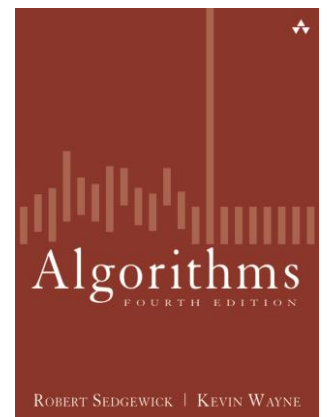


ID1020: Union-Find

Dr. Per Brand
pbrand@kth.se

kap. 1.5



Slides adapted from Algorithms 4th Edition, Sedgewick.

Att utveckla en algoritm

- Stegen för att utveckla en användbar algoritm:
 - Bygg en modell av problemet.
 - Hitta en algoritm för att lösa problemet.
 - Är algoritmen snabb nog? Tillräcklig minnessnål?
 - Om inte, kom på varför.
 - Hitta förbättring.
 - Iterera tills man är nöjd.
- Sedan, ska man analysera algoritm.

Dynamisk-konnektivitets problemet

- Givet en mängd av N objekt, programmet ska stödja två operationer:
 - **Union**: skapa en förbindelse mellan (*connect*) två objekt.
 - **Find**: Finns det en stig (*path*) som förbinder två objekt?
 - Om det finns en stig säger vi att objekten är i samma komponent

connect 4 and 3

connect 3 and 8

connect 6 and 5

connect 9 and 4

connect 2 and 1

are 0 and 7 connected? ✗

are 8 and 9 connected? ✓

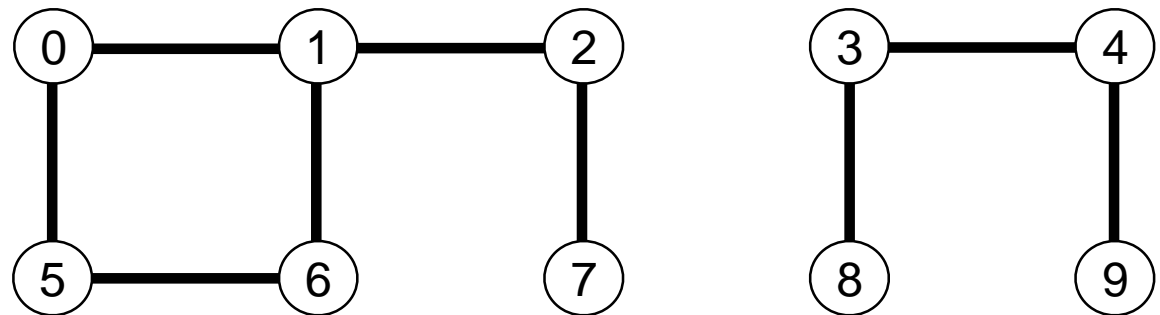
connect 5 and 0

connect 7 and 2

connect 6 and 1

connect 1 and 0

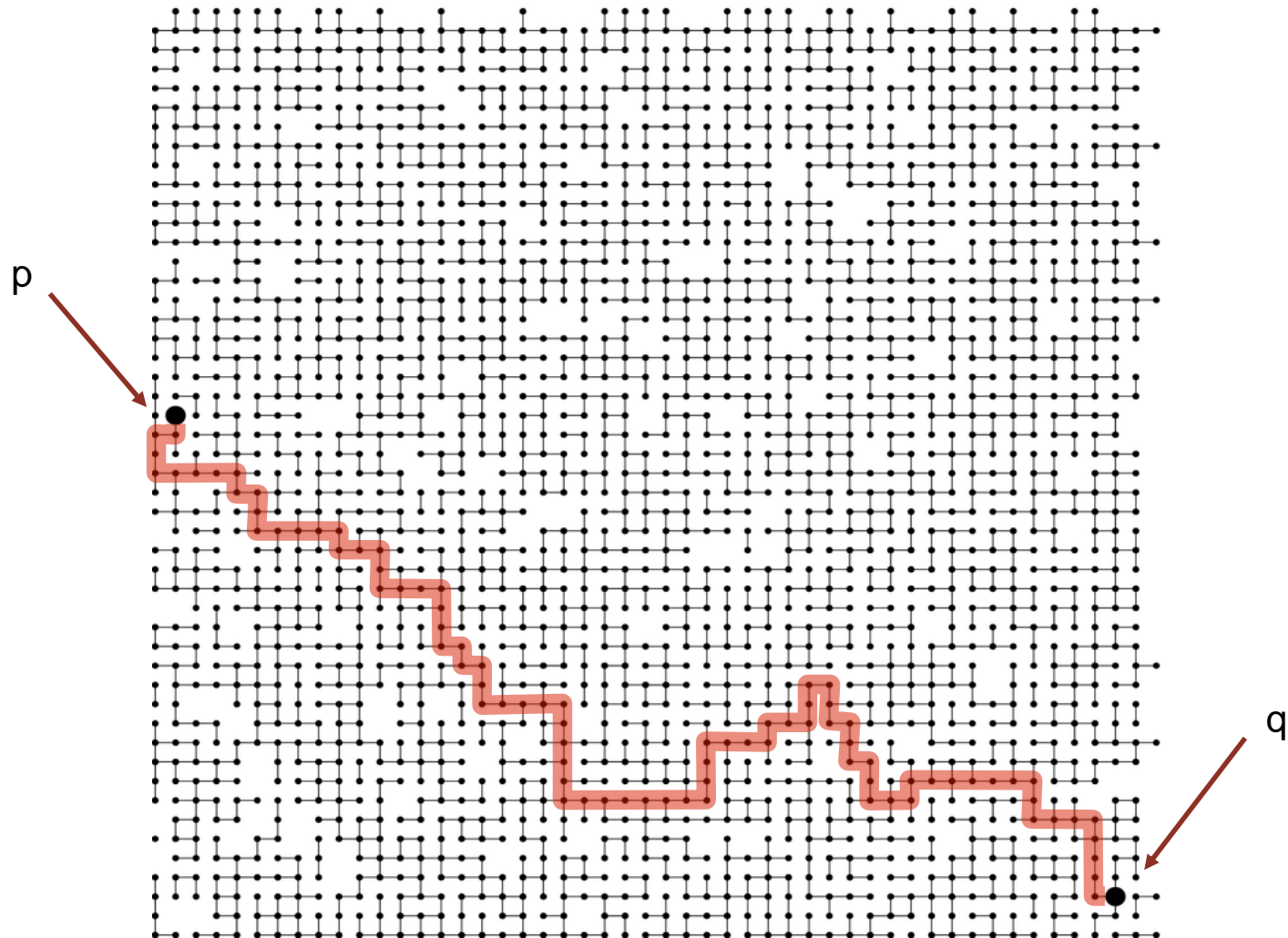
are 0 and 7 connected? ✓



Ett större konnektivitetsproblem

- Finns det en stig som förbinder p till q ?

Ja!



Bygg en modell av objekten

- Tillämpningar

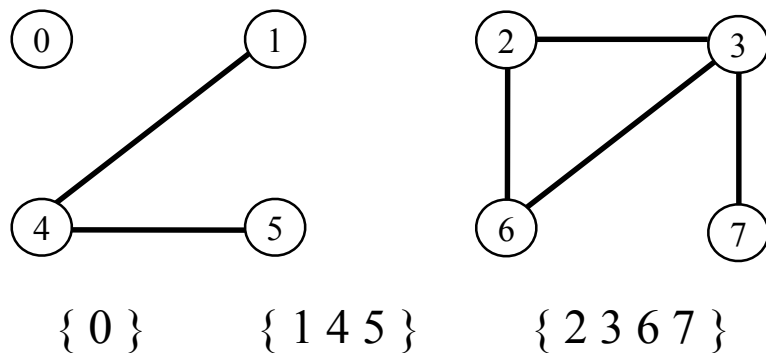
- Pixlar i en digitalbild.
- Dator i ett nätverk.
- Kompisar i ett socialt nätverk.
- Transistorer på ett datorchip.
- Element i en matematisk mängd.
- Variabelnamn i ett Fortran program.

- När man programmerar, underlättar det att benämna objekt 0 till $N - 1$.

- Använd ett heltal som en array index.
- Ta bort alla detaljer som inte är relevant till union-find.
- $a[i] == a[j]$ för alla objekt i samma komponent

Bygg en modell av förbindelserna

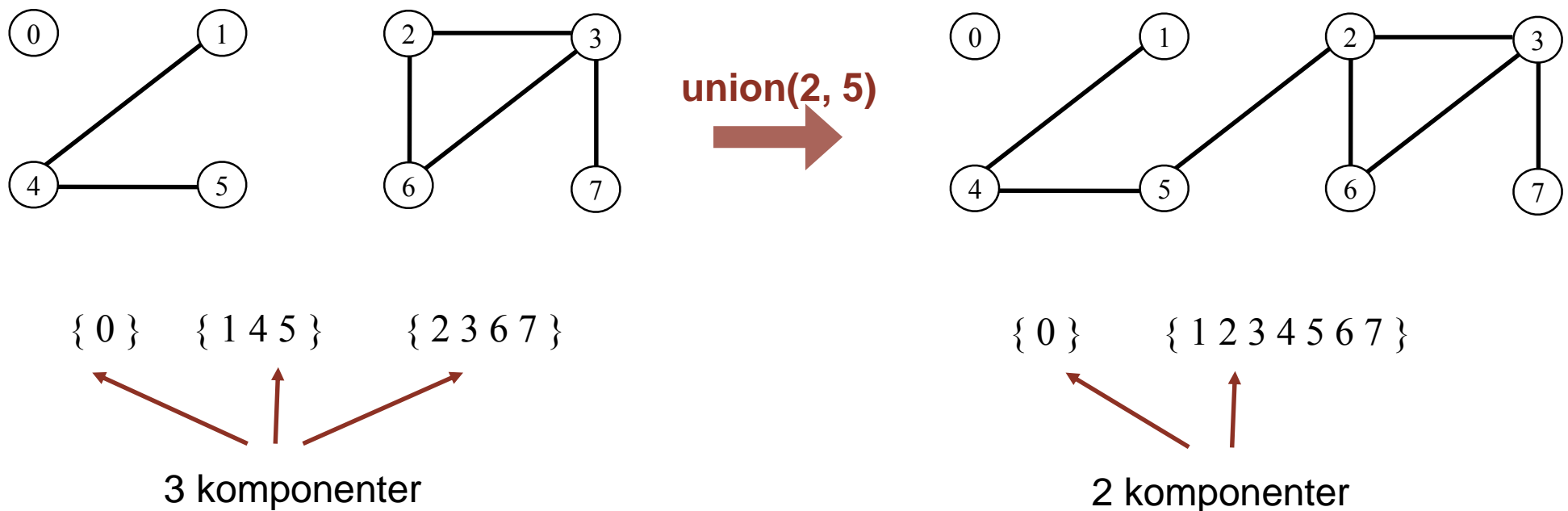
- Vi antar "är förbunden till" (*connected to*) är en ekvivalensrelation:
 - Reflexiv: p är förbunden till p .
 - Symmetrisk: om p är förbunden till q , q är förbunden till p .
 - Transitiv: om p är förbunden till q och q är förbunden till r , p är förbunden till r .
- En ekvivalensrelation partitionerar objekt i ekvivalensklasser som kallas för "connected components".
- **Sammanhängande komponent** (*connected component*). Största mängd av objekt som har ömsesidiga förbindelser.



3 (sammanhängande) komponenter

Att implementera Find och Union operationer

- **Find.** I vilken komponent hittar vi objekt p ?
- **Connected.** Finns objekt p och q i samma komponent?
- **Union.** Ersätt komponenterna som innehåller objekt p och q med en ny komponent som inkluderar både deras komponenterna , dvs deras "union".



Union-find datatyp (API)

- **Mål.** Bygg en effektiv datastruktur för union-find.
 - Antal objekt N kan vara enorm.
 - Antal operationer M kan också vara enorm.
 - Klienter kan anropa union och find operationer i vilken ordning som helst.

public class **UF**

UF(int N)

*initialize union-find data structure
with N singleton objects (0 to $N - 1$)*

void union(int p , int q)

add connection between p and q

int find(int p)

component identifier for p (0 to $N - 1$)

boolean connected(int p , int q)

are p and q in the same component?

connected()

- `connected()` kan implementeras med bara en kodrad:

```
public boolean connected(int p, int q) {  
    return find(p) == find(q);  
}
```

Ett exempel Klient till UF API:et

- Läs in ett antal objekt N från stdin (standard input).
- Upprepa stegen nedan:
 1. läs in ett par heltal från stdin
 2. om de inte är förbundna ännu, skapa en förbindelse mellan dem och skriv ut paret

```
public static void main(String[] args) {  
    int N = StdIn.readInt();  
    UF uf = new UF(N);  
    while (!StdIn.isEmpty()) {  
        int p = StdIn.readInt();  
        int q = StdIn.readInt();  
        if (!uf.connected(p, q)) {  
            uf.union(p, q);  
            StdOut.println(p + " " + q);  
        }  
    }  
}
```

% more tinyUF.txt

10 ← antal object N

4 3

3 8

6 5

9 4

2 1

8 9

5 0

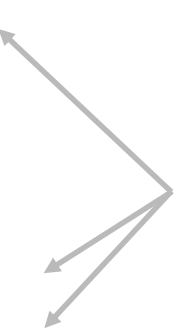
7 2

6 1

1 0

6 7

redan förbundna



Quick-find

Quick-find [ivrig metod]

- Datastruktur.

