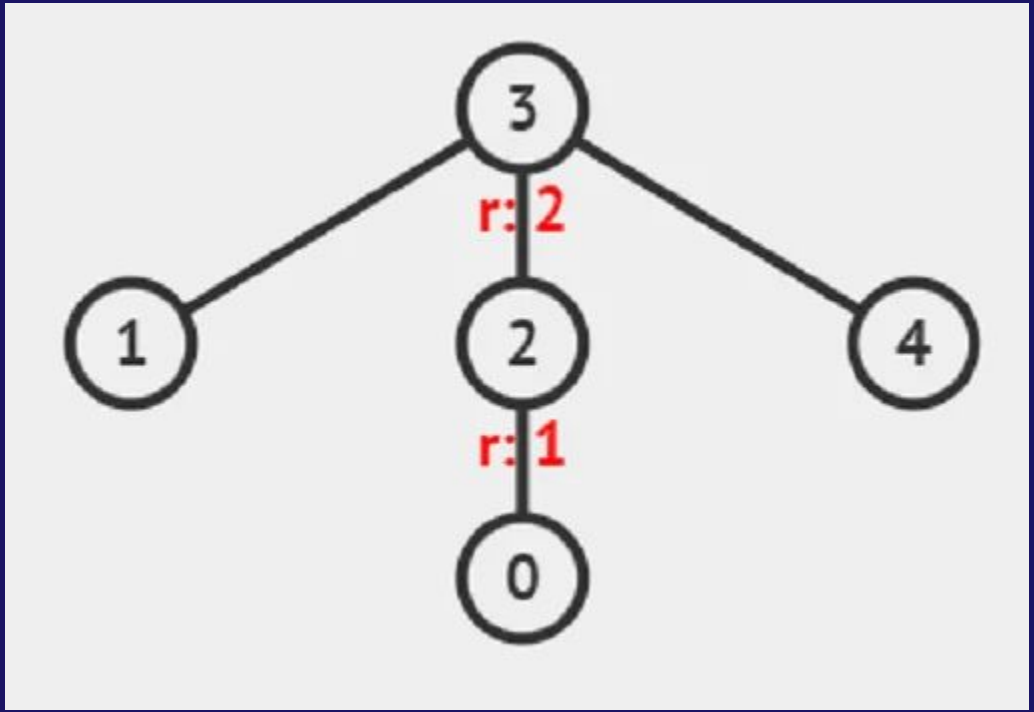
CS2040

Tutorial 6: Union-Find

Nicholas **Russell** Saerang (russellsaerang@u.nus.edu)

Let’s start with...




:D
DISJOINT SET ADT

UFDS (Union Find Disjoint Sets)

Two basic operations:

- Find: Given an element, find which set it is in.
  Time complexity is $O(\alpha(n))$*.
- Merge: Given two sets, merge them into one set.
  Time complexity is $O(\alpha(n))$*.

Application operation:

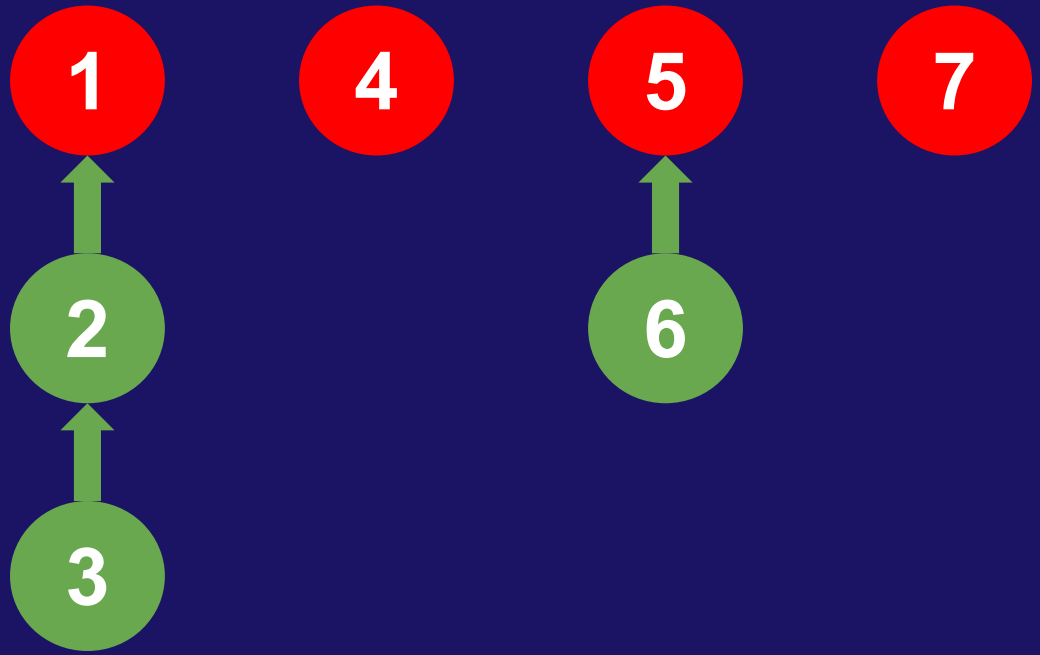
- Check if two elements belong to the same set.
  Time complexity is also $O(\alpha(n))$*.

* Path compression and union-by-rank heuristics are used.

UFDS (Union Find Disjoint Sets)

Representative

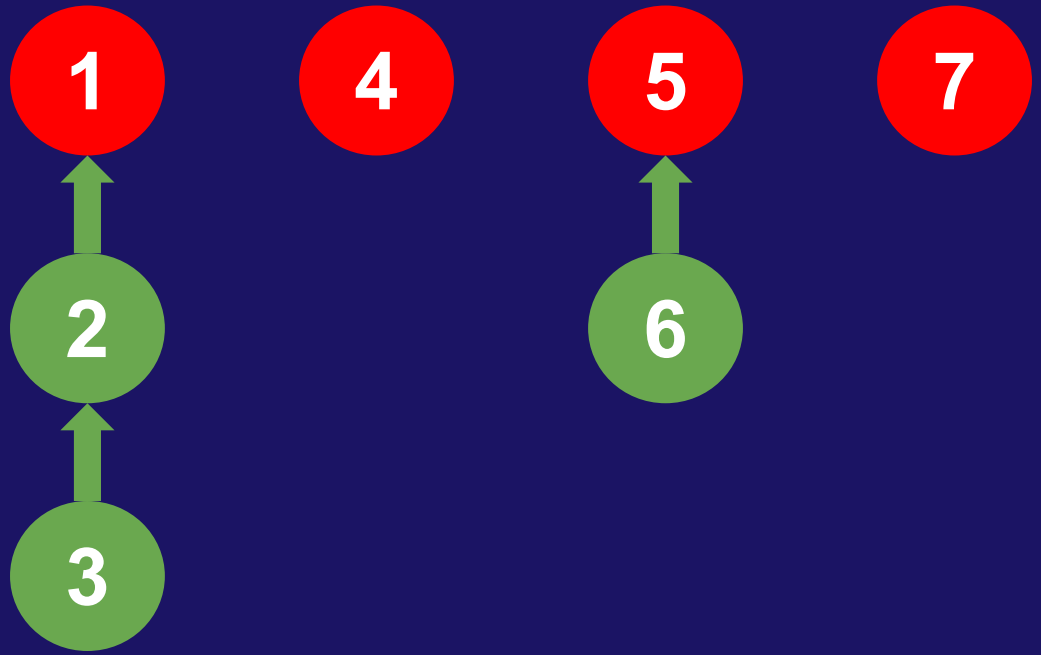
Each set is identified by a representative.



