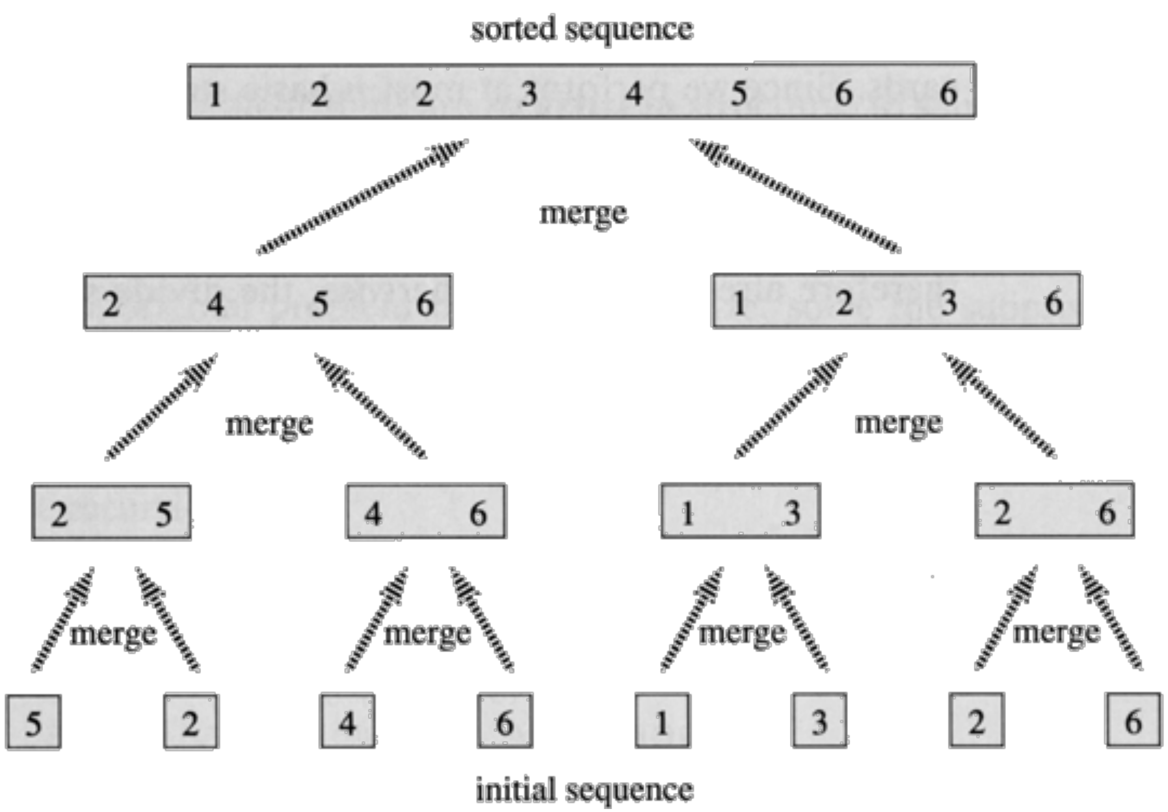
15-112
Fundamentals of Programming

Week 3 - Lecture 3:
Efficiency continued + Sets and dictionaries.



June 2, 2016

Measuring running time

How to properly measure running time

> Input length/size denoted by N (and sometimes by n)

- for lists: N = number of elements
- for strings: N = number of characters
- for ints: N = number of digits

> Running time is a function of N.

> Look at worst-case scenario/input of length N.

> Count algorithmic steps.

> Ignore constant factors. (e.g. $N^2 \approx 3N^2$)
(use big Oh notation)

Review

Give 2 definitions of $\log_2 N$

Number of times you need to divide N by 2 to reach 1.

The number *k* that satisfies $2^k = N$.

What is the big Oh notation used for?

Upper bound a function by ignoring:

- constant factors
- small *N*.

→ ignore small order additive terms.

Review

Big-Oh is the right level of abstraction!

$$8N^2 - 3n + 84$$

is analogous to "too many significant figures".

$$O(N^2)$$

"Sweet spot"

- coarse enough to suppress details like programming language, compiler, architecture,…
- sharp enough to make comparisons between different algorithmic approaches.

Review

$10^{10}n^3$ is $O(n^3)$? Yes

n is $O(n^2)$? Yes

n^3 is $O(2^n)$? Yes

n^{10000} is $O(1.1^n)$? Yes

$100n \log_2 n$ is $O(n)$? No

$1000 \log_2 n$ is $O(\sqrt{n})$? Yes

$1000 \log_2 n$ is $O(n^{0.00000001})$? Yes

When we ask "what is the running time…" you must give the tight bound!

Does the base of the log matter?

$$\log_b n = \frac{\log_c n}{\boxed{\log_c b}}$$

constant

Review

Constant:	$O(1)$
Logarithmic:	$O(\log n)$
Square-root:	$O(\sqrt{n}) = O(n^{0.5})$
Linear:	$O(n)$
Loglinear:	$O(n \log n)$
Quadratic:	$O(n^2)$
Polynomial:	$O(n^k)$
Exponential:	$O(k^n)$

Review

$$\log n <<< \sqrt{n} << n < n \log n << n^2 << n^3 <<< 2^n <<< 3^n$$

Review

Running time of Linear Search: $O(N)$

Running time of Binary Search: $O(\log N)$

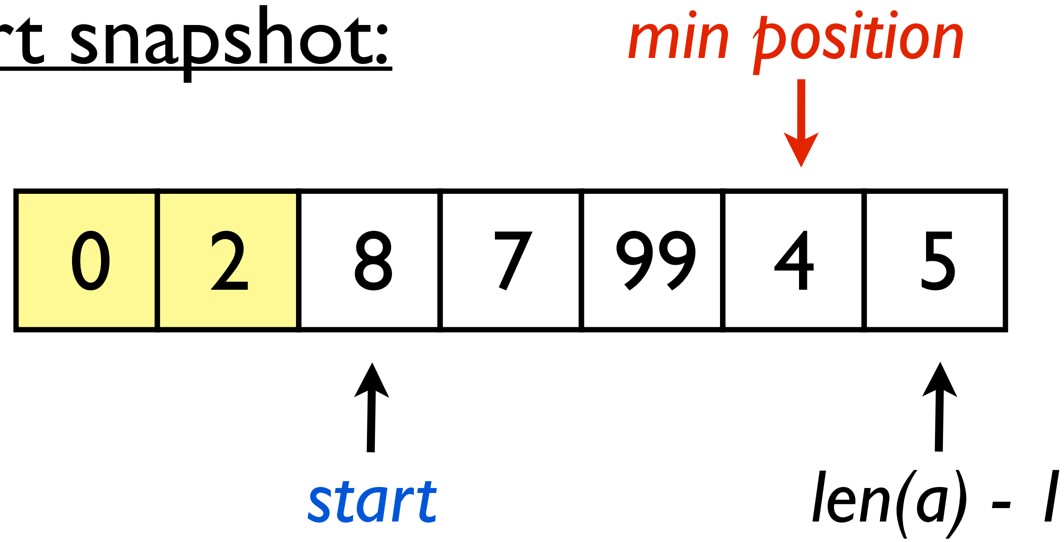
Running time of Bubble Sort: $O(N^2)$

Running time of Selection Sort: $O(N^2)$

Why is Bubble Sort slower than Selection Sort in practice?

Review: selection sort code

<u>Selection sort snapshot:</u>



Find the *min position* from *start* to *len(a) - 1*

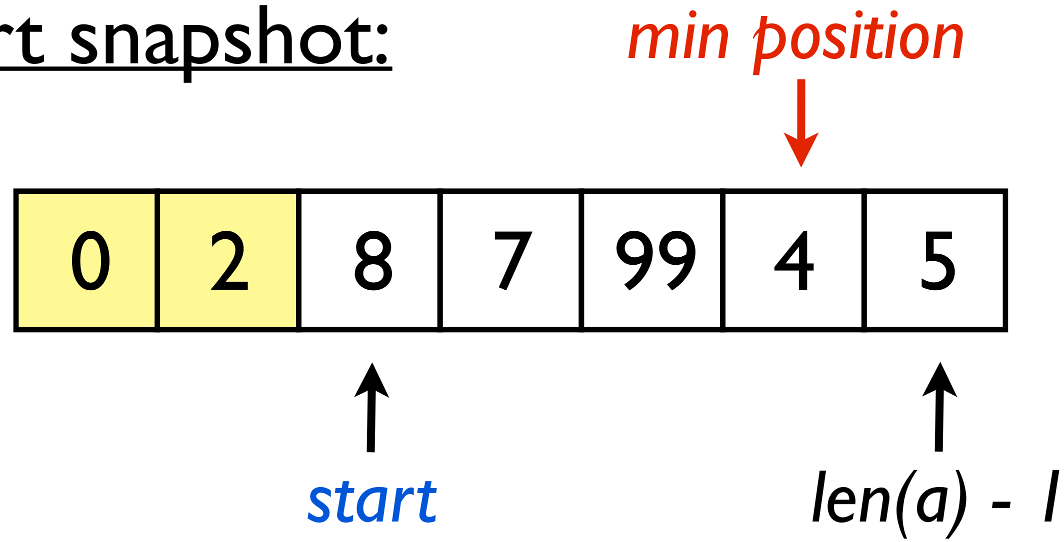
Swap elements in *min position* and *start*

Increment *start*

Repeat

Review: selection sort code

<u>Selection sort snapshot:</u>



for *start* = 0 to *len(a)-1*:

Find the *min position* from *start* to *len(a) - 1*

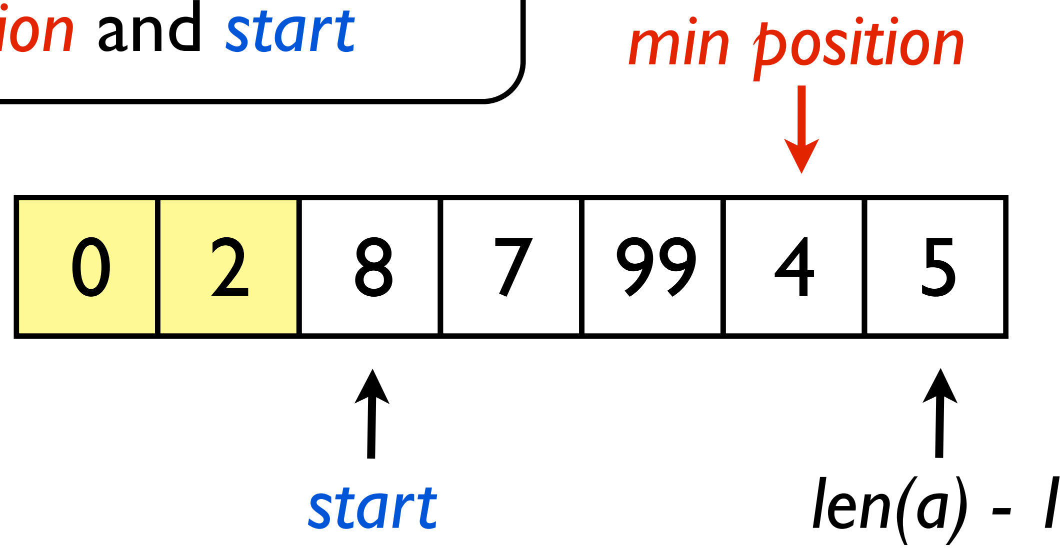
Swap elements in *min position* and *start*

Review: selection sort code

for *start* = 0 to *len(a)-1*:

  Find the *min position* from *start* to *len(a) - 1*

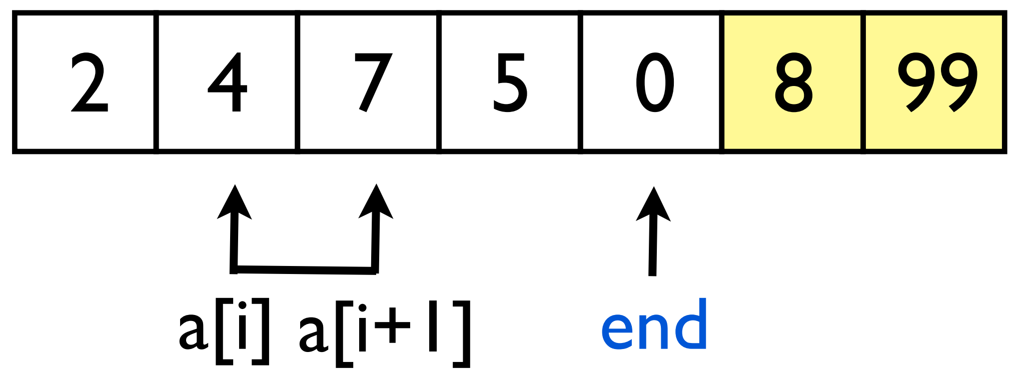
  Swap elements in *min position* and *start*




```
def selectionSort(a):
  for start in range(len(a)):
    currentMinIndex = start
    for i in range(start, len(a)):
      if(a[i] < a[currentMinIndex]):
        currentMinIndex = i
    (a[currentMinIndex], a[start]) = (a[start], a[currentMinIndex])
```

Review: bubble sort code

Bubble sort snapshot



repeat until no more swaps:

for i = 0 to end:

if a[i] > a[i+1], swap a[i] and a[i+1]

decrement end

Review: bubble sort code

repeat until no more swaps:

    for i = 0 to end:

        if a[i] > a[i+1], swap a[i] and a[i+1]

    decrement end

```
def bubbleSort(a):
    swapped = True
    end = len(a)-1
    while(swapped):
        swapped = False
        for i in range(end):
            if(a[i] > a[i+1]):
                (a[i], a[i+1]) = (a[i+1], a[i])
                swapped = True
        end -= 1
```

2	4	7	5	0	8	99

a[i] a[i+1]   end

Review

You have an algorithm with running time $O(N)$.