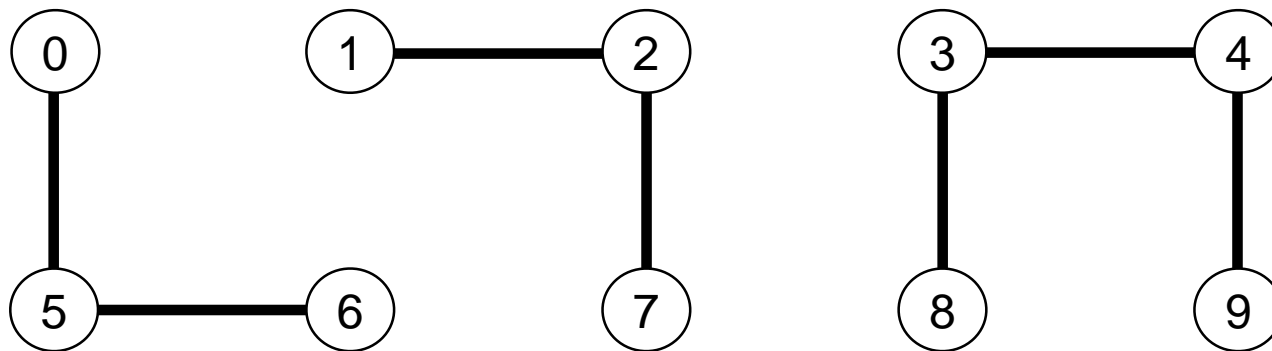
- Integer array `id[]` av längd `N`.
- Förklaring: `id[p]` är id:n av komponenten som innehåller `p`.
`p` och `q` är förbundna iff (om och endast om) de har samma `id`.

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

0, 5 och 6 är förbundna

1, 2, och 7 är förbundna

3, 4, 8, och 9 är förbundna



Quick-find

- **Datastruktur.**

- Integer array $id[]$ av längd N .
- Förklaring: $id[p]$ är id:n av komponenten som innehåller p .

	0	1	2	3	4	5	6	7	8	9
$id[]$	0	1	1	8	8	0	0	1	8	8

Find. Vad är id av p ?

Förbundna? Har p och q samma id ?

$id[6] = 0; id[1] = 1$

6 och 1 är inte förbundna

- **Union.** För att förbinda komponenter som innehåller p och q , ändra alla element vars $id = id[p]$ till $id[q]$.

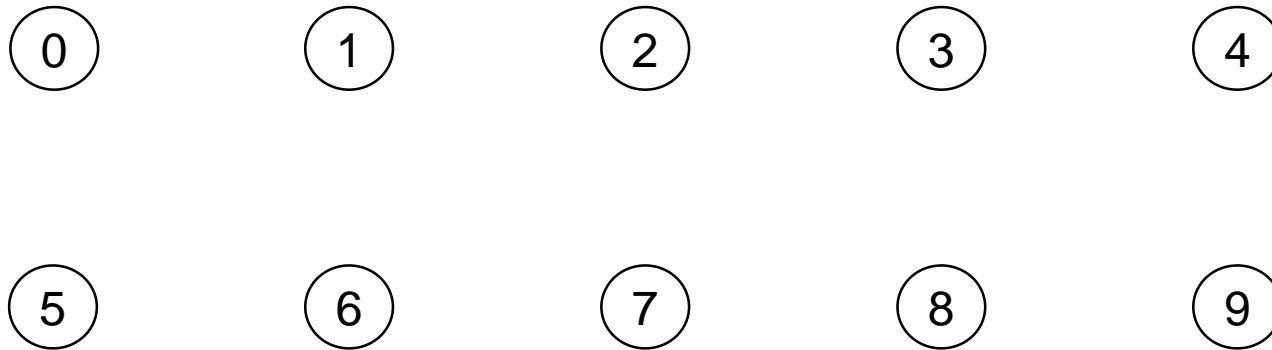
	0	1	2	3	4	5	6	7	8	9
$id[]$	1	1	1	8	8	1	1	1	8	8

↑ ↑ ↑

efter ha slagit hop 6 och 1

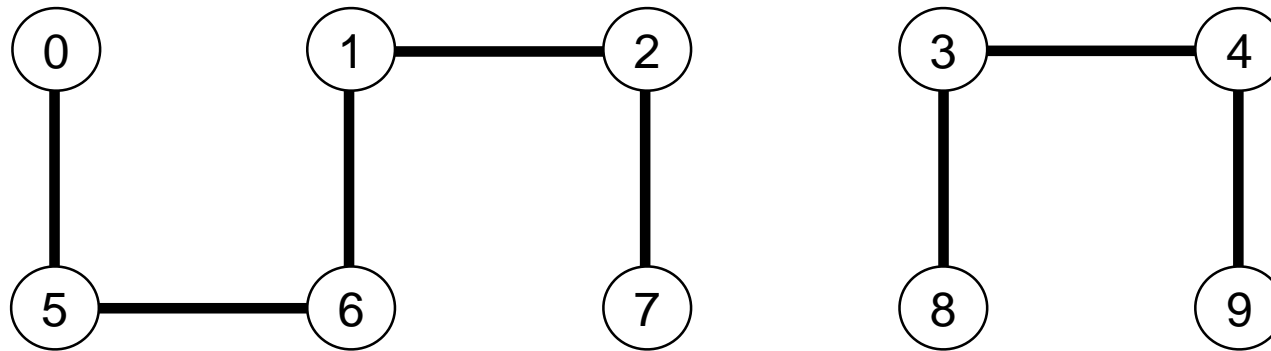
problem: många värden kan förändras

Quick-find demo



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

Quick-find demo



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

Quick-find: Java implementation

```
public class QuickFindUF {
```

```
    private int[] id;
```

```
    public QuickFindUF(int N)    {
```

```
        id = new int[N];
```

```
        for (int i = 0; i < N; i++) {
```

```
            id[i] = i;
```

```
        }
```

```
    }
```

```
    public boolean find(int p) { return id[p]; }
```

```
    public void union(int p, int q) {
```

```
        int pid = id[p];
```

```
        int qid = id[q];
```

```
        for (int i = 0; i < id.length; i++) {
```

```
            if (id[i] == pid)
```

```
                id[i] = qid;
```

```
        }
```

```
    }
```

```
}
```

← ge varje objekt samma komponent-id
som dess egen id (N array accesser)

← returnera id av p
(1 array access)


← ändra alla element med id[p] till id[q]
(tar som mest $2N + 2$ array accesser)

Quick-find är för långsam

- **Kostnadsmodell.** Antal array accesser (array läsningar och skrivningar).

algorithm	initialize	union	find	connected
quick-find	N	N	1	1

beräkningstiden för array accesser

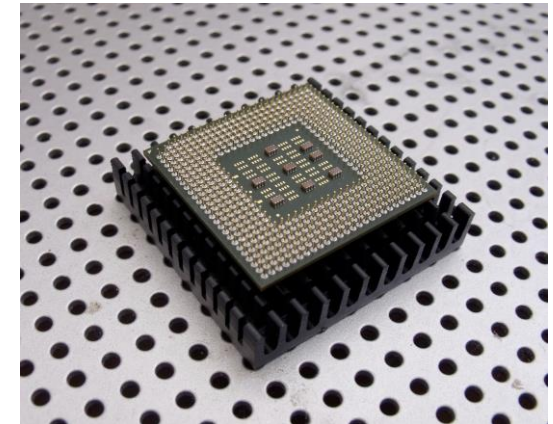
- **Union är för dyrt.** Det tar N^2 array accesser för att bearbeta en sekvens av N union operationer på N objekt.
 kvadratisk

Kvadratisk algoritmer skalar inte

- Tumregeln (för nu).

- 10^9 operationer per sekund.
- 10^9 minnesord av huvudminne.
- Kan komma åt alla minnesord på ungefär 1 sekund.

förhållanden har gällt
sedan 1950!

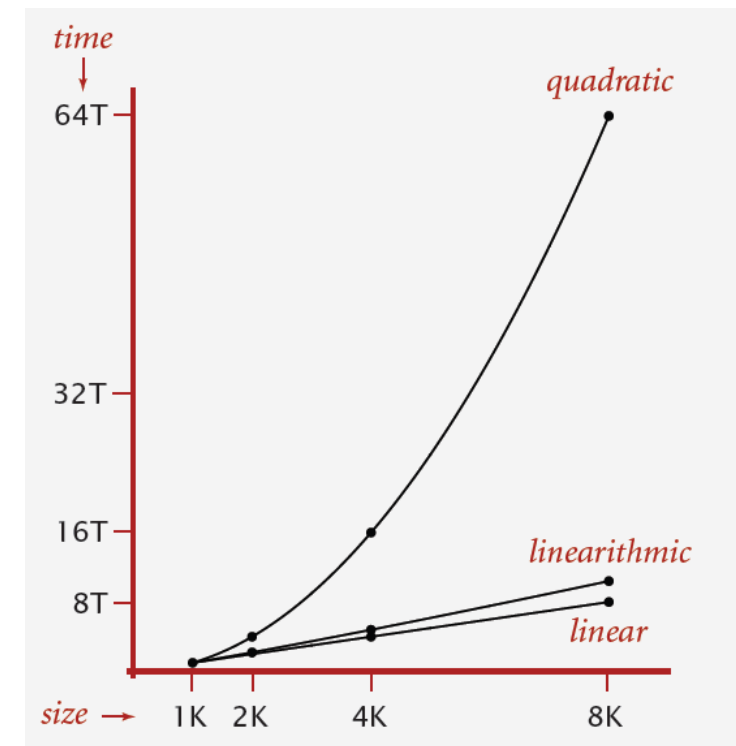


- Stort problem för quick-find:

- Med, 10^9 union operationer på 10^9 objekt, skulle quick-find exekverar mer än 10^{18} operationer \Rightarrow 30+ år av beräkningstid!

- Kvadratisk algoritmer skalar inte med bättre teknologi.

- Ny dator kan vara 10x snabbare.
- Men, datorn har 10x mer minnet \Rightarrow vi vill lösa ett problem som är 10x större.
- Med en kvadratisk algoritm, \Rightarrow 10x långsammare!



Quick Union

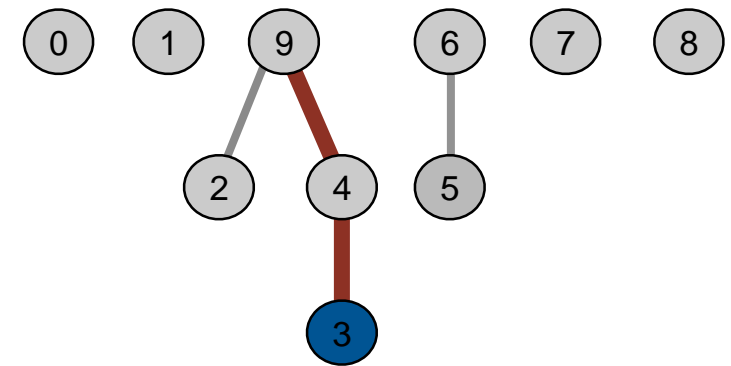
Quick-union [lat metod]

- Datastruktur.

- Integer array `id[]` av längd `N`.
- Förklaring: `id[i]` är förälder av `i`.
- **Roten (Root)** av `i` är
`id[id[id[...id[i]...]]]`.

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

fortsätta tills den inte förändras
(inga cyklar i algoritm)



föräldern av 3 är 4
roten av 3 är 9

Quick-union

- **Datastruktur.**

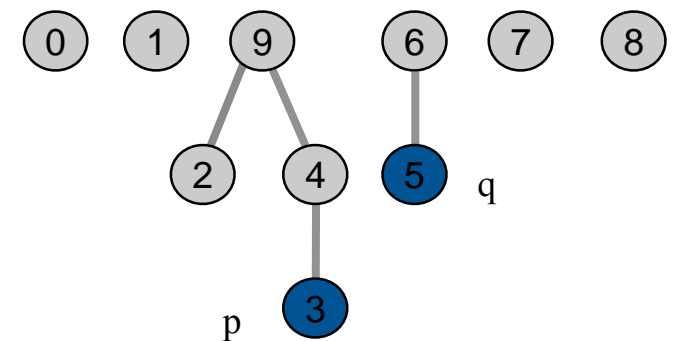
- Integer array `id[]` av längd `N`.
- Förklaring: `id[i]` är förälder av `i`.
- **Roten** av `i` är `id[id[id[...id[i]...]]]`.

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

- **Find.** Vad är roten av `p`?
- **Connected.** Delar `p` och `q` samma rot?
- **Union.** För att förbinda komponenter som innehåller `p` och `q`, tilldelar `id` av `p`'s rot till `id` av `q`'s rot.

