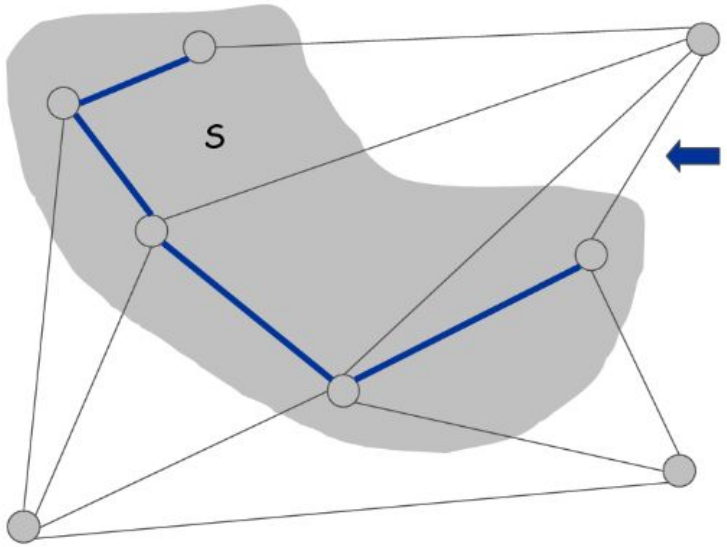
the amortized time complexity is O(α(n)),

α(n) is the inverse Ackermann function (a very slowly growing function)

Prim’s algorithm

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T.

- Initialize S = any node.
- Apply cut property to S.
- Add min cost edge in cutset corresponding to S to T, and add one new explored node u to S.



Implementation uses the priority queue similar to the one in Dijkstra algorithm

Maximum spanning tree?

The algorithms are similar except for:

- Kruskal's algorithm: sort edges in the decreasing order of the weights
- Prim's algorithm: choose the max weight edge
- The cut property: the max weight edge in a cutset is in the MST
- The cycle property: the min weight edge in a cycle is not in the MST

MST and clustering

Application: clustering

You're given n items and the distance $d(u, v)$ between each of pair.

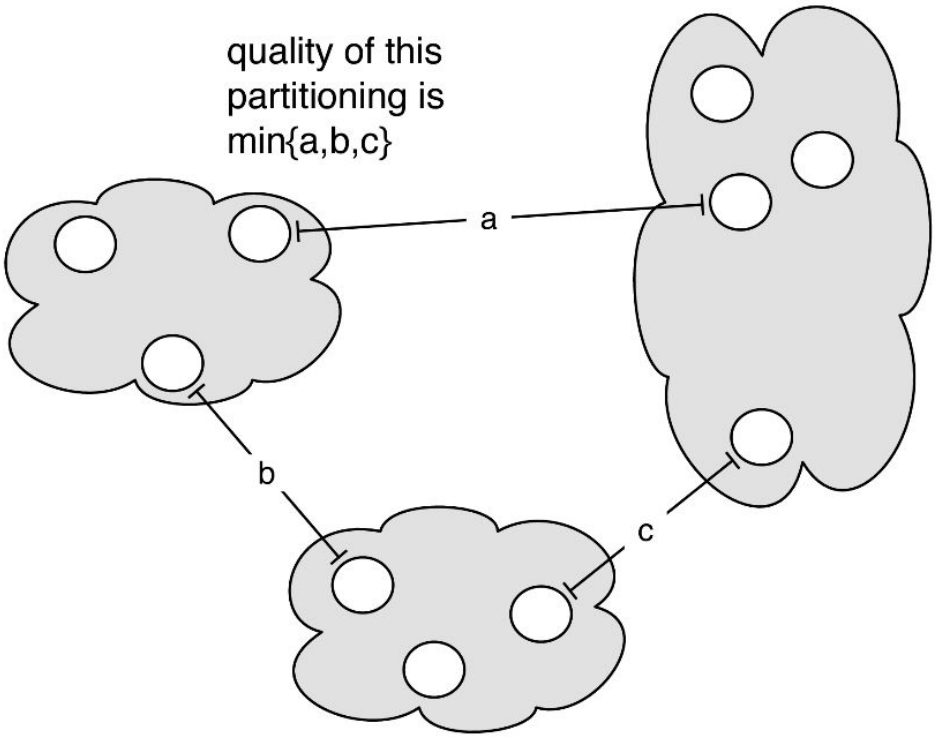
$d(u, v)$ may be an actual distance, or some abstract representation of how dissimilar two things are.

(What's the "distance" between two species?)

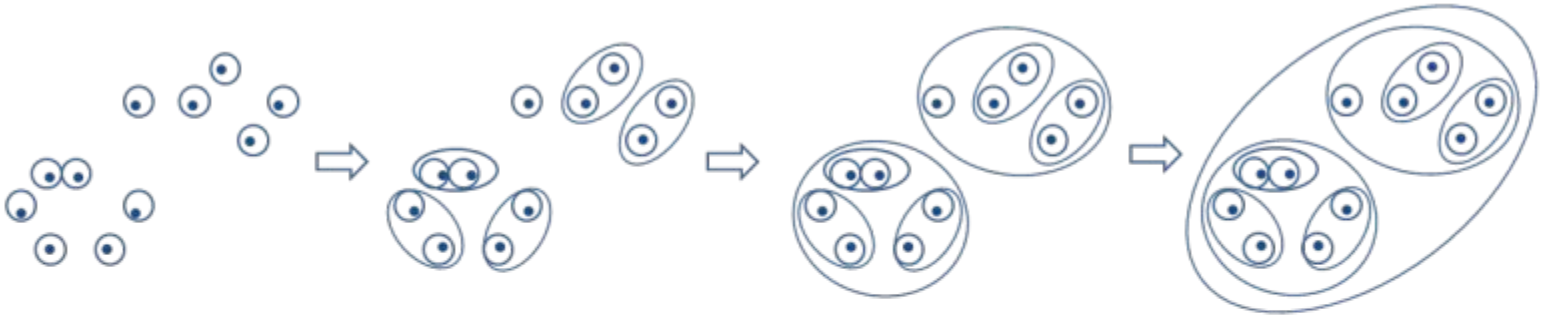
Our Goal: Divide the n items up into k groups so that the minimum distance between items in different groups is maximized.

Clustering

Our Goal: Divide the n items up into k groups so that the minimum distance between items in different groups is maximized.

