POLITECNICO
DI TORINO
Dipartimento
di Automatica e Informatica

Complexity Analysis

Paolo Camurati

Complexity Analysis

Definition:

- Forecast of resources (memory, time) needed by the algorithm for execution.
  - empirical
  - analytical

Features:

- machine-independent
- assumption: sequential single-processor model (traditional architecture)
- independent of the input data of a particular instance of the problem.

Example:

- Problem P: sort integer data
- Instance I: data are 45 10 6 7 99
- Size of instance |I|: number of bits needed to encode I, in this case 5 x the size of the integer or simply 5

- It depends on the size n of the problem. Examples:
  - number of bits of the operands for integer multiplication
  - size of the file to sort
  - number of characters in a string of text
  - number of data to sort for a sorting algorithm
- Output:
  - S(n): memory occupation
  - T(n): execution time.

Algorithm Classification

- 1: constant
- log n: logarithmic
- n: linear
- n log n: *linearithmic*
- n^2: quadratic WE TRY TO AVOID THESE
- n^3: cubic ---------
- 2^n: exponential ---------

Worst-case Asymptotic Analysis

Goal:

- to guess an upper-bound for T(n) for an algorithm on n data in the worst possible case

Asymptotic: $n \to \infty$:

- for small n, complexity is irrelevant

Why worst-case analysis?

- Conservative guess
- Worst case is very frequent
- Average case:
  - either it coincides with the worst case
  - or it is not definable, unless we resort to complex assumptions on data.

Importance of Complexity Analysis

Advantages of a lower complexity:

- it compensates hardware (in)efficiency

Example:

- Algorithm #1:
  - $T(n) = 2n^2$
  - machine #1: 10^8 instructions/second
- Algorithm #2:
  - $T(n) = 50n \lg_2 n$
  - machine 2: 10^6 instructions/second

If $n = 1M = 10^6$:

- Algorithm #1: $2 \cdot (10^6)^2 / 10^8 = 2 \cdot 10^4 = 20000$ s = 333,33 min
- Algorithm #2: $50 \cdot 10^6 \lg_2 10^6 / 10^6 = 50 \cdot 6 \cdot \lg_2 10 = 1000$ s = 16,67 min

An inefficient algorithm rapidly «wastes» the increase in hardware performance!

Examples

Discrete Fourier Transform:

- decomposition of a N-sample waveform into periodic components
- applications: DVD, JPEG, astrophysics, ….
- trivial algorithm: quadratic (N^2)
- FFT (Fast Fourier Transform): N log N

Simulation of N bodies:

- simulates gravity interaction among N bodies
- trivial algorithm: quadratic (N^2)
- Barnes-Hut algorithm: N log N

Search Algorithms on Arrays

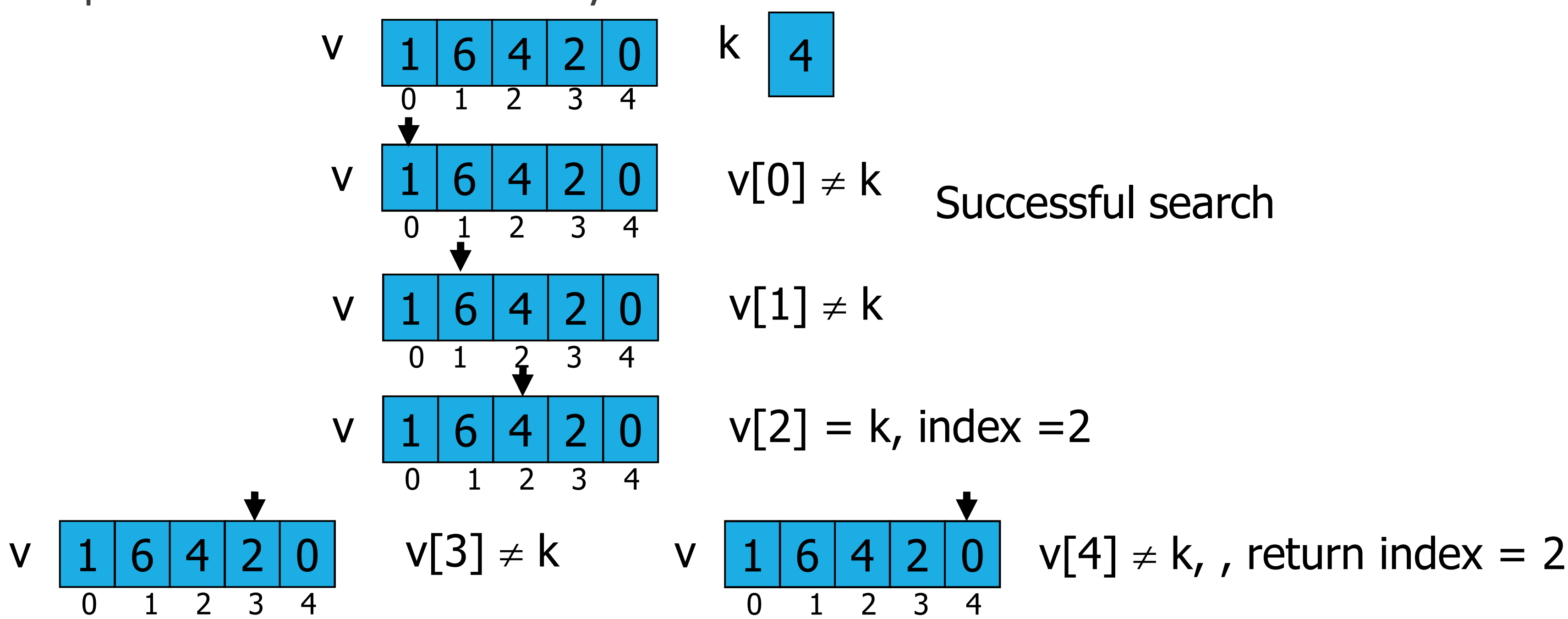
N and k are integers

Let v[N] be an array of N distinct elements, let k be a key:

- Decision problem: does key k appear in array v[N]? Yes/No
- Search problem: if k is in the array, where (at what index)?

Algorithm #1: Linear Search

Scan the array from first to possibly last element, compare at each step current element and key k.



v

1	6	4	2	0
0	1	2	3	4

k: 8

v

1	6	4	2	0
0	1	2	3	4

$v[0] \neq k$

v

1	6	4	2	0
0	1	2	3	4

$v[1] \neq k$

v

1	6	4	2	0
0	1	2	3	4

$v[2] \neq k$

Unsuccessful search

v

1	6	4	2	0
0	1	2	3	4

$v[3] \neq k$

v

1	6	4	2	0
0	1	2	3	4

$v[4] \neq k$, return index = -1

Alternatives:

- Solution #1: scan the array from first to last element: always N operations
- Solution #2: use a flag: early scan stop possible, at most N operations, in the worst case N operations.

The worst-case asymptotic complexity is the same (N), the second alternative improves the average case.

Solution #1

```
int LinearSearch1(int v[], int N, int k) {
  int i = 0, index = -1;

  for (i = 0;  i < N; i++)
    if (k == v[i])
      index = i;

  return index;
}
```

Solution #2

```
int LinearSearch2(int v[], int N, int k) {
  int i = 0;
  int found = 0;

  while (i < N && found == 0)
    if (k == v[i])
      found = 1;
    else
      i++;

  if (found == 0)
    return -1;
  else
    return i;
}
```

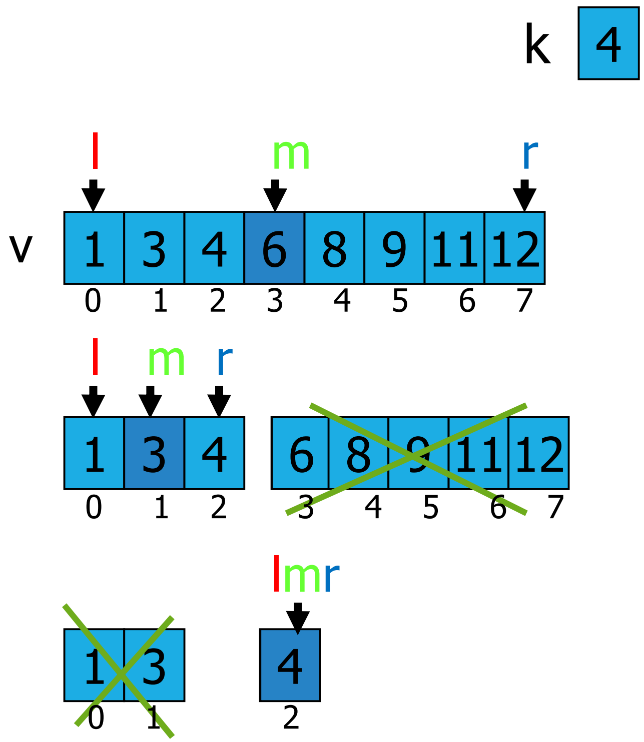
Algorithm #2: Binary Search

Let v[N] be a sorted array of N distinct elements and let k be a key k:

- Decision problem: does key k appear in array v[N]? Yes/No
- Search problem: if k is in the array, where (at what index)?

We work on a subarray identified by the contiguous elements of v whose indices range from a leftmost one (l) and a rightmost one (r). Initially array and subarray coincide (l = 0 and r = N-1). The middle element of the subarray is at index m= (l+r)/2. integer division

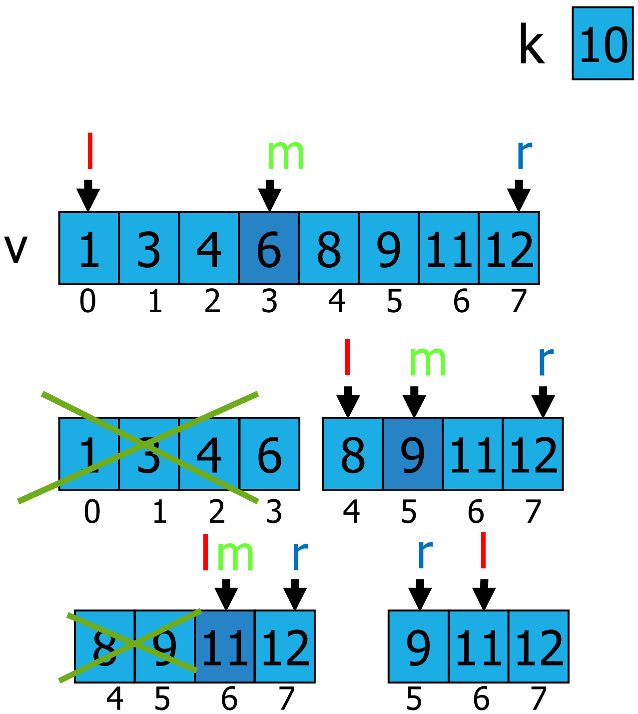
- Loop: at each step compare k to the middle element v[m] of the subarray
- Loop condition: l ≤ r && found==0: the key has not yet been found and the subarray is meaningful (l doesn't exceed r)
- Body of the loop:
  - if v[m] == k: termination with success, found = 1
  - if v[m] < k: search continues in the right subarray: l=m+1, r unchanged
  - if v[m] > k: search continues in the left subarray: l unchanged, r = m-1
- Upon exiting the loop, test found, return -1 for failure or m for success.



l = leftmost index, initially l = 0
r = rightmost index, initially r = N-1
m = index of middle element
v[m] = middle element

l=0, r=7, m= (l+r)/2=3, v[3]>k, r=m-1
l=0, r=2, m= (l+r)/2=1, v[1]<k, l=m+1
l=2, r=2, m= (l+r)/2=2, v[2]=k

Successful search, return index =2



l = leftmost index, initially l = 0
r = rightmost index, initially r = N-1
m = index of middle element
v[m] = middle element

l=0, r=7, m= (l+r)/2=3, v[3]<k, l=m+1
l=4, r=7, m= (l+r)/2=5, v[5]<k, l=m+1
l=6, r=7, m= (l+r)/2=6, v[6]>k, r=m-1
r < l, exit from loop, search failed

```c
int BinSearch(int v[], int N, int k) {
  int m, found= 0, l=0, r=N-1;

while(l <= r && found == 0){
    m = (l+r)/2;
    if(v[m] == k)
      found = 1;
    if(v[m] < k)
      l = m+1;
    else
      r = m-1;
  }
  if (found == 0)
    return -1;
  else
    return m;
}
```

Analysis of Linear Search

linear time complexity

- We consider n numbers for a search miss and in average n/2 for a search hit
- T(n) grows linearly with n.