UFDS (Union Find Disjoint Sets)

Find

Given an element, we can find the representative of the set it is in.

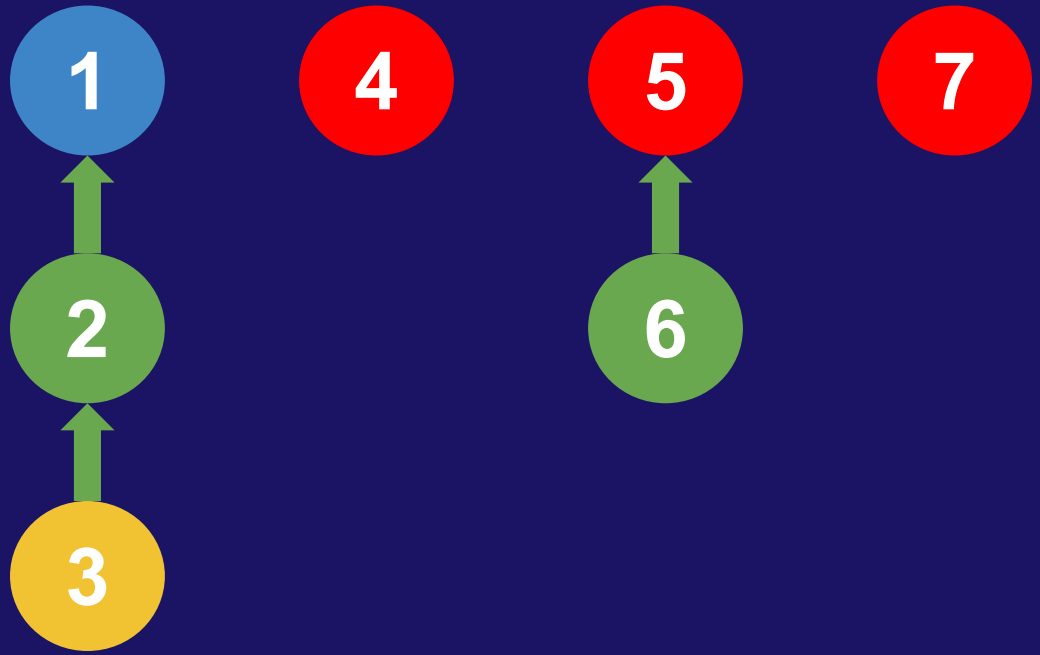


UFDS (Union Find Disjoint Sets)

Find

Given an element, we can find the representative of the set it is in.

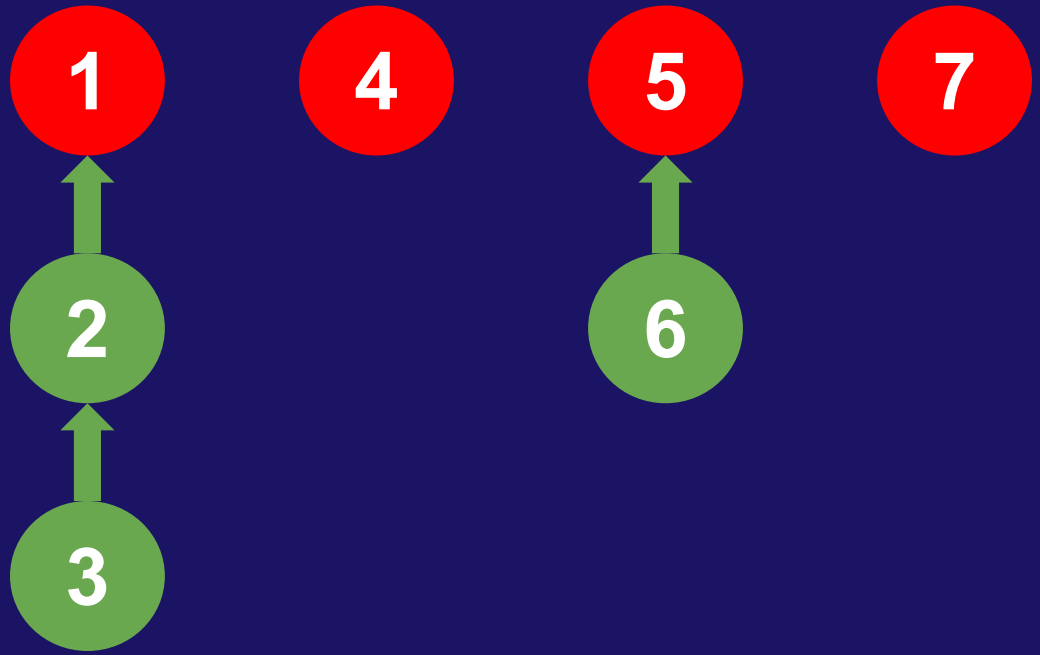
findSet(3) → 1



UFDS (Union Find Disjoint Sets)

Merge

When we need to merge two sets, we do the following two steps.