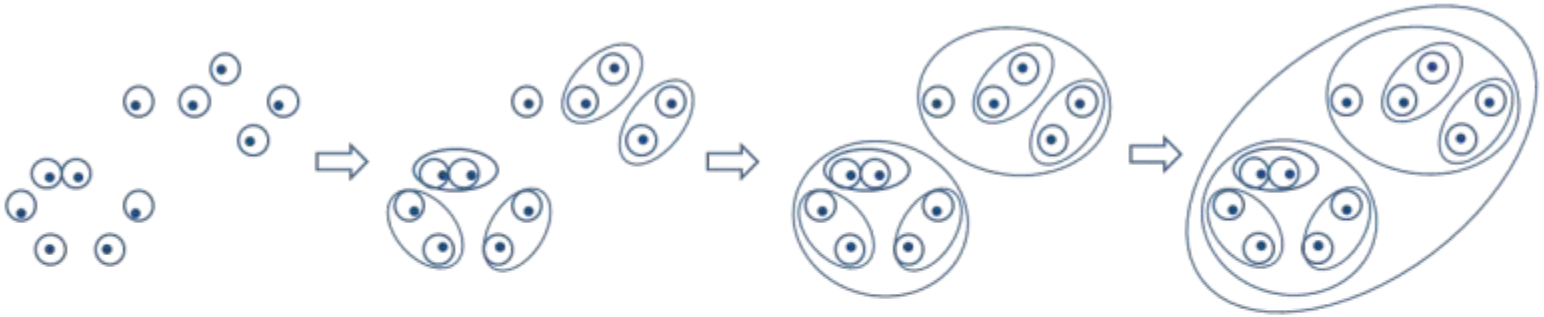


Agglomerative clustering: from bottom to top



1. Compute distance between each pairs of objects
2. Choose a pair (A, B) with the shortest distance
3. Combine A & B into the same cluster A’
4. Remove A and B from the distance matrix and add A’
5. Compute distances from A’ to all other objects
6. Repeat from 2

Agglomerative clustering: from bottom to top

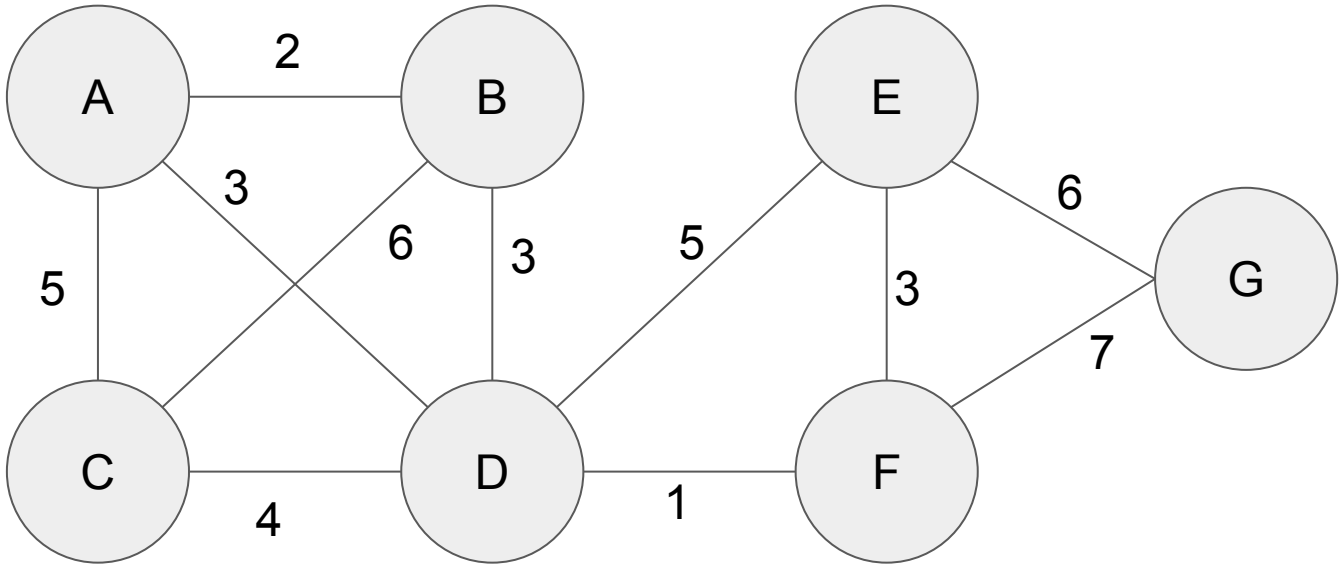


1. Compute distance between each pairs of objects
2. Choose a pair (A, B) with the shortest distance
3. Combine A & B into the same cluster A’
4. Remove A and B from the distance matrix and add A’
5. **Compute distances from A’ to all other objects**
6. Repeat from 2

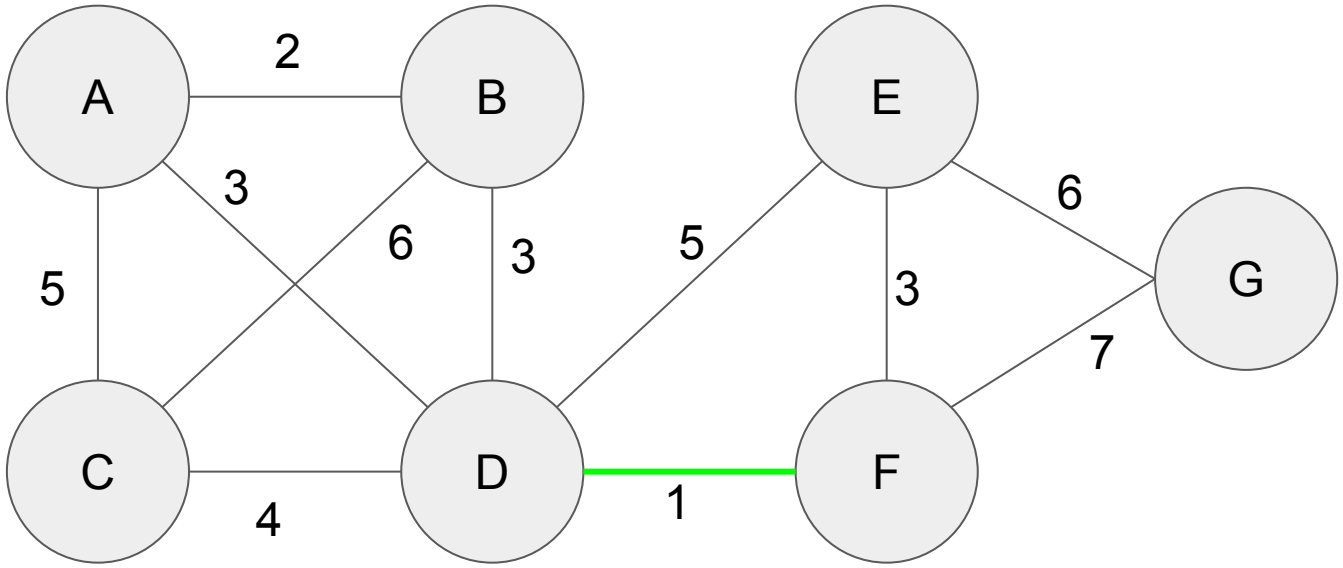
Single linkage: min{ d(x, y): x in A, y in B }

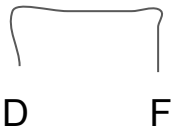
Average clustering: sum{ d(x, y): x in A, y in B } / |A| / |B|