Analysis of Binary Search

logarithmic time complexity
but sorted array is needed

- At the beginning the array to be examined contains n numbers
- At the 2nd iteration the array to be examined contains about n/2 numbers
- ….
- At the i-th iteration the array to be examined contains about $n/2^i$ numbers
- Termination occurs when the array to be examined contains 1 number, thus $n/2^i = 1$, $i = \log_2(n)$
- T(n) grows logarithmically with n.

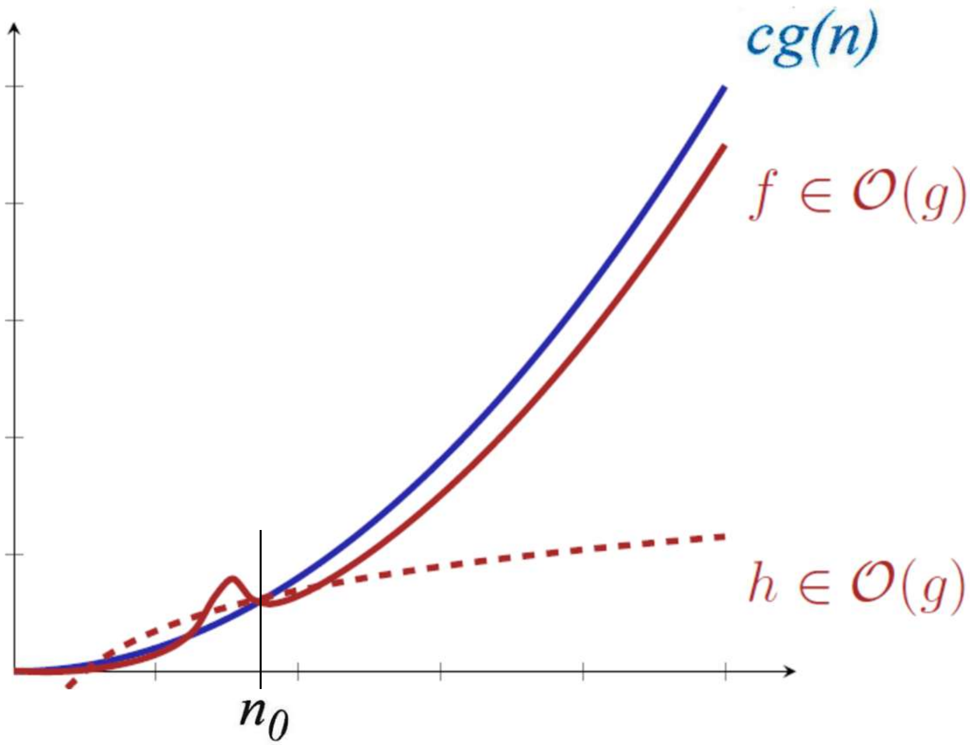
Big-Oh Asymptotic Notation

attribute complexity to a class g(n) which is an upper bound of our complexity (upper bound --> at most)

Definition:

$$T(n) = O(g(n)) \Leftrightarrow$$
$$\exists\, c>0,\ \exists\, n_0>0 \text{ such that } \forall n \geq n_0$$
$$0 \leq T(n) \leq cg(n)$$

g(n) = loose upper bound for T(n). The number of steps is at most g(n) (constant c doesn't count in asymptotic analysis).


$cg(n)$
$f \in \mathcal{O}(g)$
$h \in \mathcal{O}(g)$
n_0

Examples:

$T(n) = 3n+2 = O(n)$, $c=4$ and $n_0=2$: $\quad 3n+2 \leq 4n \quad \forall n \geq 2$

$T(n) = 10n^2+4n+2 = O(n^2)$, $c=11$ and $n_0=5$ $\quad 10n^2+4n+2 \leq 11n^2 \; \forall n \geq 5$

$T(n) = 3n+2 = O(n^2)$, $c=3$ and $n_0=2$ $\quad 3n+2 \leq 3n^2 \quad \forall n \geq 2$

Theorem:

if $T(n) = a_m n^m + \ldots. + a_1 n + a_0$

then $T(n) = O(n^m)$

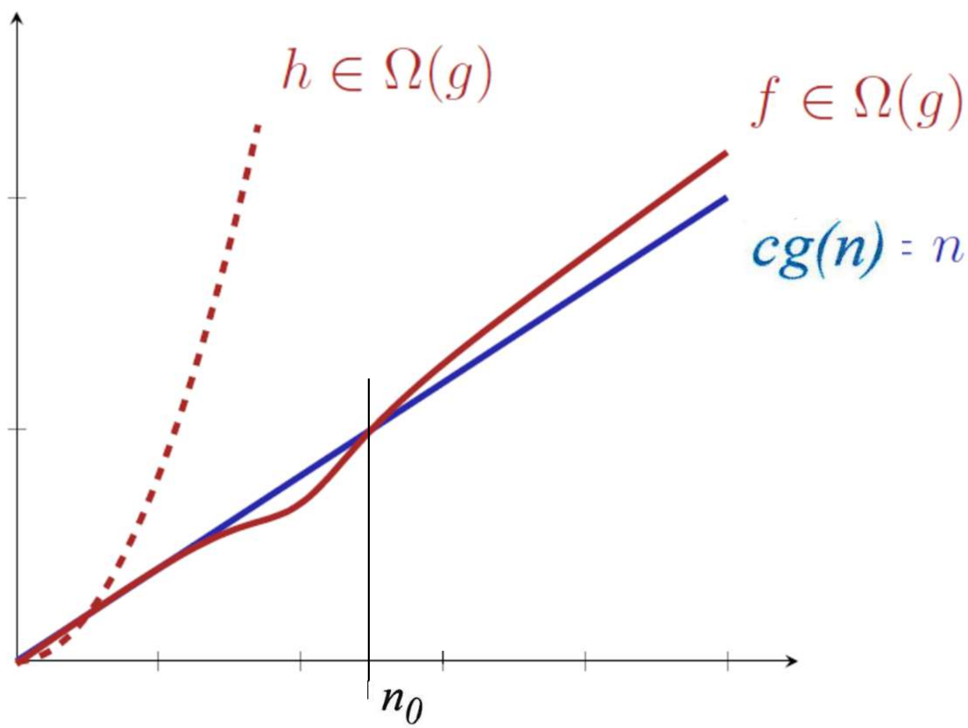
Big-Omega (Ω) Asymptotic Notation

Definition:

$$T(n) = \Omega(g(n)) \Leftrightarrow$$
$$\exists\ c>0,\ \exists\ n_0>0 \text{ such that } \forall n \geq n_0$$
$$0 \leq c\ g(n) \leq T(n)$$

loose lower bound

g(n) = loose lower bound for T(n). The number of steps is at least g(n) (constant c doesn't count in asymptotic analysis).


$h \in \Omega(g)$
$f \in \Omega(g)$
$cg(n) = n$
n_0

Examples:

$T(n) = 3n+2 = \Omega(n)$, c=3 and $n_0=1$ $\qquad 3n \leq 3n+2 \qquad \forall n \geq 1$

$T(n) = 10n^2+4n+2 = \Omega(n^2)$, c=1 and $n_0=1$ $\qquad n^2 \leq 10n^2+4n+2 \qquad \forall n \geq 1$

$T(n) = 10n^2+4n+2 = \Omega(n)$, c=30 and $n_0=3$ $\qquad 30n \leq 10n^2+4n+2 \qquad \forall n \geq 3$

Theorem:

if $T(n) = a_m n^m + \ldots. + a_1 n + a_0$

then $T(n) = \Omega(n^m)$

Big-Theta (Θ) Asymptotic Notation

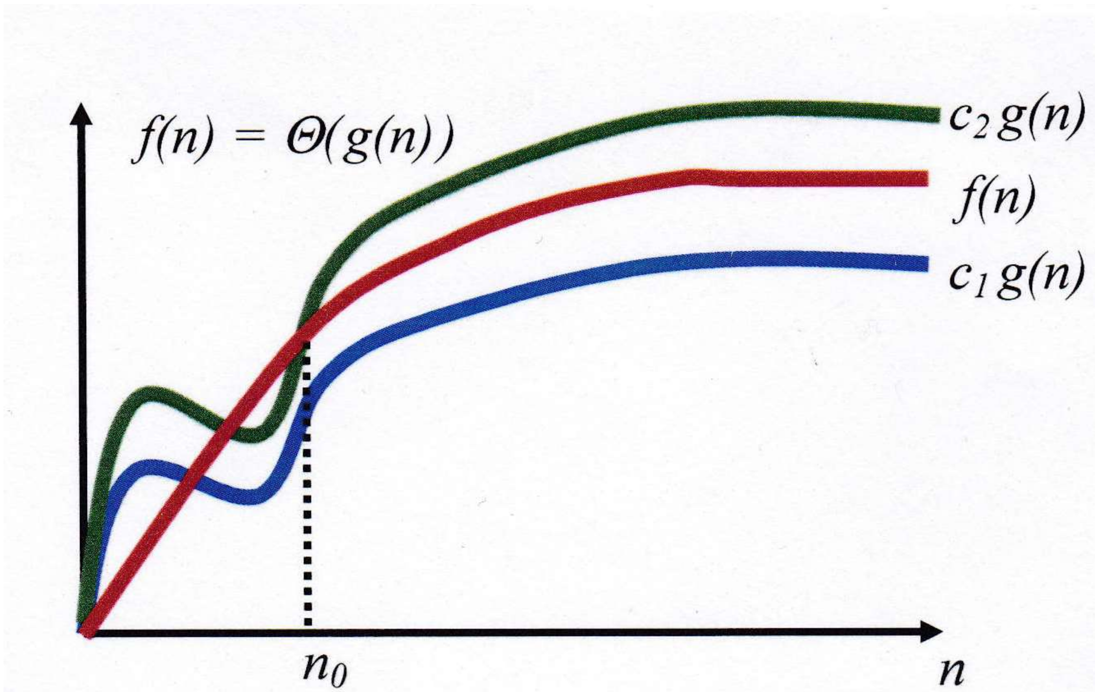
Definition:

$$T(n) = \Theta(g(n)) \Leftrightarrow$$
$$\exists\, c_1, c_2 > 0, \exists\, n_0 > 0 \text{ such that } \forall n \geq n_0$$
$$0 \leq c_1\, g(n) \leq T(n) \leq c_2\, g(n)$$

strict upperbound

sandwiched between g multiplied by different constants

g(n) = tight asymptotic bound for T(n). The number of steps is exactly g(n) (constants c_1 and c_2 do not count in asymptotic analysis).



$f(n) = \Theta(g(n))$
$c_2 g(n)$
$f(n)$
$c_1 g(n)$
n_0
n

Examples:

$T(n) = 3n+2 = \Theta(n)$, $c_1=3$, $c_2=4$ and $n_0=2$ $\quad 3n \leq 3n+2 \leq 4n \quad \forall n \geq 1$

$T(n) = 3n+2 \neq \Theta(n^2)$, $T(n) = 10n^2+4n+2 \neq \Theta(n)$

Theorems:

- If $T(n) = a_m n^m + \ldots + a_1 n + a_0$, then $T(n) = \Theta(n^m)$
- Let g(n) and T(n) be 2 functions,

$T(n) = \Theta(g(n)) \Leftrightarrow T(n) = O(g(n))$ and $T(n) = \Omega(g(n))$.