If we double the input size,
by what factor does the running time increase?

If we quadruple the input size,
by what factor does the running time increase?

You have an algorithm with running time $O(N^2)$.

If we double the input size,
by what factor does the running time increase?

If we quadruple the input size,
by what factor does the running time increase?

Review

To search for an element in a list, it is better to:

- sort the list, then do binary search, or
- do a linear search?

Give an example of an algorithm that requires exponential time.

Exhaustive search for the Subset Sum Problem.

Can you find a polynomial time algorithm for Subset Sum?

The Plan

> Merge sort

> Measuring running time when the input is an int

> Efficient data structures: sets and dictionaries

Merge Sort: Merge

Merge

The key subroutine/helper function:

merge(a, b)

Input: two sorted lists a and b

Output: a and b merged into a single list, all sorted.

Turns out we can do this pretty efficiently.

And that turns out to be quite useful!

Merge Sort: Merge Algorithm

Merge

a =

4	8	9	11

b =

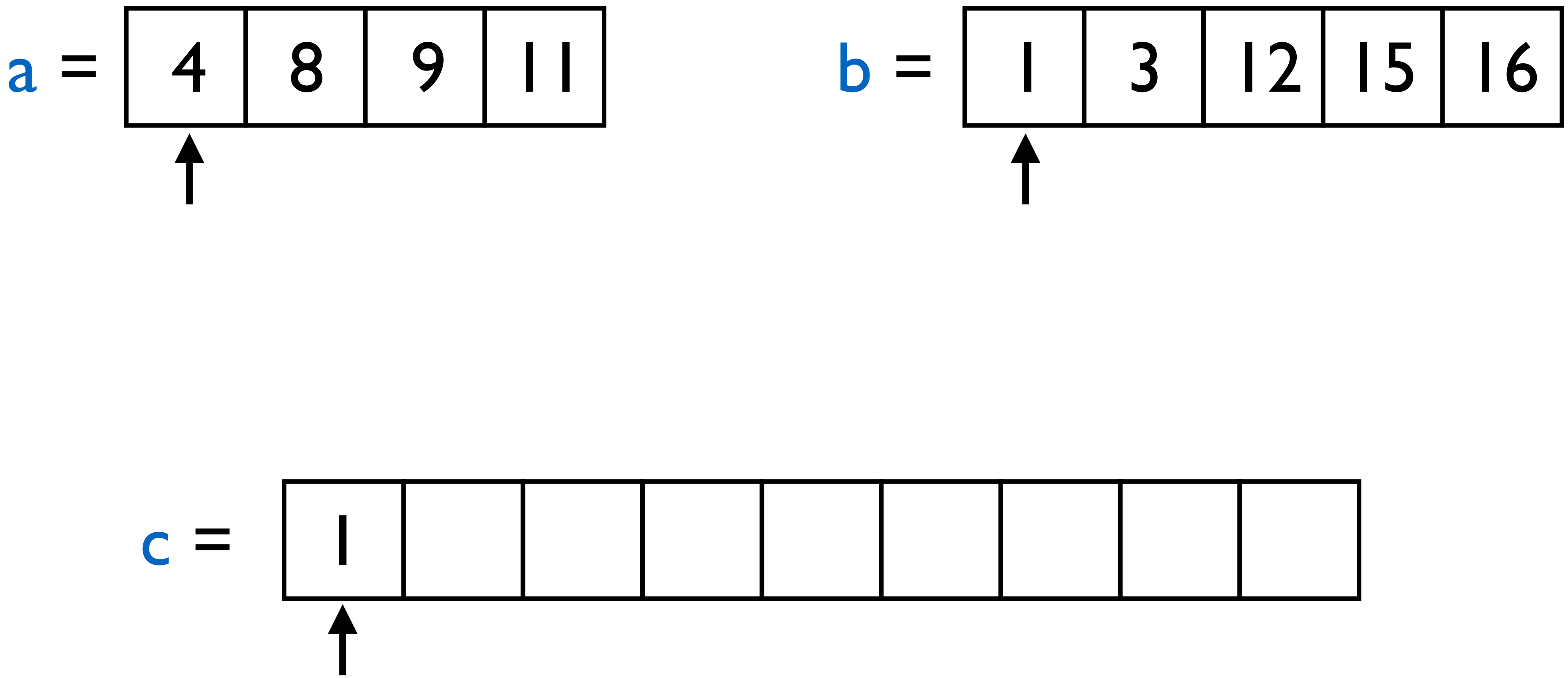
1	3	12	15	16

c =

Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

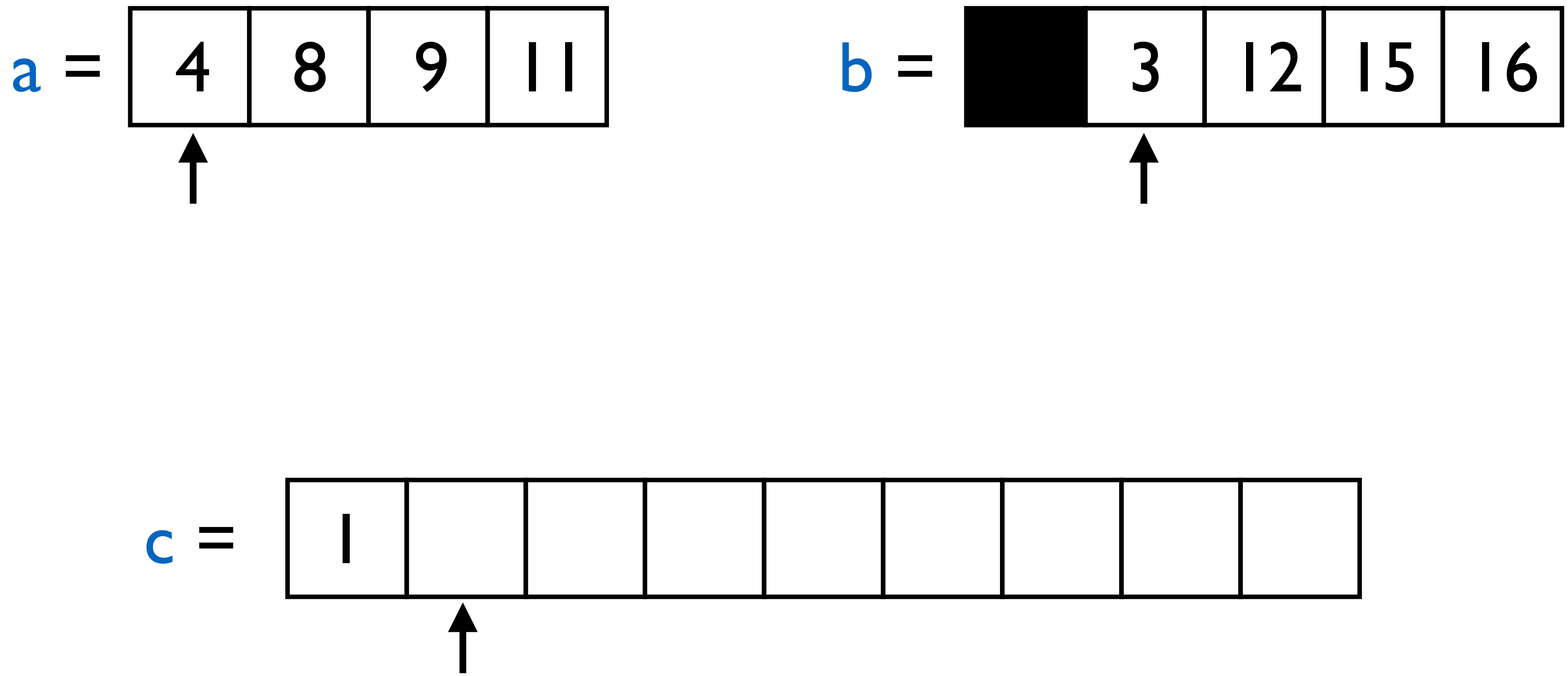
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

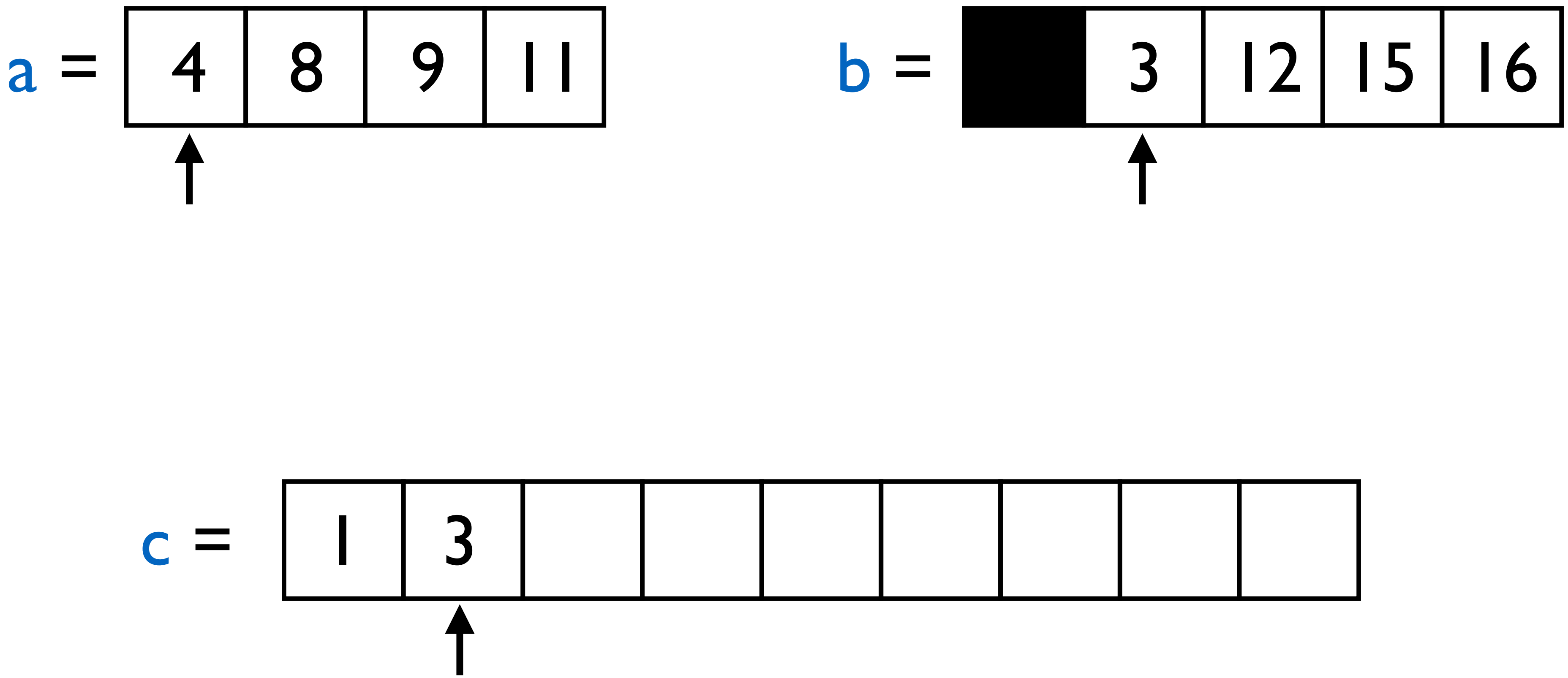
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