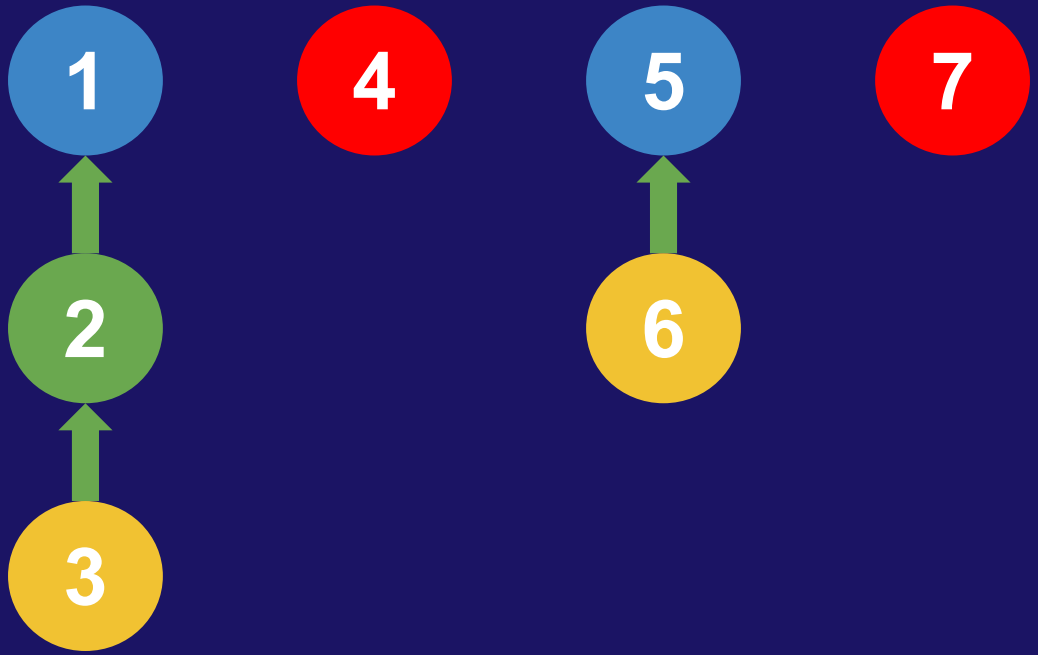


UFDS (Union Find Disjoint Sets)

Merge

First, find the representative of the two sets that we want to merge.

unionSet(3,6)

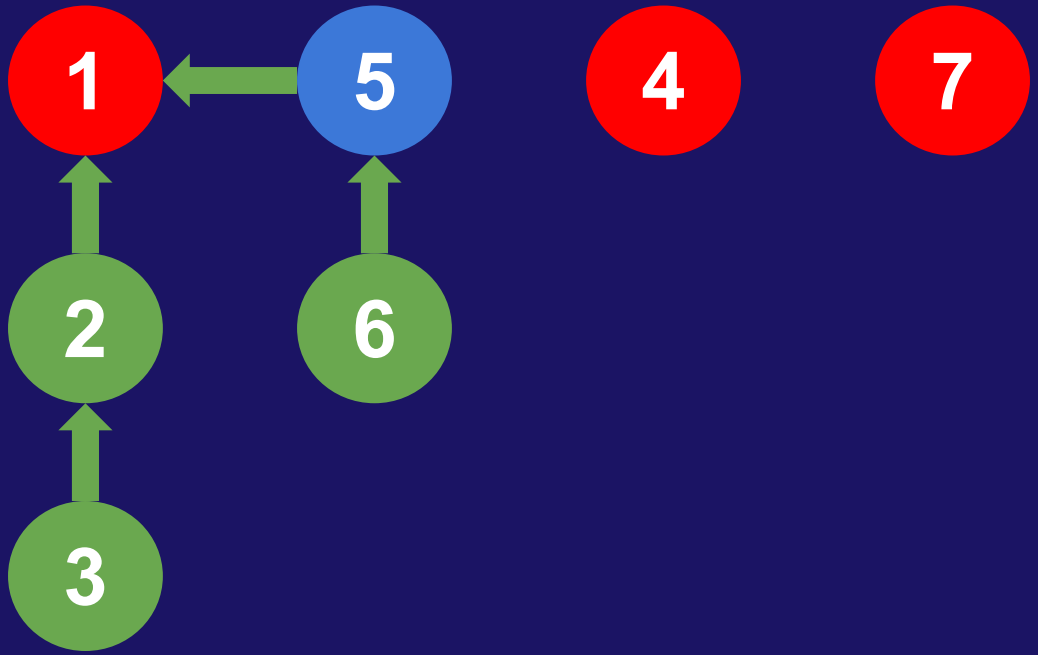


UFDS (Union Find Disjoint Sets)

Merge

Second, make one of them representative of both sets.

unionSet(3,6)



Optimizations

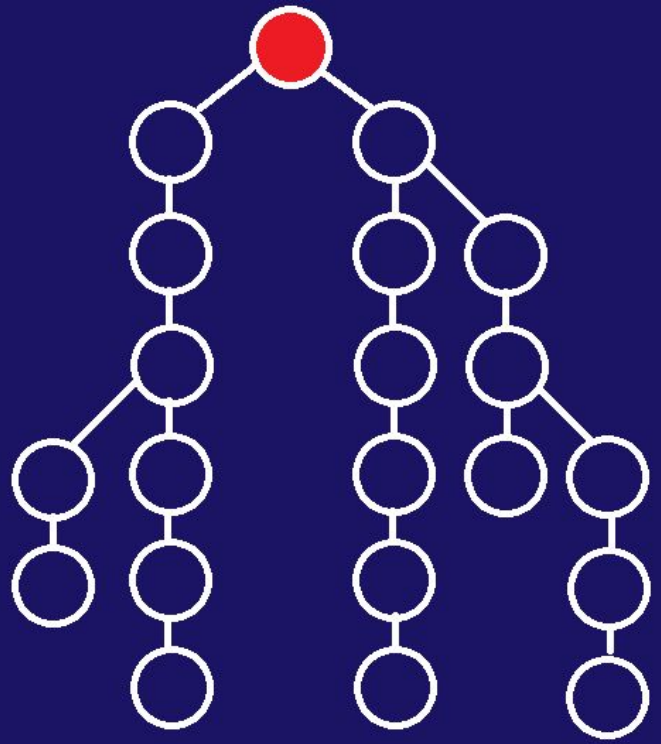
We can optimize our UFDS in a few ways:

- Path compression
- Union by rank
- Union by size (off-CS2040(S))

Motivation

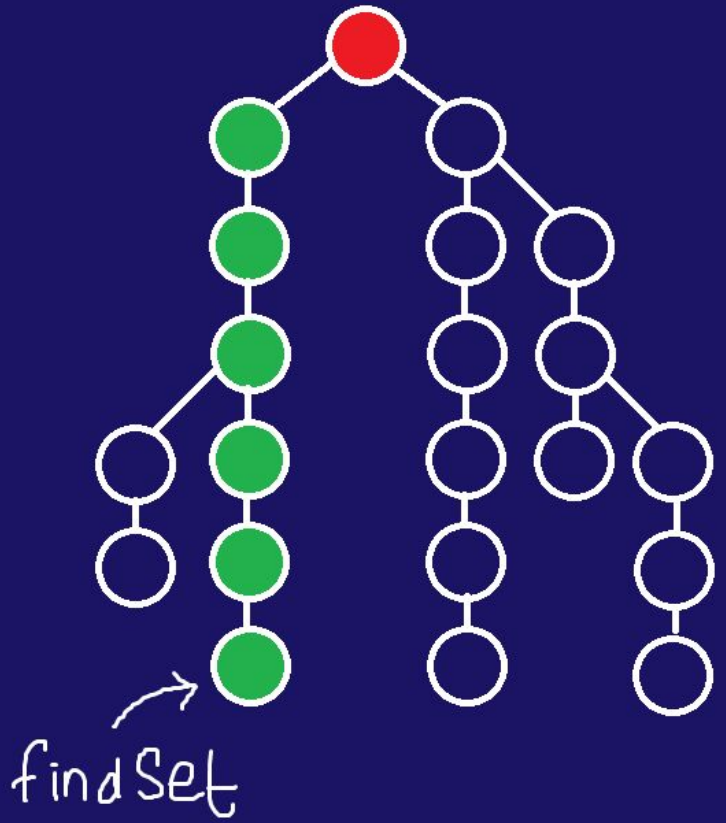
If we are not careful, the height of the UFDS structure may be large.