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

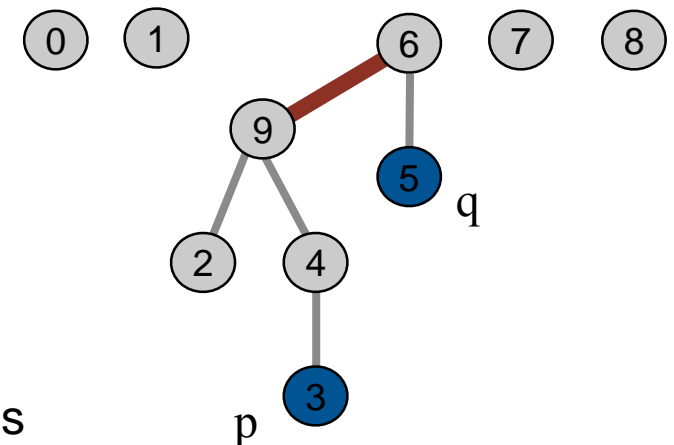
bara ett värde förändras



roten av 3 är 9

roten av 5 är 6

3 och 5 är inte förbundna

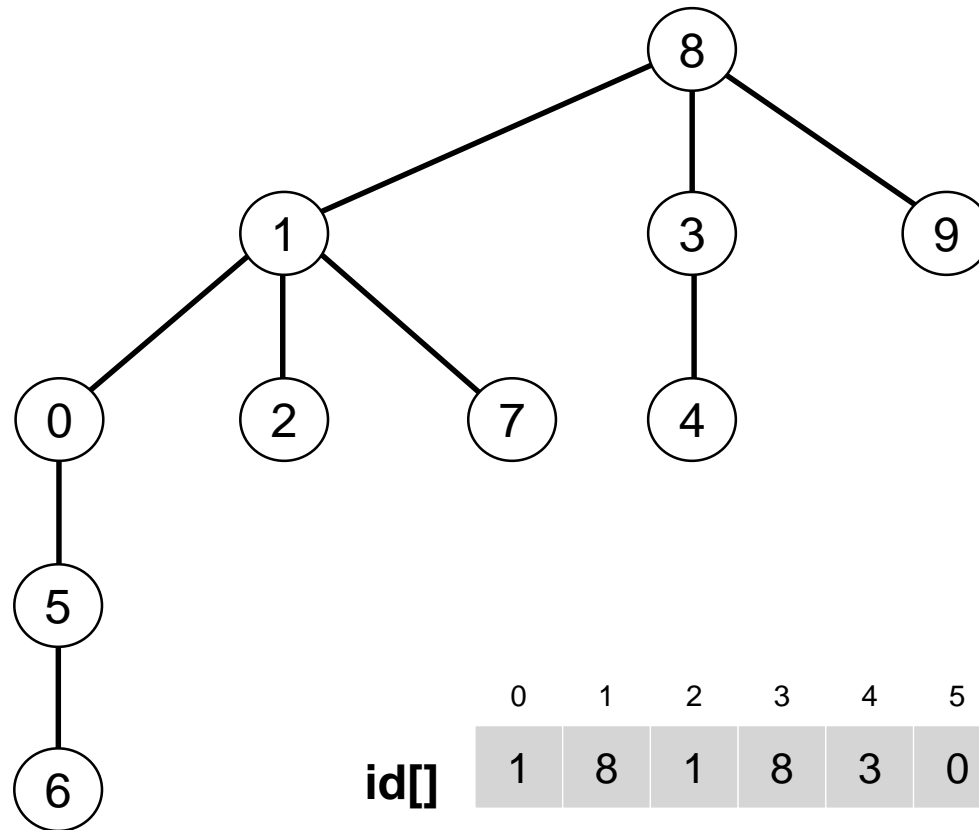


Quick-union demo



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

Quick-union demo



Quick-union: Java implementation

```
public class QuickUnionUF {  
    private int[] id;  
    public QuickUnionUF(int N) {  
        id = new int[N];  
        for (int i = 0; i < N; i++) {  
            id[i] = i;  
        }  
    }  
    public int find(int i) {  
        while (i != id[i]) {  
            i = id[i];  
        }  
        return i;  
    }  
    public void union(int p, int q) {  
        int i = find(p);  
        int j = find(q);  
        id[i] = j;  
    }  
}
```

← ge varje objekt samma komponent-id
som dess egen id (N array accesser)

← följ föräldrapekare tills man når roten
(djupet av trädet array accesser)

← byt roten av p för att peka till roten av q
(det tar djupet av p och q array accesser)

Quick-union är också för långsamt

Kostnadsmodell. Antal array accesser.

algorithm	initialize	union	find	connected
quick-find	N	N	1	1
quick-union	N	N [†]	N	N ← worst case

† inkluderar kostnaden av att hitta roten

Quick-find defekt.

- Union är för dyrt. (N array accesser).
- Träden är "flat", men det kostar för mycket att behålla dem *flat*.

Quick-union defekt.

- Träden kan bli för djupa.
- Find/connected för dyrt (kan ta N array accesser).

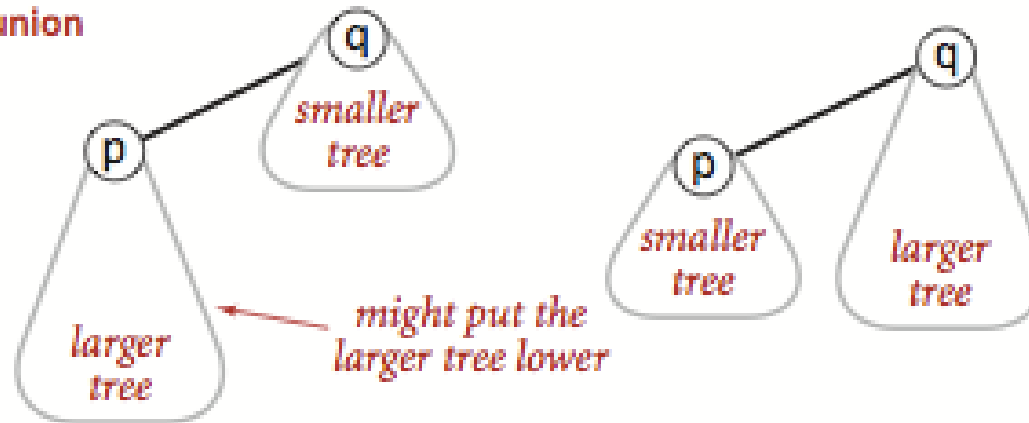
Quick-union förbättringar

Förbättring 1: Viktning

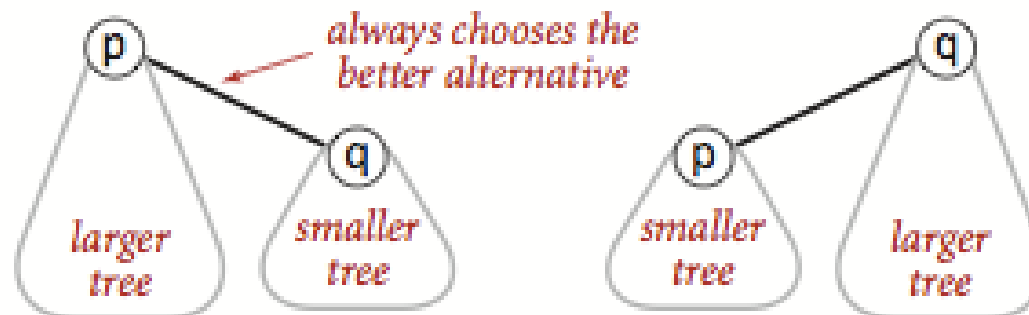
- Viktad quick-union.

- Ändra på quick-union för att undvika djupa träd.
- Spara storleken av trädet (dvs., antal objekt i trädet) i ett attribut.
- Balansera träden genom att länka roten av det mindre trädet till roten av det större trädet.

quick-union



weighted



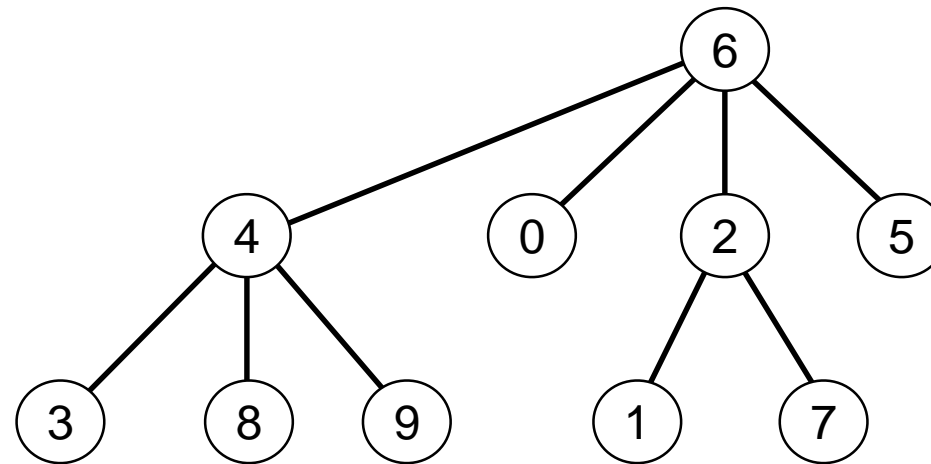
Weighted quick-union

Viktad quick-union demo



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

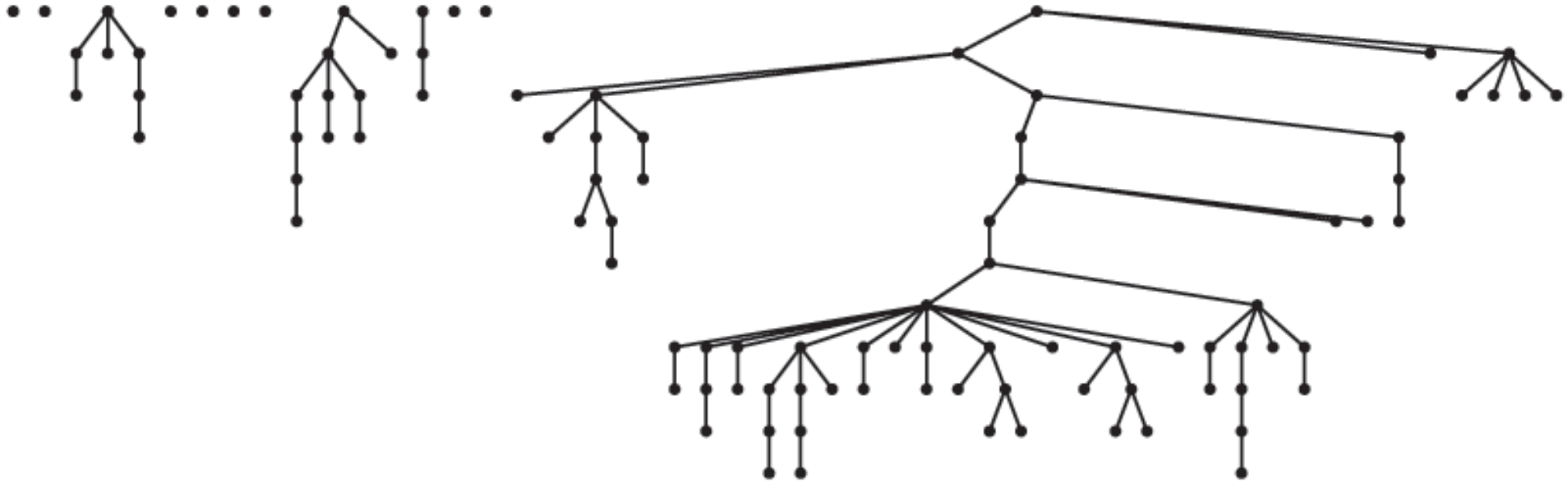
Viktad quick-union demo



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

Quick-union och viktad quick-union exempel

quick-union



average distance to root: 5.11

weighted



average distance to root: 1.52

Quick-union and weighted quick-union (100 sites, 88 union() operations)

Viktad quick-union: Java implementation

- **Datastruktur.** Samma som quick-union, men vi behöver en extra array `sz[i]` för att räkna antal objekt i trädet med roten `i`.
- **Find/connected.** Samma som quick-union.
- **Union.** Ändringar till quick-union:
 - Länka roten av det mindre trädet till roten av det större trädet.
 - Uppdatera `sz[]` array.

```
int i = find(p);
int j = find(q);
if (i == j) return;
if (sz[i] < sz[j]) {
    id[i] = j; sz[j] += sz[i];
} else {
    id[j] = i; sz[i] += sz[j];
}
```

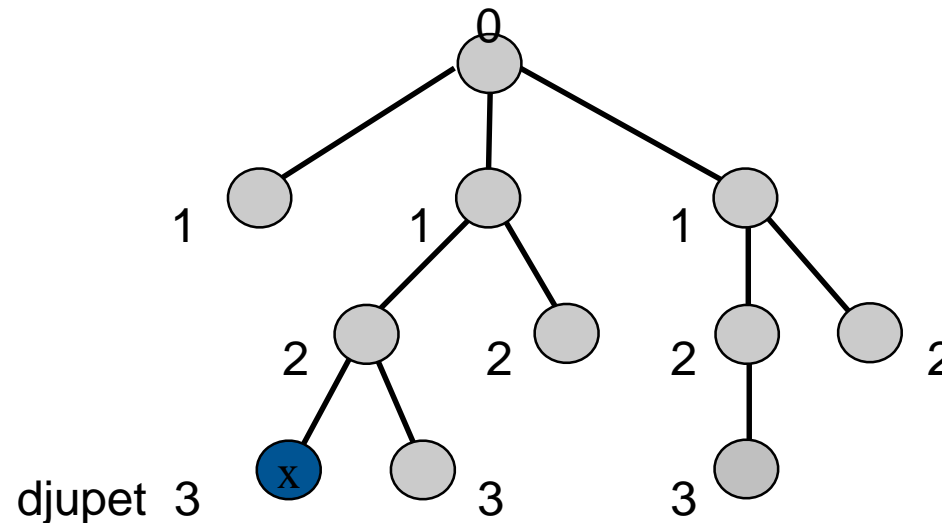
Viktad quick-union analys

- Körtiden.

- Find: tar tid proportionellt mot djupet av p .
- Union: tar konstant tid, givet rotnoderna.

\lg = base-2 logarithm

- Sats. Djupet av nod x är maximalt $\lg N$.



$$N = 11$$

$$\text{djupet}(x) = 3 \leq \lg N$$

Viktad quick-union analys

- **Körtiden.**

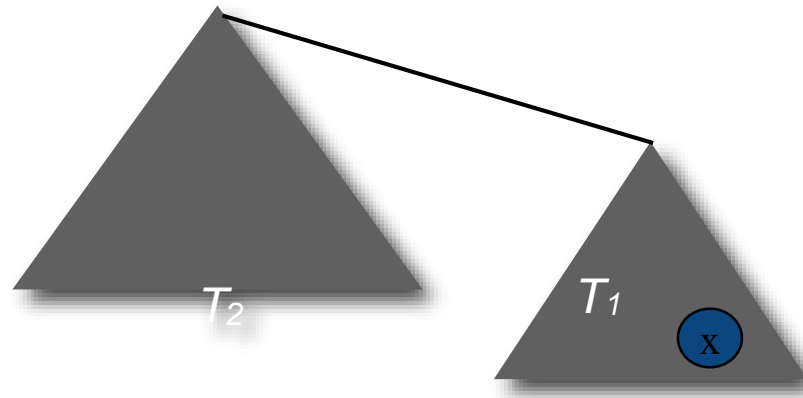
- Find: tar tid proportionellt mot djupet av p .
- Union: tar konstant tid, givet rotnoderna.