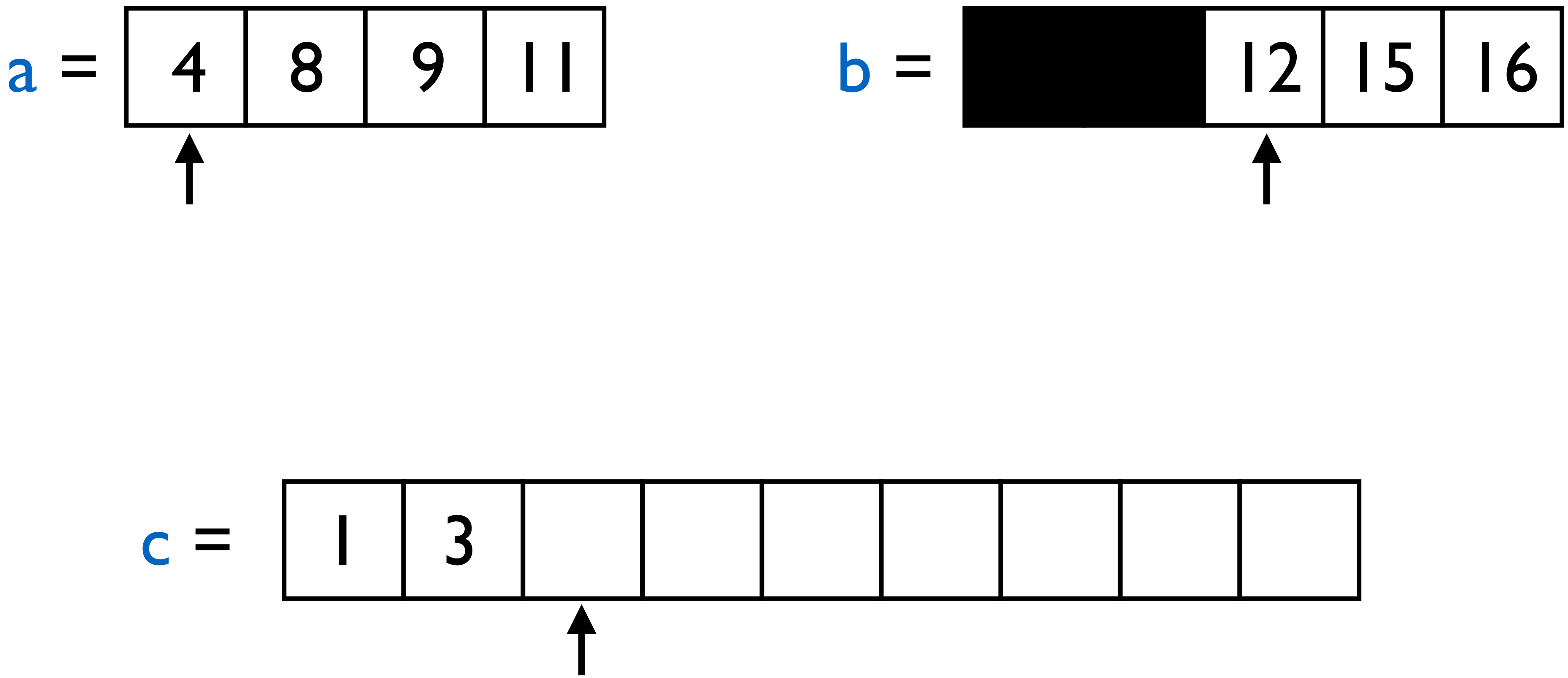
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

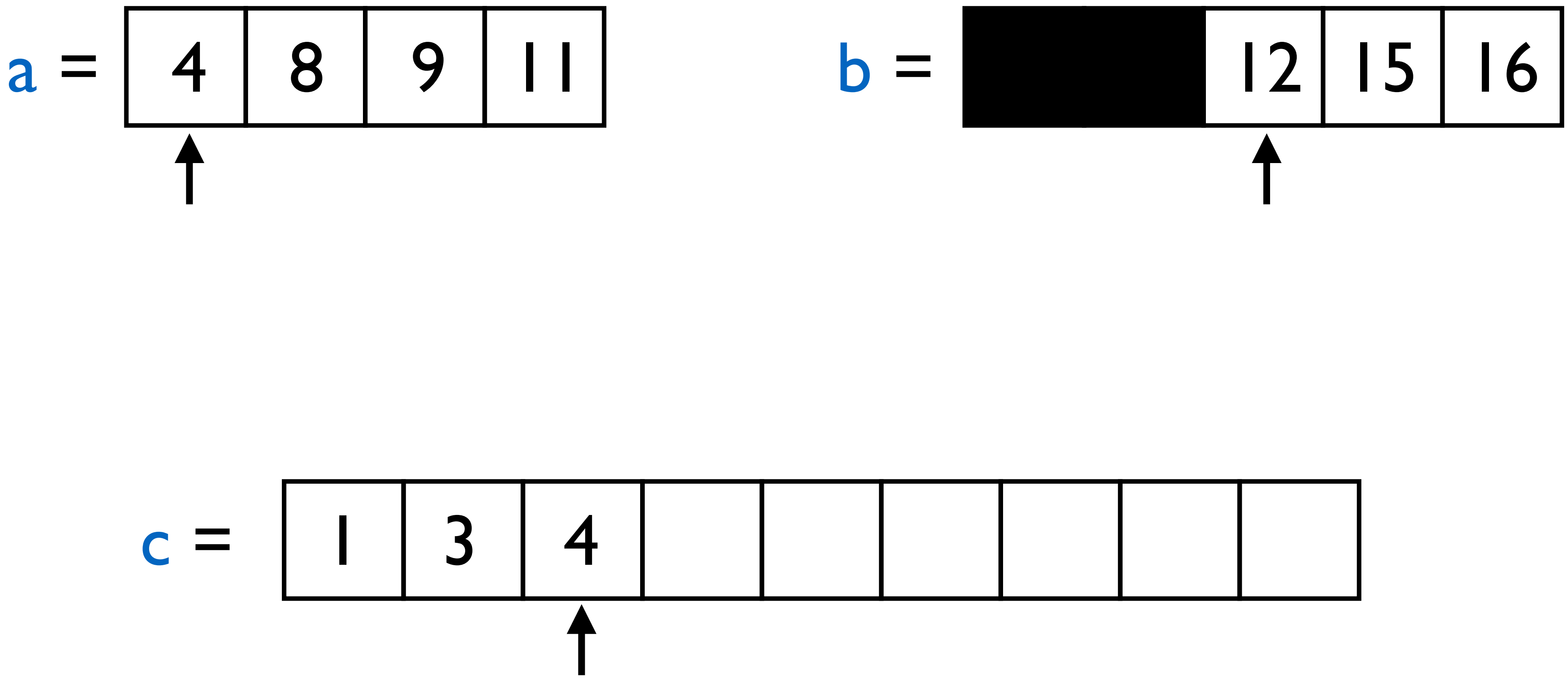
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

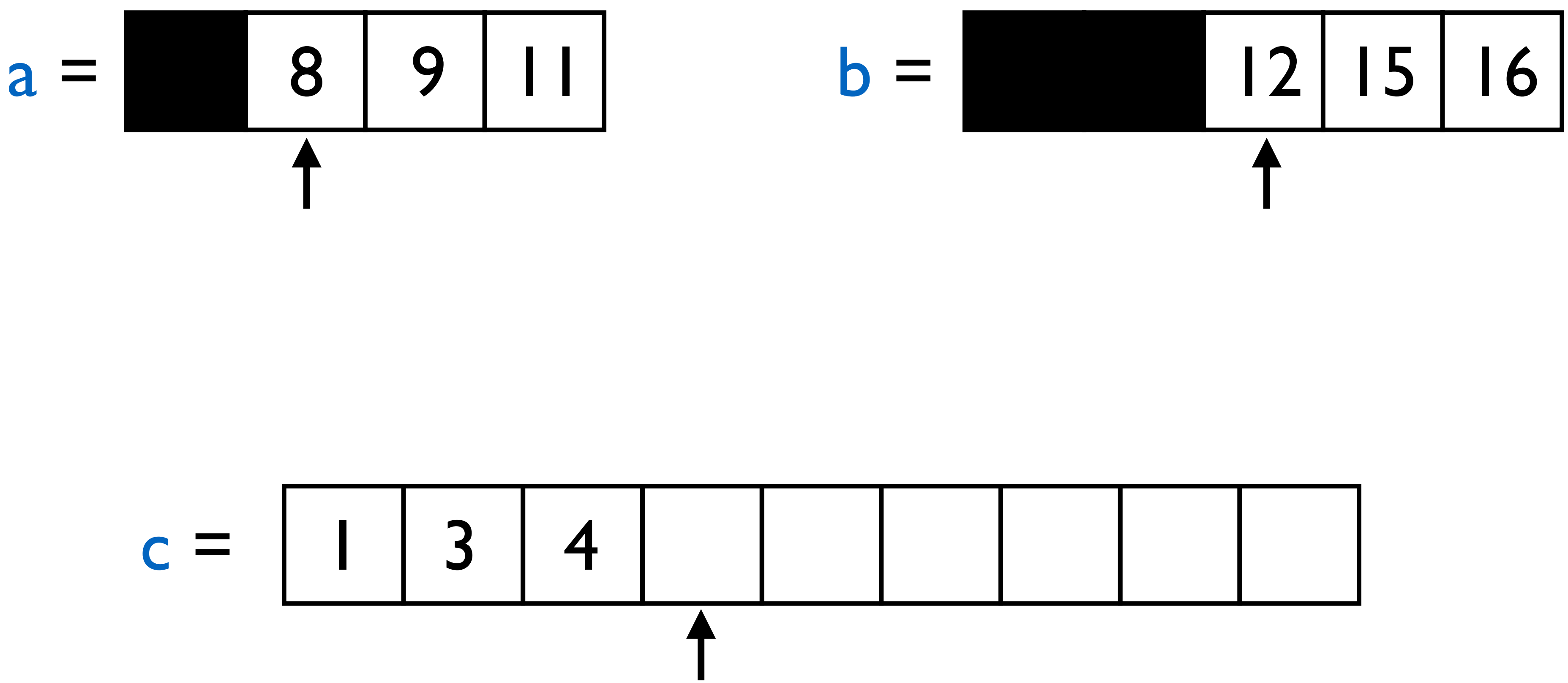
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