The findSet() operation will be slow → O(N) where N is the number of vertices

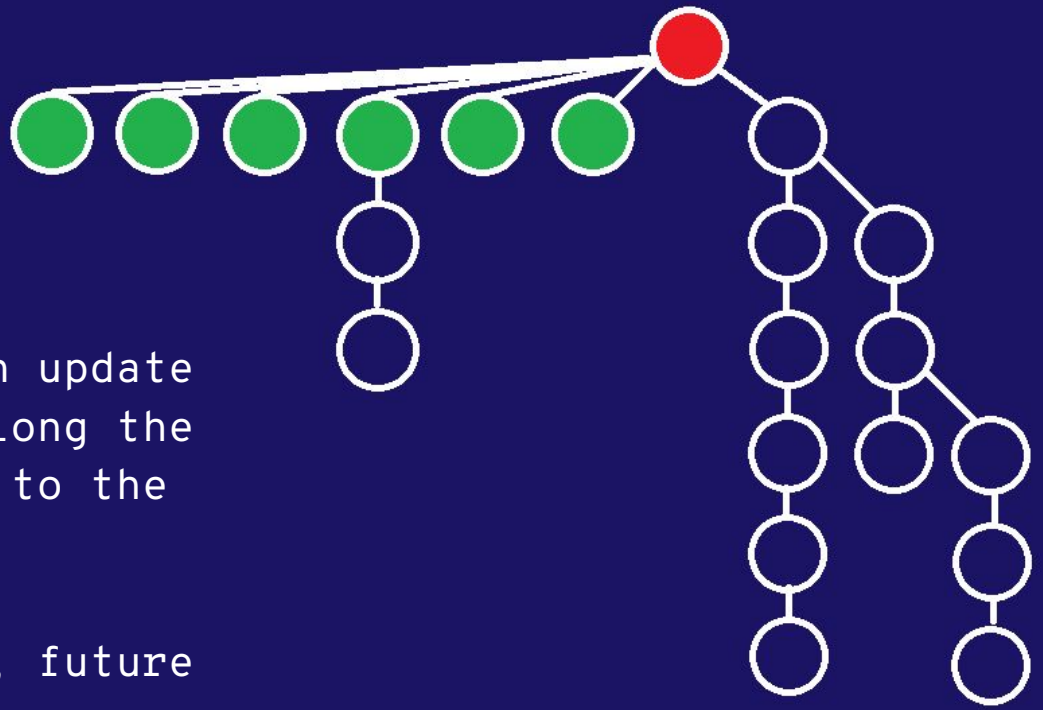


Path Compression

When we call the findSet operation on a node, all nodes along the path towards the representative will eventually point to the representative.



Path Compression

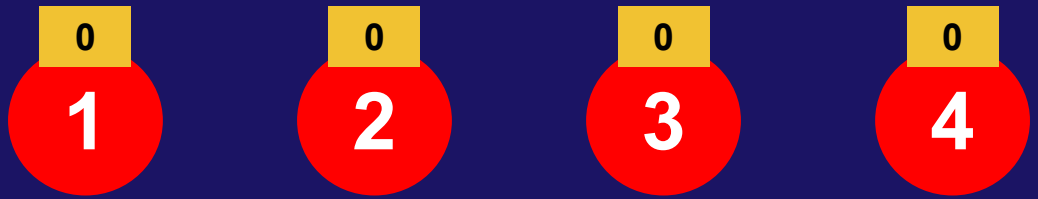


After we find the representative, we can update all the nodes found along the way to point directly to the representative.

With path compression, future finds will be much faster.

Union by Rank

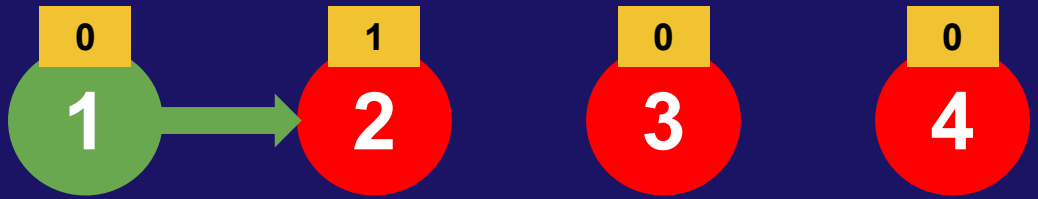
Initially, every representative has rank 0.



Union by Rank

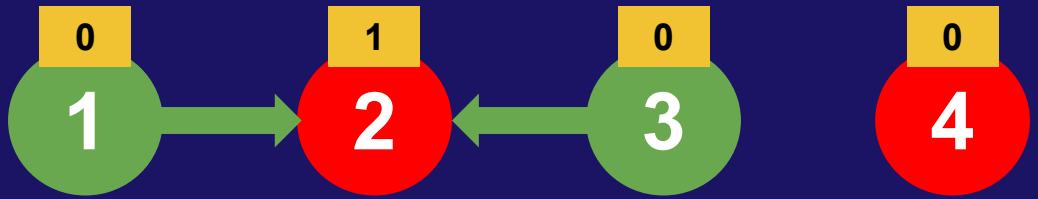
When merging two representatives of the same rank, merge in either direction, and increase the rank of the new representative by 1.

E.g. `unionSet(1,2)`



Union by Rank

When merging two representatives of the different ranks, merge the one with the smaller rank to that with the greater rank.
Rank of new representative is unchanged.
E.g. `unionSet(2,3)`



Analysis of Union by Rank

If we are using union by rank only (without path compression), notice that the rank actually represents the height of tree rooted at the representative.