1. Linear search
2. Binary search

Online Connectivity

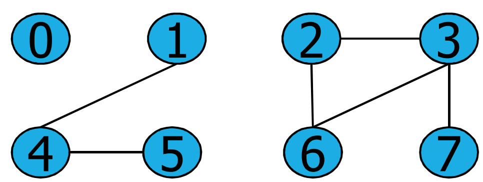
Real problem to understand the impact of the choice of the algorithm and of the data structure on complexity:

- Undirected graph whose vertices are integers and whose edges are pairs of integers
- Input: sequence of integer pairs (p, q)
- Interpretation: p is connected to q
- A connectivity relation is:
  - reflexive: *p is connected to p*
  - symmetrical: *if p is connected to q, q is connected to p*
  - transitive: *if p is connected to q and q is connected to r, then p is connected to r*

thus it is an equivalence relation.

- Output: list of previously unknown connections (or not transitively implied by the previous ones):
  - null if p and q are already connected (directly or indirectly)
  - else (p, q)

Connected component in an undirected graph : maximal subset of mutually reachable nodes



{0} {1, 4, 5} {2, 3, 6, 7}

3 connected components

Applications

- Pixels in digital pictures
- Computer networks (computers, links)
- Electrical networks (components, wires)
- Social networks (friends)
- Mathematical sets
- Program variables.

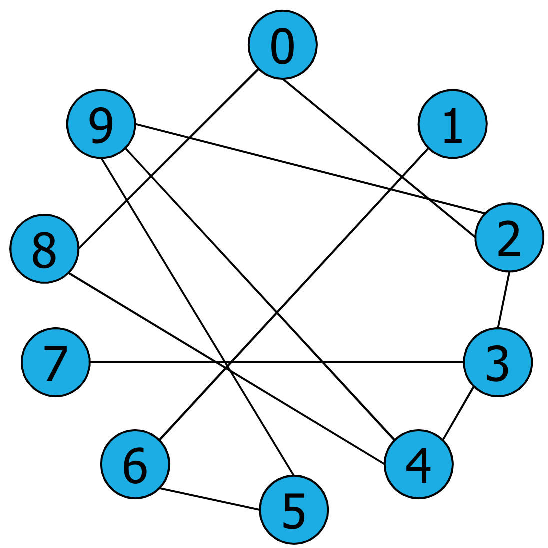
Example

we need to delete redundant connections

Input sequence:

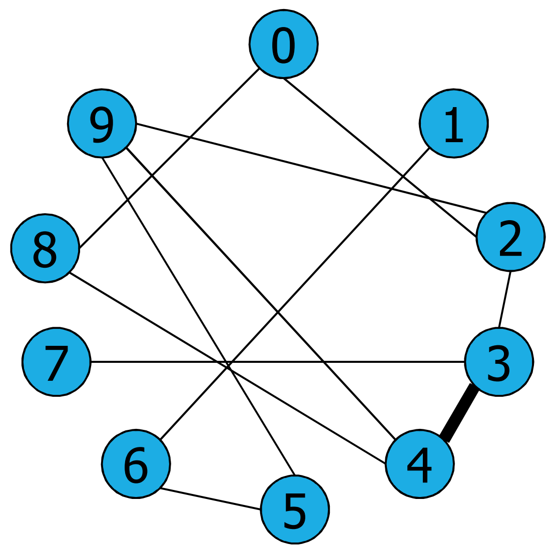
3-4, 4-9, 8-0, 2-3, 5-6, 2-9, 5-9, 7-3, 4-8, 5-6, 0-2, 6-1

Corrisponding graph:



Input

3 4

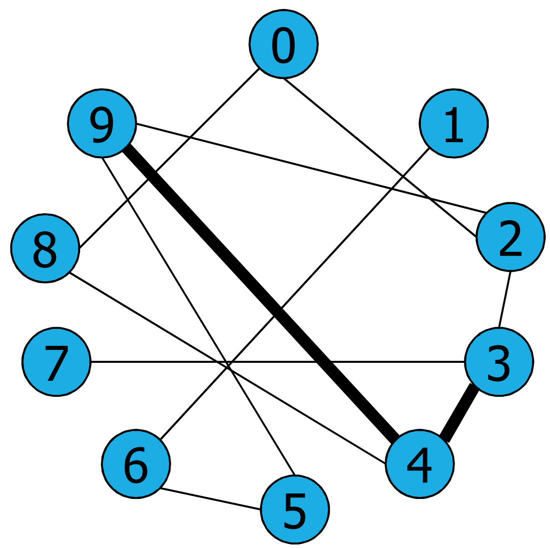


Output

3 4

Input

9 4



Output

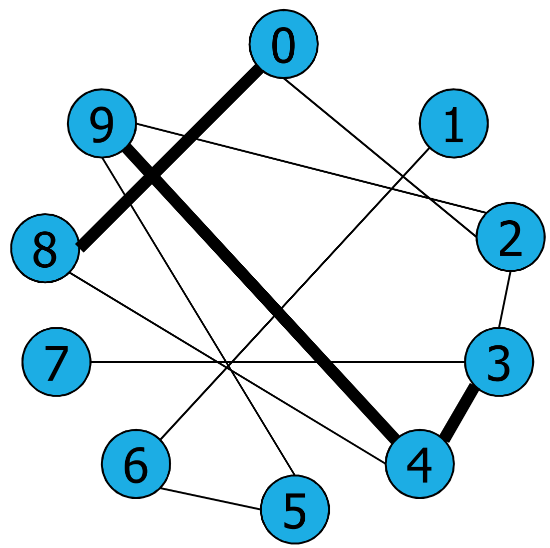
9 4

Input

8 0

Output

8 0

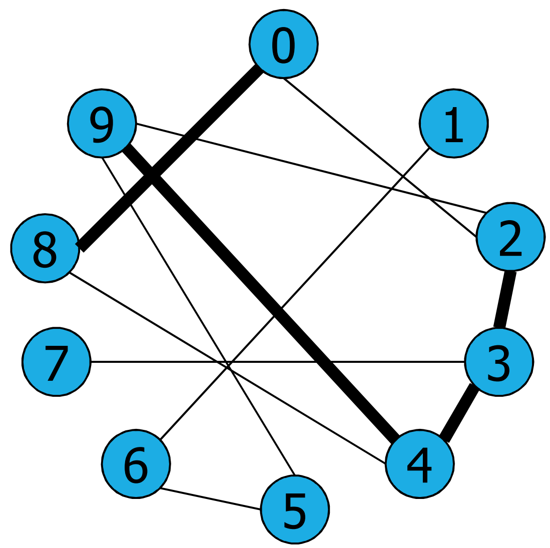


Input

2 3

Output

2 3

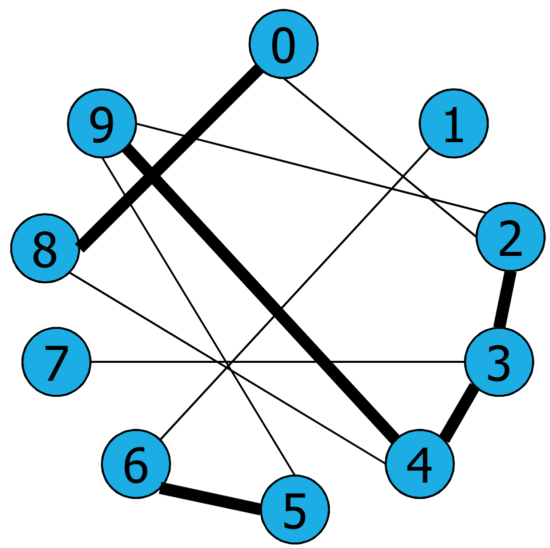


Input

5 6

Output

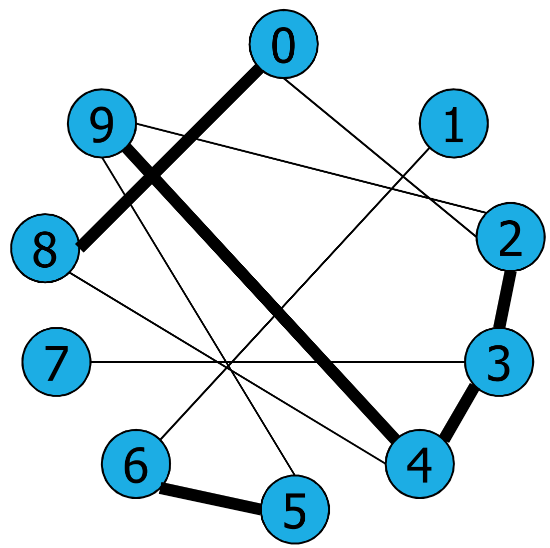
5 6



Input

2 9

Output



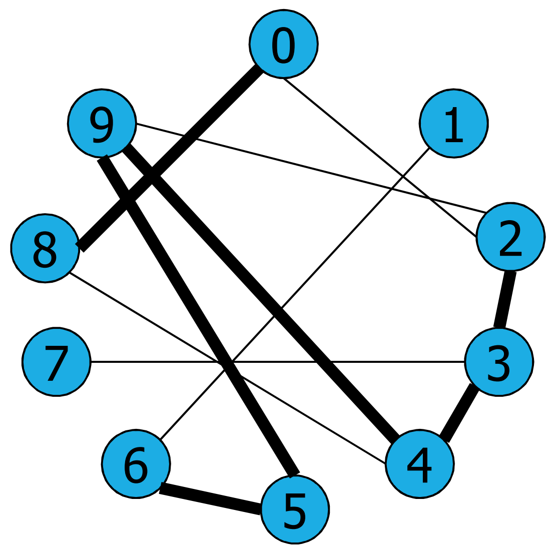
Path 2-3-4-9 already exists

Input

5 9

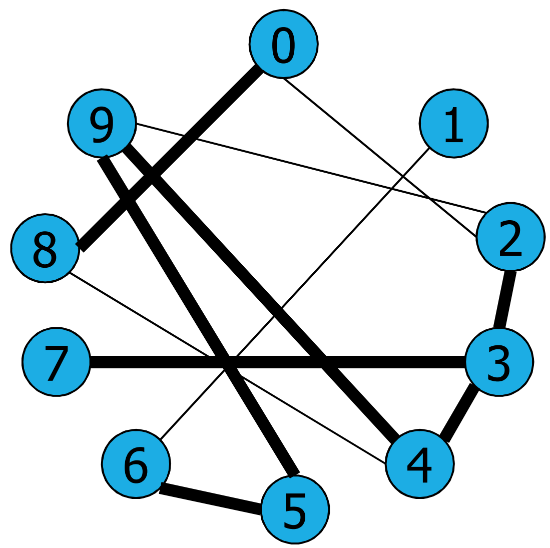
Output

5 9



Input

7 3



Output

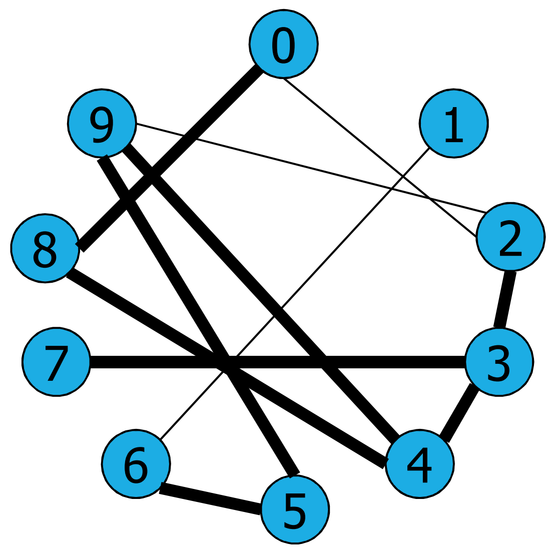
7 3

Input

4 8

Output

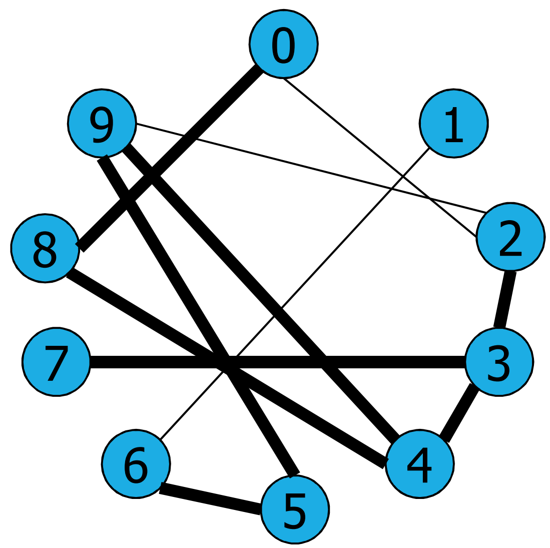
4 8



Input

5 6

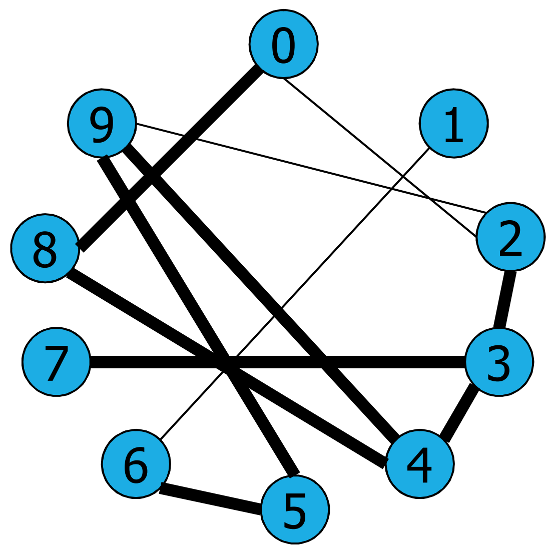
Output



Path 5-6 already exists

Input

0 2

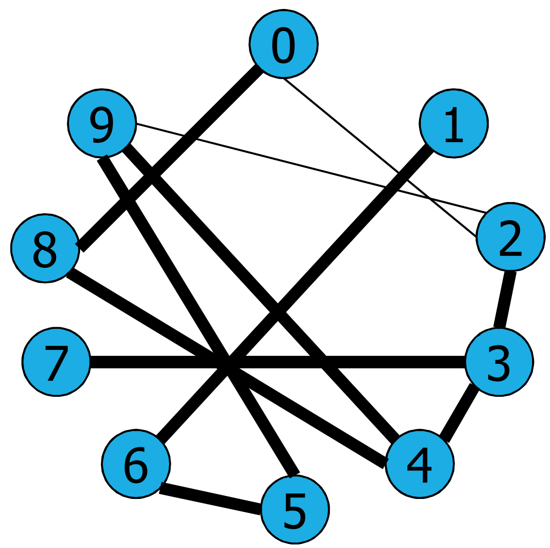


Output

Path 0-8-4-3-2 already exists

Input

6 1



Output

6 1

On-line approach

Assumptions:

- We don't have the graph
- We work online pair by pair, keeping and updating information necessary to find out connectivity.
- Each pair is made of 2 integers in the range from 0 to N-1

Sets S_i of connected pairs, initially as many sets as nodes, each node being connected just to itself.

Abstract operations:

- find: find the set an object belongs to
- union: merge two sets

high level idea of the algorithms --> working on sets: FIND & UNION.

How do I implement these sets? DATA STRUCTURE
How do I find and unite? ALGORITHM

- Algorithm: repeat for all pairs (p, q)
  - read the pair (p, q)
  - execute find on p: find an S_p such that $p \in S_p$
  - execute find on q: find an S_q such that $q \in S_q$
  - if S_p and S_q coincide, consider the next pair, otherwise execute union on S_p and S_q

Two approaches giving privilege either to find or union find quickly if a union is needed but repetitive, and therefore slow, in the union

Quick find

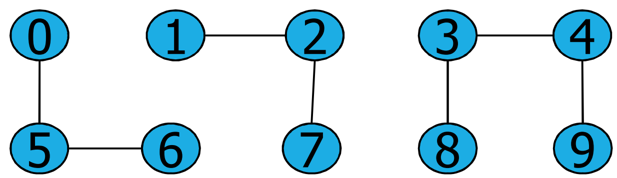
Represent sets S_i of connected pairs with array id:

- initially id[i] = i (no connection)
- if p and q are connected, id[p] = id[q]

Example: the following graph

if different representative, the second group becomes represented by the first

no rule regarding who changes but be consistent



only one element represents the set

is represented like this:

id	0	1	1	8	8	0	0	1	8	8
	0	1	2	3	4	5	6	7	8	9

Algorithm:

- repeat for all pairs (p, q):
  - read pair (p, q)
  - if pair is connected (id[p] = id[q]), do nothing and move to the next pair, else scan the array, replacing id[p] values with id[q] values

- find: simple reference to cell in array id[index], unit cost O(1)
- union: scan array to replace id[p] values with id[q] values, cost linear in array size O(n)
- overall number of operations related to

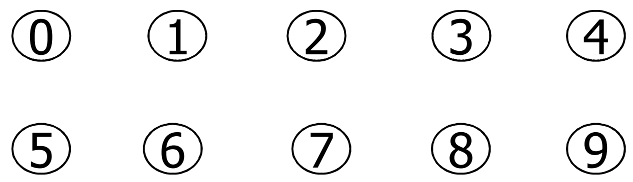
  # pairs * array size

O(n)

Tree representation

- Some objects represent the set they belong to
- Other objects point to the the object that represents the set they belong to.

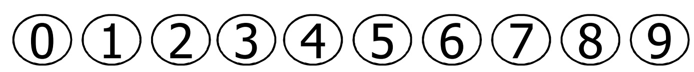
Example

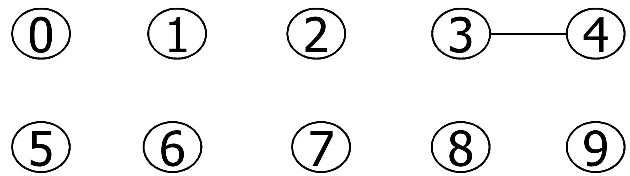


Initially

id	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9

S_0 = {0}, S_1 = {1}, S_2 = {2}, S_3 = {3}, S_4 = {4}
S_5 = {5}, S_6 = {6}, S_7 = {7}, S_8 = {8}, S_9 = {9}





p q = 3 4

id

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

id[p]=3 $\neq$ id[q]=4

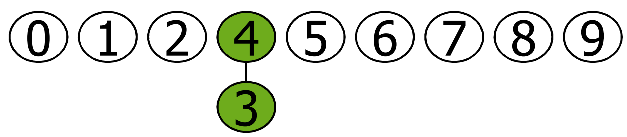
change all id[p] values in id[q]

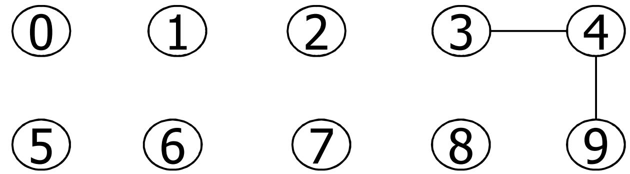
id

0	1	2	4	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

S_0 = {0}, S_1 = {1}, S_2 = {2}, $S_{3\text{-}4}$ = {3,4},

S_5 = {5}, S_6 = {6}, S_7 = {7}, S_8 = {8}, S_9 = {9}





p q = 4 9

id

0	1	2	4	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

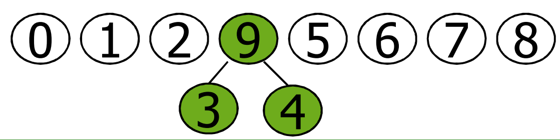
id[p]=4 $\neq$ id[q]=9

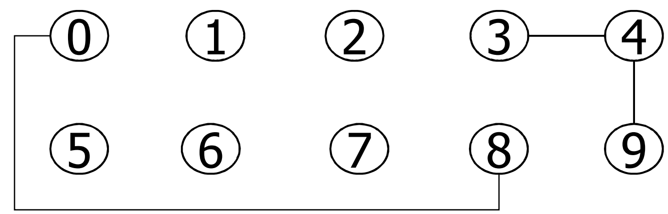
change all id[p] values in id[q]

id

0	1	2	9	9	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

S_0 = {0}, S_1 = {1}, S_2 = {2}, $S_{3\text{-}4\text{-}9}$ = {3,4,9},
S_5 = {5}, S_6 = {6}, S_7 = {7}, S_8 = {8}





p q = 8 0

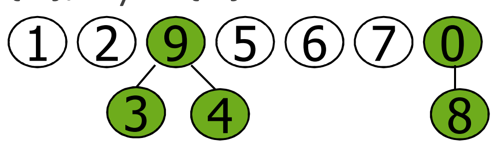
id	0	1	2	9	9	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9

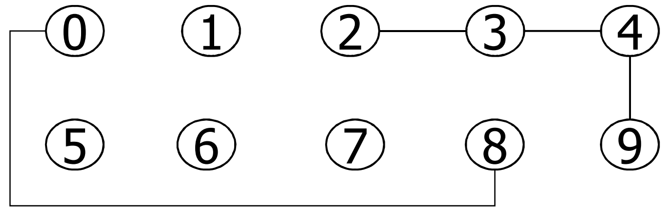
id[p]=8 $\neq$ id[q]=0

change all id[p] values in id[q]

id	0	1	2	9	9	5	6	7	0	9
	0	1	2	3	4	5	6	7	8	9

$S_{0\text{-}8} = \{0,8\}$, $S_1 = \{1\}$, $S_2 = \{2\}$, $S_{3\text{-}4\text{-}9} = \{3,4,9\}$,
$S_5 = \{5\}$, $S_6 = \{6\}$, $S_7 = \{7\}$





p q = 2 3

id

0	1	2	9	9	5	6	7	0	9
0	1	2	3	4	5	6	7	8	9

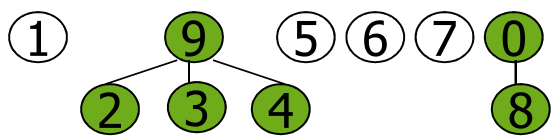
id[p]=2 $\neq$ id[q]=9

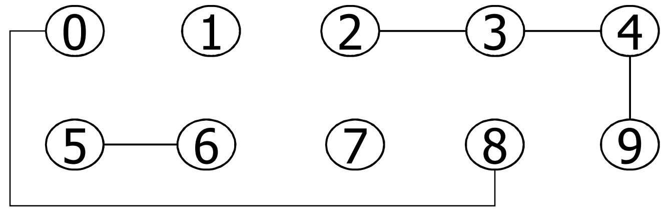
change all id[p] values in id[q]

id

0	1	9	9	9	5	6	7	0	9
0	1	2	3	4	5	6	7	8	9

$S_{0\text{-}8} = \{0,8\}$, $S_1 = \{1\}$, $S_{2\text{-}3\text{-}4\text{-}9} = \{2,3,4,9\}$,
$S_5 = \{5\}$, $S_6 = \{6\}$, $S_7 = \{7\}$





p q = 5 6

id

0	1	9	9	9	5	6	7	0	9
0	1	2	3	4	5	6	7	8	9

id[p]=5 $\neq$ id[q]=6

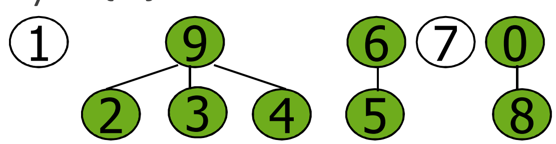
change all id[p] values in id[q]