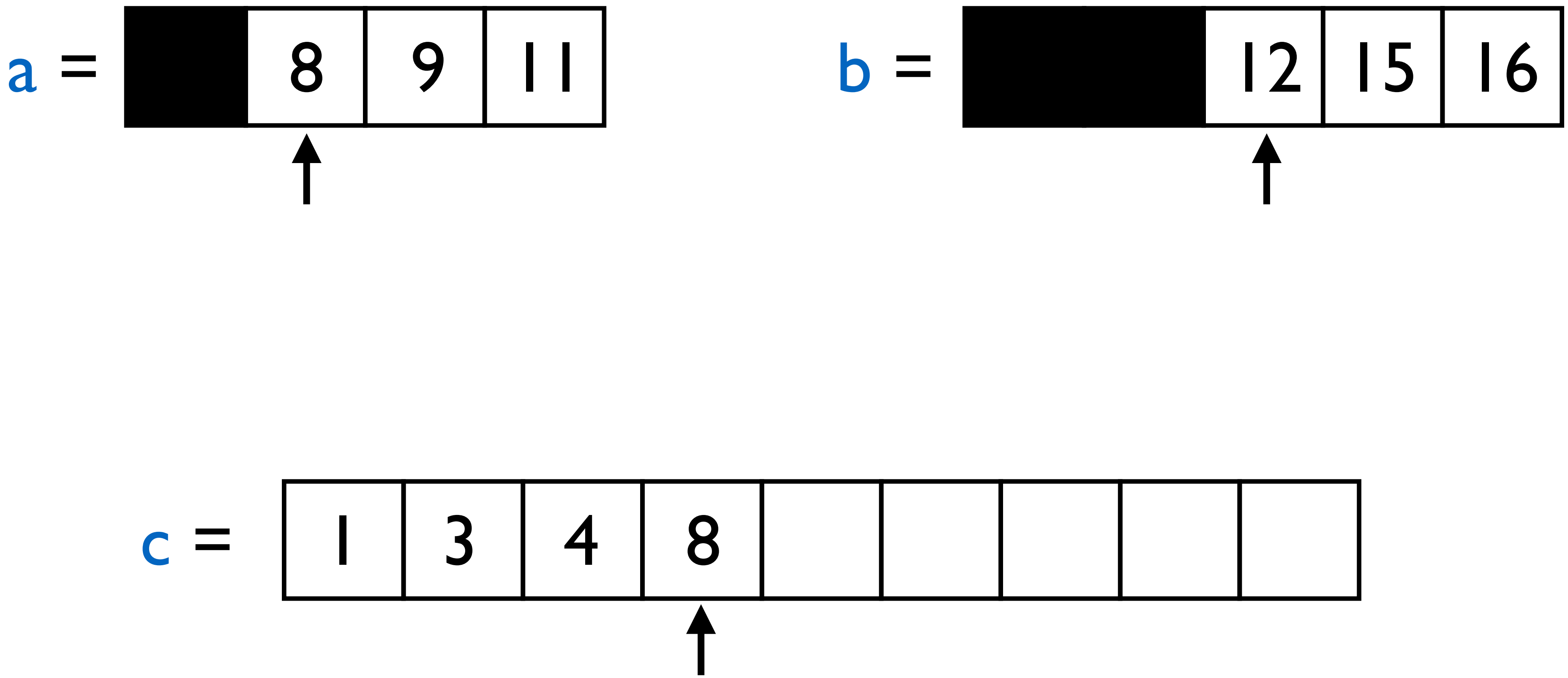
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

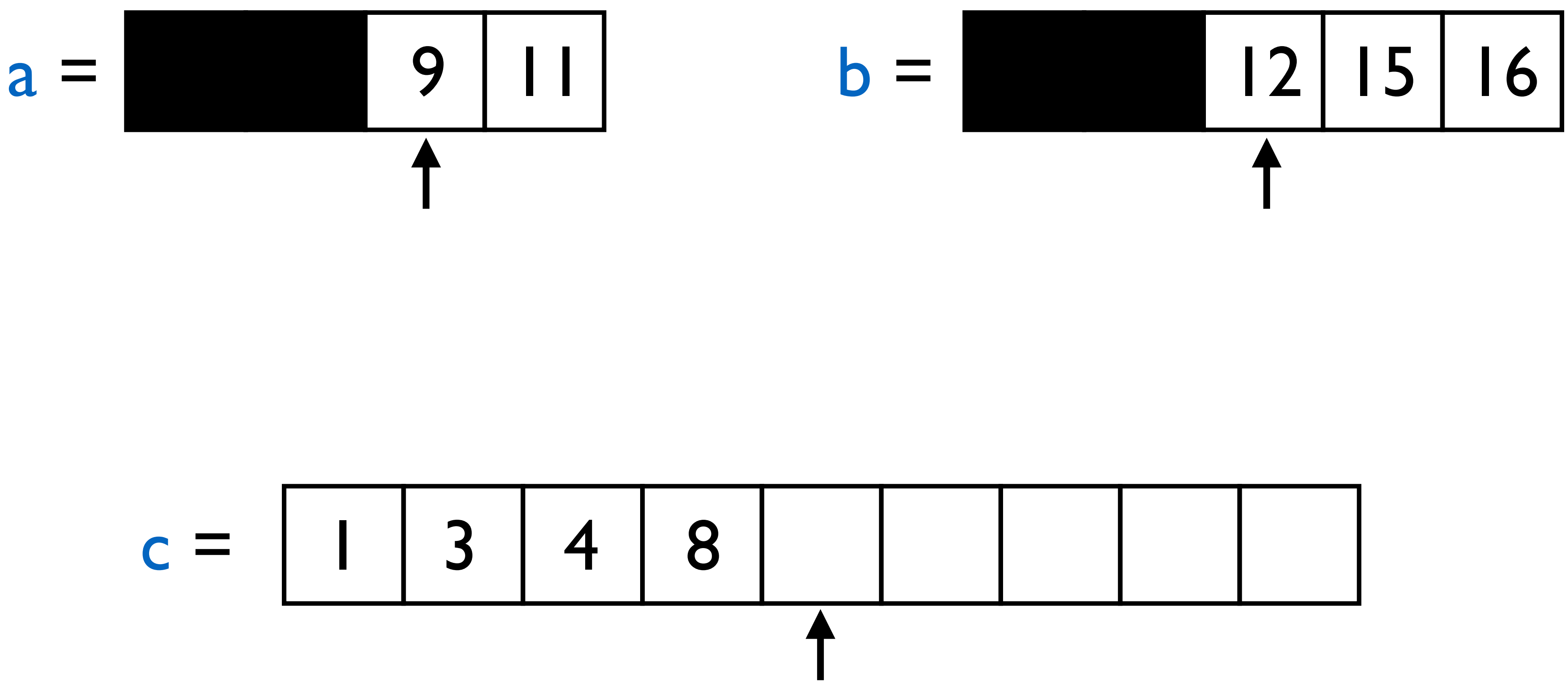
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

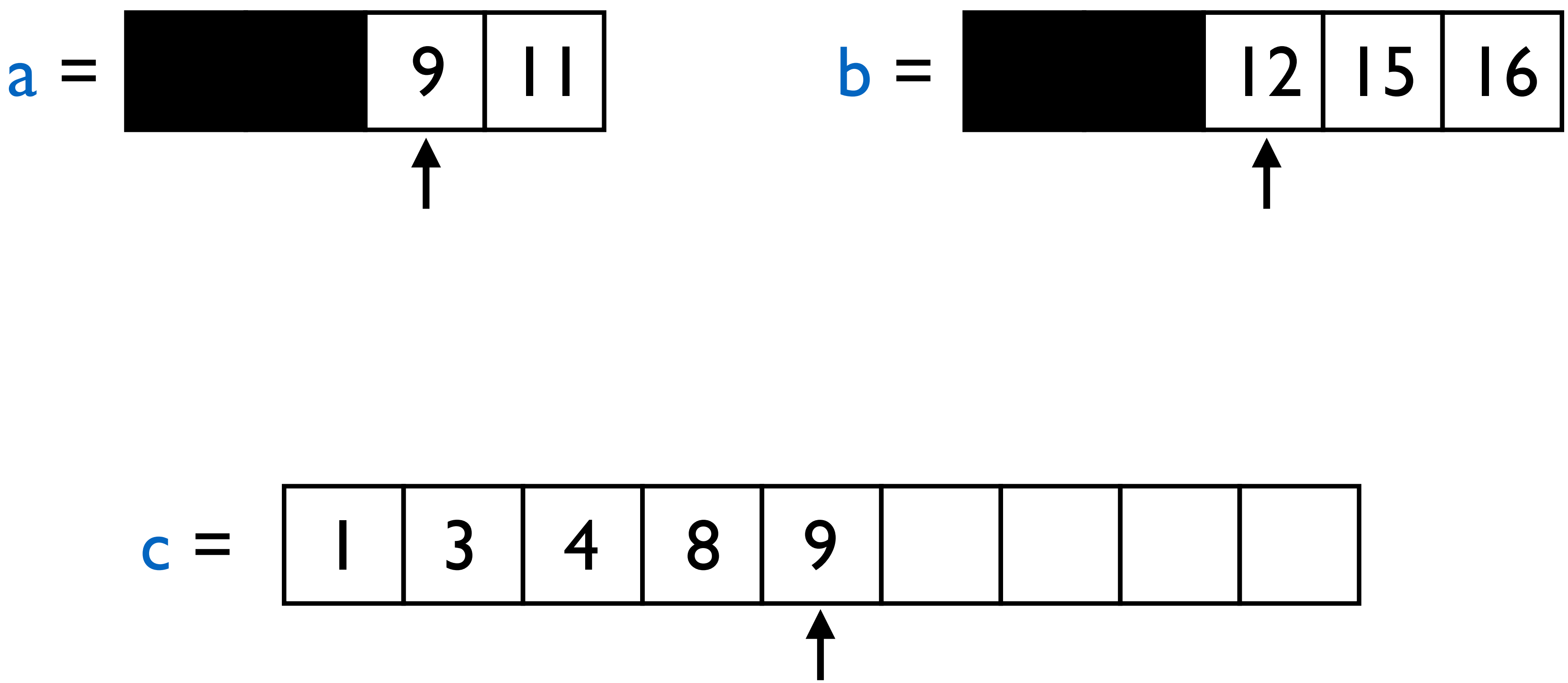
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

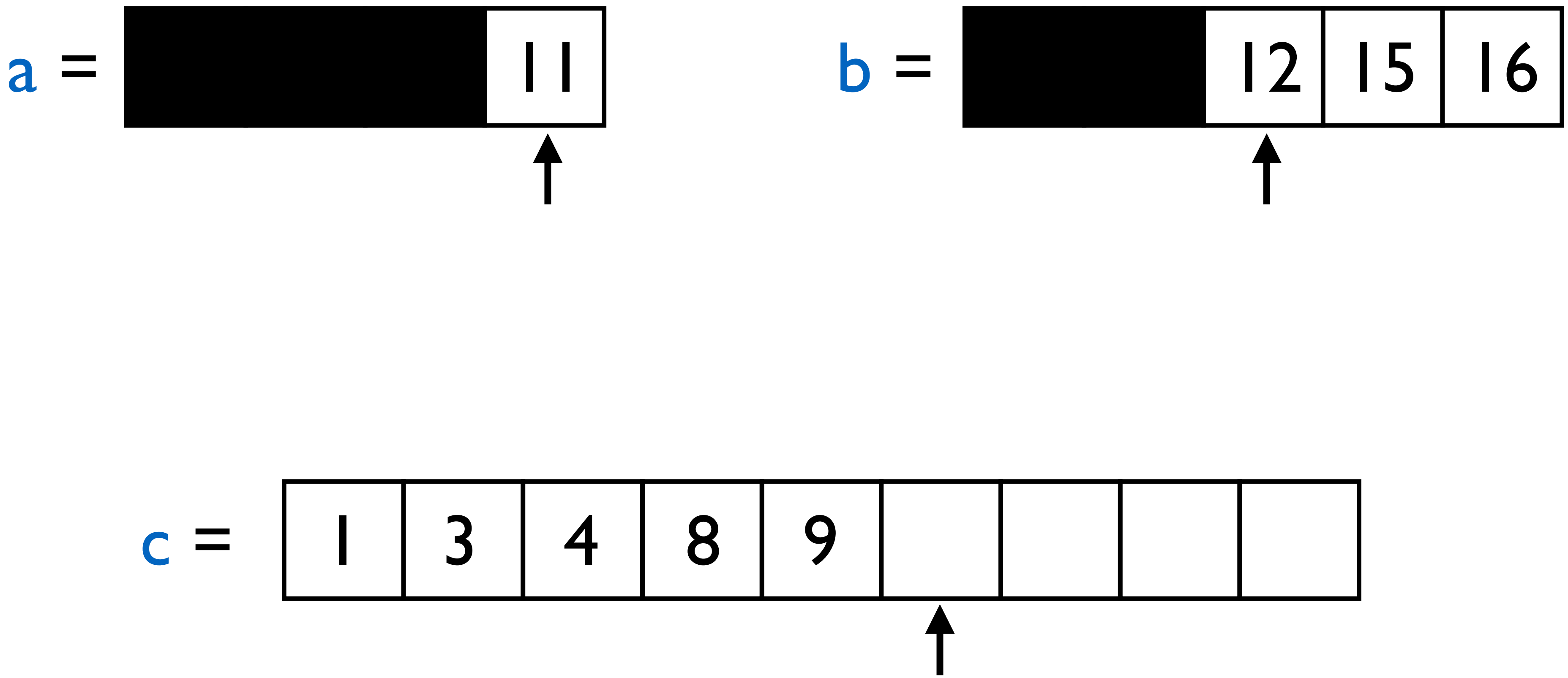
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

