Disjoint Sets

Union Find / Quick Find (Based on id's)

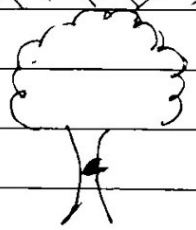
To start:
- lst = [0,1,2,3,4] - implicit form
- (0) (1) (2) (3) (4) - explicit form

Ω(n) union(0,2) -> (0) (1) (0) (3) (4)
O(n) union(0,4) -> (0) (1) (0) (3) (0)

```
def union(lst, a, b):
    for i in range(len(lst)):
        if lst[i] == lst[b]:
            lst[i] = lst[a]
```

Ω(1)
O(1) connected(2,4) -> lst[2] == lst[4] = True

Quick Union: (Basic Tree, Based on root's)



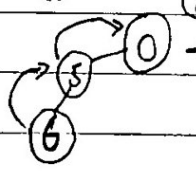
To start:
- lst = [0,1,2,3,4,5,6] - implicit form
- (0) (1) (2) (3) (4) - explicit form
- (5) - (6)

Ω(1) union(0,3) -> (5) - (0) - (6) (3) (1) (2) (4) -> lst[root(3)] = root(0)
O(n)

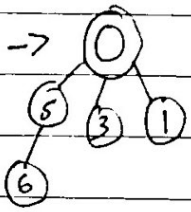
union(3,1) -> (6) - (5) - (3) - (1) (2) (4) -> lst[root(1)] = root(3)

```
def root(lst, a):
    while (lst[a] ! a:
        a = lst[a]
    return a
```

Ω(1)
O(n) root(6) ->



Ω(1)
O(n) connected(5,4) ->



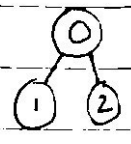
(2) (4) ->

```
def connected(lst, a, b):
    root_a = root(lst, a)
    root_b = root(lst, b)
    return root_a == root_b
```

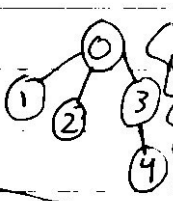
* Can do the same thing with union by height but has issues with integrating with path compression.

Weighted Quick Union/Union by Size (Attach smaller to larger, Tree Based)

Ω(1)
O(n) { union(0,3) ->
Θ(n log n)

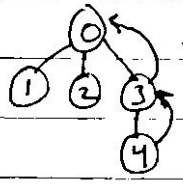


->



```
def union(lst, a, b)
root_a = root(lst, a)
root_b = root(lst, b)
if sz[root_a] ≤ sz[root_b]:
    lst[root_b] = root_a
    sz[root_a] += sz[root_b]
else:
    lst[root_a] = root_b
    sz[root_b] += sz[root_a]
```

Ω(1)
O(n) { root(4) ->

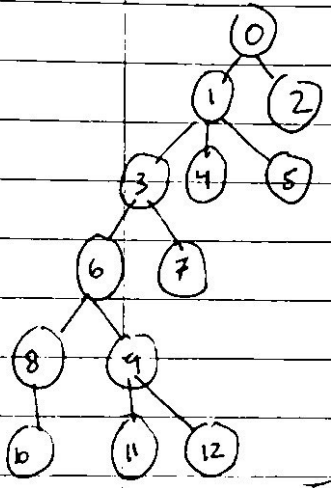


->

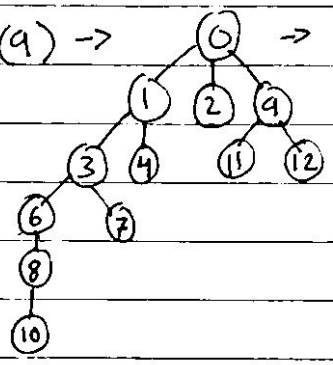
```
def root(lst, a):
    while lst[a] != a:
        a = lst[a]
    return a
```

Ω(1)
O(n) { connected(0,4) -> root(a) == root(b)