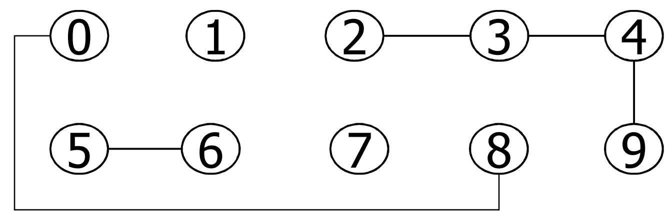
id

0	1	9	9	9	6	6	7	0	9
0	1	2	3	4	5	6	7	8	9

$S_{0\text{-}8}$ = {0,8}, S_1 = {1}, $S_{2\text{-}3\text{-}4\text{-}9}$ = {2,3,4,9},

$S_{5\text{-}6}$ = {5,6}, S_7 = {7}





p q = 2 9

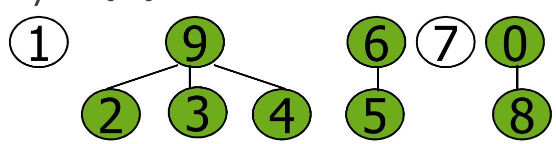
id	0	1	9	9	9	6	6	7	0	9
	0	1	2	3	4	5	6	7	8	9

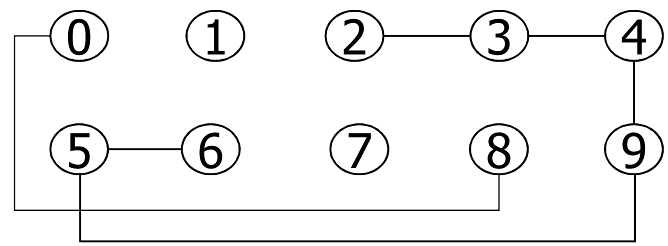
id[p]=9 = id[q]=9

no change

id	0	1	9	9	9	6	6	7	0	9
	0	1	2	3	4	5	6	7	8	9

$S_{0\text{-}8}$ = {0,8}, S_1 = {1}, $S_{2\text{-}3\text{-}4\text{-}9}$ = {2,3,4,9},
$S_{5\text{-}6}$ = {5,6}, S_7 = {7}





p q = 5 9

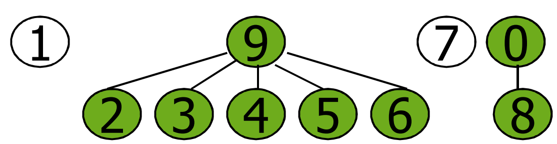
id	0	1	9	9	9	6	6	7	0	9
	0	1	2	3	4	5	6	7	8	9

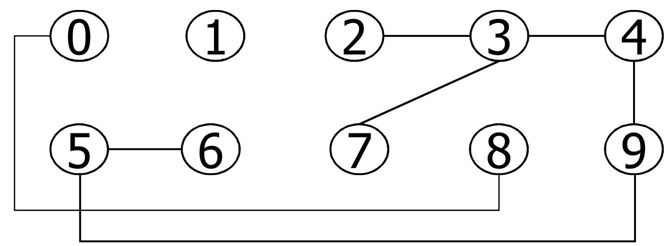
id[p]=6 $\neq$ id[q]=9

change all id[p] values in id[q]

id	0	1	9	9	9	9	9	7	0	9
	0	1	2	3	4	5	6	7	8	9

$S_{0\text{-}8} = \{0,8\}$, $S_1 = \{1\}$, $S_{2\text{-}3\text{-}4\text{-}5\text{-}6\text{-}9} = \{2,3,4,5,6,9\}$, $S_7 = \{7\}$





p q = 7 3

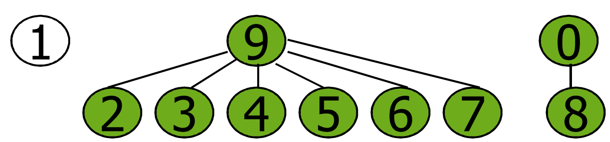
	id									
value	0	1	9	9	9	9	9	7	0	9
index	0	1	2	3	4	5	6	7	8	9

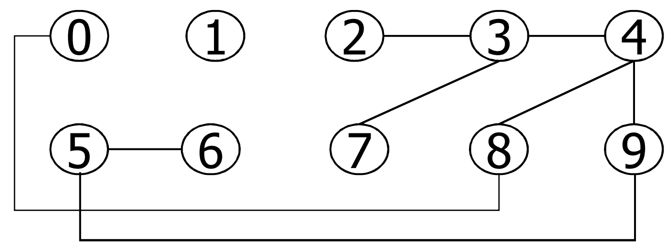
id[p]=7 $\neq$ id[q]=9

change all id[p] values in id[q]

	id									
value	0	1	9	9	9	9	9	9	0	9
index	0	1	2	3	4	5	6	7	8	9

$S_{0\text{-}8}$ = {0,8}, S_1 = {1}, $S_{2\text{-}3\text{-}4\text{-}5\text{-}6\text{-}7\text{-}9}$ = {2,3,4,5,6,7,9}





p q = 4 8

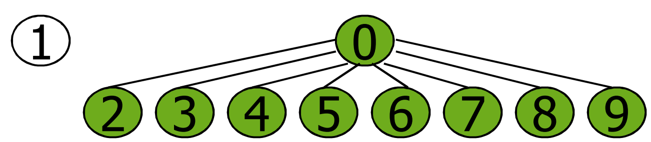
id	0	1	9	9	9	9	9	9	0	9
	0	1	2	3	4	5	6	7	8	9

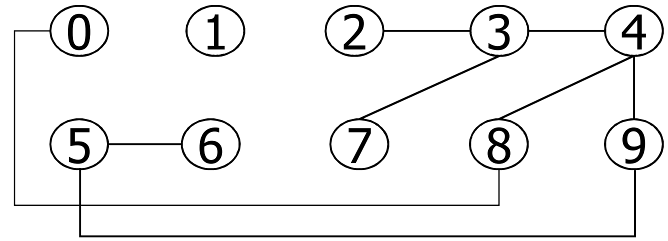
id[p]=9 $\neq$ id[q]=0

change all id[p] values in id[q]

id	0	1	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9

$S_1 = \{1\}$, $S_{0\text{-}2\text{-}3\text{-}4\text{-}5\text{-}6\text{-}7\text{-}8\text{-}9} = \{0,2,3,4,5,6,7,8,9\}$





p q = 5 6

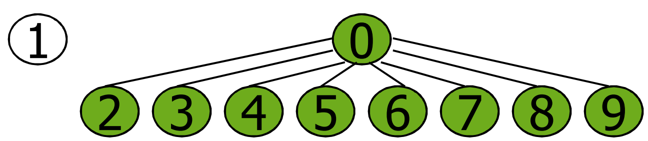
id	0	1	2	3	4	5	6	7	8	9
	0	1	0	0	0	0	0	0	0	0

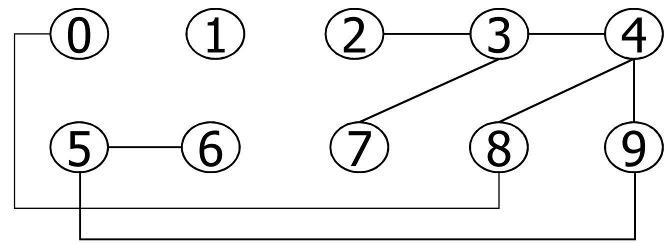
id[p]=0 = id[q]=0

no change

id	0	1	2	3	4	5	6	7	8	9
	0	1	0	0	0	0	0	0	0	0

$S_1 = \{1\}$, $S_{0\text{-}2\text{-}3\text{-}4\text{-}5\text{-}6\text{-}7\text{-}8\text{-}9} = \{0,2,3,4,5,6,7,8,9\}$





p q = 0 2

id

0	1	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

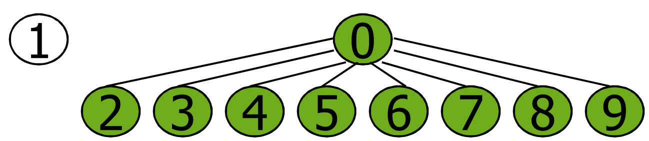
id[p]=0 = id[q]=0

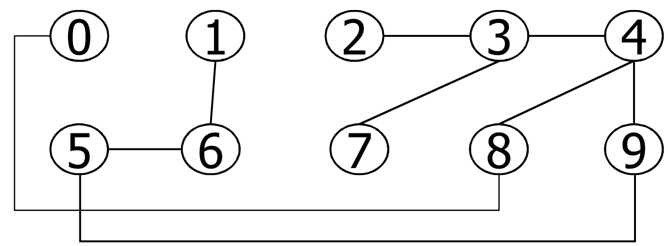
no change

id

0	1	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

S_1 = {1}, $S_{0\text{-}2\text{-}3\text{-}4\text{-}5\text{-}6\text{-}7\text{-}8\text{-}9}$ = {0,2,3,4,5,6,7,8,9}





p q = 6 1

id

0	1	2	3	4	5	6	7	8	9
0	1	0	0	0	0	0	0	0	0

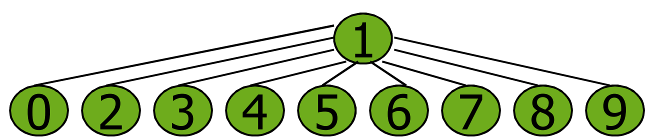
id[p]=0 = id[q]=1

change all id[p] values in id[q]

id

0	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1	1

$S_{0\text{-}1\text{-}2\text{-}3\text{-}4\text{-}5\text{-}6\text{-}7\text{-}8\text{-}9} = \{0,1,2,3,4,5,6,7,8,9\}$



```
#include <stdio.h>
#define N 10000
main() {
  int i, t, p, q, id[N];
  for (i=0; i<N; i++) # initialization
    id[i] = i;
  printf("Input pair p q:  ");
  while (scanf("%d %d", &p, &q) ==2) {#condition if inputs are two integers
    if (id[p] == id[q]) #if already connected
      printf("%d %d already connected\n", p,q);
    else {#if not connected --> merge
      for (t = id[p], i = 0; i < N; i++)
        if (id[i] == t)
          id[i] = id[q];
        printf("pair %d %d not yet connected\n", p, q);
      }
      printf("Input pair p q:  ");
  }
}
```

Quick union

find and (if necessary) merge

the representative of the group

Represent sets S_i of connected pairs with an array id:

- initially all the objects point to themselves

  id[i] = i (no connection)

- each object points either to an object to which it is connected or to itself (no loops).

Notation (id[i])* stands for id[id[id[… id[i]]]]

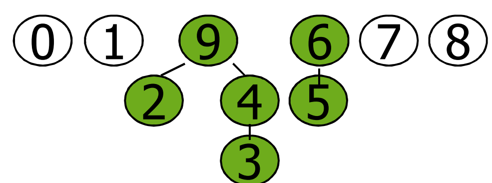
If objects i are j connected

(id[i])* = (id[j])*

Example

id

0	1	9	4	9	6	6	7	8	9
0	1	2	3	4	5	6	7	8	9



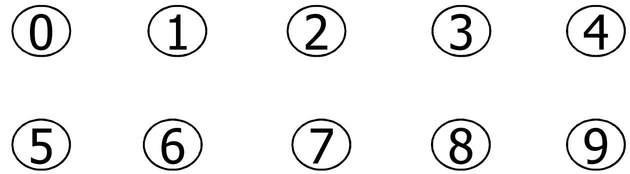
Algorithm:

- repeat for all the pairs (p, q):
  - read pair (p, q)
  - if(id[p])* = (id[q])* do nothing (the pair is already connected) and move on to the next pair, else id[(id[p])*] = (id[q])* (connect the pair).