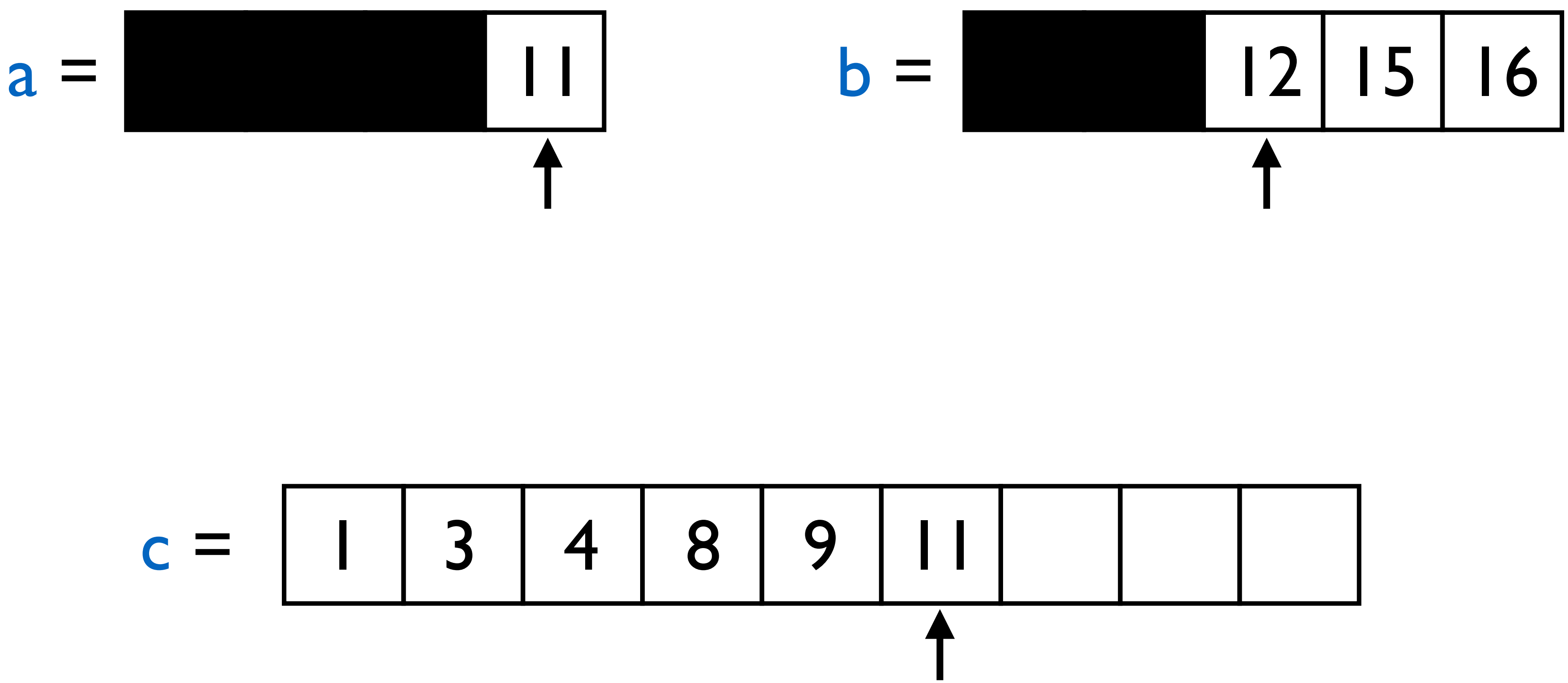
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

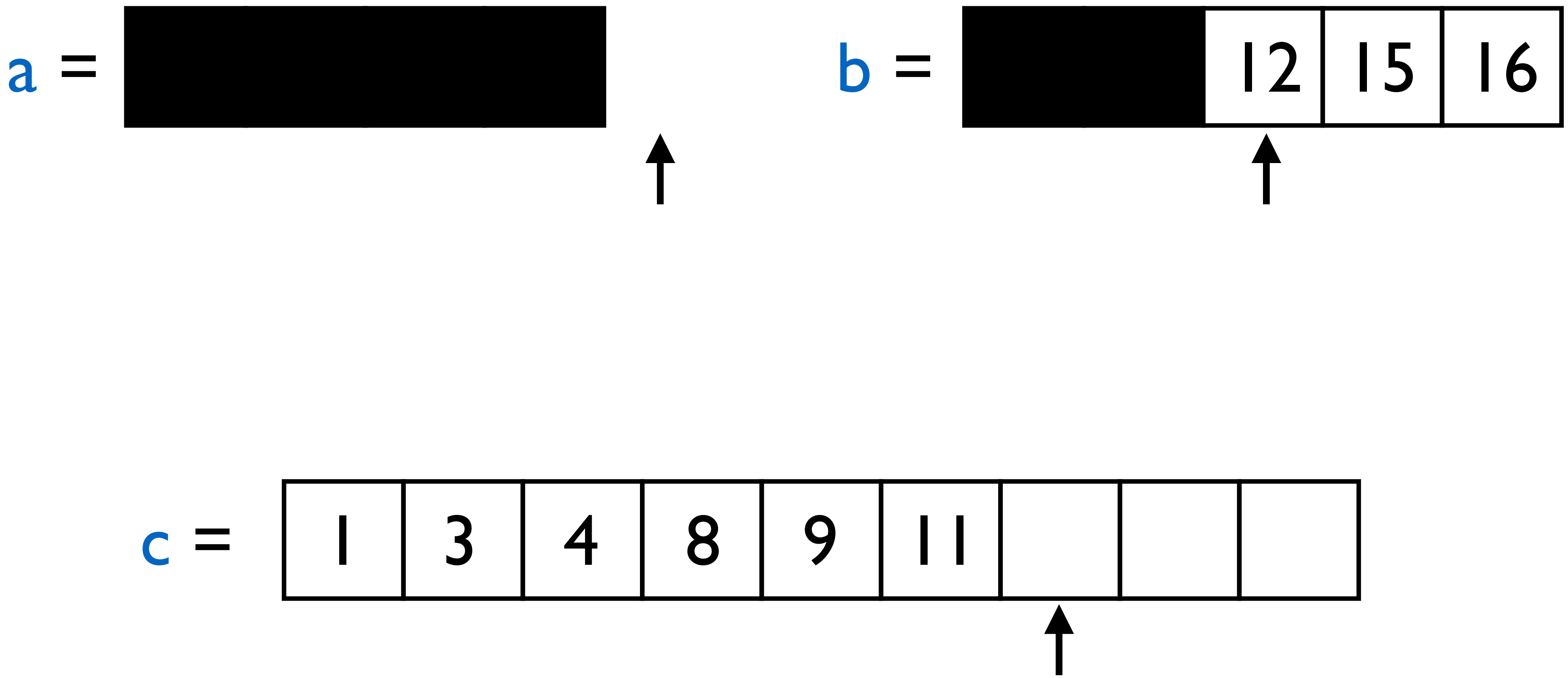
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

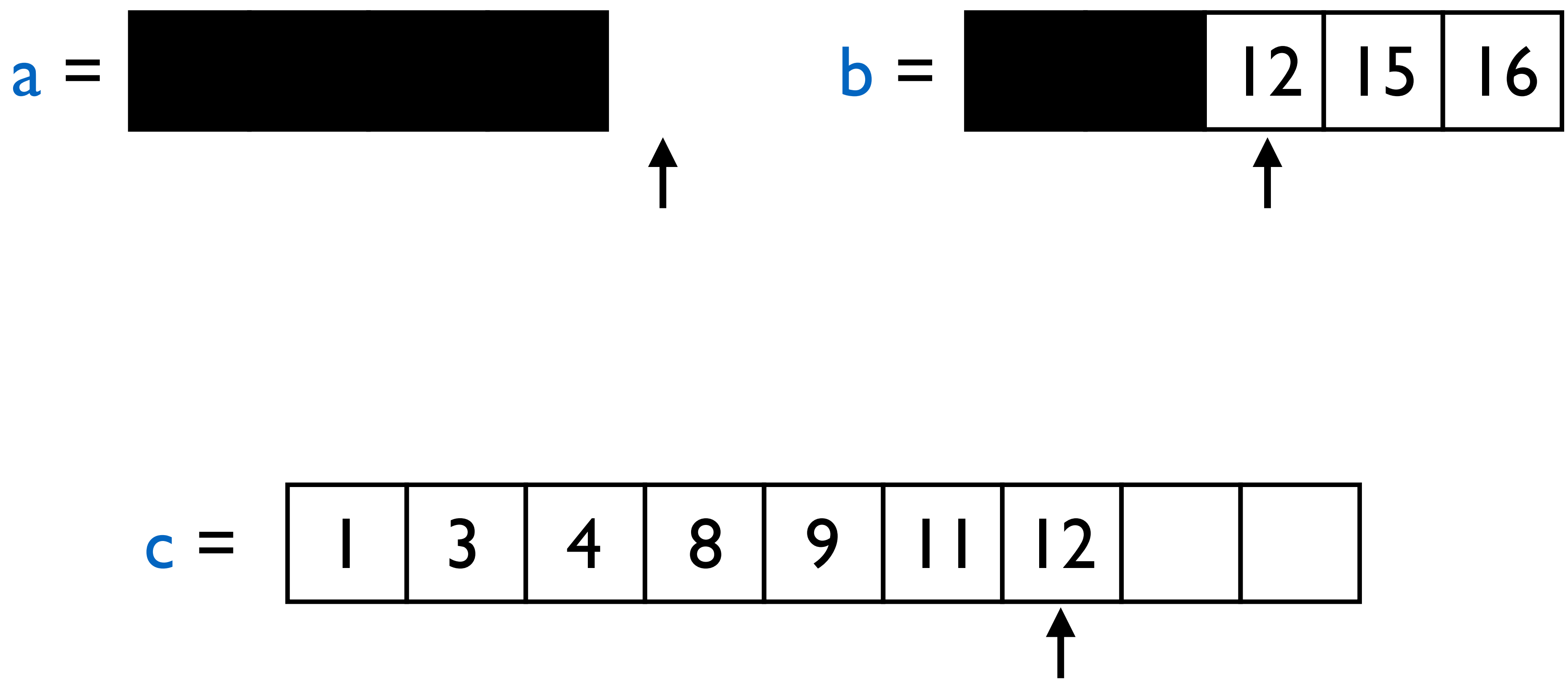
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