Kruskal's algorithm aka single-linkage clustering

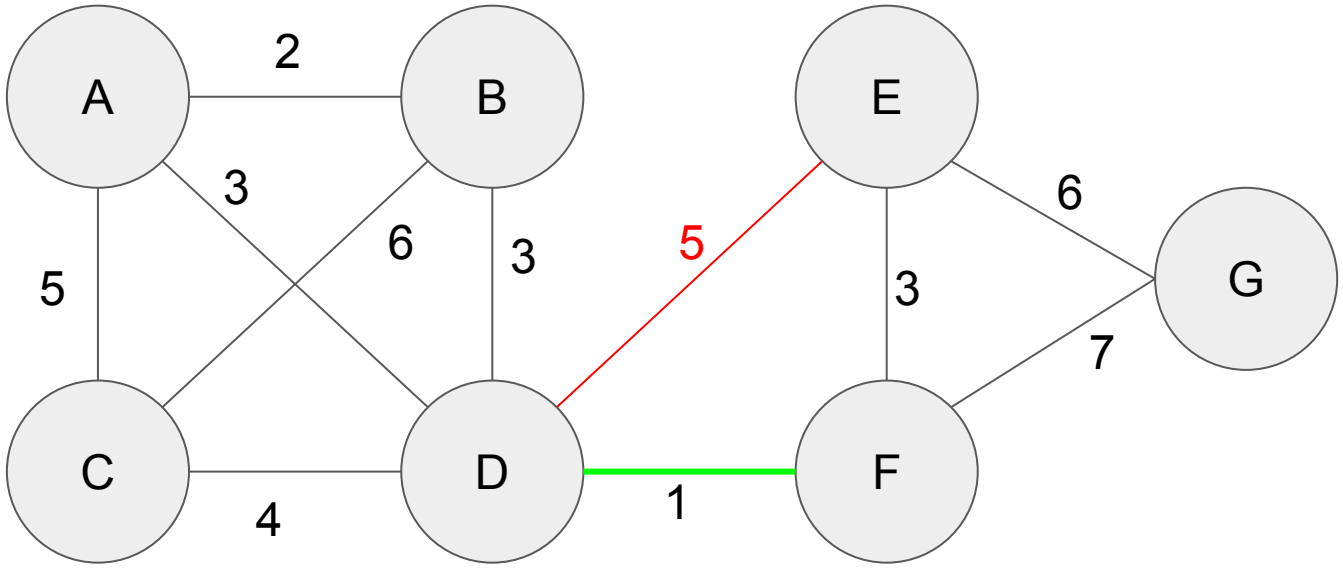


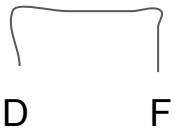
Kruskal’s algorithm - weight 1



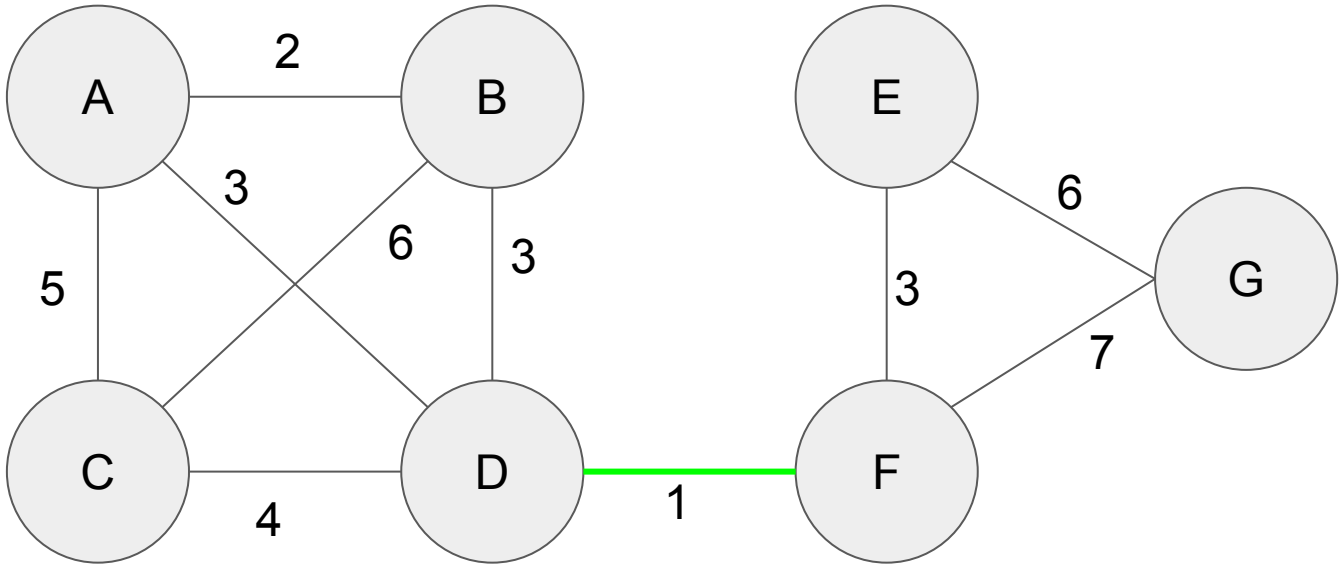


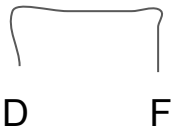
Kruskal’s algorithm - weight 1