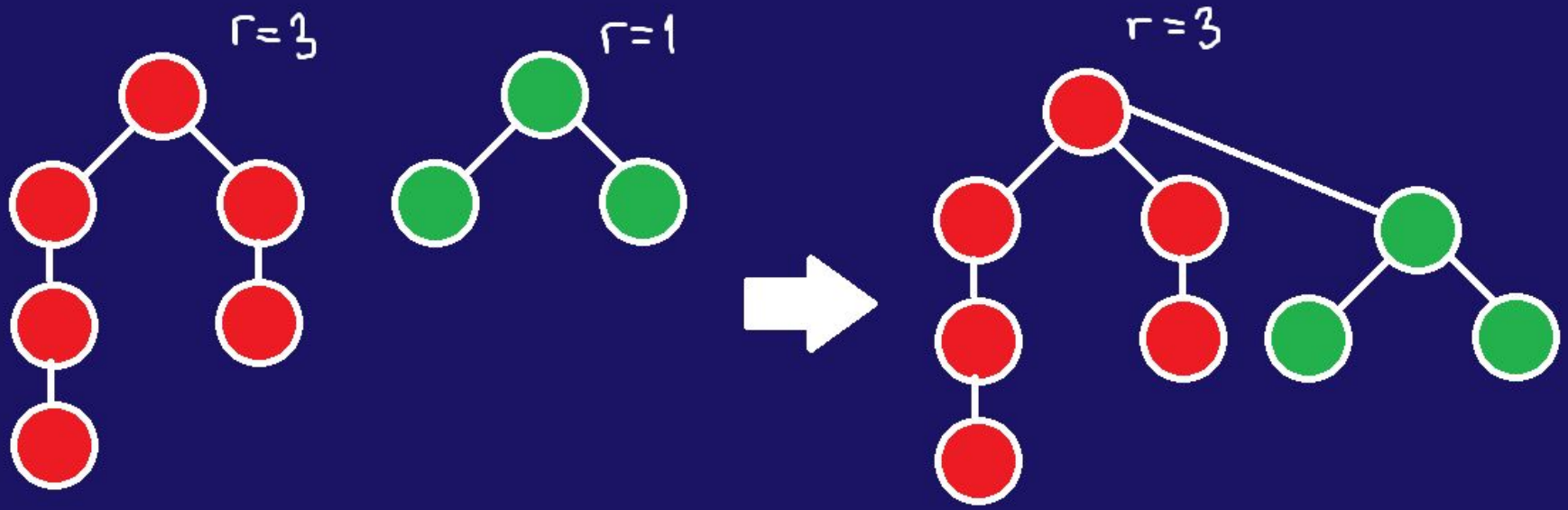
For a representative of rank n, the minimum number of nodes in its tree is n + 1.

Since we increase the rank only when merging trees of equal rank, notice that every time the rank of a representative increases, the number of elements in the tree doubles.

The maximum attainable rank would thus be bounded by the number of time we can double the size of a tree, which would be $O(\log n)$ given n elements.

Analysis of Union by Rank

Thus the maximum height of a tree would be O(log n) given a tree with n elements. As such, if we use union by rank only, both find and merge will run in O(log n).



Time Complexities

Optimization(s)	Time complexity of findSet and unionSet
None	O(N)
Path compression only	O(N)
Union-by-rank	O(log N)
Path compression + union-by-rank	O(α(N))

Inverse Ackermann Function

The inverse Ackermann function, $\alpha(N)$, is a function with a very slow growing rate. It can be shown (proof is off-CS2040(S)) that the time complexity for both operations given path compression and union-by-rank is $O(\alpha(N))$.

N	$\alpha(N)$
1	1
3	2
7	3
61	4
$2^{2^{2^{2^{16}}}} - 3$	5

"Effectively" O(1)

01

LARGEST SET

Largest Set

Given a UFDS initialised with n disjoint sets, what is the maximum possible rank h that can be obtained from calling any combination of unionSet(i, j) and/or findSet(i) operations?

Assume that both the path-compression and union-by-rank heuristics are used.

Largest Set

- findSet(i)
  Useless, even instead decreases the rank due to path compression.

- unionSet(i, j)
  The rank of the resulting set will be higher than that of set i and set j if and only if the size of both sets are the same. The resulting set will have 1 rank higher than that of the two sets.

Largest Set

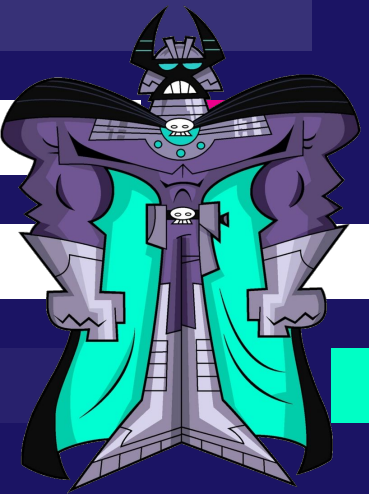
Thus the maximum height is obtained when we call unionSet(i, j) on $\left\lfloor \frac{n}{2} \right\rfloor$ distinct pairs from the n disjoint sets of rank 0 to obtain $\left\lfloor \frac{n}{2} \right\rfloor$ disjoint sets of rank 1.

Similarly, call unionSet(i, j) again on the $\left\lfloor \frac{n}{4} \right\rfloor$ distinct pairs to obtain $\left\lfloor \frac{n}{4} \right\rfloor$ disjoint sets of rank 2.

Repeatedly applying this set of operations, we will eventually obtain one set of rank $\left\lfloor \log_2 n \right\rfloor$ (with some leftovers).

02

INTERGALACTIC WARS



Intergalactic Wars

Problem 2a

Given n warlords and n worlds, where initially, each warlord has one unique world under its domain, support the following two types of queries in sub-O(log n) time:

- Warlord x conquers warlord y. Add all worlds of warlord y under warlord x.
- Answer the query of whether a world z is under a given warlord x.

The first type of query looks like a unionSet operation, and the second type of query looks like a findSet operation. This hints us towards a UFDS solution.

Intergalactic Wars

Problem 2a
Construct a UFDS on the worlds.

Add an additional parameter to every set: the warlord that is in charge of the worlds in the set.

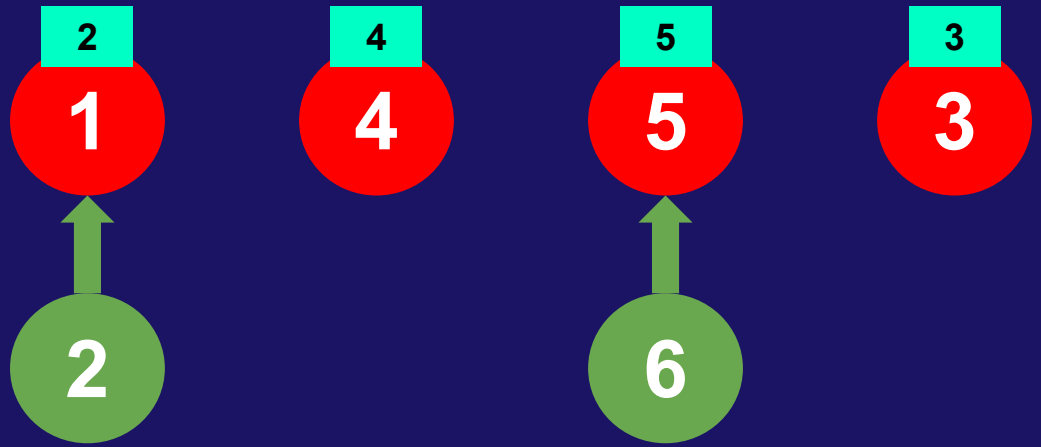
When we merge two sets of worlds together, ensure that the representative of the merged set is updated with the warlord in charge of the worlds in the entire set.