Path Compression



root(a) ->



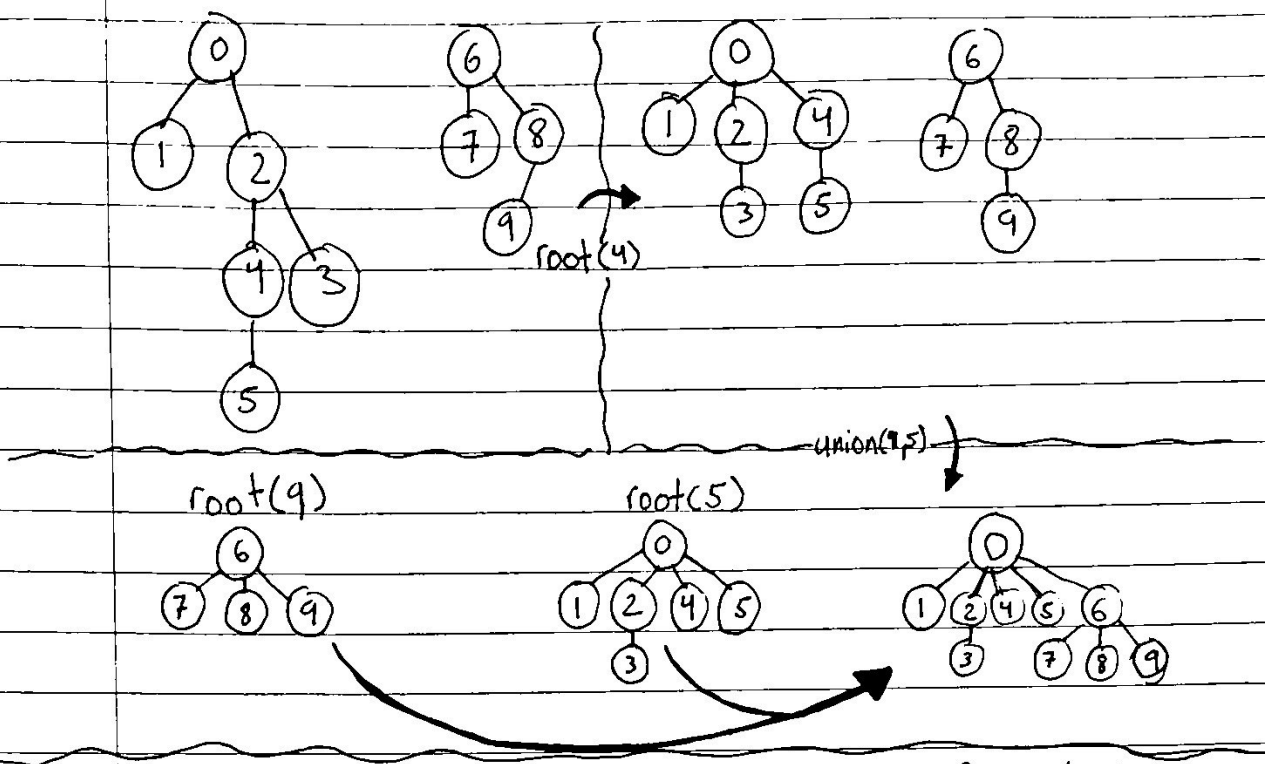
->

```
def root(lst, a):
    while lst[a] != a:
        lst[a] = lst[lst[a]]
        a = lst[a]
```

union ->

```
def union(lst, a, b):
    root_a = root(a)
    root_b = root(b)
    lst[root_b] = root_a
```

Visualize Path Compression:



* When we find the root of a node we drag it up & attach it to the root directly to flatten the tree. This makes future look ups faster.

Note: root happens as well when we union two nodes, those calls to root() also help to flatten the tree.

* Path compression can be used in coliboration with other methods to enhance it. A typical one is Weighted Path Compression as it works well together. However, union by height does not work well with path compression in its basic form since path compression changes the tree heights so you may have to recompute them. We can instead use an estimate of the height instead to solve this problem. When we do this estimation it becomes just as efficient as union by size in theory.

* Using path compression the amortized cost of nearly constant. This is because path compression greatly limits the height of a tree especially after many root() calls. Even say that the tree is 4 deep then it takes 3 lookups to find the root which is essentially O(1). So finally O(4) = O(1).

Summary:

- The purpose of disjoint set problem is to efficently support the root and union operations.
- Used to track elements partioned into non-overlapping sub-sets.
- Implicit = the array representation
  Explicit = the node drawing representation
- 4 main techniques: (some may be combined)
  1. Union-Find
  2. Quick-Union
  3. Weighted Quick union (union by size)
  4. Path Compression
- With path compression & other optimization amortized cost is essentially constant as the tree is very flat. O(1)
- Practical Applications
  - Can be used to solve mazes
  - Network connectivity
  - Kruskal minimum spanning tree

Algorithm Analysis

- $1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

BigOh (upper bound)

def: $f(n) = O(g(n))$ iff $\exists$ pos constants c & n_0, such that $f(n) \leq c * g(n)$ $\forall$ $n \geq n_0$

example: $f(n) = 2n+3$ (a better way to get this)

$2n+3 \leq 10n$ $n \geq 1$ ✓ | $2n+3 \leq 2n+3n$

$f(n) \leq c * g(n)$ | $2n+3 \leq 5n$ ✓

$f(n) = O(n)$

* Since big-oh is an upper bound if a function $f(n) = O(n)$ then anything greater than that are true such a $f(n) = O(n^2)$ or $O(2^n)$ or $O(n^n)$

- $1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

BigOmega (lower bound)

def: $f(n) = \Omega(g(n))$, iff $\exists$ pos constants c & n_0, such that $f(n) \geq c * g(n)$ $\forall$ $n \geq n_0$

example: $f(n) = 2n+3$

$2n+3 \geq 1n$ $\forall$ $n \geq 1$

$f(n)$ $c(g(n))$

$f(n) = \Omega(n)$

* Since big omega is a lower bound if a function $f(n) = \Omega(n)$ then anything smaller than that are true such as $f(n) = \Omega(\sqrt{n})$ or $\Omega(1)$.