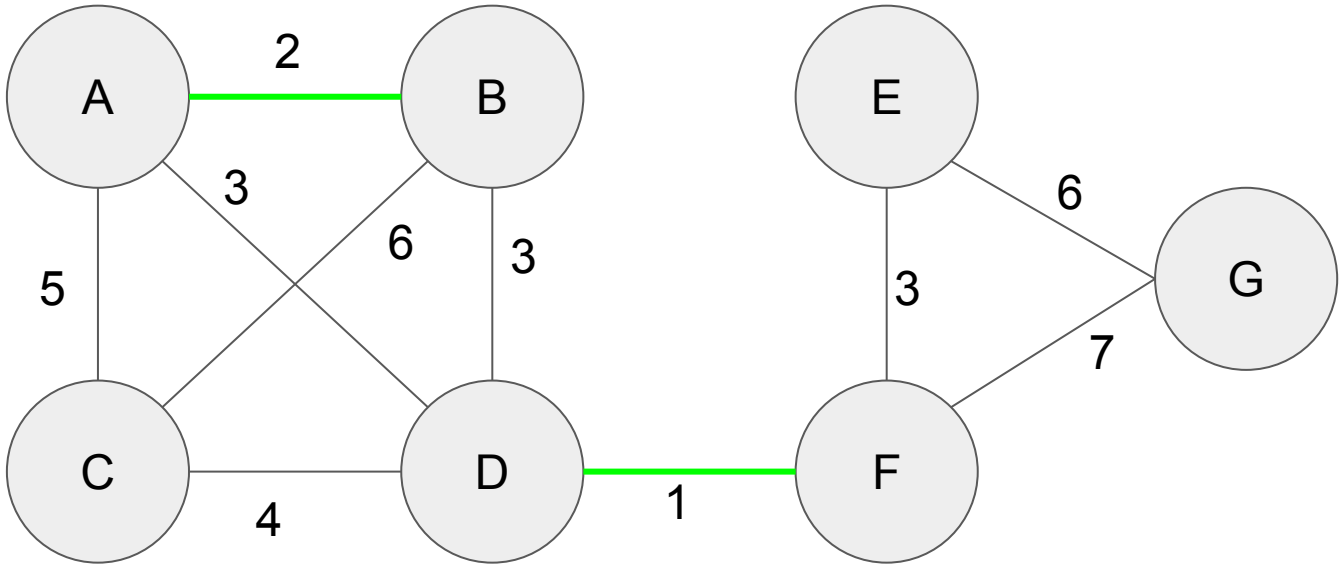


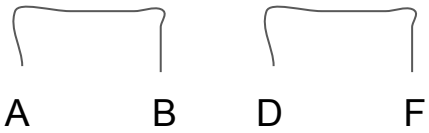
Kruskal’s algorithm - weight 1



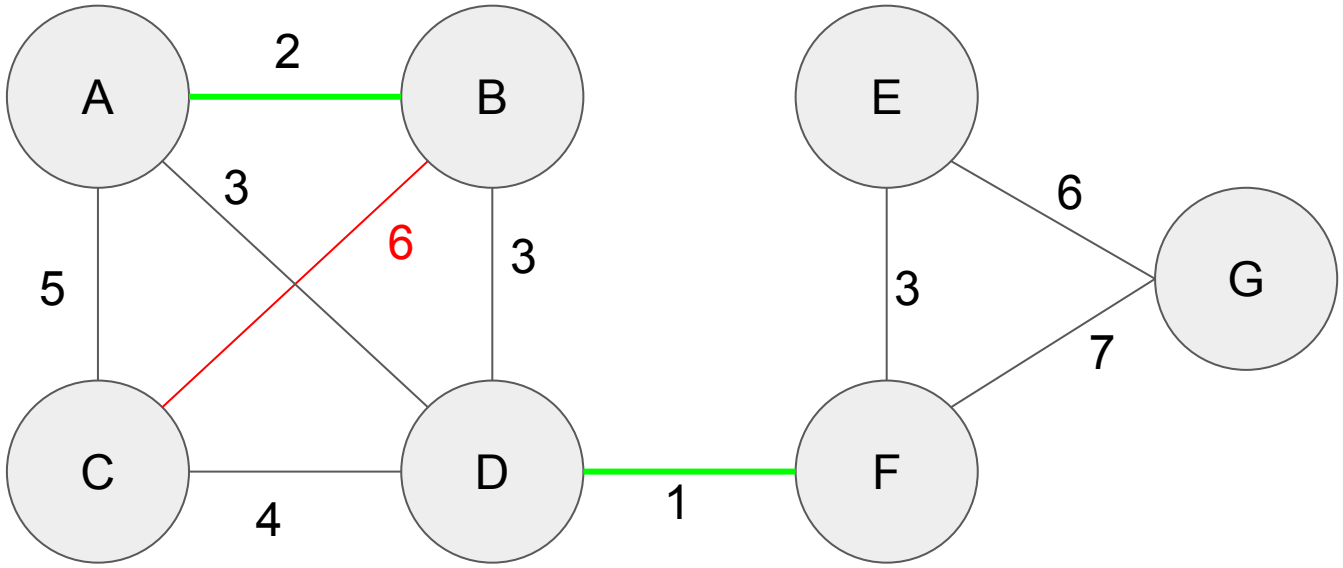


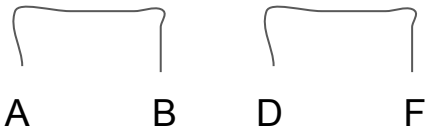
Kruskal’s algorithm - weight 2



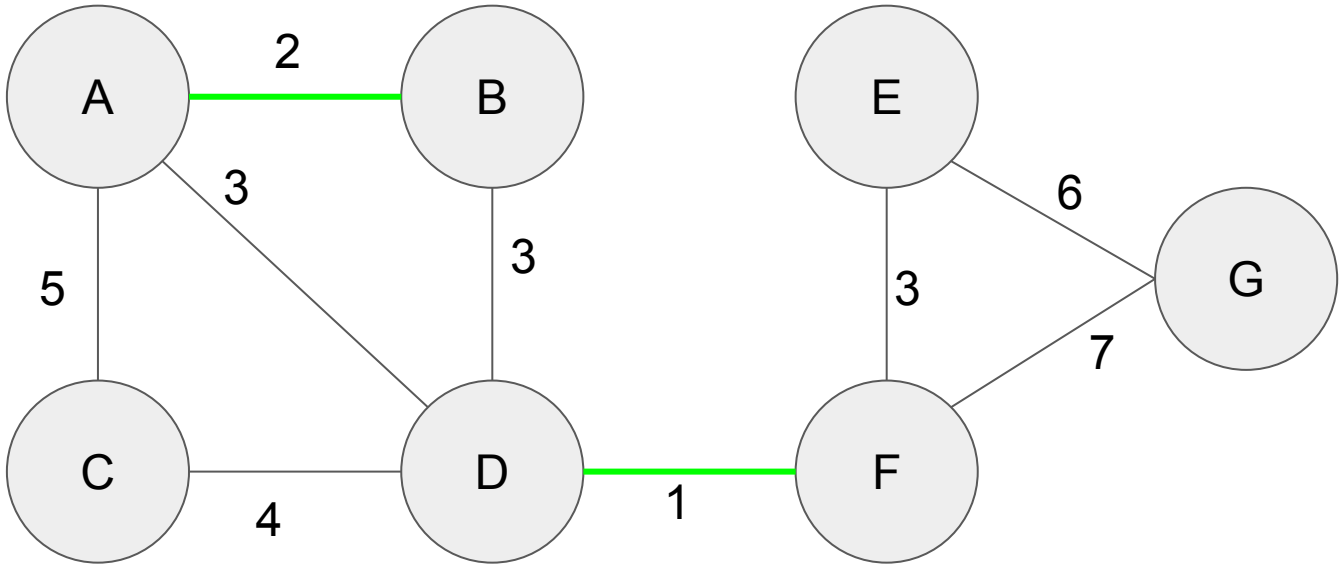