The second type of queries can then be answered by first doing a findSet operation to find the representative of the set, then checking the warlord in charge of all the worlds in the set.

Intergalactic Wars

Problem 2a

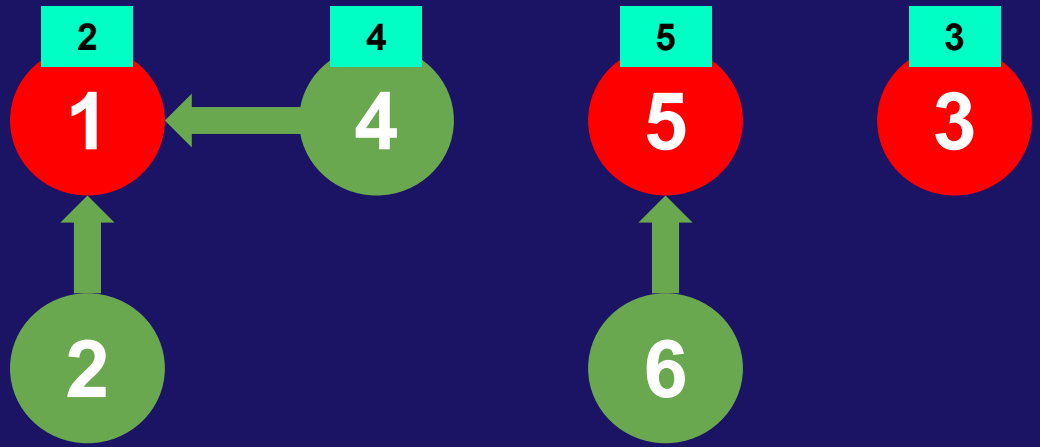
Suppose this is our initial state.



Intergalactic Wars

Problem 2a

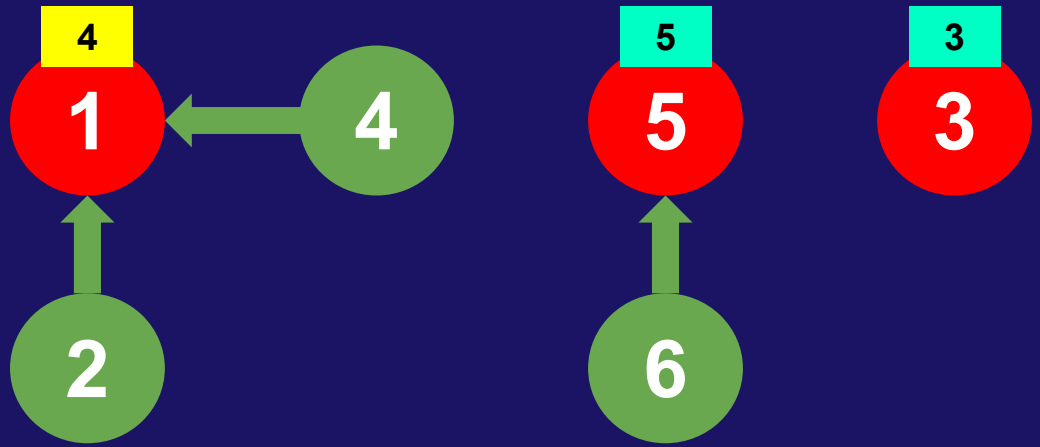
Warlord 4 conquers Warlord 2.
Merge the two sets together using the merge operation in UFDS.



Intergalactic Wars

Problem 2a

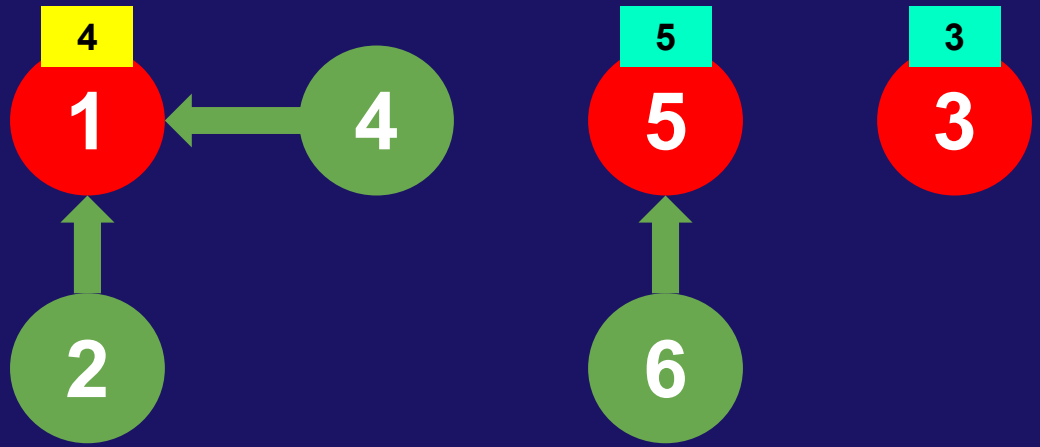
Warlord 4 conquers Warlord 2.
Update the new representative with the new ruler.



Intergalactic Wars

Problem 2a

To check if a world is under a certain warlord, simply find the representative of the set that the world is in, and check the warlord that rules the worlds in that set.



Intergalactic Wars

Problem 2b

Make modifications / additional operations to support an additional type of query: Are all n worlds under warlord x?

Note that since merges are performed, after every merge, the number of different sets of worlds decreases by 1. After n − 1 merges, no more merges can be performed, since all worlds will be under one warlord.

Thus, we can simply check if n − 1 merges have been performed, and then finding the warlord in the representative of the whole set of worlds.

03

NUMBER CANDIES

Number Candies

There are m types of candies. Initially, you have no candies.