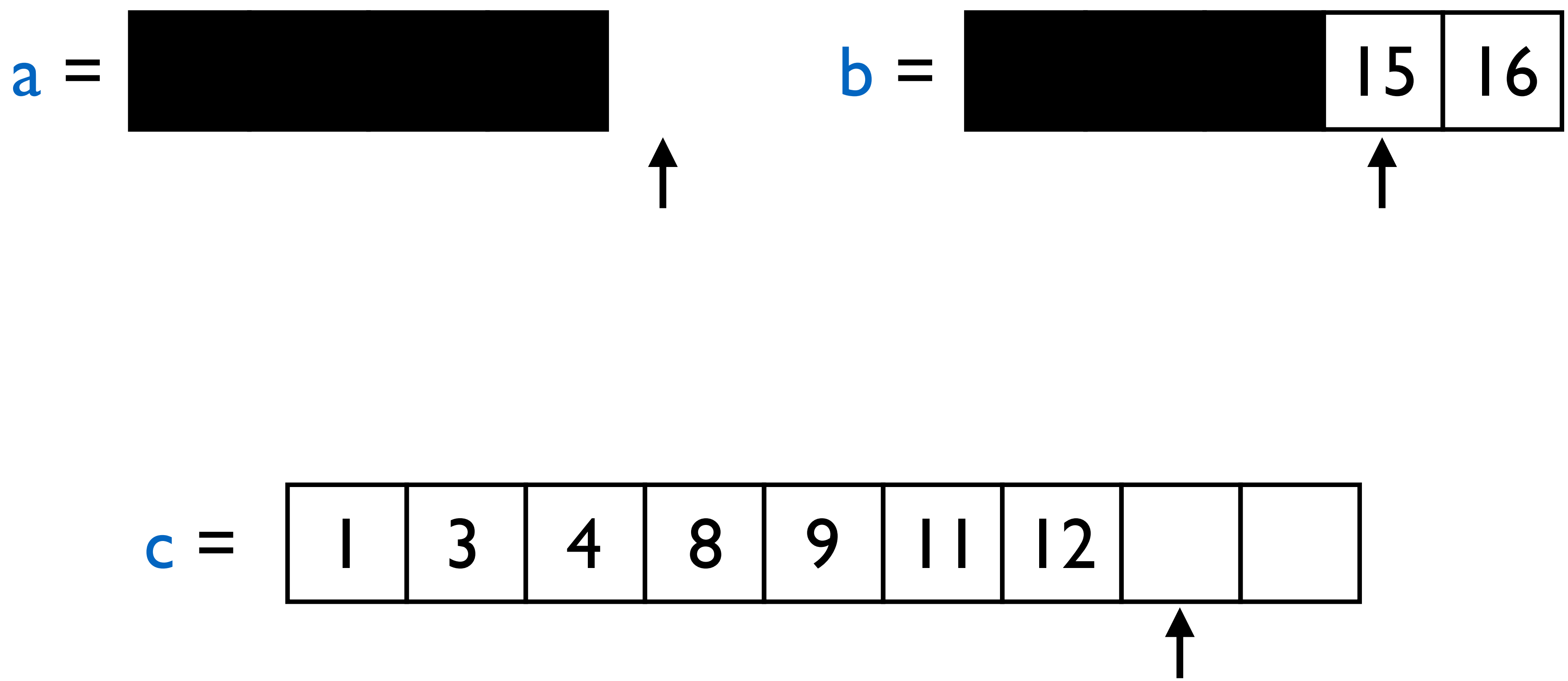
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

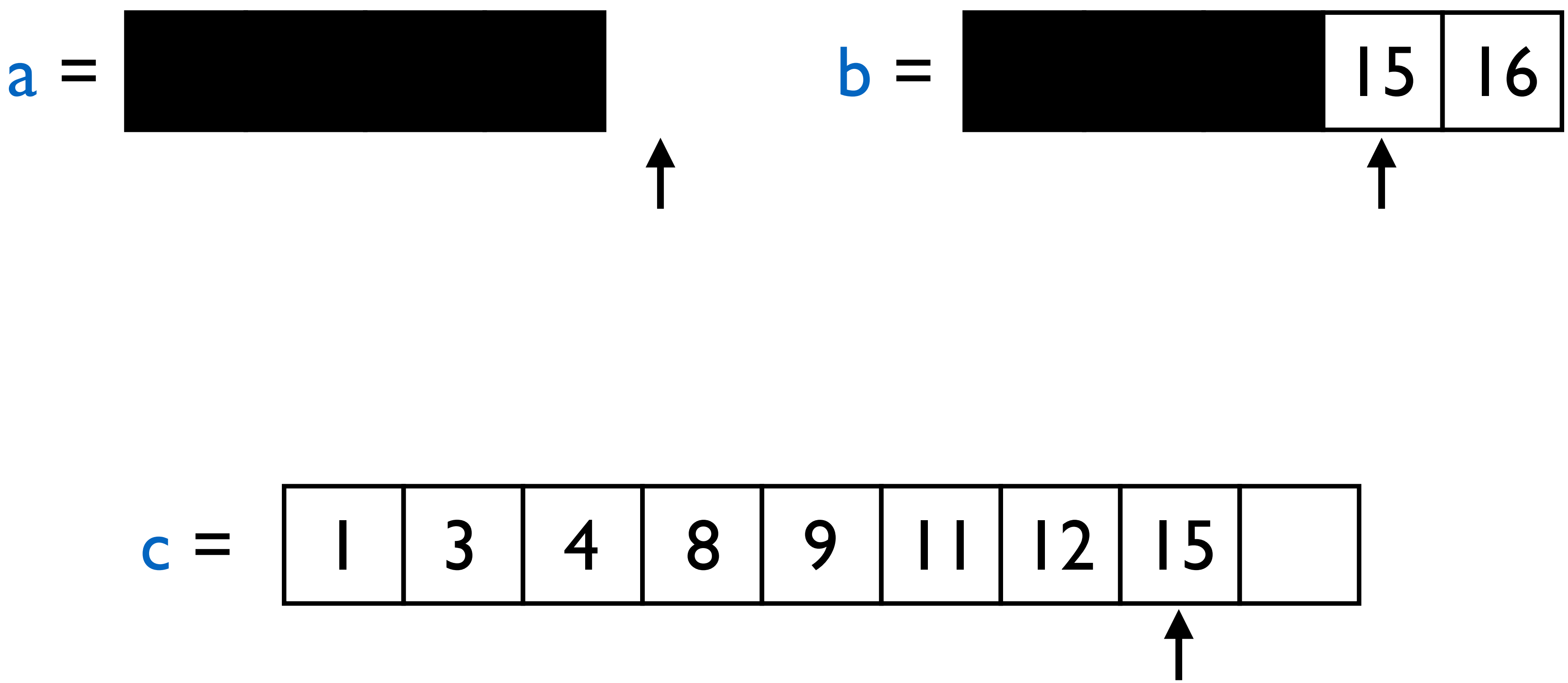
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

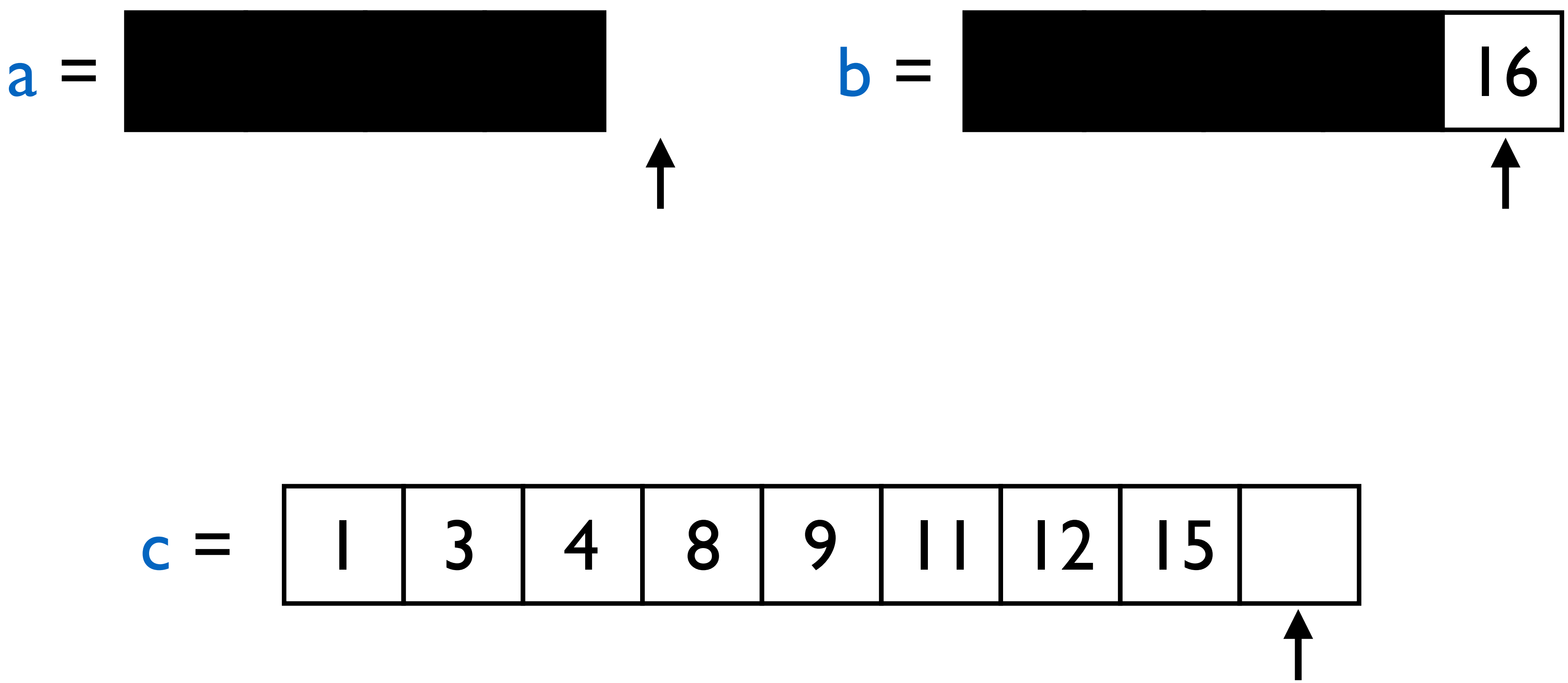
Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

Merge



Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

Merge

a = [redacted]

b = [redacted]

[redacted]	16

c =

1	3	4	8	9	11	12	15	16

Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm

Merge

a = [redacted] b = [redacted]

c =

1	3	4	8	9	11	12	15	16

Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Running Time

Merge

a = [redacted]

b = [redacted]