\lg = base-2 logarithm

- **Sats.** Djupet av nod x är maximalt $\lg(N)$.

- **Bevis.** Vad är det som gör att djupet av objekt x ökar?

- Ökar med 1 när trädet T_1 som innehåller x sammanfogar (*merges*) med ett annat träd T_2 .
- Storleken av trädet som innehåller x minst fördubblas eftersom $|T_2| \geq |T_1|$.
- Storleken av trädet som innehåller x kan fördubblas som mest $\lg N$ gånger. Varför?



1
2
4
8
16
:
N

} $\lg N$

Viktad quick-union analys

- **Körtiden.**
 - Find: tar tid proportionellt mot djupet av p .
 - Union: tar konstant tid, givet rotnoderna.
- **Sats.** Djupet av nod x är maximalt $\lg(N)$.

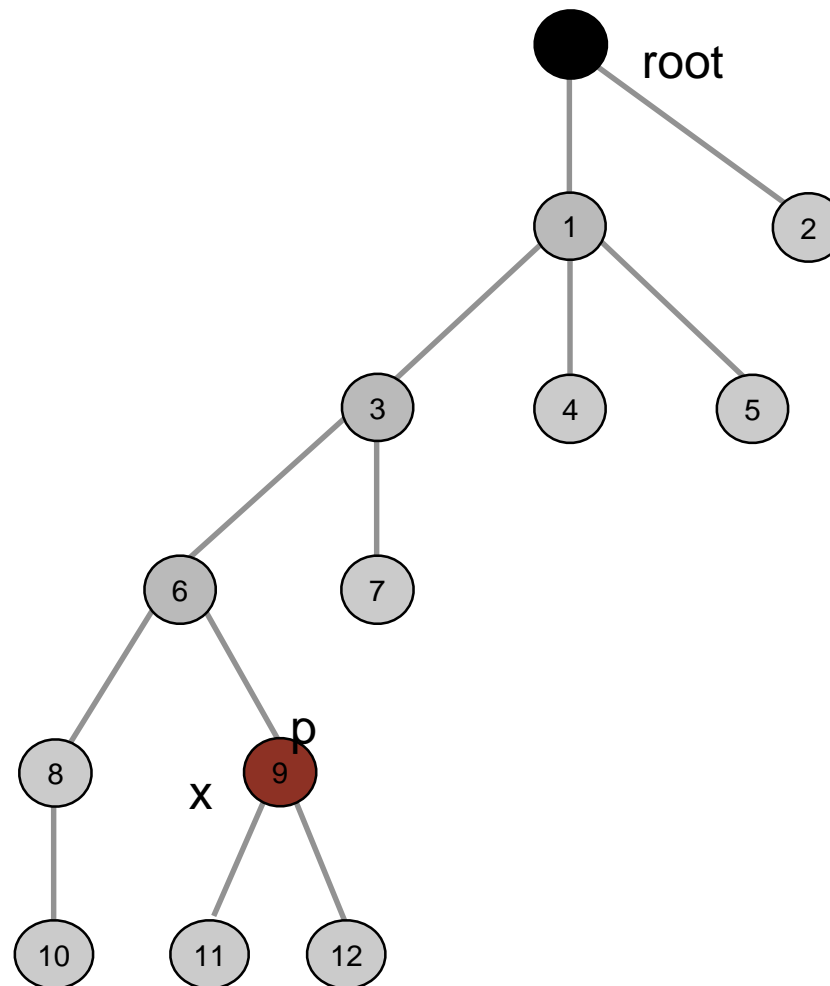
algorithm	initialize	union	find	connected
quick-find	N	N	1	1
quick-union	N	N^{\dagger}	N	N
viktad QU	N	$\lg N^{\dagger}$	$\lg N$	$\lg N$

\dagger inkluderar kostnaden av att hitta roten

Kommer algoritmen att terminera med garanterad acceptabelt prestanda?
Nej, men det går lätt att förbättra den.

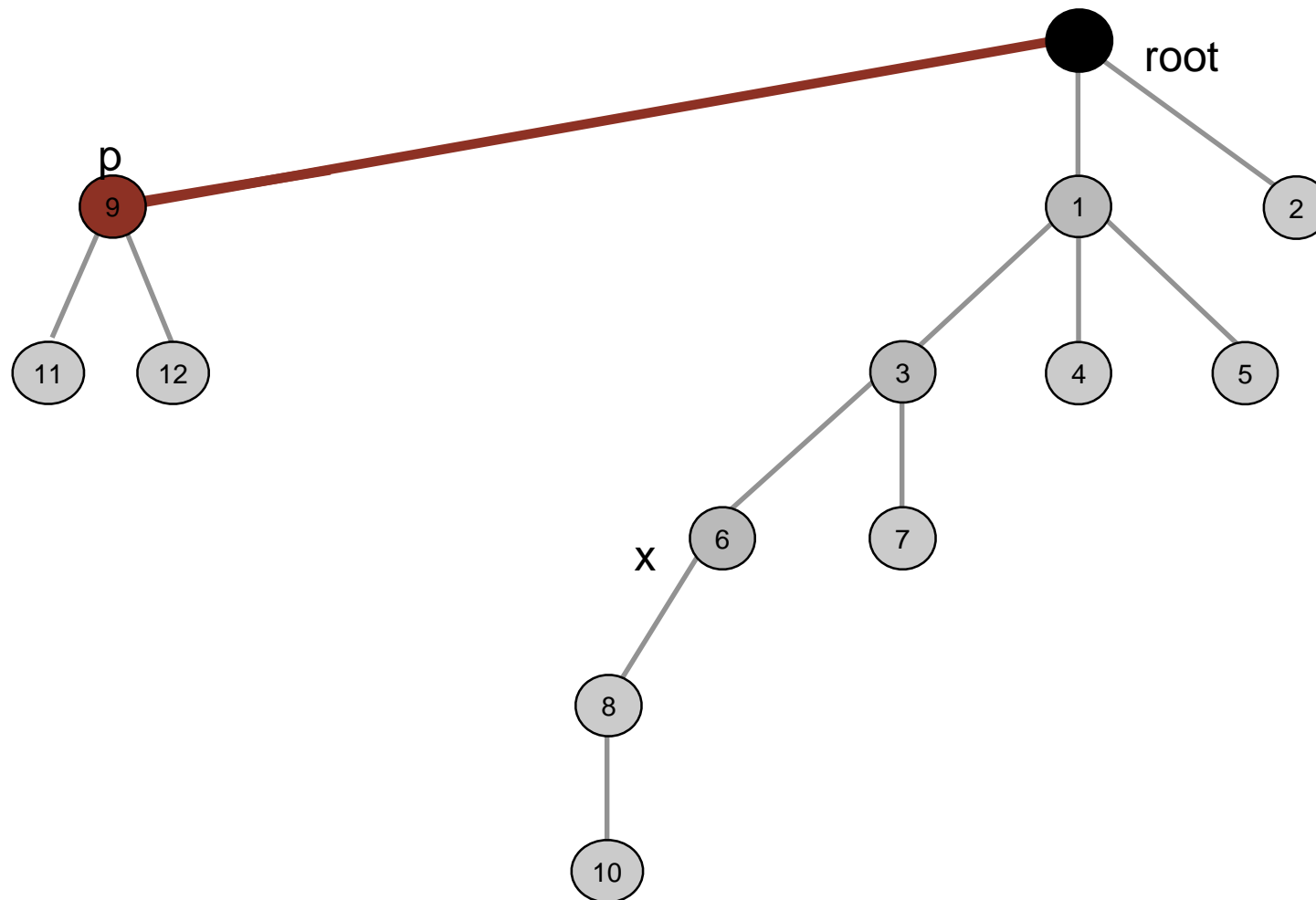
Förbättring 2: stig komprimering

- Quick union med stig komprimering (*path compression*). Precis efter ha beräknat roten av p , sätt $id[]$ av varje examinerade nod så att den pekar till denna roten.



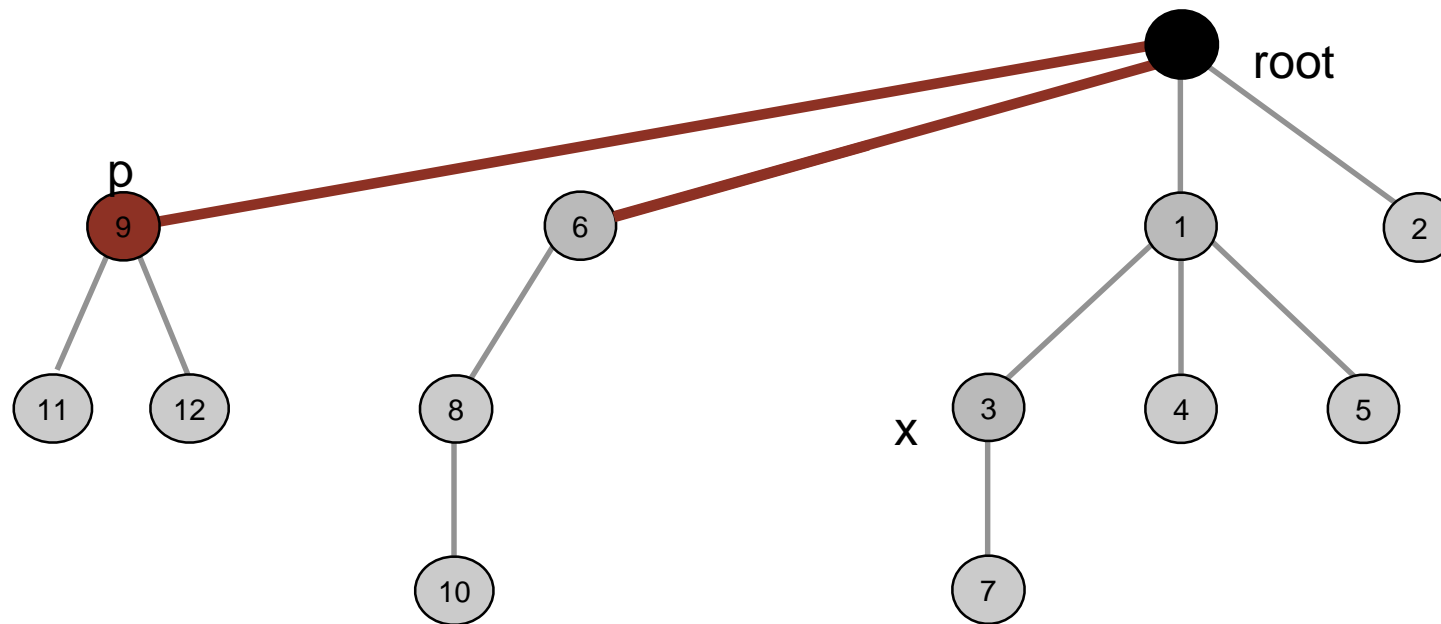
Förbättring 2: stig komprimering

- **Quick union med stig komprimering.** Precis efter ha beräknat roten av p , sätt $id[]$ av varje examinerade nod så att den pekar till denna roten.



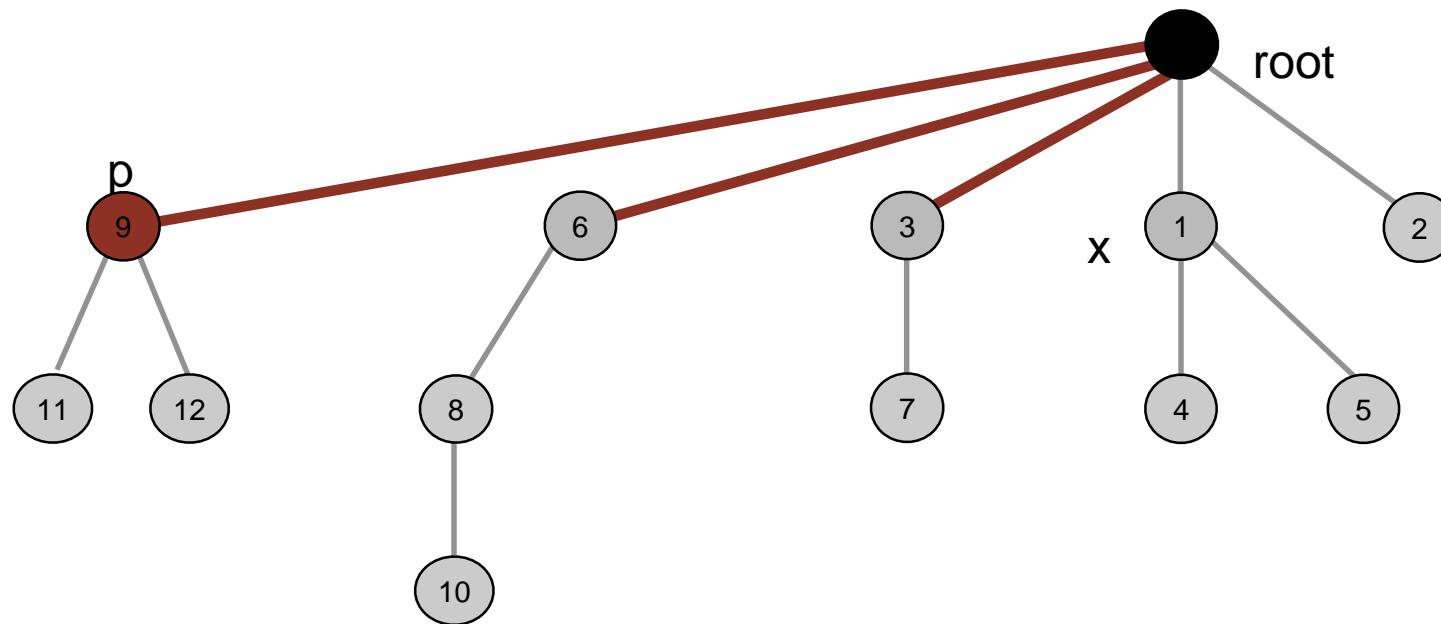
Förbättring 2: stig komprimering

- **Quick union med stig komprimering.** Precis efter ha beräknat roten av p , sätt $id[]$ av varje examinerade nod så att den pekar till denna roten.