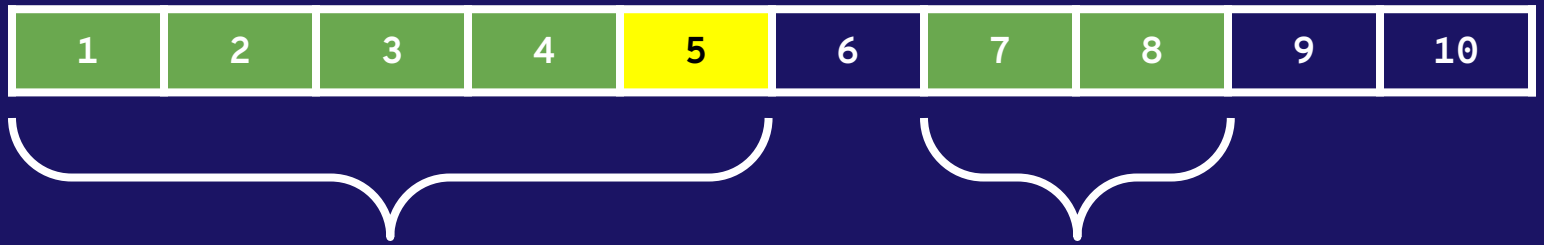
You will buy h candies. After buying each candy, output the longest unbroken sequence of candies you have from candy 1.

e.g. If you have the following sweets, the longest unbroken sequence has length 4.

Number Candies

Observations:

- Each unbroken sequence of candies is like a group of candies.
- When a new candy is bought, if it's next to a candy that was bought before, it joins the unbroken sequence / group of candies. e.g. Buys candy 5 (shown below)
  - Joining a group is like the 'merge' operation in UFDS!

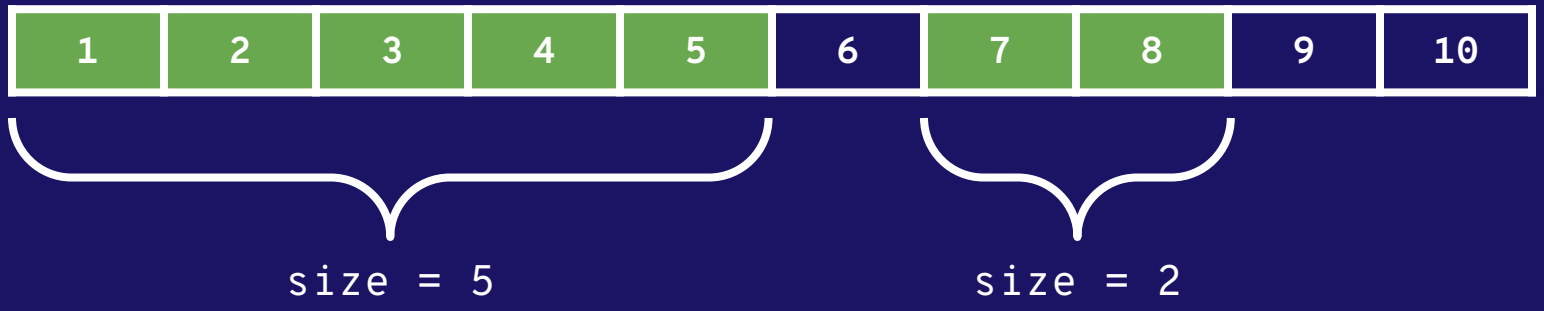


Number Candies

Observations:

- What is the longest unbroken sequence of candies that you have starting from candy 1?
  The number of candies in the same group as candy 1!

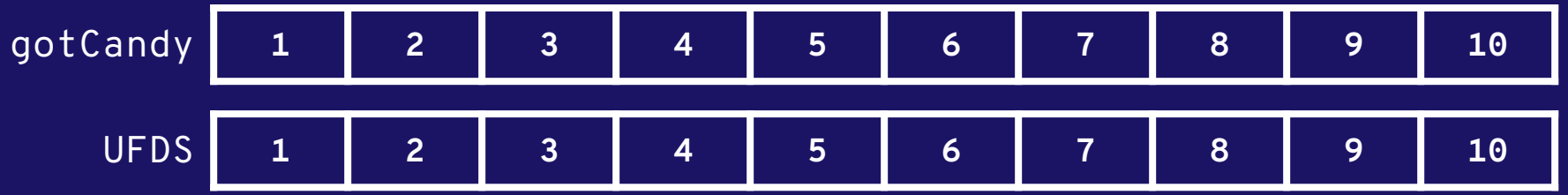
  Augment the UFDS to maintain the size of each set. (You can also use max instead of size)



Number Candies

Suppose we have an array to keep track whether a candy is bought and the UFDS itself.

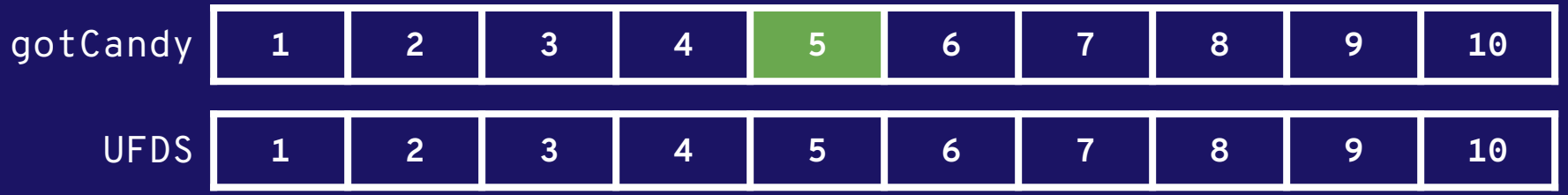
We shall simulate some operations over time.



Number Candies

Suppose we have an array to keep track whether a candy is bought and the UFDS itself.

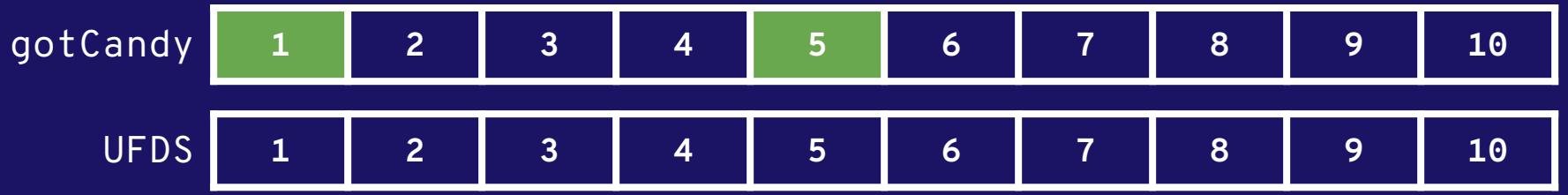
Buy candy 5. Since we don't have 4 and 6, do not merge.
Since gotCandy(1) = false, longest = 0.



Number Candies

Suppose we have an array to keep track whether a candy is bought and the UFDS itself.