c =

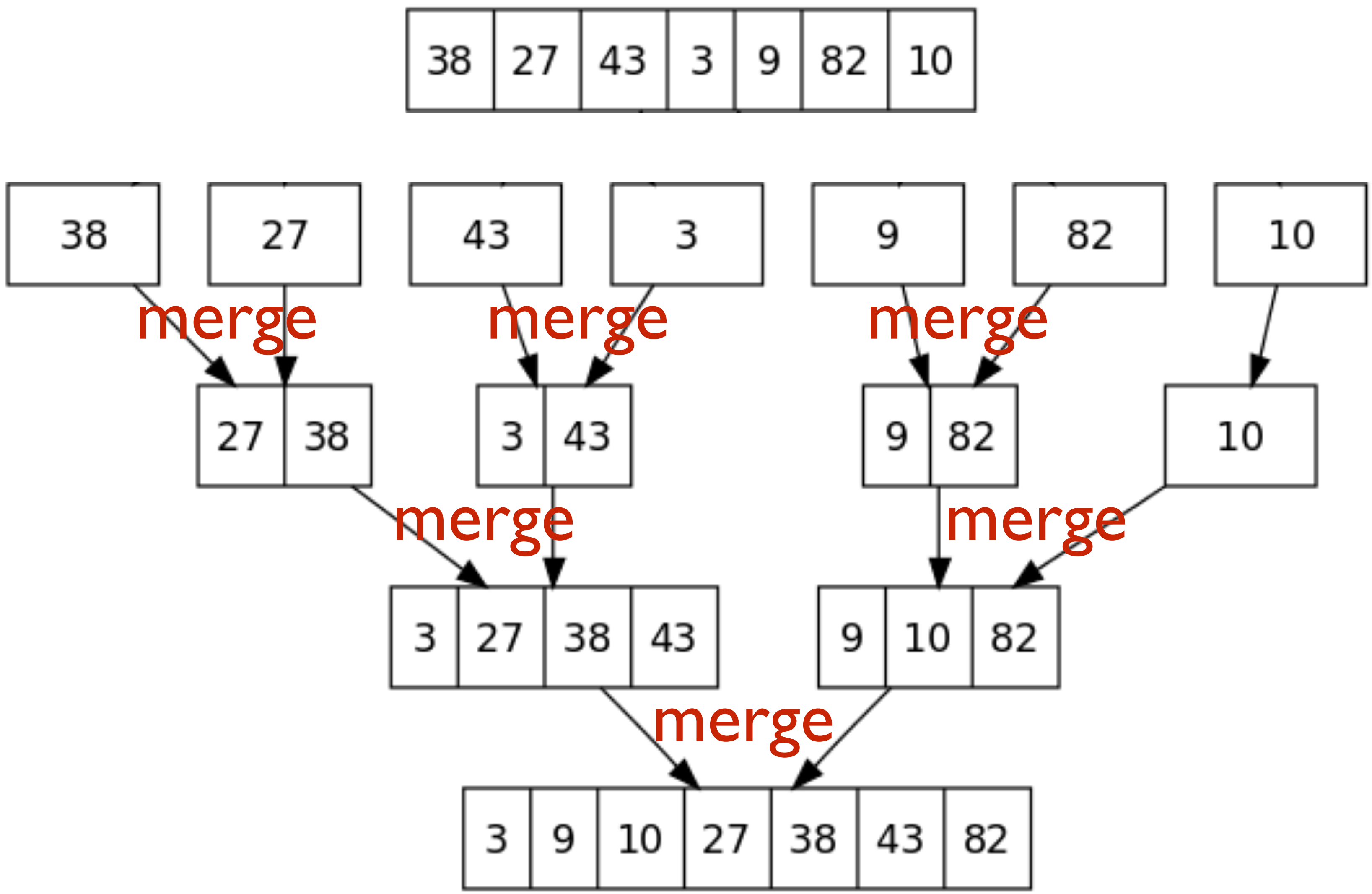
1	3	4	8	9	11	12	15	16

Running time? $N = \mathrm{len}(a) + \mathrm{len}(b)$

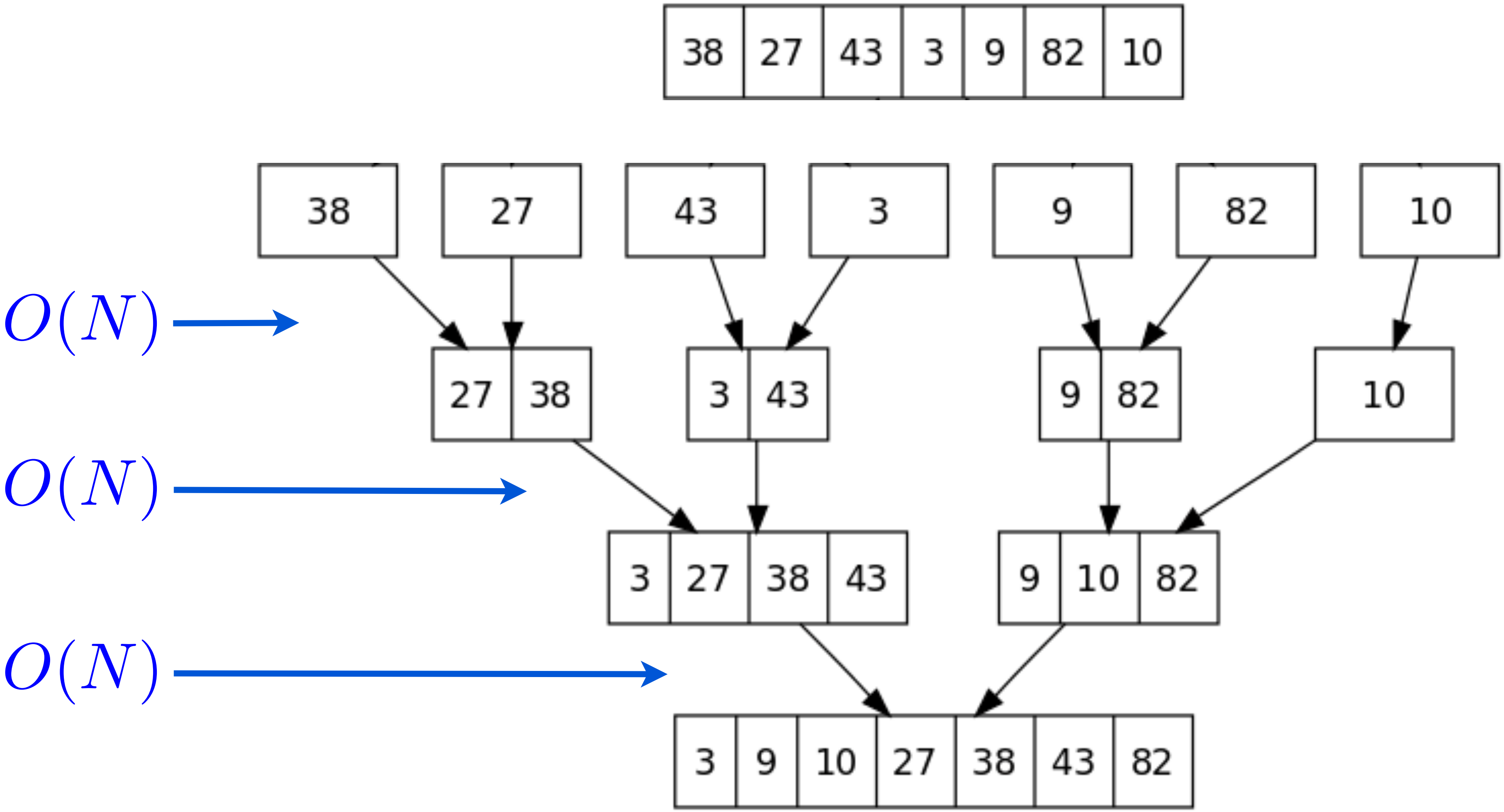
steps: $O(N)$

Merge Sort: Algorithm

Merge Sort



Merge Sort: Running Time



$O(\log N)$ levels

Total: $O(N \log N)$

The Plan

> Merge sort

> Measuring running time when the input is an int

> Efficient data structures: sets and dictionaries

Integer inputs

```
def isPrime(n):
    if (n < 2):
        return False
    for factor in range(2, n):
        if (n % factor == 0):
            return False
    return True
```

Simplifying assumption in 15-112:

Arithmetic operations take constant time.

Integer inputs

```
def isPrime(n):
    if (n < 2):
        return False
    for factor in range(2, n):
        if (n % factor == 0):
            return False
    return True
```

What is the input length?

= number of digits in n

$\sim \log_{10} n$

Integer Inputs

```
def isPrime(m):
    if (m < 2):
        return False
    for factor in range(2, m):
        if (m % factor == 0):
            return False
    return True
```

What is the input length?

= number of digits in m

~ $\log_{10} m$ (actually $\log_2 m$ because it is in binary)

So $N \sim \log_2 m$ i.e., $m \sim 2^N$

What is the running time? $O(m) = O(2^N)$

Integer Inputs

```
def fasterIsPrime(m):
    if (m < 2):
        return False
    maxFactor = int(round(m**0.5))
    for factor in range(3, maxFactor+1):
        if (m % factor == 0):
            return False
    return True
```

What is the running time? $O(2^{N/2})$

isPrime

Amazing result from 2002:

There is a polynomial-time algorithm for primality testing.

Agrawal, Kayal, Saxena

undergraduate students at the time

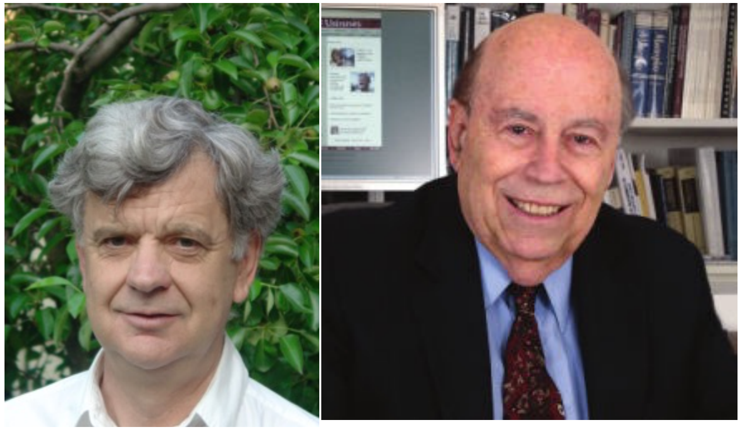
However, best known implementation is ~$O(N^6)$ time.

Not feasible when $N = 2048$.

isPrime

So that's not what we use in practice.

Everyone uses the Miller-Rabin algorithm (1975).



CMU
Professor

The running time is ~ $O(N^2)$.

It is a randomized algorithm with a tiny error probability.

(say $1/2^{300}$)

The Plan

> Merge sort

> Measuring running time when the input is an int

> Efficient data structures: sets and dictionaries

Tangent

Can we cheat exponential time?

What is a data structure?

A data structure allows you to store and maintain a collection of data.

It should support basic operations like:

- add an element to the data structure
- remove an element from the data structure
- find an element in the data structure

…

What is a data structure?

A list is a data structure.

It supports basic operations:

- append( ) $O(1)$
- remove( ) $O(N)$
- **in** operator, index( ) $O(N)$

$\ldots$

One could potentially come up with a different structure which has different running times for basic operations.

Motivating example

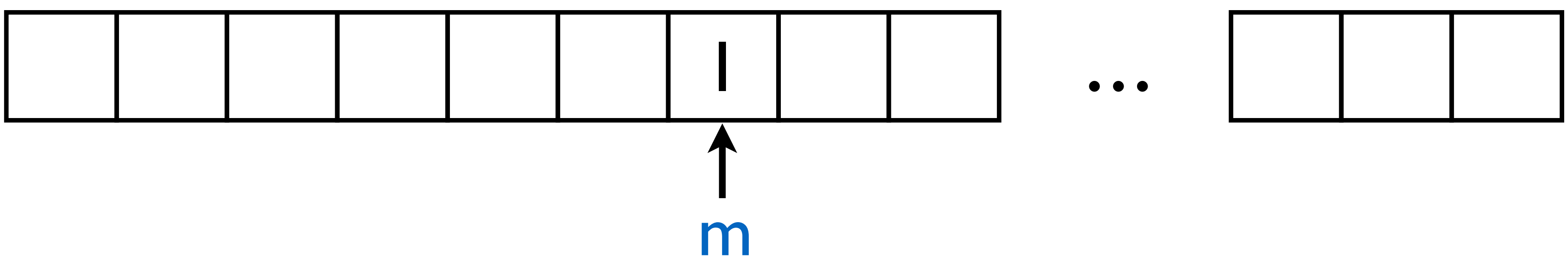
Sorting a list of numbers.

What if I know all the numbers are less than 1 million.

Solution:

Create a list of size 1 million.

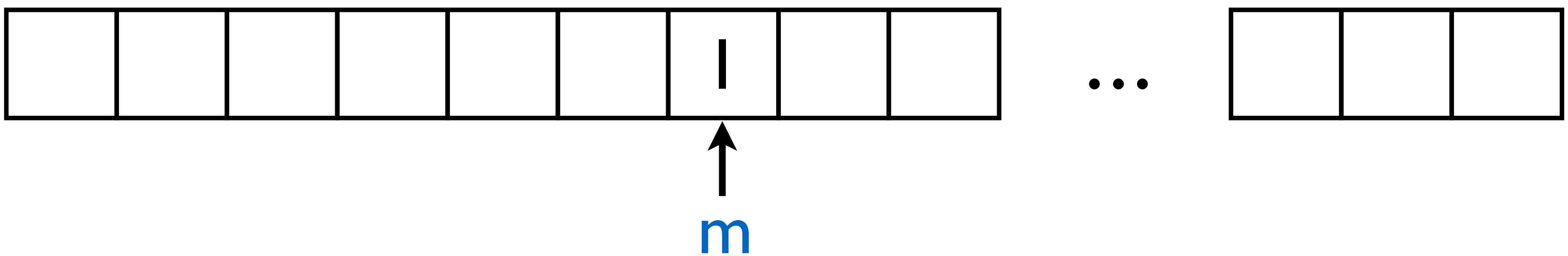
Put number m at index m.



What is the running time for searching for an element?

$O(1)$

Motivating example



The sweet idea:

Connecting value to index.