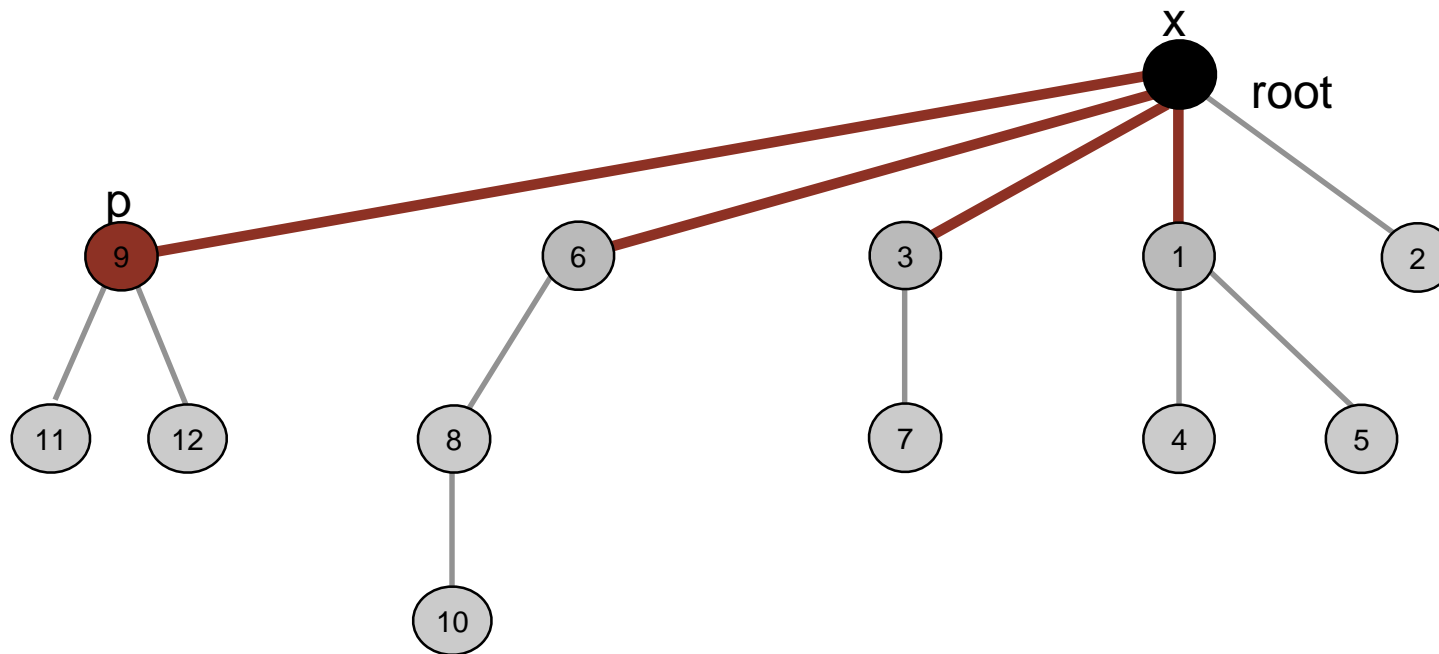
Förbättring 2: stig komprimering

- **Quick union med stig komprimering.** Precis efter ha beräknat roten av p , sätt $id[]$ av varje examinerade nod så att den pekar till denna roten.



Förbättring 2: stig komprimering

- **Quick union med stig komprimering.** Precis efter ha beräknat roten av p , sätt $id[]$ av varje examinerade nod så att den pekar till denna roten.




Resultatet. Nu, `find()` har bi-effekten att den komprimerar trädets djupet av trädets .

Stig komprimering: Java implementation

- **Tvåstegs implementation:** lägg till en andra loop till `find()` för att tilldela `id[]` av varje examinerade nod till roten.
- **Enklare ettstegs-variant (stighalvering):** Gör att varannan nod i stigen pekar till sin farfar (*grandparent*).

```
public int find(int i) {  
    while (i != id[i]) {  
        id[i] = id[id[i]];  
        i = id[i];  
    }  
    return i;  
}
```

bara en extra rad av kod
behövdes



I praktiken. Det finns ingen anledning att inte komprimera stigen! Den behåller träden nästan "flat".

Viktad quick-union with stig komprimering: amorterad analys

- **Sats.** [Hopcroft-Ulman, Tarjan] Om man börjar med en tom datastruktur, vilken sekvens som helst av M union-find operationer på N objekt gör mindre än $c (N + M \lg^* N)$ array accesser.
 - Analys kan förbättras till $N + M \alpha(M, N)$.
 - Enkel algoritm med fascinerande matematik.

N	$\lg^* N$
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

itererade lg funktion

Linjärtid algoritm för M union-find operationer på N objekt?

- Kostnad inom en konstant faktor av inläsningen av data.
- I teori är WQUPC (weighted quick-union with path compression) inte helt linjär.
- I praktiken är WQUPC linjär.

Häpnadsväckande fakta: Ingen linjärtid ~~algoritm~~ finns [Fredman-Saks] .
i "cell-probe" beräkningsmodellen

Sammanfattning

- **Slutsats.** Viktad quick union (och/eller stig komprimering) möjliggör att lösa problem som annars inte hade kunnat lösas.

algorithm	worst-case tid
quick-find	$M \cdot N$
quick-union	$M \cdot N$
viktad QU	$N + M \log N$
QU + stig komprimering	$N + M \log N$
viktad QU + stig komprimering	$N + M \lg^* N$

beräkningstid för M union-find operationer på en mängd av N objekt

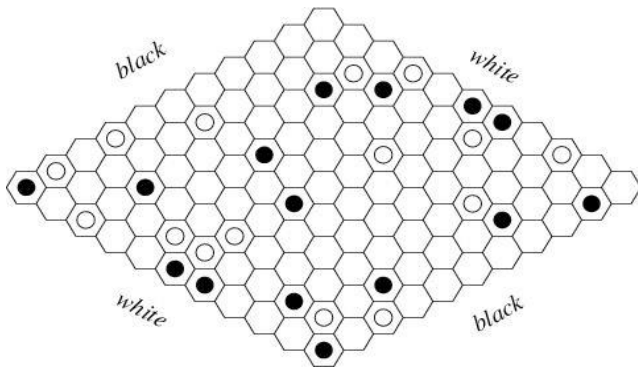
Ex. [10^9 unions och finds med 10^9 objekt]

- WQUPC minskar tiden från 30 år till 6 sekunder.
- En Supercomputer hjälper inte mycket; det är en bra algoritm som möjliggör lösningen.

Union-Find applikationer

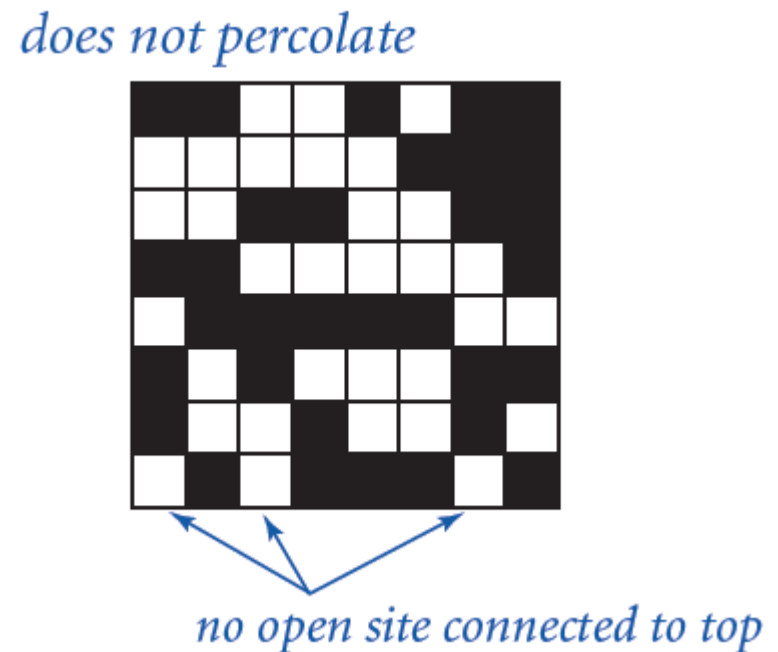
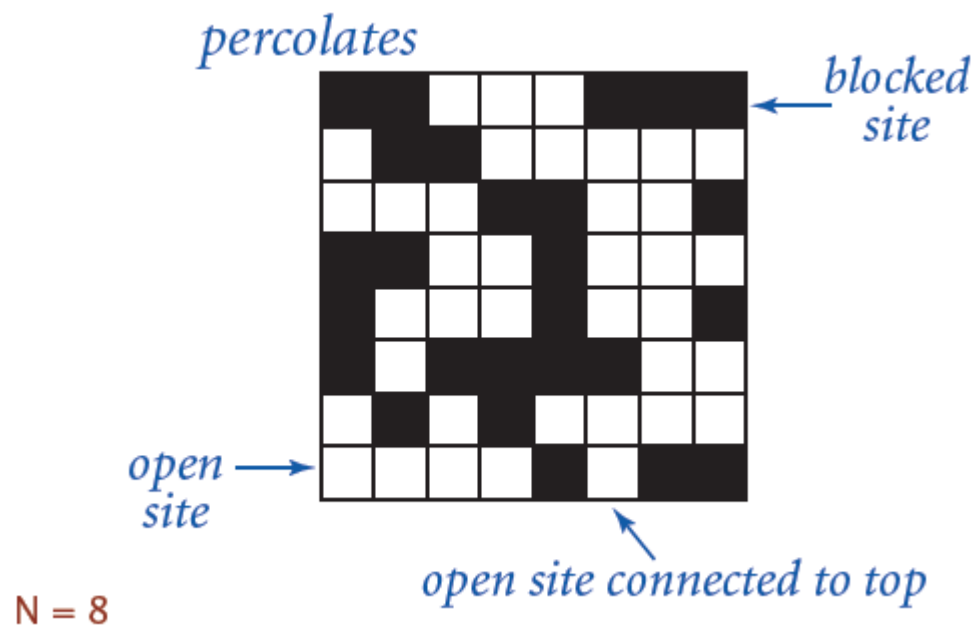
Union-find applikationer

- Perkolation.
- Spel (Go, Hex).
- ✓ **Dynamisk konnektivitet.**
- Least common ancestor.
- Ekvivalens av ändliga tillståndsautomater.
- Hinley-Milner polymorphiskt typinferens.
- Kruskal's minsta uppspännande träd algoritm.
- Matlab's `bwlabel()` funktion i bildbehandling.



Perkolation

- En abstrakt modell av många fysiska system:
 - $N \times N$ rutnät av platser (*sites*).
 - Varje plats är öppen med sannolikheten p (och blockerad med sannolikheten $1 - p$).
 - Systemet **perkolerar** iff topp och botten är förbundna genom öppna platser.



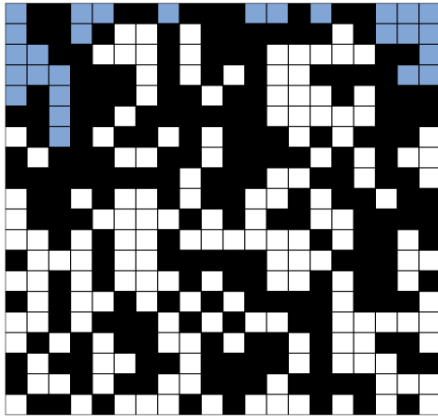
Perkolation

- En abstrakt modell av många fysiska system:
 - $N \times N$ rutnät (grid) av platser (*sites*).
 - Varje plats är öppen med sannolikheten p (och blockerad med sannolikheten $1 - p$).
 - Systemet **perkolerar** iff topp och botten är förbundna genom öppna platser.

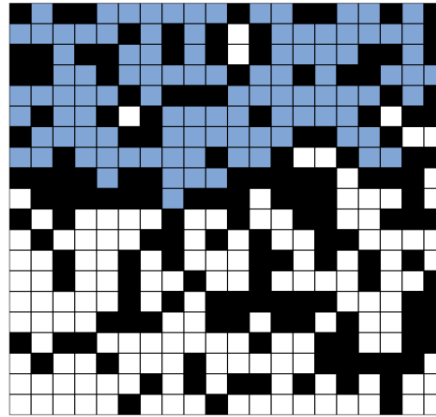
model	system	vacant site	occupied site	percolates
electricity	material	conductor	insulated	conducts
fluid flow	material	empty	blocked	porous
social interaction	population	person	empty	communicates

Sannolikheten av perkolation

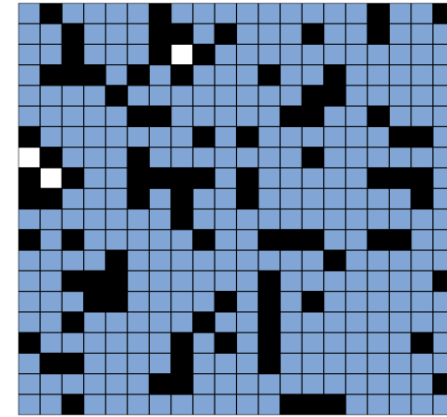
- Beror på grid storleken N och ledig plats (*site vacancy*) sannolikheten p .