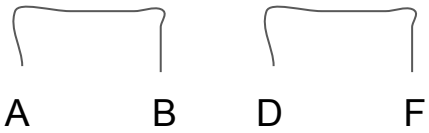
Kruskal's algorithm - weight 2



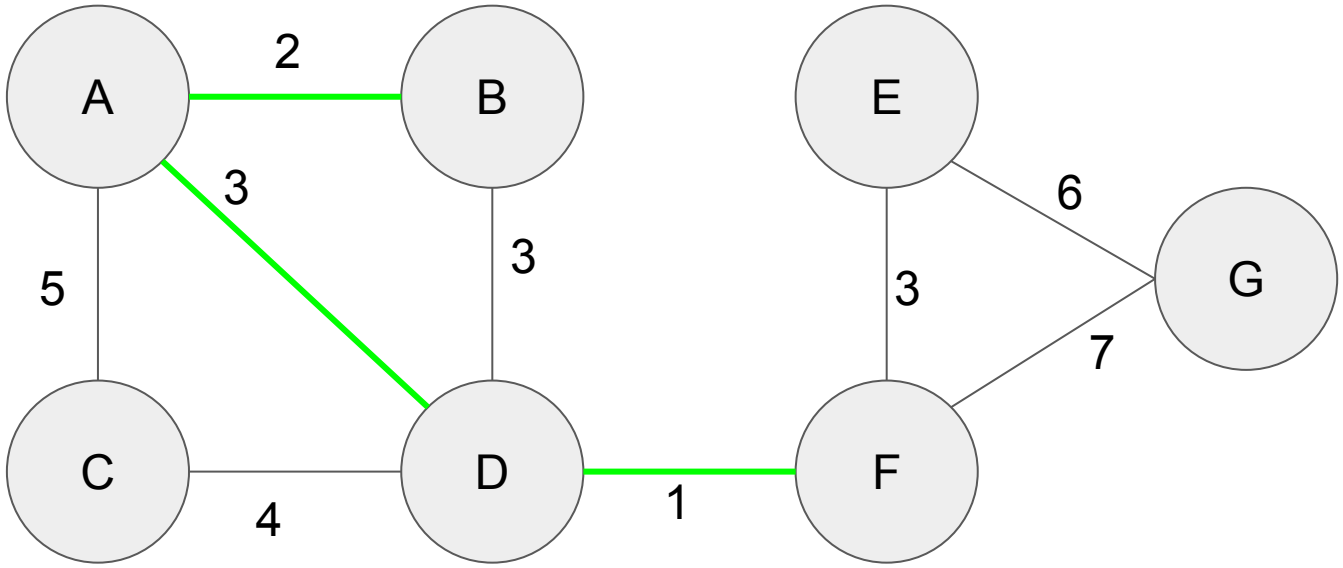


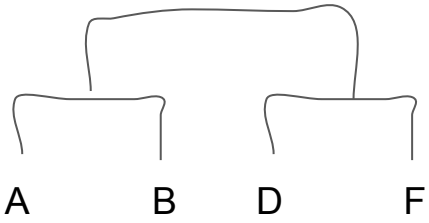
Kruskal’s algorithm - weight 2



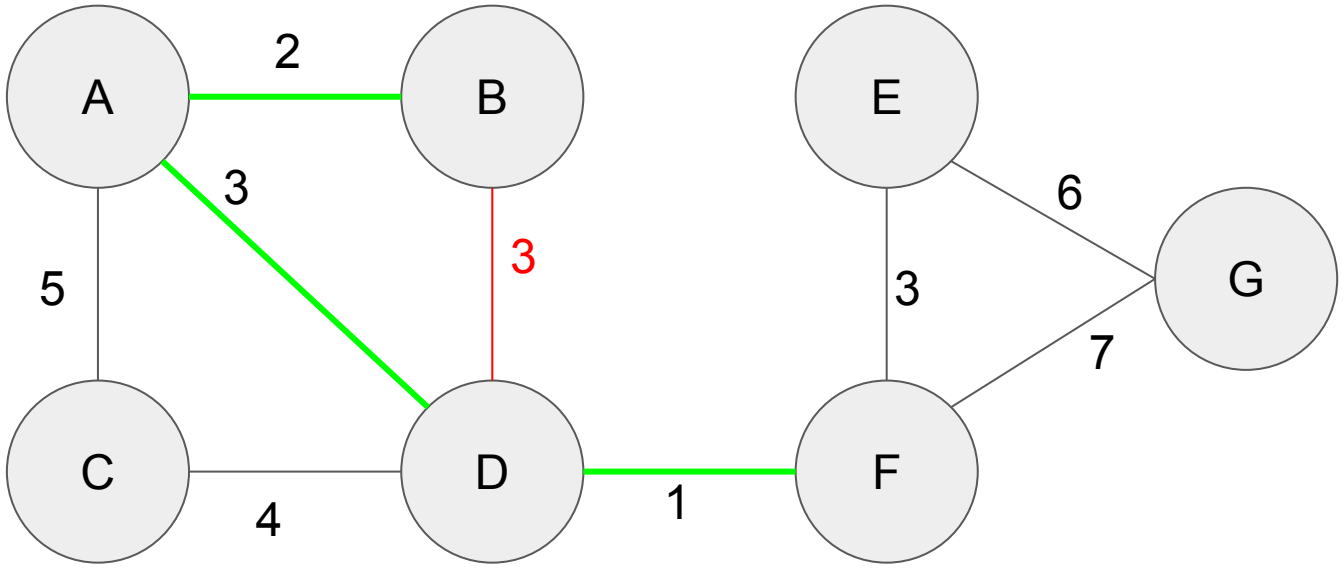


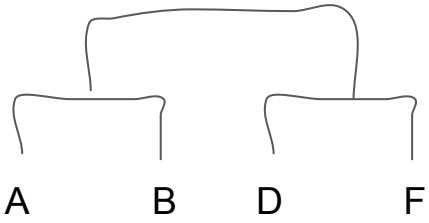
Kruskal’s algorithm - weight 3