- find: scan a “chain” of objects, upper bound linear cost in the number of objects, in general well below upper bound O(n)
- union: simple, as it is enough that an object points to another object, unit cost O(1)
- overall number of operations related to

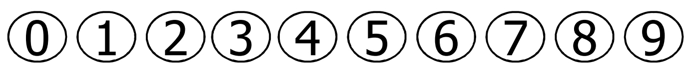
  # pairs * chain length

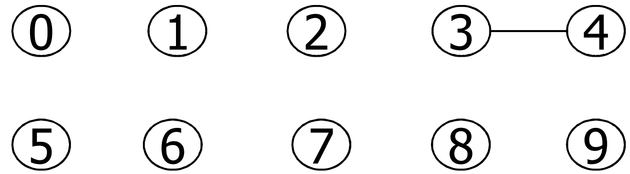
Example



Initially

id	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9





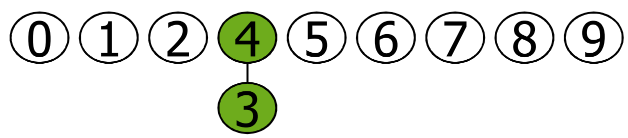
p q = 3 4

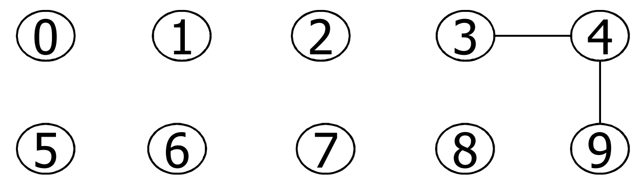
id	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9

id[p]=3 $\neq$ id[q]=4

let p point to q: id[p]=4

id	0	1	2	4	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9





p q = 4 9

id

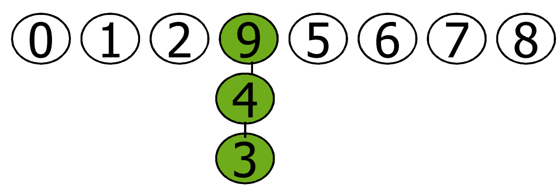
0	1	2	3	4	5	6	7	8	9
0	1	2	4	4	5	6	7	8	9

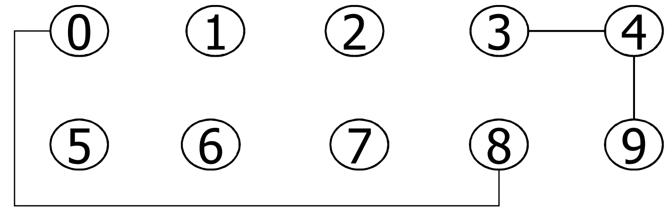
id[p]=4 $\neq$ id[q]=9

let p point to q: id[p]=9

id

0	1	2	3	4	5	6	7	8	9
0	1	2	4	9	5	6	7	8	9





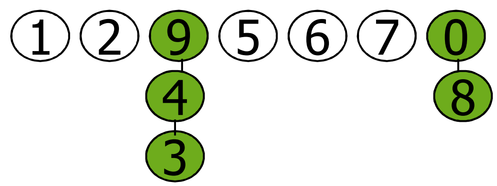
p q = 8 0

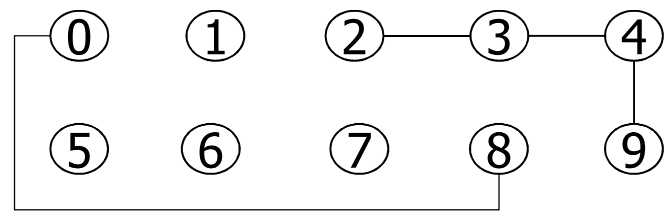
id	0	1	2	4	9	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9

id[p]=8 $\neq$ id[q]=0

let p point to q: id[p]=0

id	0	1	2	4	9	5	6	7	0	9
	0	1	2	3	4	5	6	7	8	9





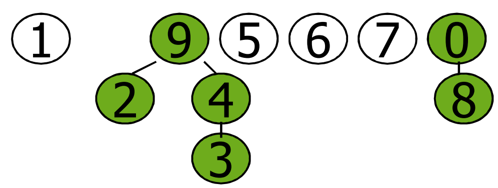
p q = 2 3

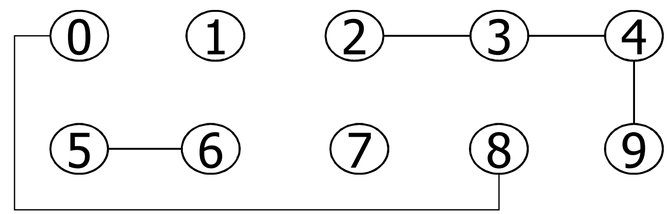
id	0	1	2	4	9	5	6	7	0	9
	0	1	2	3	4	5	6	7	8	9

$id[p]=2 \neq id[id[id[q]]]=9$

let p point to q: $id[p]=9$

id	0	1	9	4	9	5	6	7	0	9
	0	1	2	3	4	5	6	7	8	9





p q = 5 6

id

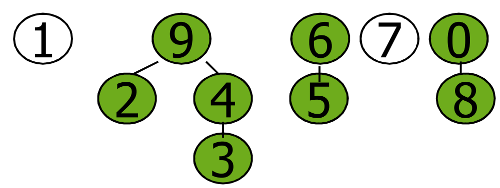
0	1	2	3	4	5	6	7	8	9
0	1	9	4	9	5	6	7	0	9

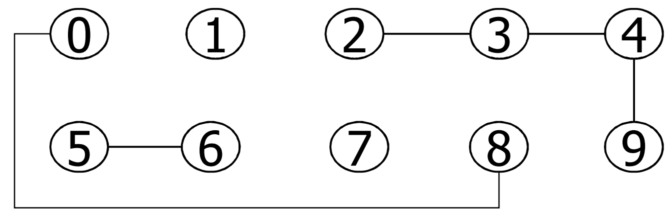
id[p]=5 $\neq$ id[q]=6

let p point to q: id[p]=6

id

0	1	2	3	4	5	6	7	8	9
0	1	9	4	9	6	6	7	0	9





p q = 2 9

id

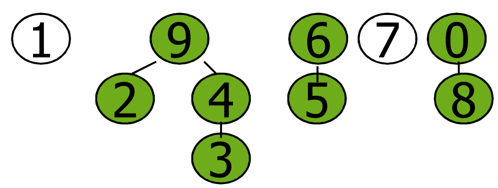
0	1	9	4	9	6	6	7	0	9
0	1	2	3	4	5	6	7	8	9

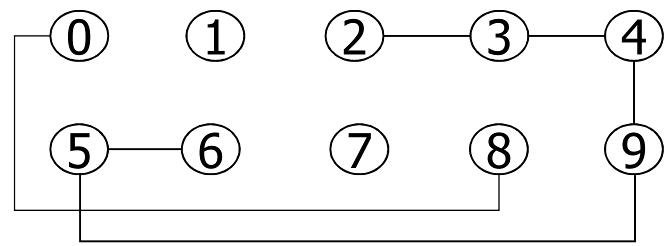
id[id[p]]=9 = id[q]=9

unchanged

id

0	1	9	4	9	6	6	7	0	9
0	1	2	3	4	5	6	7	8	9





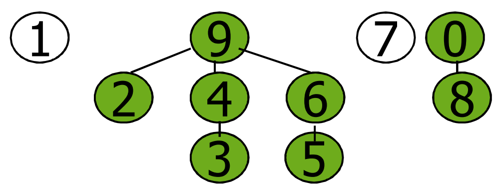
p q = 5 9

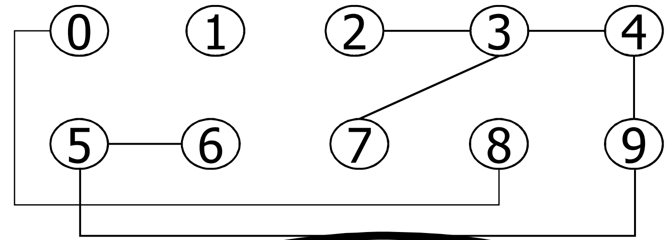
id	0	1	9	4	9	6	6	7	0	9
	0	1	2	3	4	5	6	7	8	9

id[id[p]]=6 $\neq$ id[q]=9

let p point to q: id[id[p]]=9

id	0	1	9	4	9	6	9	7	0	9
	0	1	2	3	4	5	6	7	8	9





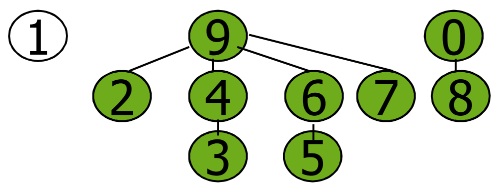
p q = 7 3

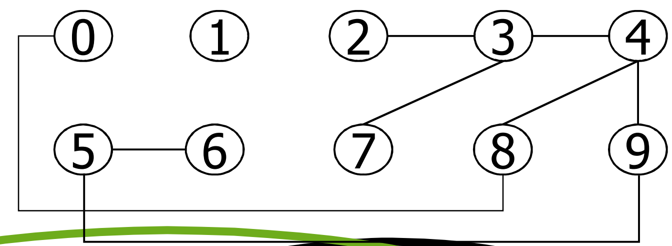
id	0	1	9	4	9	6	9	7	0	9
	0	1	2	3	4	5	6	7	8	9