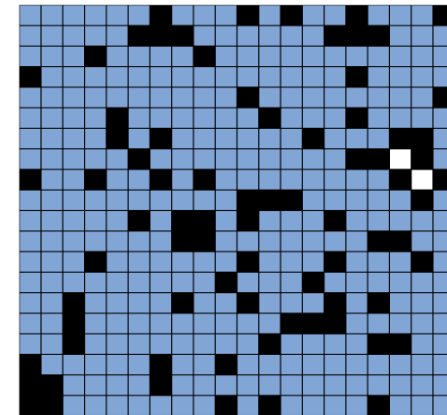
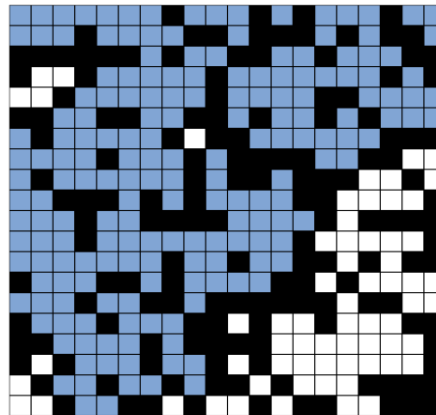
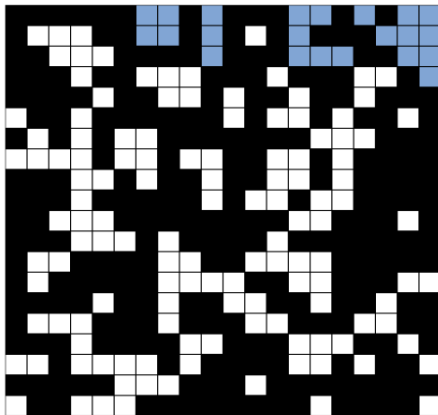
p low (0.4)
does not percolate



p medium (0.6)
percolates?



p high (0.8)
percolates

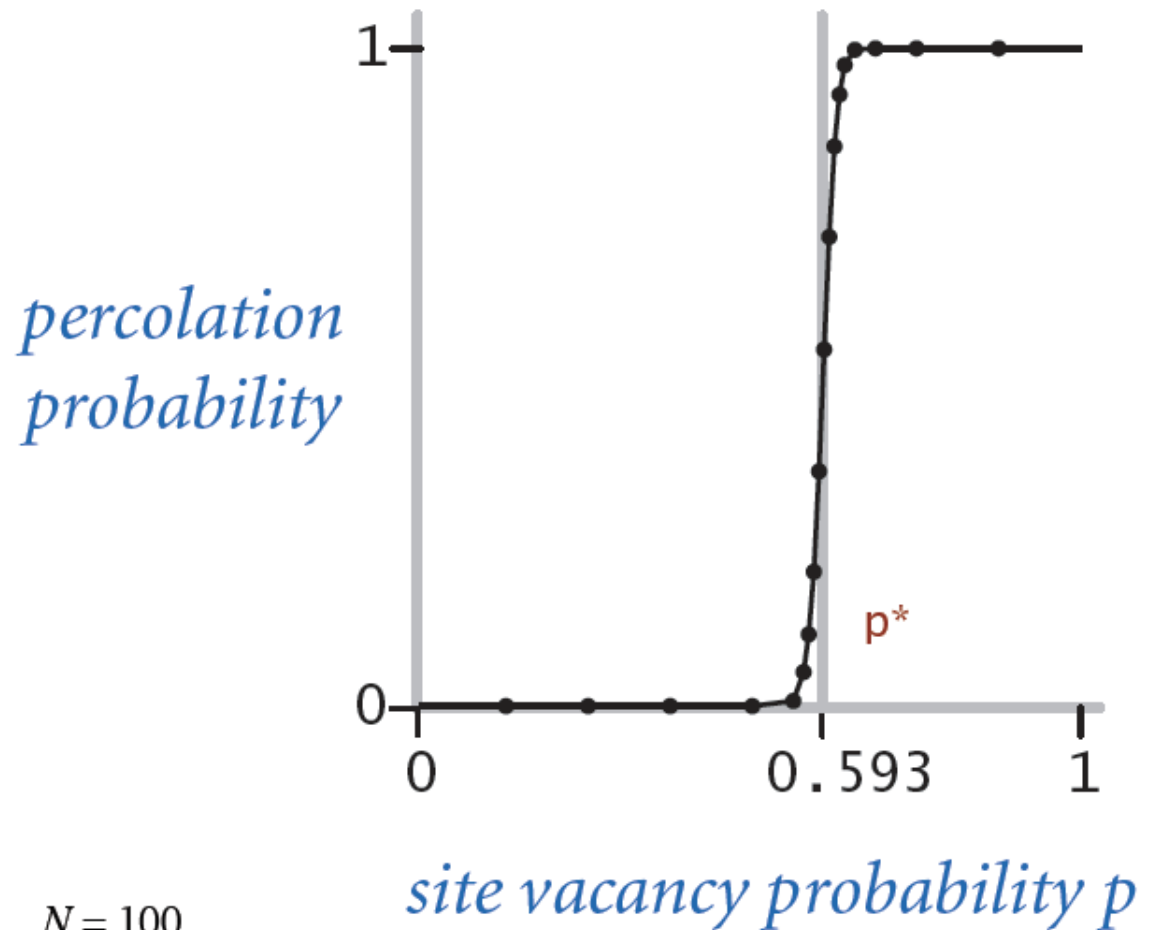


Perkolation fasövergång

- När N är stor, teori garanterar en skarp tröskel p^* .

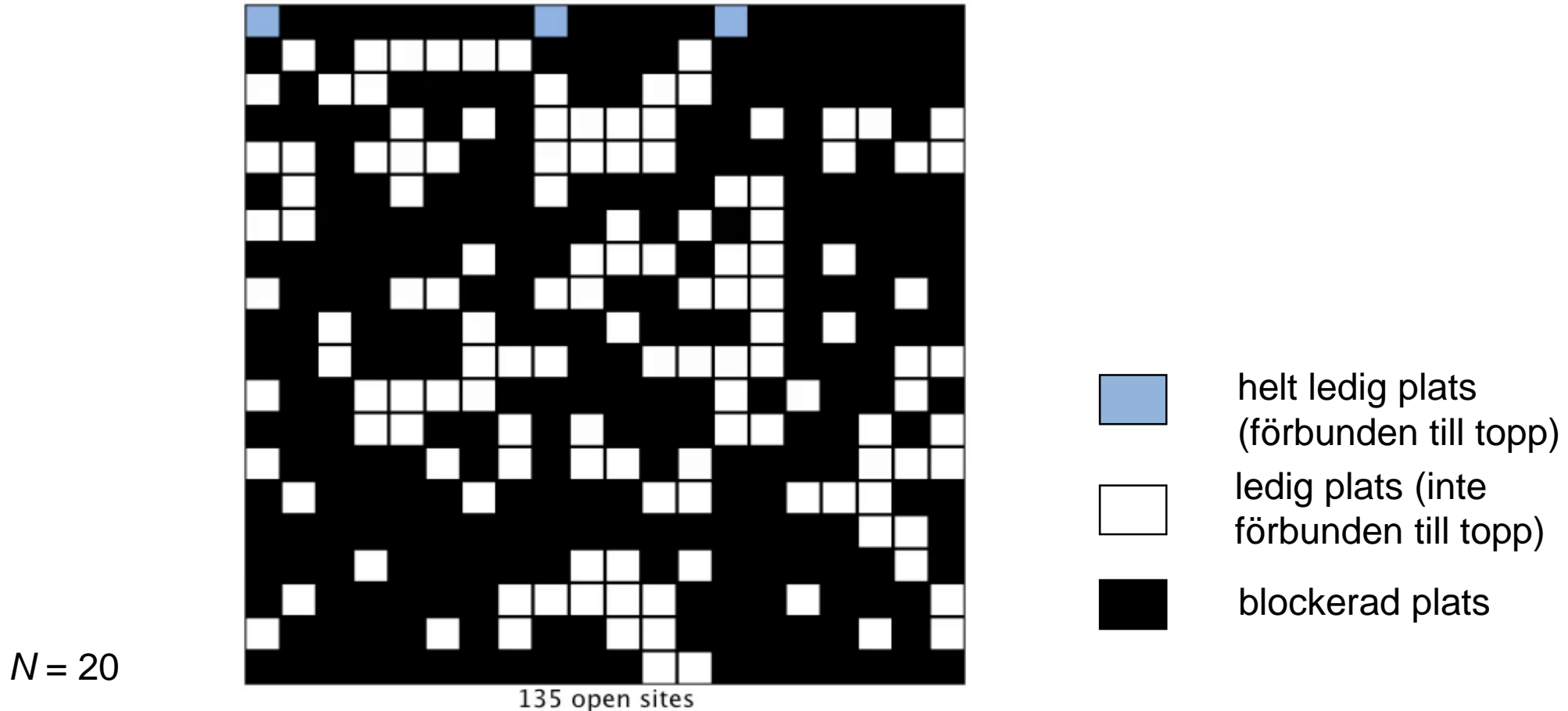
- $p > p^*$: perkulerar med väldigt hög sannolikhet.
- $p < p^*$: perkulerar med väldigt låg sannolikhet.

- Vad är värdet av p^* ?



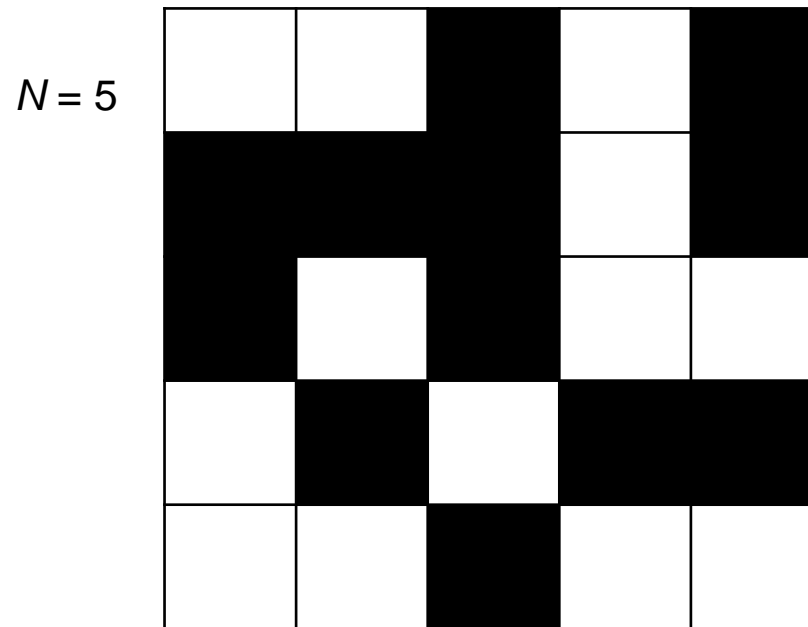
Monte Carlo simulering



- Initialisera alla platser i en $N \times N$ grid så att deras tillstånd är blockerad (*blocked*).
- Gör slump platser lediga tills det finns en stig mellan toppen och botten.
- Ledighet procentsats (*vacancy percentage*) uppskatter p^* .



Dynamisk konnektivitet lösning för att uppskatta perkolationströskeln

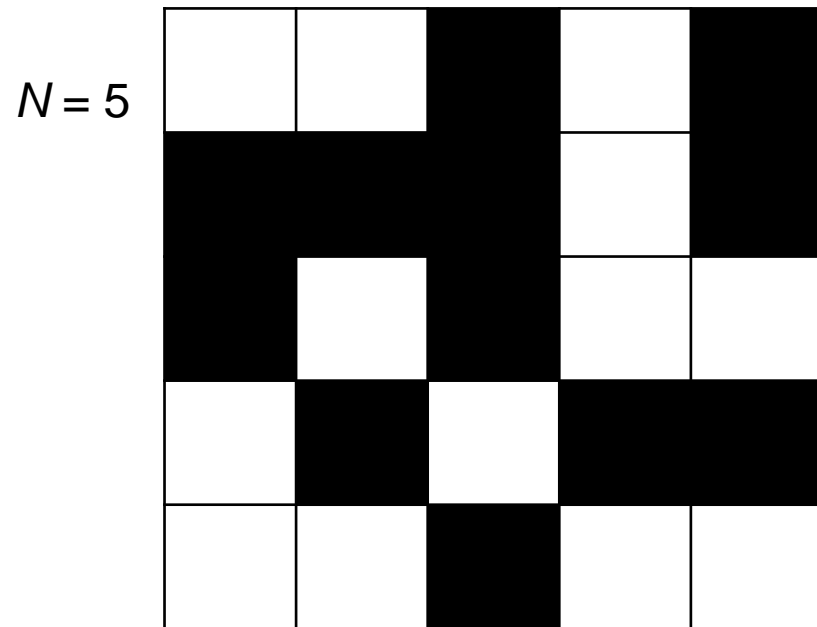
- Hur kan man kontrollera om ett $N \times N$ system perkolerar?
Modellera som ett **dynamisk konnektivitet** problem och använd **union-find**.




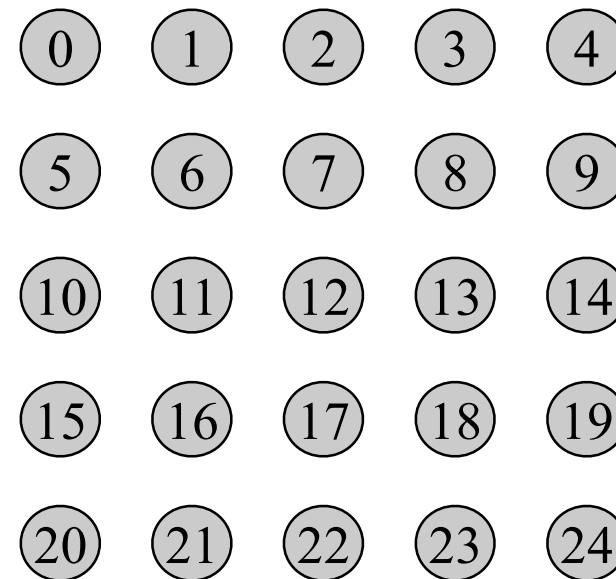
 ledig plats
 blockerad plats

Dynamisk konnektivitet lösning för att uppskatta perkolationströskeln

- Hur kan man kontrollera om ett $N \times N$ system perkolerar?
- ✓ Skapa ett objekt för varje plats och tilldela de namn från 0 to $N^2 - 1$.