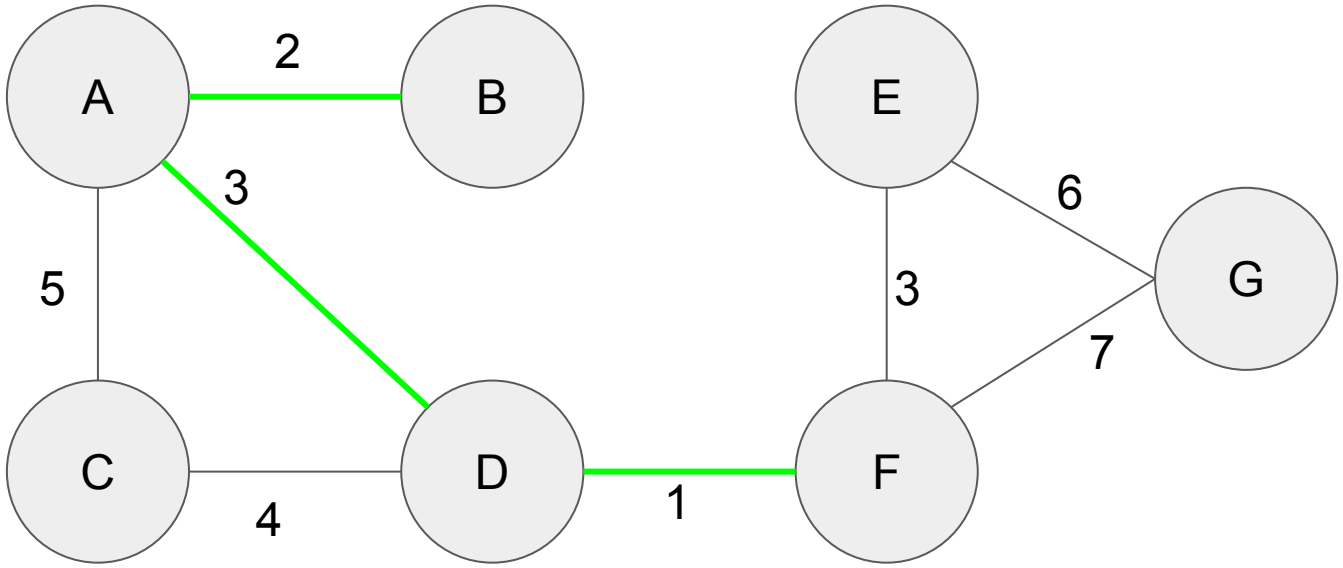


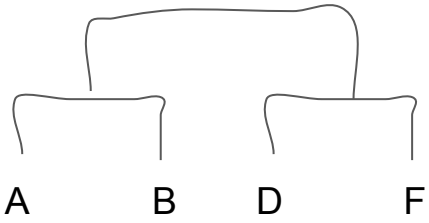
Kruskal's algorithm - weight 3



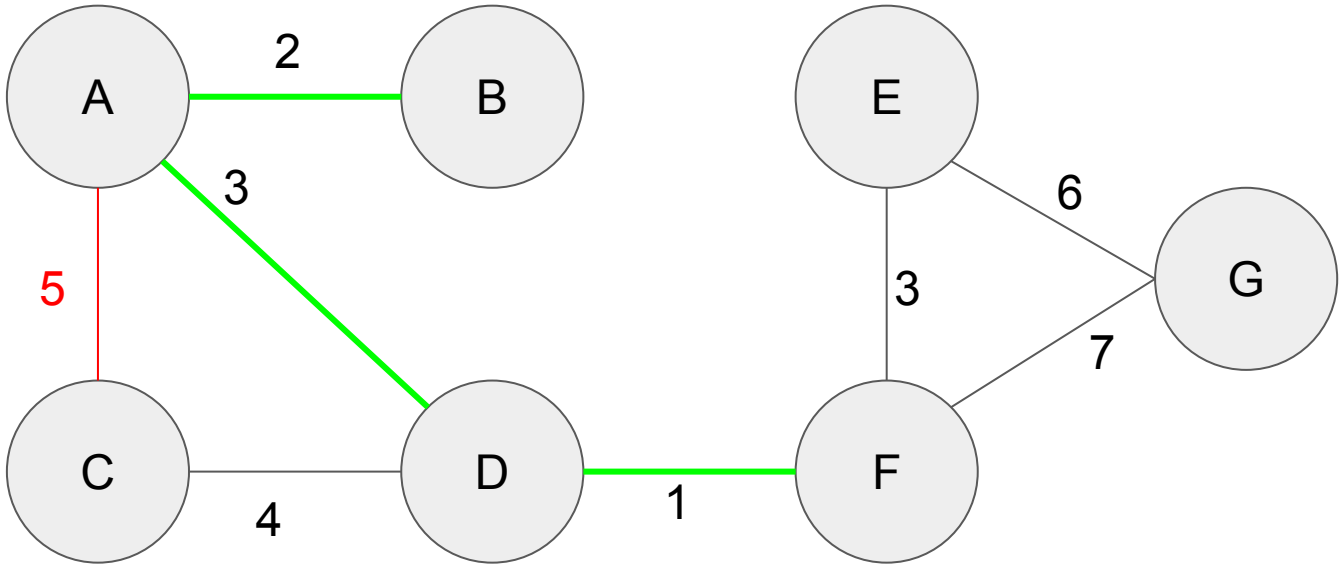


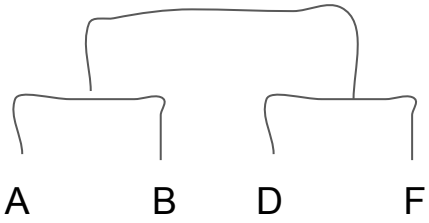
Kruskal’s algorithm - weight 3



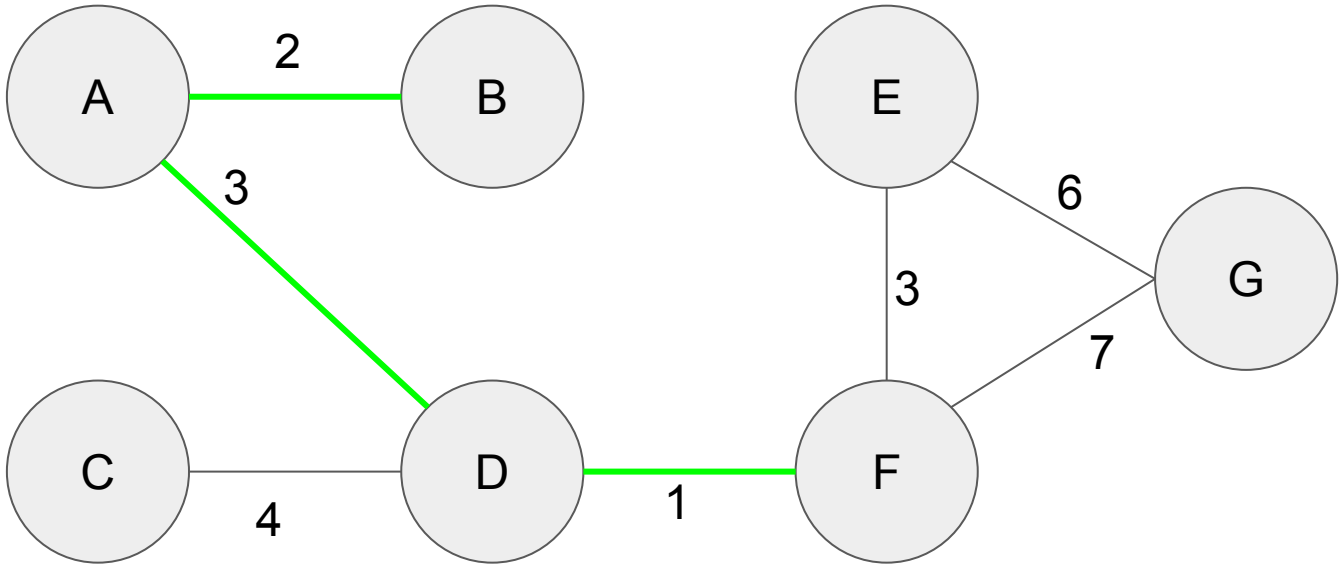