- $1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

Big Theta (average)

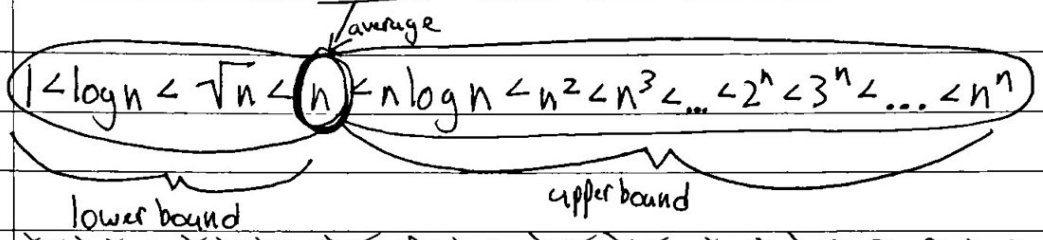
def: $f(n) = \Theta(g(n))$ iff $\exists$ positive constants c_1, c_2, and n_0, such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

example: $f(n) = 2n+3$

$1 \times n \leq 2n+3 \leq 5 \times n$

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

$f(n) = \Theta(n)$



Better Examples

- $f(n) = 2n^2 + 3n + 4$
  - $2n^2+3n+4 \leq 9n^2 \rightarrow f(n) = O(n^2)$
  - $2n^2+3n+4 \geq 1 \times n^2 \rightarrow f(n) = \Omega(n^2)$
  - $1n^2 \leq 2n^2+3n+4 \leq 9n^2 \rightarrow f(n) = \Theta(n^2)$

- $f(n) = n^2 \log n + n$
  - $n^2 \log n \leq n^2 \log n + n \leq 10 n^2 \log n$
  - $f(n) = \Theta(n^2 \log n)$

* If we do not get same term like n^2 or $n^2 \log n$ on both sides we can not find average or Θ of $f(n)$.

Asymptotic properties

General

works for all O, Θ, Ω

- if $f(n)$ is $O(g(n))$ then $a * f(n)$ is $O(g(n))$
  - example: $f(n) = 2n^2+5 \rightarrow f(n) = O(n^2)$
  - then $7 * f(n) = 7 \times (2n^2+5) = 14n^2 + 35 = O(n^2)$
- The same rule applies for omega as big-oh above.
- The same rule applies for theta.

Reflexive

works for O and Ω

- if $f(n)$ is given then $f(n)$ is $O(f(n))$
  - example: $f(n) = n^2 \rightarrow O(n^2)$

Transitive

works for all O, Θ, Ω

- if $f(n) = O(g(n))$ and $g(n)$ is $O(h(n))$
  - then $f(n) = O(h(n))$
  - example: $f(n) = n$ $g(n) = n^2$ $h(n) = n^3$
  - n is $O(n^2)$ and n^2 is $O(n^3)$
  - then n is $O(n^3)$

Symmetric

only works for theta

- if $f(n)$ is $\Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$
  - example: $f(n) = n^2$ $g(n) = n^2$
  - $f(n) = \Theta(n^2)$
  - $g(n) = \Theta(n^2)$

Transpose Symmetric

works for O and Ω

- if $f(n)$ is $O(g(n))$ the $g(n)$ is $\Omega(f(n))$
  - example: $f(n) = n$ $g(n) = n^2$
  - then n is $O(n^2)$ and
  - n^2 is $\Omega(n)$

K-sum k=2

"Obvious" k=2 $O(n^2)$

- check every index for a pair with every other index.
- nested for loops $O(n^2)$, returns duplicates without "smart iteration" method

```
def dumb_twosum(lst):
    res = []
    for i in range(len(lst)):
        for j in range(len(lst)):
            if i != j and lst[i] + lst[j] == 0:
                res.append([lst[i], lst[j]])
```

Smart Iteration k=2 $O(n^2)$

- simply change inner loop indexing to i+1 instead of i

```
for i ... blah blah
    for j in range(i+1, len(lst))
        ...
```

- Still $O(n^2)$ but will perform better, avoids duplicates

Sorting Pointers k=2 $O(n \log n)$

- sort first then use pointers at each end

```
def pointer_ksum(lst):
    res, fp, ep, lst = [], 0, len(lst)-1, sorted(lst)
    while fp < ep:
        currSum = lst[fp] + lst[ep]
        if currSum == 0:
            res.append([lst[fp], lst[ep]])
            fp += 1
            ep -= 1
        elif currSum < 0:
            fp += 1
        else: ep -= 1
```

- look almost constant only because sorted hides costwall.

Caching k=2 O(n)

- cache seen nums
- if we've seen inverse of curr number then we found a pair.
- takes a bit more space

cache = set()

check if -1st(i) in cache

k-sum k=3

k=3 using k=2 $O(n^2)$

- ~~duplicates included~~
- adjust k=2 sum to accept a target value not just 0
- slice the array you give two sum so it doesn't include the current value

k=3 with 2sum cache | k=3 with pointer 2sum

- Both are $O(n^2)$
- nothing to much to note here just combining the two functions