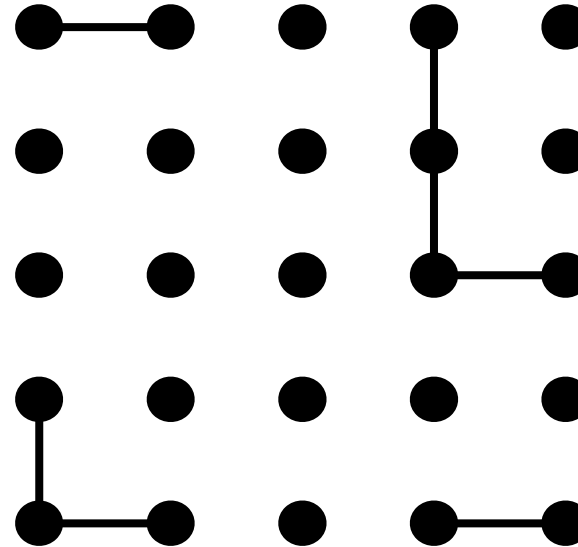
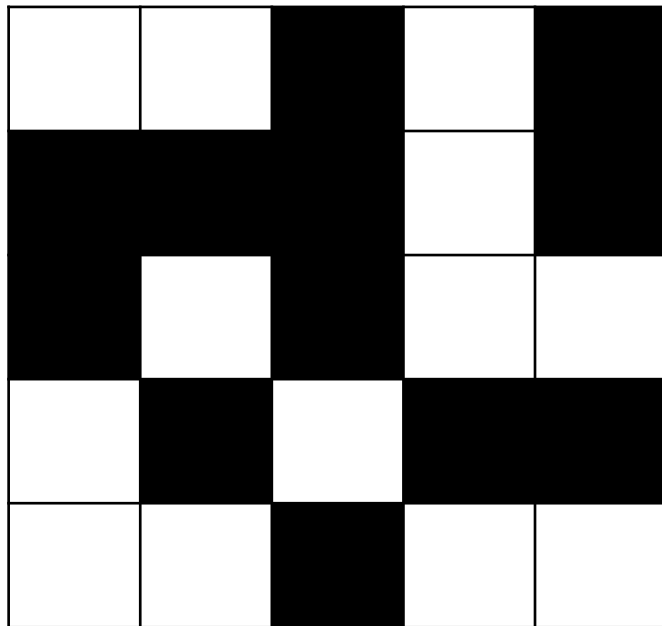
 ledig plats
 blockerad plats



Dynamisk konnektivitet lösning för att uppskatta perkolationströskeln

- Hur kan man kontrollera om ett $N \times N$ system perkolerar?
- ✓ Skapa ett objekt för varje plats och tilldela de namn från 0 till $N^2 - 1$.
- ✓ Platser befinner sig i samma komponent iff de är förbundna med lediga platser.

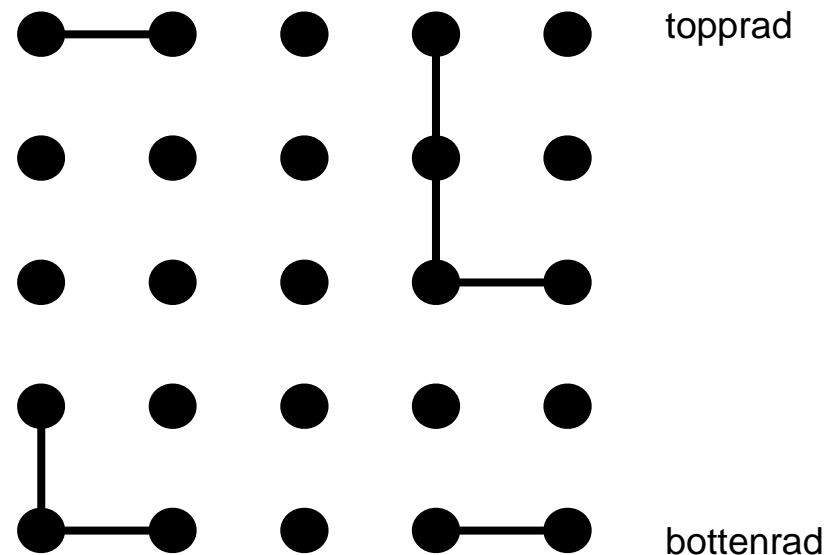
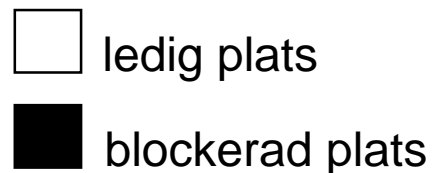
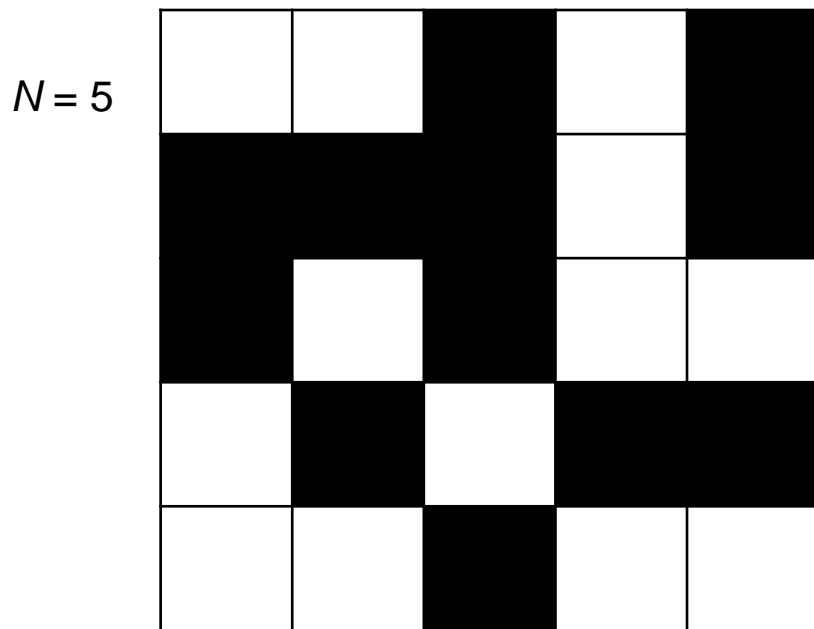
$N = 5$



Dynamisk konnektivitet lösning för att uppskatta perkolationströskeln

- Hur kan man kontrollera om ett $N \times N$ system perkolerar?
- ✓ Skapa ett objekt för varje plats och tilldela de namn från 0 till $N^2 - 1$.
- ✓ Platser befinner sig i samma komponent iff de är förbundna med lediga platser.
- ✓ Perkolerar iff en av platsen på bottenraden (vilken som helst) är förbunden till en på toppraden (vilken plats som helst).

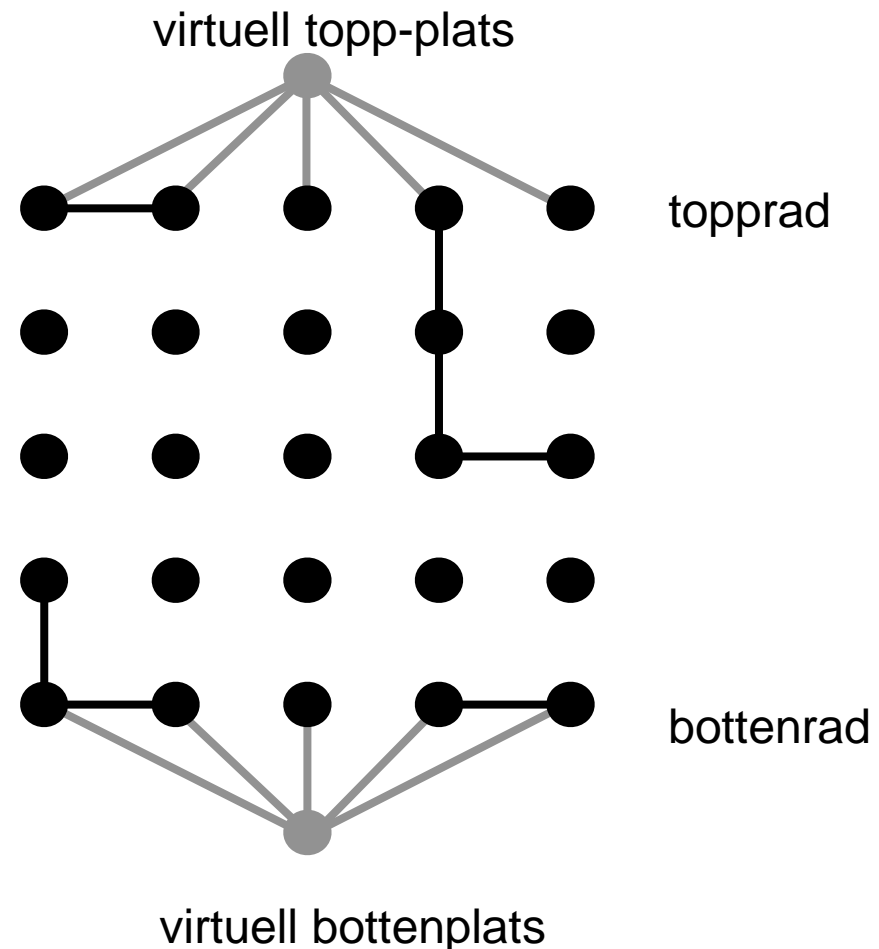
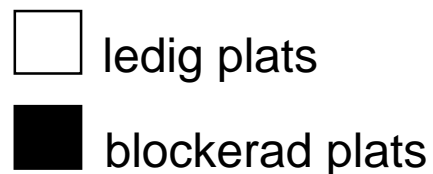
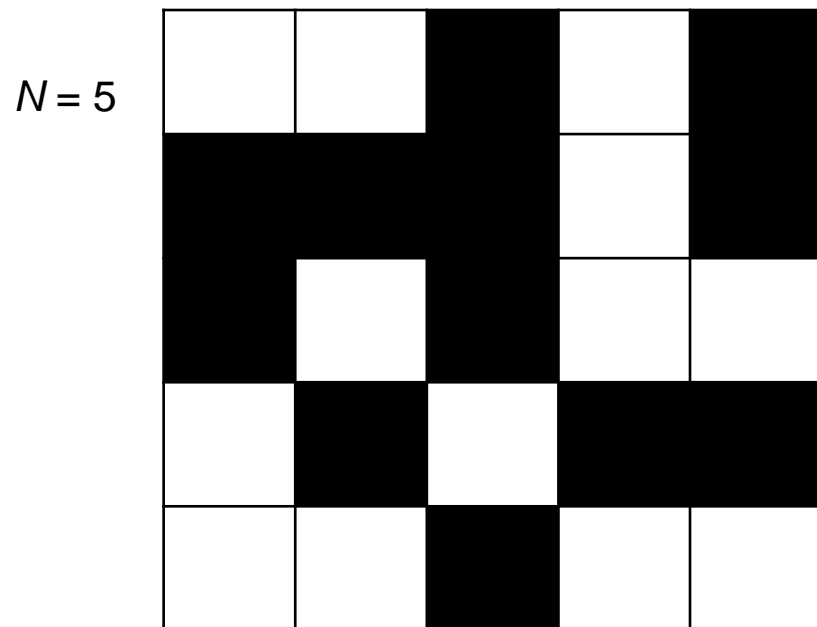
brute-force algoritm: N^2 anrop till `connected()`