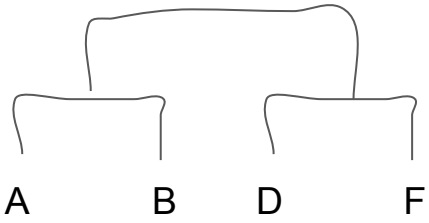
Kruskal's algorithm - weight 3



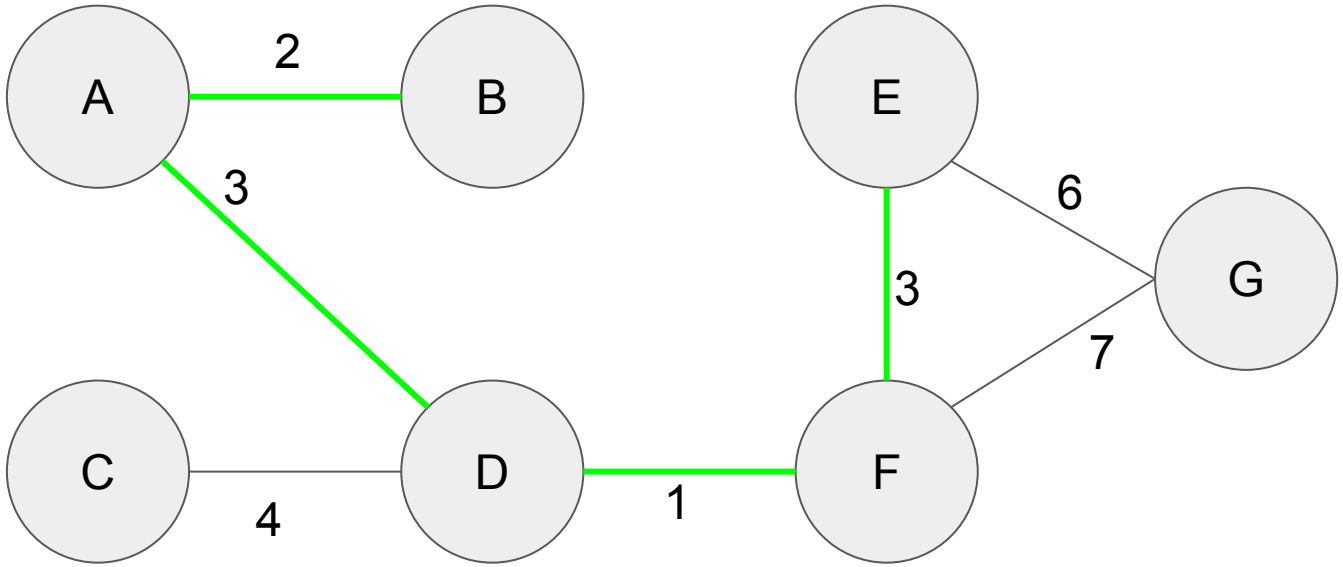


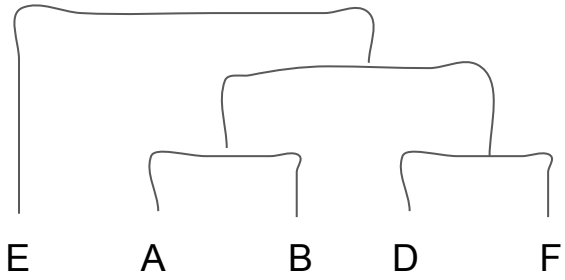
Kruskal’s algorithm - weight 3



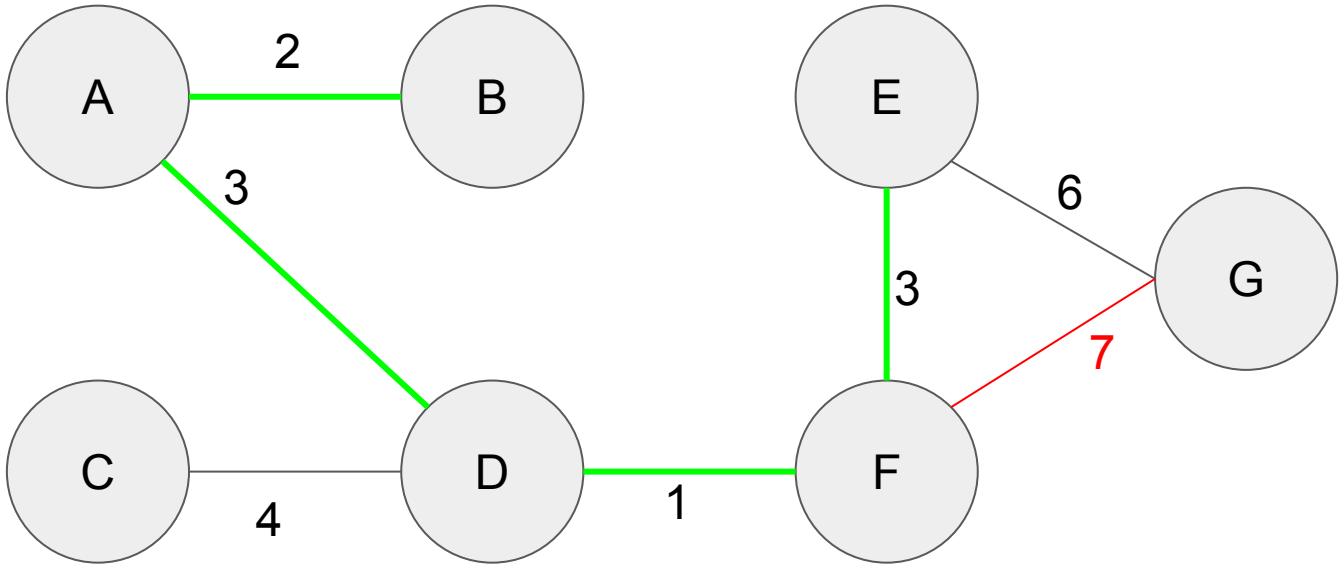


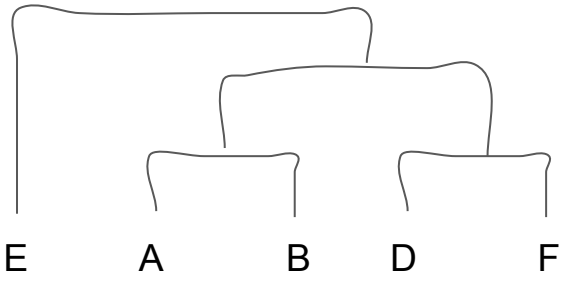