$id[p]=7 \neq id[id[id[q]]]=9$

let p point to q: id[p]=9

id	0	1	9	4	9	6	9	9	0	9
	0	1	2	3	4	5	6	7	8	9





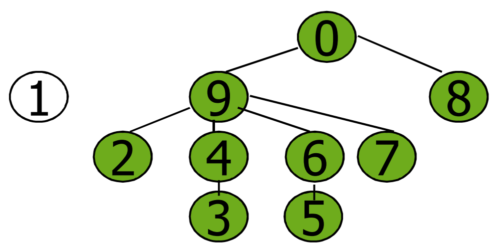
p q = 4 8

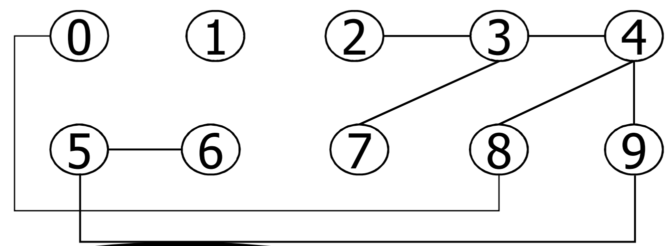
id	0	1	9	4	9	6	9	9	0	9
	0	1	2	3	4	5	6	7	8	9

id[id[p]]=9 $\neq$ id[id[q]]=0

let p point to q: id[id[p]]=0

id	0	1	9	4	9	6	9	9	0	0
	0	1	2	3	4	5	6	7	8	9





p q = 5 6

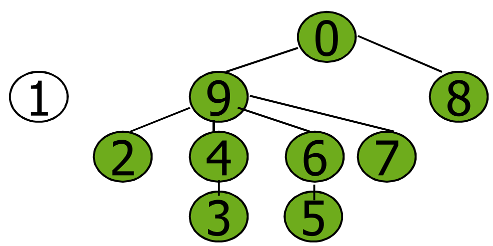
id

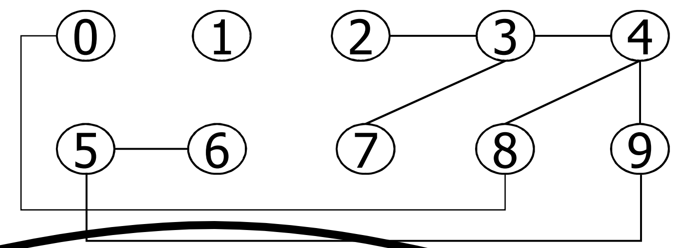
id	0	1	9	4	9	6	9	9	0	0
	0	1	2	3	4	5	6	7	8	9

id[id[id[id[p]]]]=0 = id[id[q]]=0

unchanged

id	0	1	9	4	9	6	9	9	0	0
	0	1	2	3	4	5	6	7	8	9





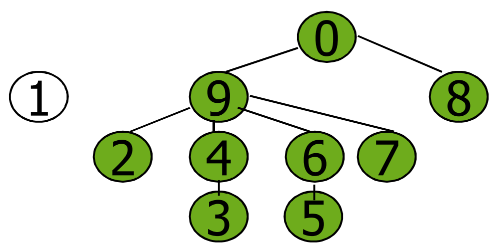
p q = 0 2

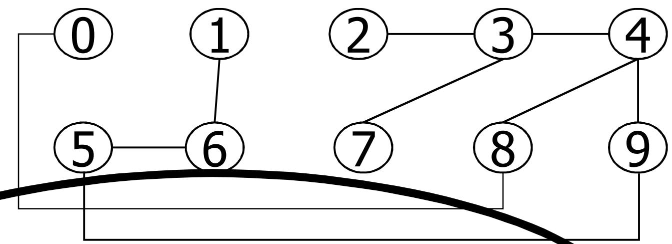
id	0	1	9	4	9	6	9	9	0	0
	0	1	2	3	4	5	6	7	8	9

id[p]=0 = id[id[id[q]]]=0

unchanged

id	0	1	9	4	9	6	9	9	0	0
	0	1	2	3	4	5	6	7	8	9





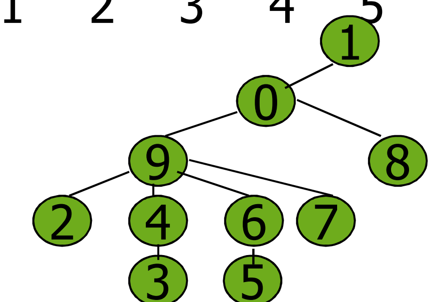
p q = 6 1

id	0	1	9	4	9	6	9	9	0	0
	0	1	2	3	4	5	6	7	8	9

id[id[id[p]]]=0 $\neq$ id[q]=1

let p point to q: id[id[id[p]]]=1

id	1	1	9	4	9	6	9	9	0	0
	0	1	2	3	4	5	6	7	8	9



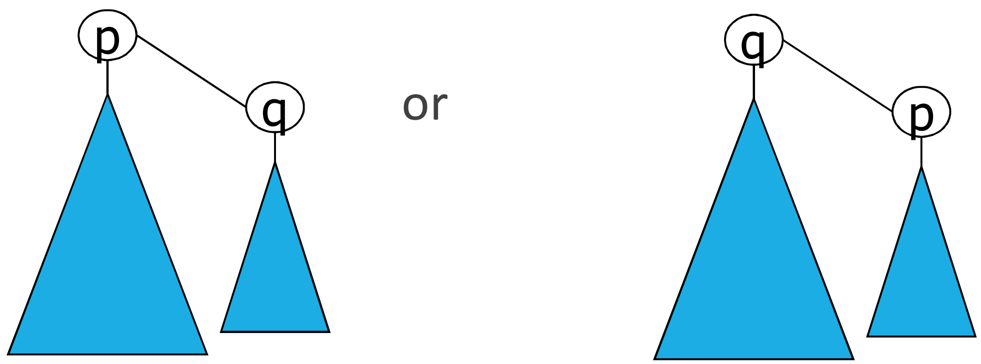
```c
#include <stdio.h>
#define N 10000
main() {
  int i, j, p, q, id[N];
  for(i=0; i<N; i++)
    id[i] = i;
  printf("Input pair p q:  ");
  while (scanf("%d %d", &p, &q) ==2) {
    for (i = p; i!= id[i]; i = id[i]);
    for (j = q; j!= id[j]; j = id[j]);
    if (i == j)
      printf("pair %d %d already connected\n", p,q);
    else {
      id[i] = j;
      printf("pair %d %d not yet connected\n", p, q);
    }
    printf("Input pair p q:  ");
  }
}
```

Quick union Optimization

Weighted quick union:

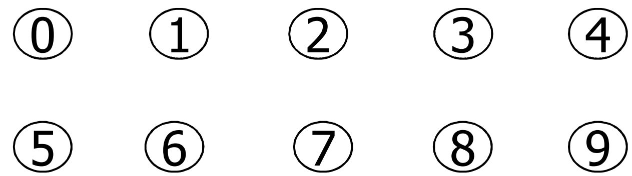
- To shorten the chain's length, keep track of the number of elements in each tree (array `sz`) and connect the smaller tree to the larger one.
- According to which one is the larger, there might be 2 solutions:

we need to keep track of the sizes of the subarrays



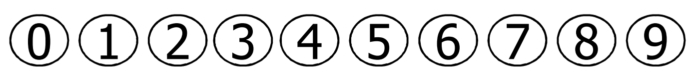
NB: it doesn't matter whether if p appears at the right or at the left of q.

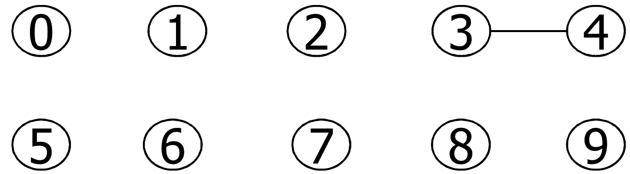
Example



Initially

id									
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9





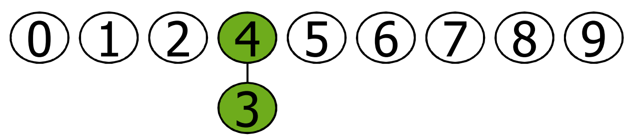
p q = 3 4

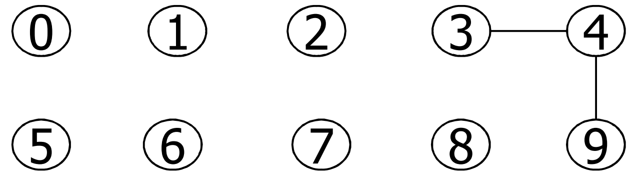
id	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9

id[p]=3 $\neq$ id[q]=4

let p point to q: id[p]=4

id	0	1	2	4	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9





p q = 4 9

id

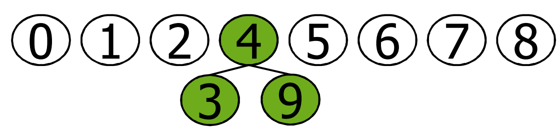
0	1	2	4	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

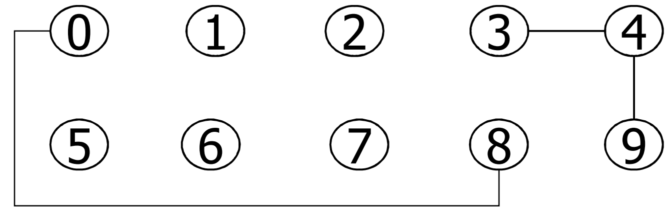
id[p]=4 $\neq$ id[q]=9

let the smaller tree q pointo to the larger tree p: id[q]=4

id

0	1	2	4	4	5	6	7	8	4
0	1	2	3	4	5	6	7	8	9





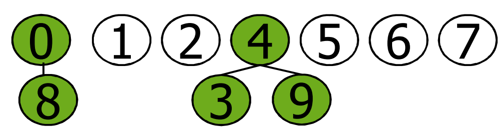
p q = 8 0

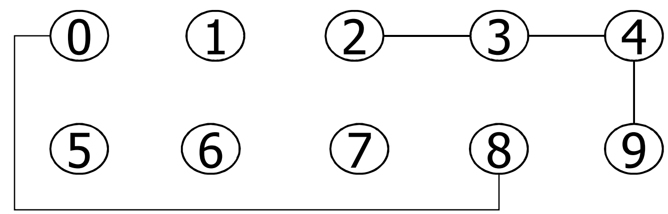
id	0	1	2	4	4	5	6	7	8	4
	0	1	2	3	4	5	6	7	8	9

id[p]=8 $\neq$ id[q]=0

let p point to q: id[p]=0

id	0	1	2	4	4	5	6	7	0	4
	0	1	2	3	4	5	6	7	8	9





p q = 2 3

id

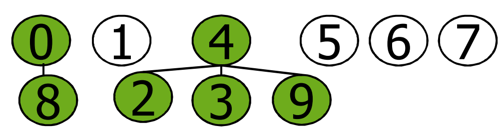
0	1	2	3	4	5	6	7	8	9
0	1	2	4	4	5	6	7	0	4

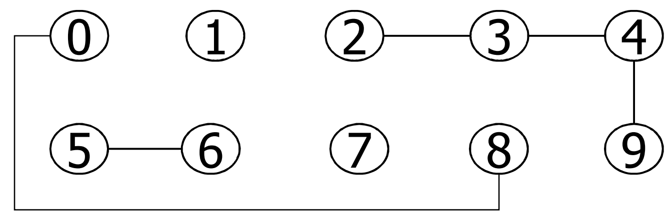
id[p]=2 $\neq$ id[id[q]]=4

let the smaller tree q pointo to the larger tree p: id[p]=4

id

0	1	2	3	4	5	6	7	8	9
0	1	4	4	4	5	6	7	0	4





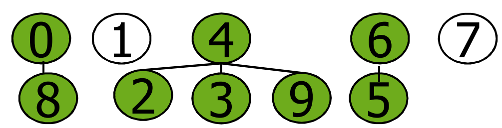
p q = 5 6

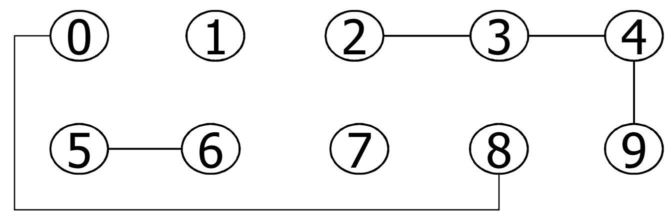
id	0	1	4	4	4	5	6	7	0	4
	0	1	2	3	4	5	6	7	8	9

id[p]=5 $\neq$ id[q]=6

let p point to q: id[p]=6

id	0	1	4	4	4	6	6	7	0	4
	0	1	2	3	4	5	6	7	8	9





p q = 2 9

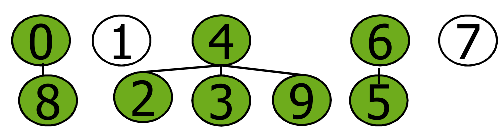
id

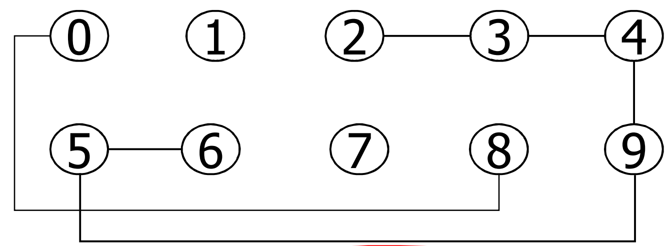
id	0	1	4	4	4	6	6	7	0	4
	0	1	2	3	4	5	6	7	8	9