Dynamisk konnektivitet lösning för att uppskatta perkolationströskeln

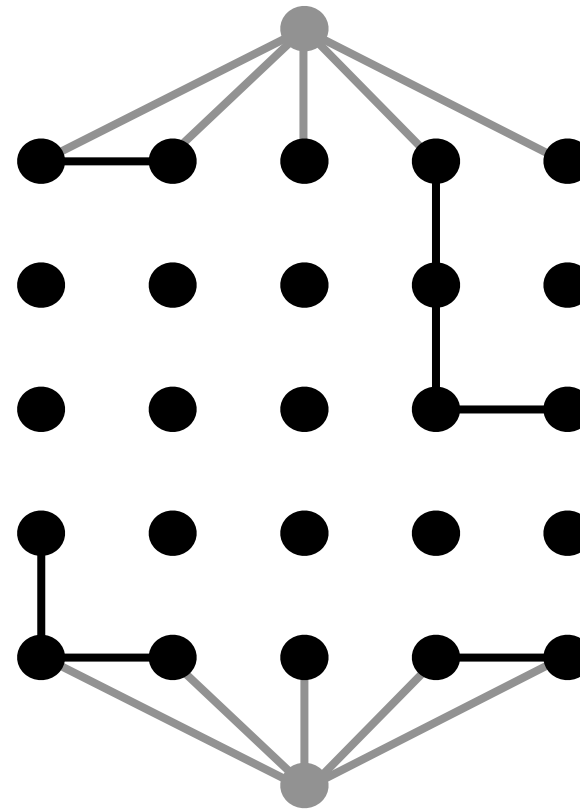
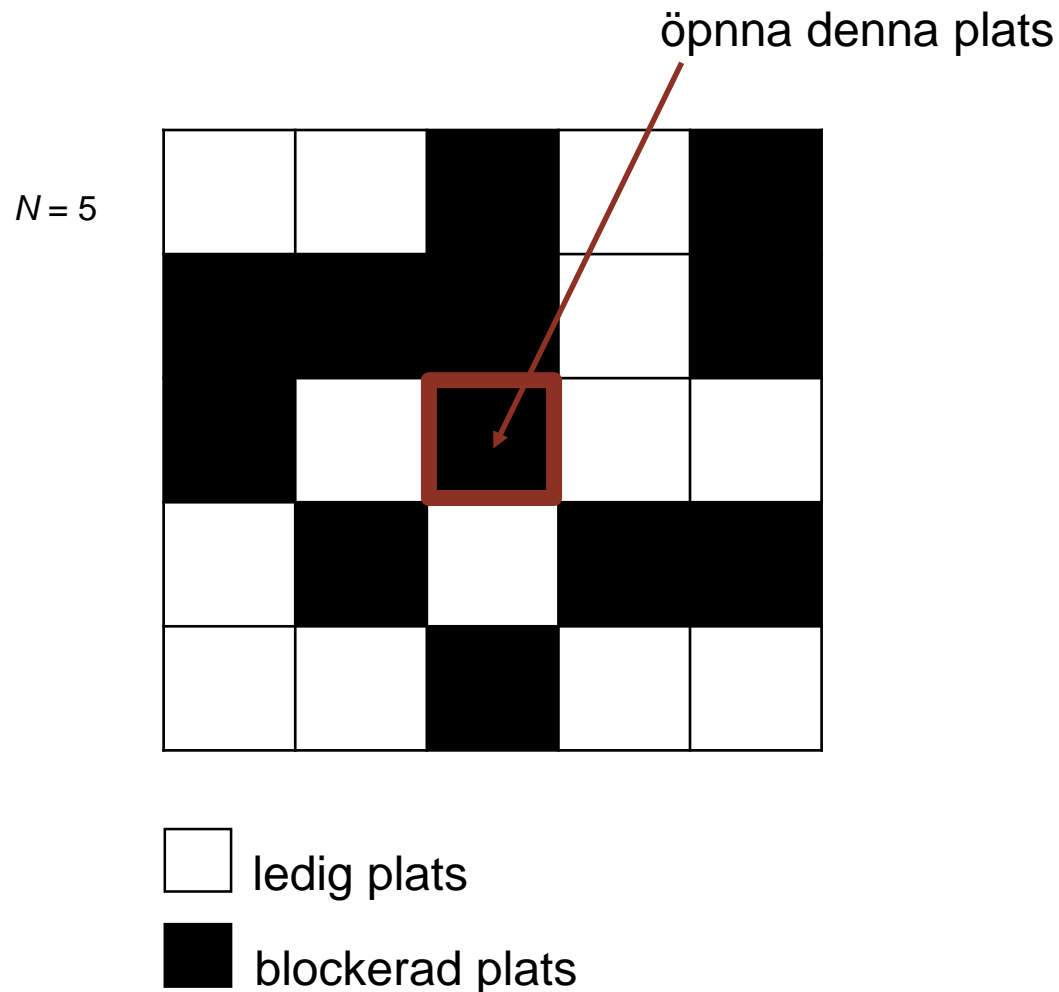
- **Knep.** Introducera 2 virtuella platser (virtual sites) med förbindelser till topp och botten.
- ✓ Systemet perkolerar iff den virtuella topp-platsen är förbunden till en virtuell bottenplats

mer effektivt algoritm: bara 1 anrop till connected()



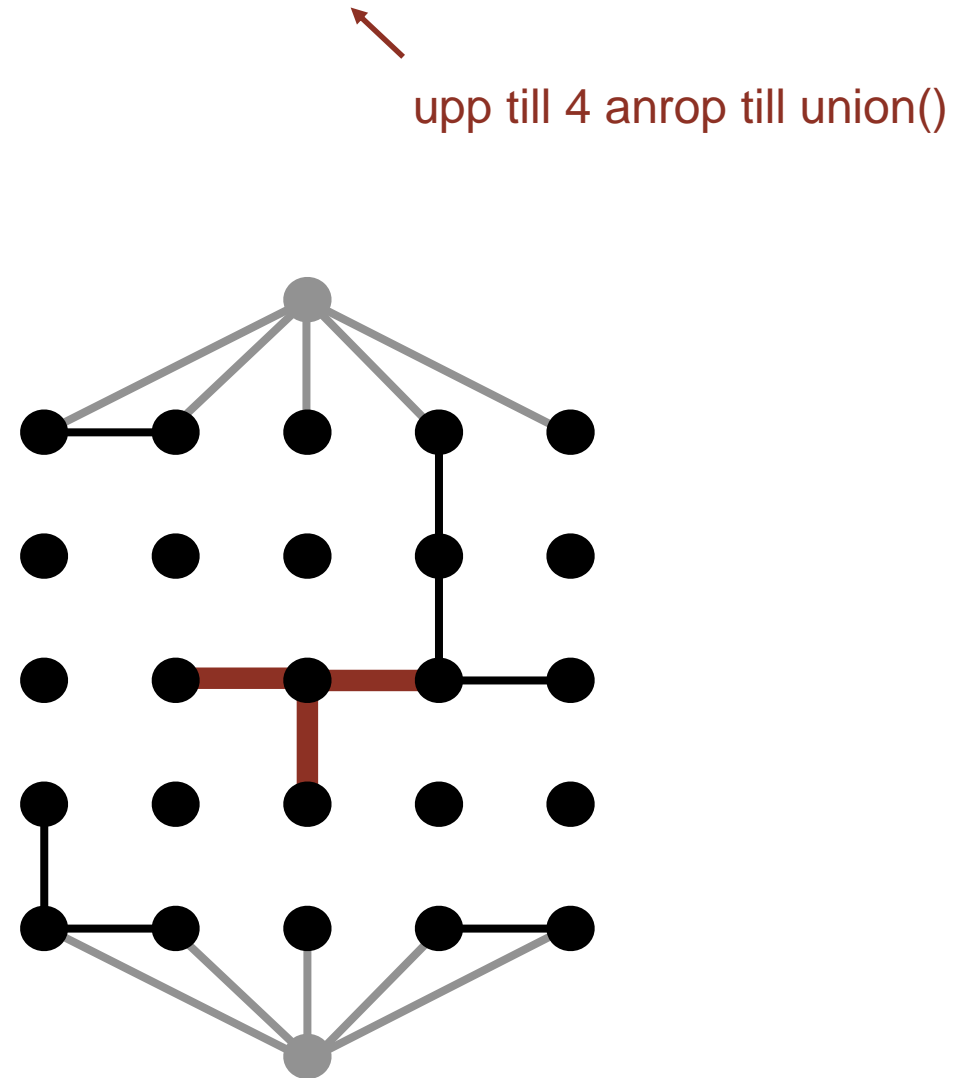
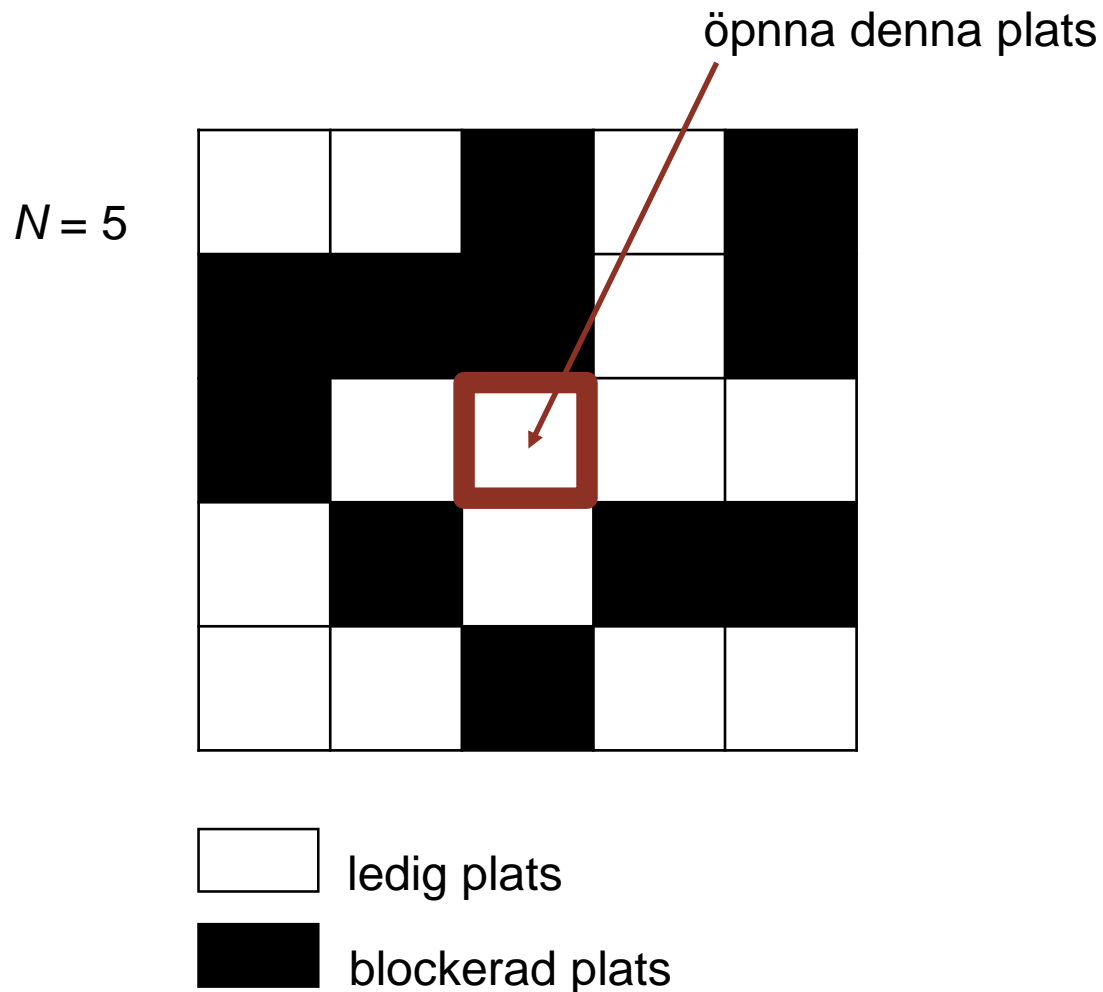
Dynamisk konnektivitet lösning för att uppskatta perkolationströskeln

- Vi vill omvandla en blockerad plats till en öppen plats. Hur bygger man en model av detta?



Dynamisk konnektivitet lösning för att uppskatta perkolationströskeln

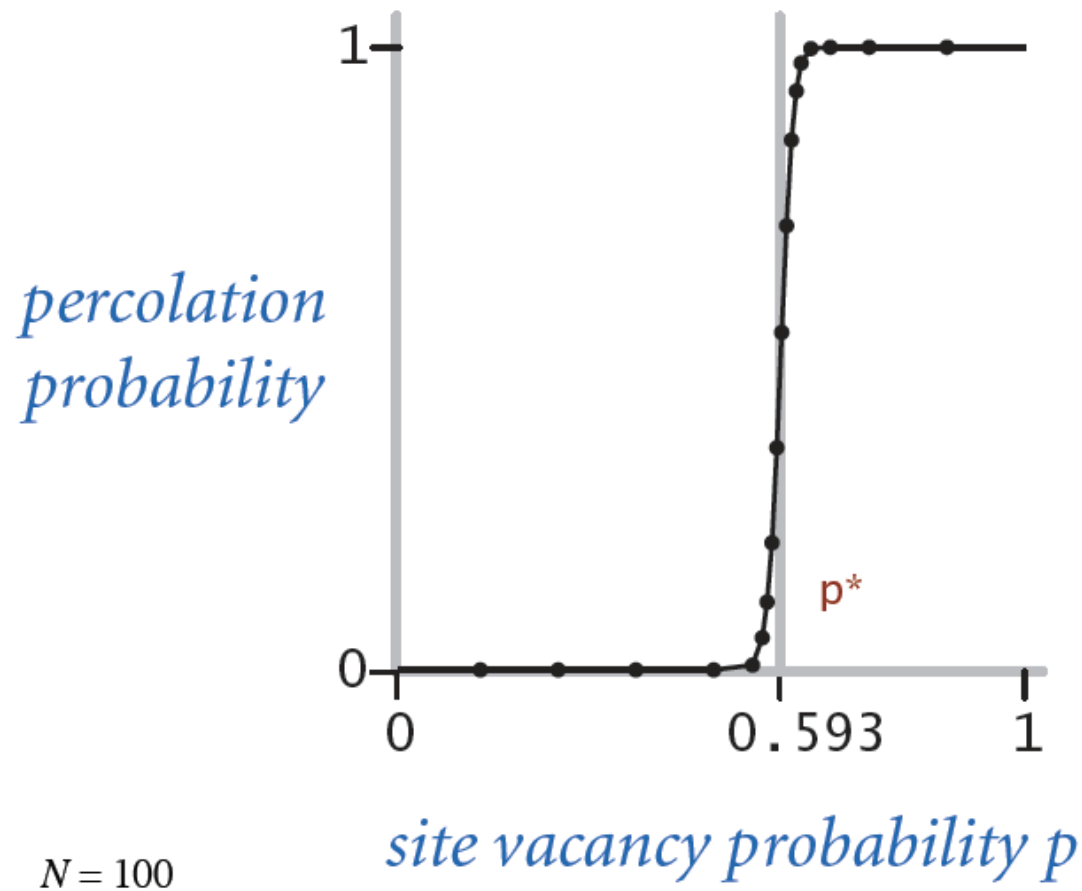
- Markera den nya platsen som ledig; förbinda alla sina angränsande lediga platser.



Perkolationströskeln

- Vad är perkolation tröskeln p^* ?
- Ungfär 0.592746 för *large square lattices*.

konstant känd bara med hjälp av simuleringar



- En snabb algoritm möjliggör ett noggrant svar till en vetenskaplig fråga.

Sammanfattning

- Union-Find är en intressant algoritm med många praktiska applikationer
 - Quick-Find, Quick-Union, Weighted Quick-Union, Weighted Quick-Union with Path Compression.
- Med union-find som ett exempel, har vi gått igenom följande steg i utvecklingen av en algoritm:
 - Bygga en modell av problemet.
 - Hitta en algoritm för att lösa problemet.
 - Är den snabb nog? Passar den i minnet?
 - Om inte, kom på varför.
 - Hitta ett sätt att fixa problemet.
 - Iterera tills man är njöd
- Nästa föreläsningen:
 - Enkel-sortering