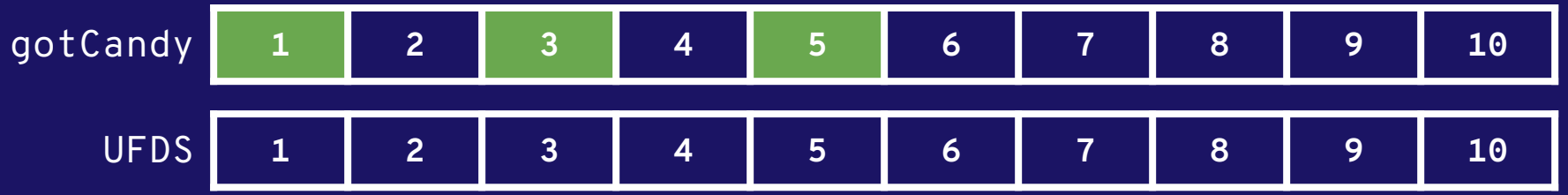
Buy candy 1. Since we don't have 2, don't merge. longest = 1.



Number Candies

Suppose we have an array to keep track whether a candy is bought and the UFDS itself.

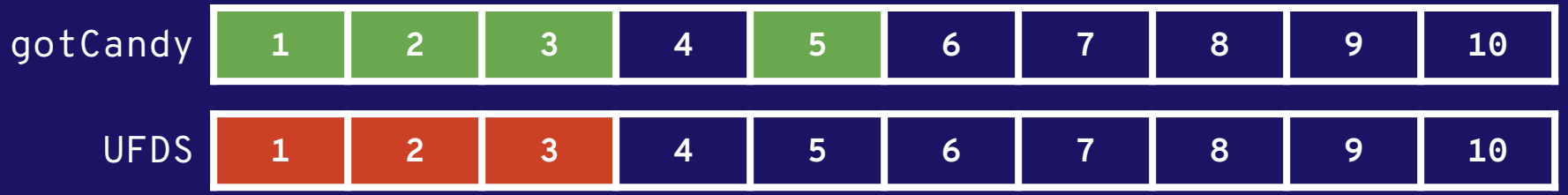
Buy candy 3. Since we don't have 2 and 4, do not merge.
longest = still 1.



Number Candies

Suppose we have an array to keep track whether a candy is bought and the UFDS itself.

Buy candy 2. Since we have 1 and 3, merge with both. Size of set of 1 is 3, so longest = 3.



Number Candies

Suppose we have an array to keep track whether a candy is bought and the UFDS itself.

Buy candy 6. Since there is 5, merge with 5. longest = 3.

Number Candies

Suppose we have an array to keep track whether a candy is bought and the UFDS itself.

Buy candy 4. Since we have 3 and 5, merge with both sets. Size of set of 1 = 6 so longest = 6.

Number Candies

Suppose we have an array to keep track whether a candy is bought and the UFDS itself.

Note that each merge can be done in $O(\alpha(h))$ time. Checking the size of the set of 1 also takes $O(\alpha(h))$ time.

04

FUNFAIR RIDE

Funfair Ride

There is a **circular** ring of n seats (seat 1 is next to seat 2 and n, seat 2 is next to seat 1 and 3, and so on). m children will arrive one by one.

Each child has a desired seat. For each child, do the following:

- If the desired seat is not taken, assign that seat to the child.
- If the desired seat is taken, ‘linear probe’ for the next empty seat, and assigned that seat to the child.

For each child, output the seat assigned to the child.

Funfair Ride

The idea is very similar to Problem 3.

- A sequence of consecutive children form a ‘group’
- If a child wants a seat in the middle of the group, the child gets the seat at the end of the group.

Funfair Ride

The idea is very similar to Problem 3.

- A sequence of consecutive children form a ‘group’
- If a child wants a seat in the middle of the group, the child gets the seat at the end of the group.

Funfair Ride

The idea is very similar to Problem 3.

- If a new seat is assigned and adjacent seats are occupied, merge the two groups of seats together.
  → **Merge operation in UFDS.**

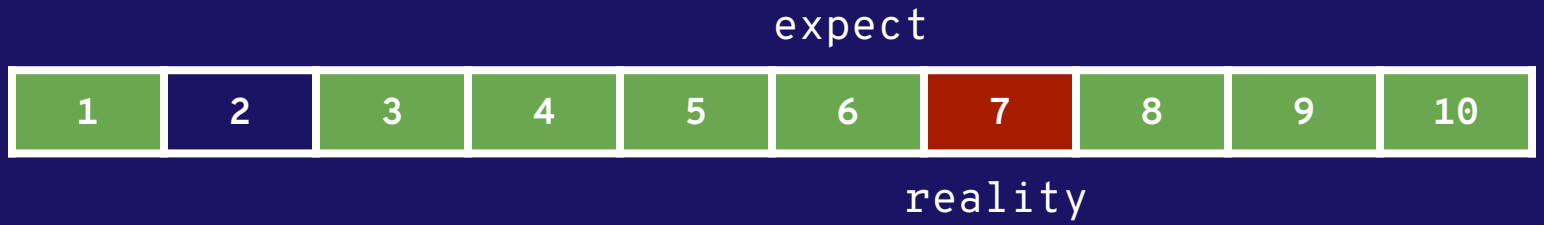
- How to find seat assigned if child wants a seat in a group?

Funfair Ride

The idea is very similar to Problem 3.

- How to find seat assigned if child wants a seat in a group?
  Maximum seat number in that group + 1

- What if that exceeds n?



Funfair Ride

The idea is very similar to Problem 3.

- What if that exceeds n?
  If seat 1 is not assigned, then seat 1 it is!
  Otherwise, (maximum seat number of group with seat 1) + 1.

- Augment UFDS to maintain the maximum seat number in each group.

UFDS Merge

What you get from the lecture:

```
function merge(x, y)
    x = find(x)
    y = find(y)

    if rank[x] < rank[y]
        parent[x] = y

    else
        parent[y] = x

        if rank[x] == rank[y]
            rank[x] ← rank[x] + 1
```

UFDS Merge

What you actually need for Problem 4 (and 3):

```
function merge(x, y)
    x = find(x)
    y = find(y)

    if rank[x] < rank[y]
        parent[x] = y
        max[y] = max(max[x], max[y])
    else
        parent[y] = x
        max[x] = max(max[x], max[y])
        if rank[x] == rank[y]
            rank[x] ← rank[x] + 1
```

UFDS Merge

Extra: what if we need to maintain size of the set instead?

Not provided. For you to think about :)

```
function merge(x, y)
    x = find(x)
    y = find(y)

    if rank[x] < rank[y]
        parent[x] = y
        ???
    else
        parent[y] = x
        ???
        if rank[x] == rank[y]
            rank[x] ← rank[x] + 1
```

THE END!