id[id[p]]=4 = id[q]=4

unchanged

id	0	1	4	4	4	6	6	7	0	4
	0	1	2	3	4	5	6	7	8	9





p q = 5 9

id

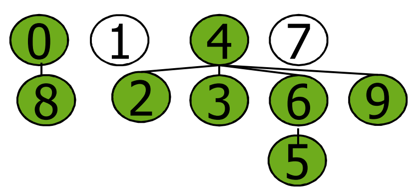
0	1	4	4	4	6	6	7	0	4
0	1	2	3	4	5	6	7	8	9

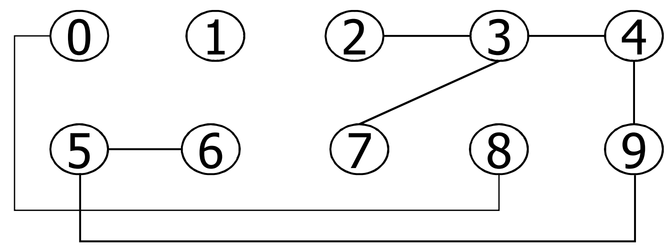
id[id[p]]=6 $\neq$ id[id[q]]=4

let the smaller tree q pointo to the larger tree p: id[id[p]]=4

id

0	1	4	4	4	6	4	7	0	4
0	1	2	3	4	5	6	7	8	9





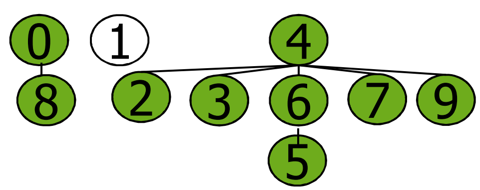
p q = 7 3

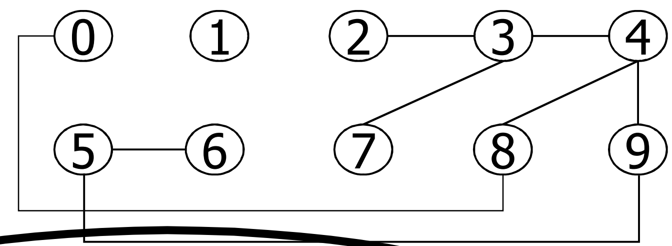
id										
	0	1	4	4	4	6	4	7	0	4
	0	1	2	3	4	5	6	7	8	9

id[p]=7 $\neq$ id[id[q]]=4

let the smaller tree q pointo to the larger tree p: id[p]=4

id										
	0	1	4	4	4	6	4	4	0	4
	0	1	2	3	4	5	6	7	8	9





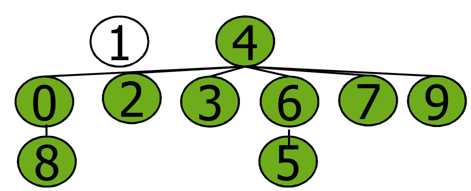
p q = 4 8

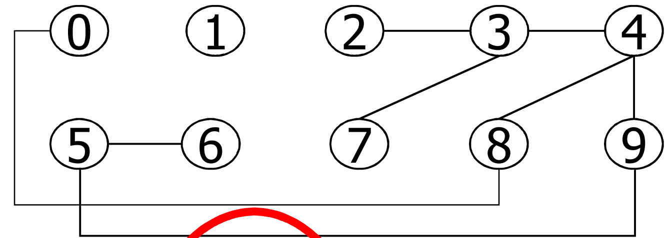
id									
0	1	4	4	4	6	4	4	0	4
0	1	2	3	4	5	6	7	8	9

id[p]=4 $\neq$ id[id[q]]=0

let the smaller tree q pointo to the larger tree p: id[id[q]]=4

id									
4	1	4	4	4	6	4	4	0	4
0	1	2	3	4	5	6	7	8	9





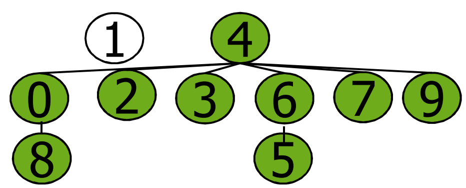
p q = 5 6

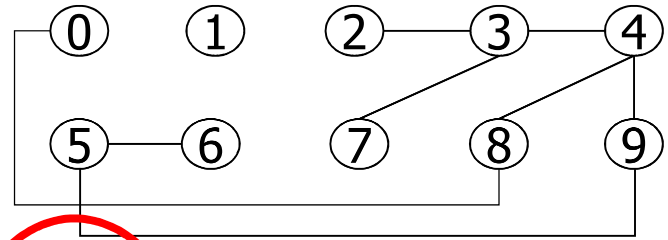
id	4	1	4	4	4	6	4	4	0	4
	0	1	2	3	4	5	6	7	8	9

id[id[id[p]]]=4 = id[id[q]]=4

unchanged

id	4	1	4	4	4	6	4	4	0	4
	0	1	2	3	4	5	6	7	8	9





p q = 0 2

id

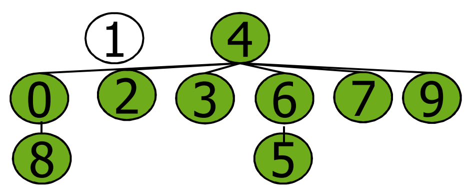
4	1	4	4	4	6	4	4	0	4
0	1	2	3	4	5	6	7	8	9

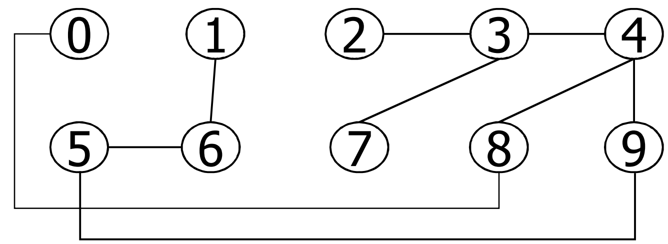
id[id[p]]=4 = id[id[q]]=4

unchanged

id

4	1	4	4	4	6	4	4	0	4
0	1	2	3	4	5	6	7	8	9





p q = 6 1

id

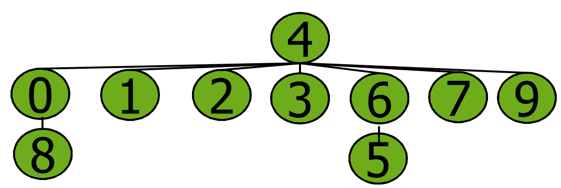
4	1	4	4	4	6	4	4	0	4
0	1	2	3	4	5	6	7	8	9

id[id[p]]=4 $\neq$ id[q]=1

let the smaller tree q pointo to the larger tree p: id[q]=4

id

4	4	4	4	4	6	4	4	0	4
0	1	2	3	4	5	6	7	8	9



```
...
  int i, j, p, q, id[N], sz[N];
  for(i=0; i<N; i++) { id[i] = i; sz[i] =1; }
  printf("Input pair p q:  ");
  while (scanf("%d %d", &p, &q) ==2) {
    for (i = p; i!= id[i]; i = id[i]);
    for (j = q; j!= id[j]; j = id[j]);
    if (i == j)
      printf("pair %d %d already connected\n", p,q);
    else {
      printf("pair %d %d not yet connected\n", p, q);
      if (sz[i] <= sz[j]) {
        id[i] = j; sz[j] += sz[i]; }
       else { id[j] = i; sz[i] += sz[j];}
    }
    printf("Input pair p q:  ");
  }
...
```

- find: scanning a “chain” of objects, cost at most logarithmic in the number of objects O(logn)
- union: simple, because it is enough that an object points to another object, unit cost O(1)
- globally the number of operations is bounded by

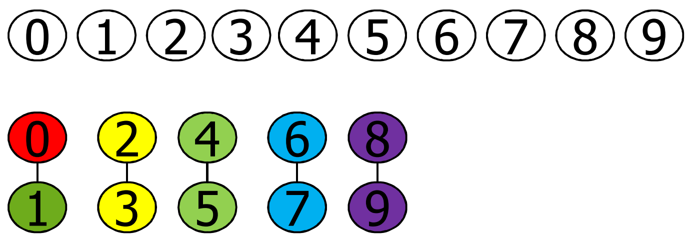
numb. of pairs * “chain” length

but the chain’s length grows logarithmically!

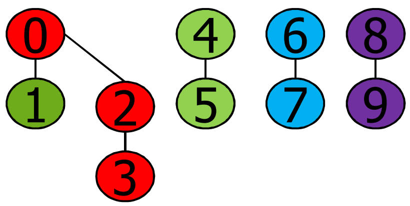
Why logatitithmic?

Worst-case: given n elements, each union connects 2 trees of the same size

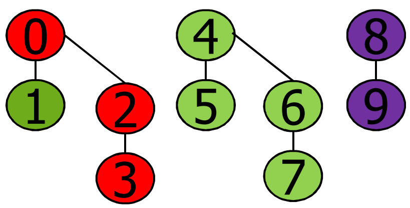
at least double the size of the set



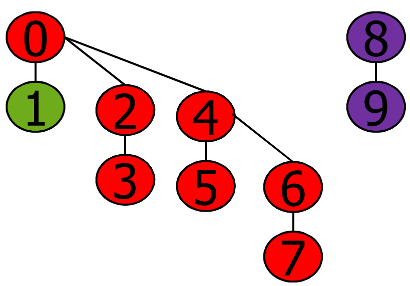
p q = 0 1
p q = 2 3
p q = 4 5
p q = 6 7
p q = 8 9



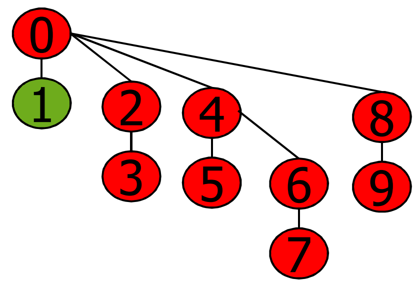
p q = 1 3



p q = 5 7



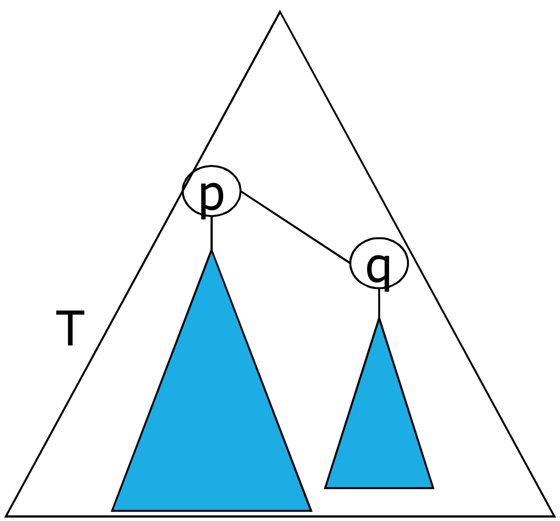
p q = 3 7



p q = 7 9

Each tree containing 2^h nodes has height h.
With a union operation, in the worst case, we merge 2 trees with the same number of nodes 2^h. The result is a tree with 2^{h+1} nodes, thus its height is h+1.
Height grows linearly with the number of union operations.

How many union operations are required?
If $T_1 \geq T_2$, each time we merge a smaller tree into a larger one, we create a tree whose size T is at least twice the size of T_2.



If, at each step, the number of elements in the tree doubles at least and if there are N elements, after i steps there will be at least 2^i elements in the tree. As the inequality $2^i \leq N$ holds, the number of union operations required is $i \leq \log_2 N$.