Kruskal’s algorithm - weight 3



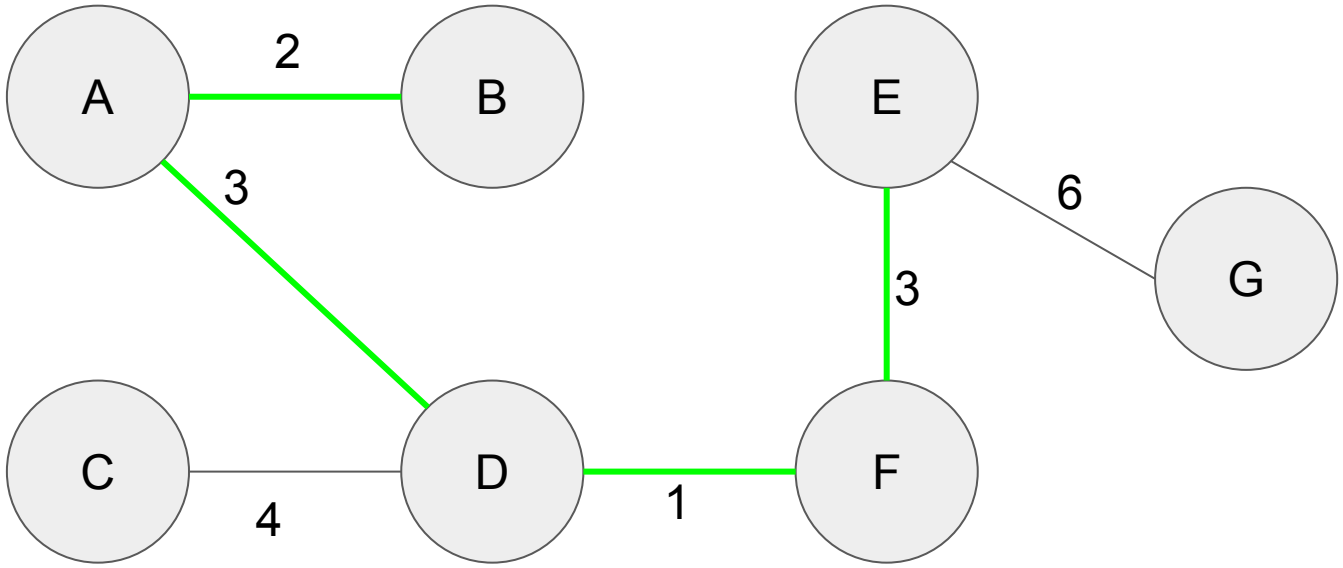


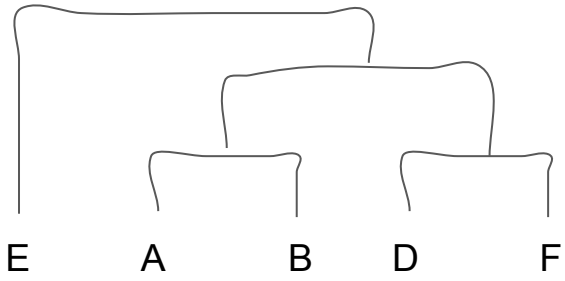
Kruskal’s algorithm - weight 3



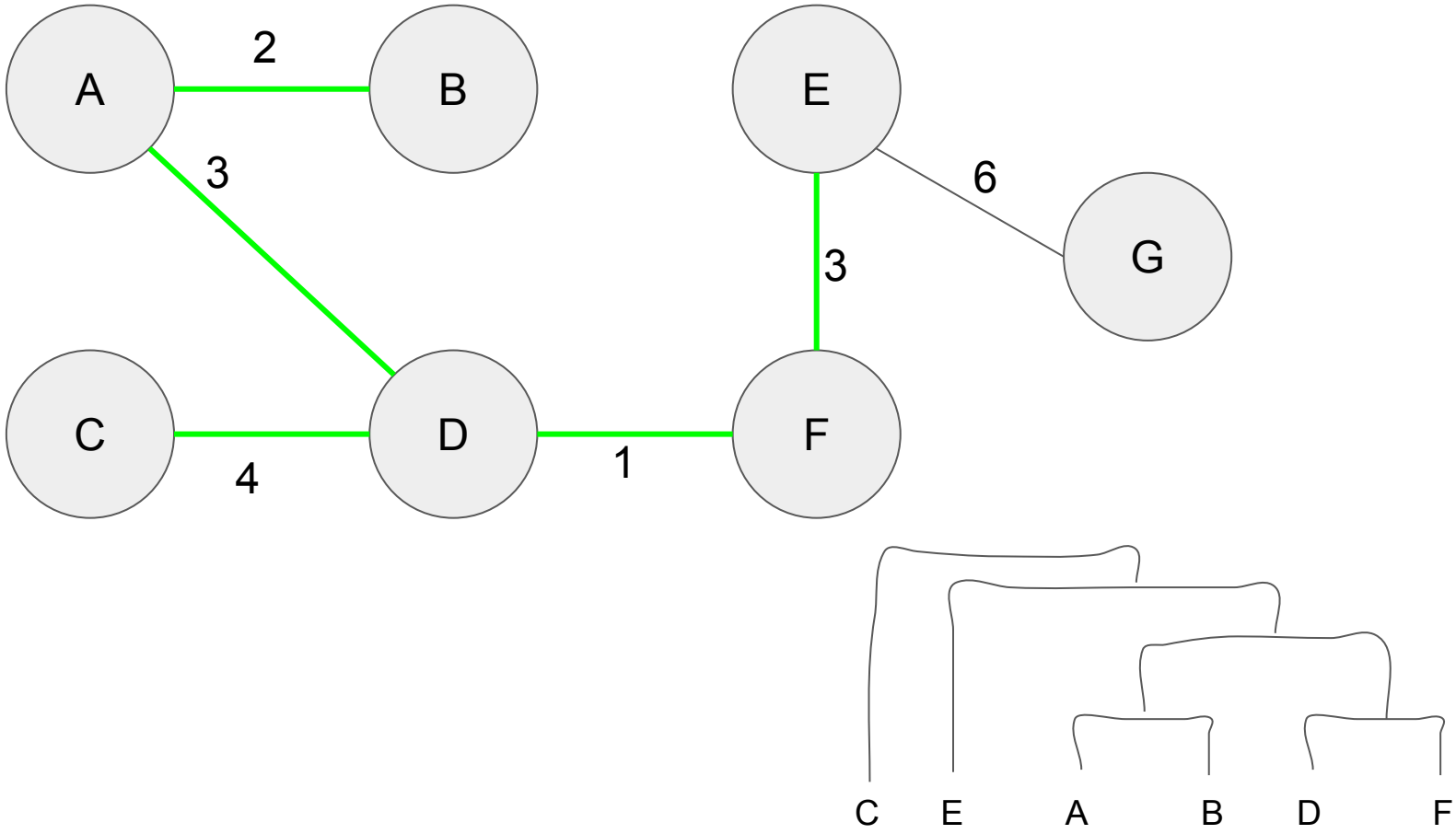


Kruskal’s algorithm - weight 3





Kruskal’s algorithm - weight 4



Kruskal’s algorithm - weight 6