Motivating example

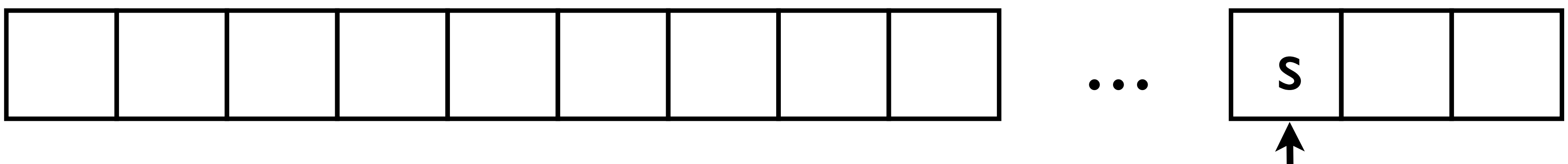
Questions

What if the numbers are not bounded by a million?

What if you want to store strings rather than numbers?

Extending the sweet idea

Storing a collection of strings?



s $\longrightarrow$ h(s) mod (size of list)

Start with a certain size list (e.g. 100)

Pick a function h that maps strings to numbers.

Store s at index h(s) mod (size of list)

h is called a **hash function**.

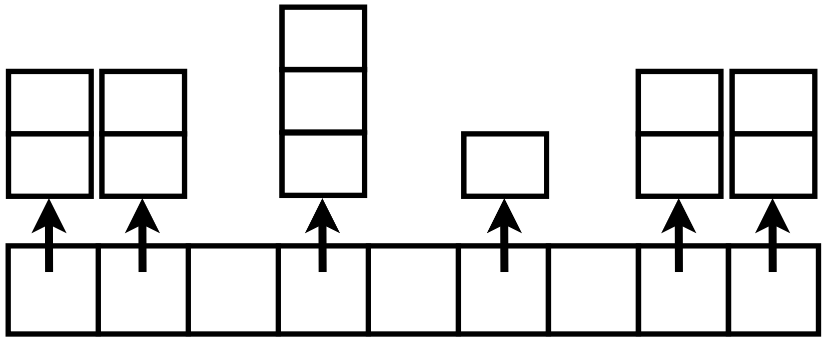
Extending the sweet idea

<u>Potential Problems</u>

Collision: two strings map to the same index

List fills up

HASH TABLE



<u>Fixes</u>

The hash function should be "random"
so that the likelihood of collision is not high.

Store multiple values at one index (bucket)
(e.g. use 2d list)

When buckets get large (say more than 10),
resize and **rehash**: pick a larger list, rehash everything

Extending the sweet idea

What did we gain:

Basic operations add, remove, find/search **super fast**

(sometimes (infrequently) we need to resize/rehash)

What did we lose:

No mutable elements

No order

Repetitions are not good

Sets

Introducing sets

Sets:

- a non-sequential (unordered) collection of objects
- immutable elements
- no repetitions allowed
- look up by object's value
  - finding a value is super efficient

- supports basic operations like:

  s.add(x), s.remove(x), s.union(t), s.intersection(t)
  x **in** s

Creating a set

s = set()

s = set([2, 4, 8])   **# {8, 2, 4}**

s = set([“hello”, 2, True, 3.14])   **# {“hello”, True, 2, 3.14}**

s = set([2, 2, 4, 8])   **# {8, 2, 4}**

s = set([2, 4, [8]])   **# Error**

(sets are mutable, but its elements must be immutable.)

s = set(“hello”)   **# {‘e’, ‘h’, ‘l’, ‘o’}**

s = set((2, 4, 8))   **# {8, 2, 4}**

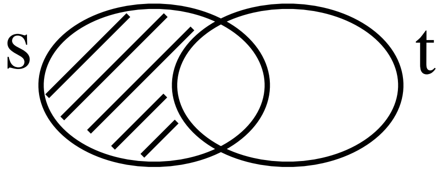
s = set(range(10))   **# {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}**

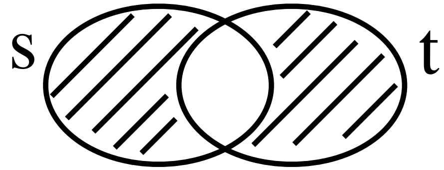
Set methods

Returns a new set (non-destructive):

s.copy()

s.union(t), s.intersection(t),
s.difference(t), s.symmetric_difference(t)





Modifies s (destructive):

s.pop(), s.clear()

s.add(x), s.remove(x), s.discard(x)

s.update(t), s.intersection_update(t),
s.difference_update(t), s.symmetric_difference_update(t)

Other:

s.issubset(t), s.issuperset(t)

The advantage over lists

```
s = set()
for x in range(10000):
    s.add(x)
```

```
print(5000 in s)        # Super fast
print(-1 not in s)      # Super fast
s.remove(100)           # Super fast
```

Essentially $O(1)$

Example: checking for duplicates

Given a list, want to check if there is any element appearing more than once.

Dictionaries (Maps)

Dictionaries / maps

Lists:

- a sequential collection of objects
- can do look up by index (the position in the collection)

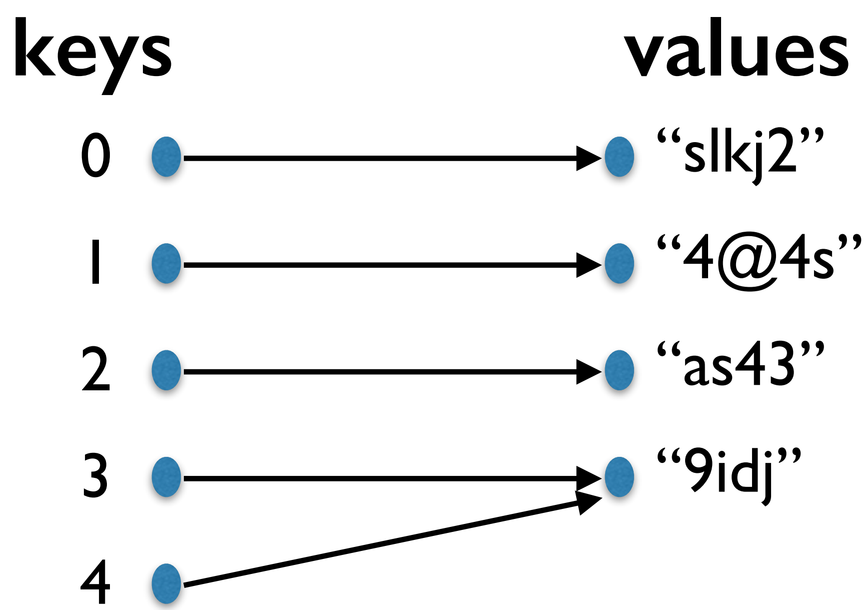
Dictionaries:

- a non-sequential (unordered) collection of objects
- a more flexible look up by keys
- 

Dictionaries / maps

a = [None]*5
a[0] = “slkj2”
a[1] = “4@4s”
a[2] = “as43”
a[3] = “9idj”
a[4] = “9idj”

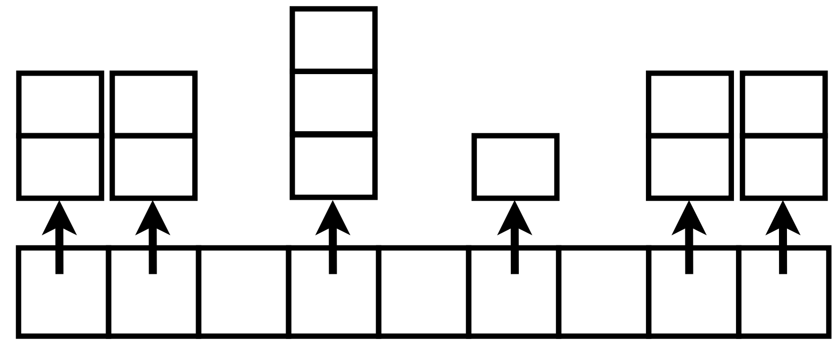
List



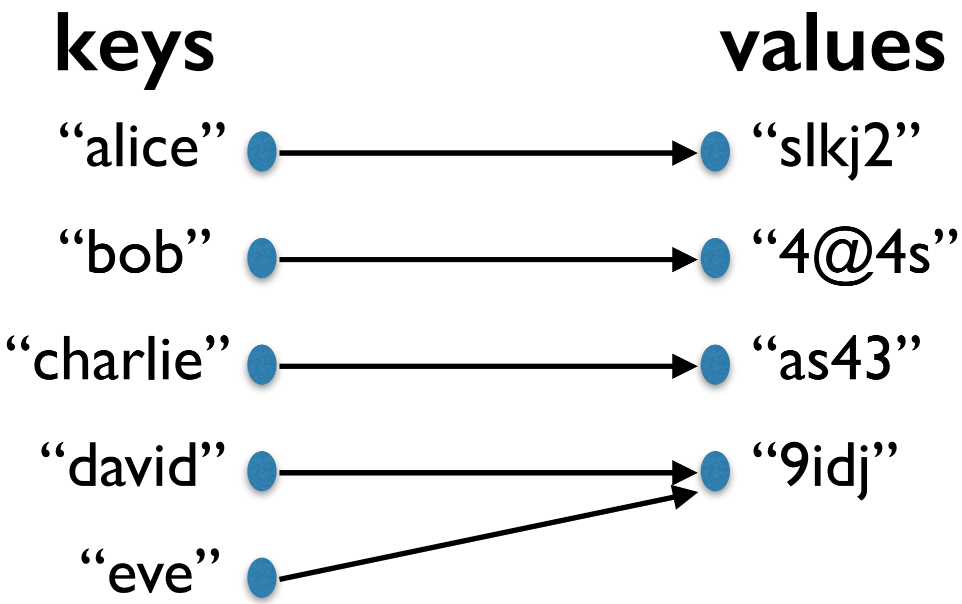
Dictionaries / maps

```
d = dict()
d["alice"] = "slkj2"
d["bob"] = "4@4s"
d["charlie"] = "as43"
d["david"] = "9idj"
d["eve"] = "9idj"
```

HASH TABLE



- hash using the key
- store (key, value) pair



Properties:

- unordered
- values are mutable
- keys form a set (immutable, no repetition)

Dictionaries / maps

Creating dictionaries

users = dict()

users[“alice”] = “sl@3”

users[“bob”] = “#$ks”

users[“charlie”] = “slk92”

---

users = {“alice”: “sl@3”, “bob”: “#$ks”, “charlie”: “slk92”}

---

users = [(“alice”, “sl@3”), (“bob”, “#$ks”), (“charlie”, “#242”)]

users = dict(users)

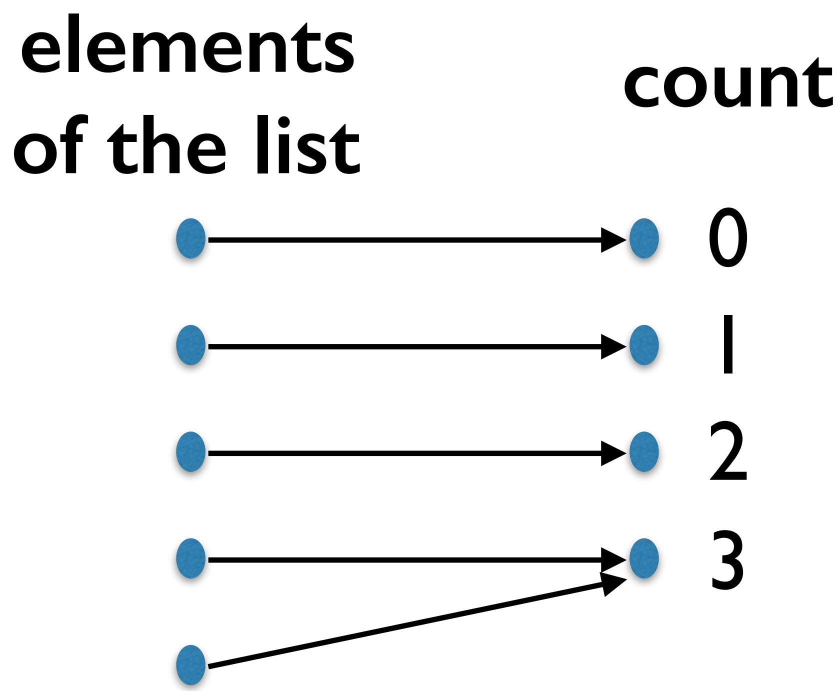
Dictionaries / maps

```
users = {“alice”: “sl@3”, “bob”: “#$ks”, “charlie”: “slk92”}

for key in users:
    print(key, d[key])

print(users[“frank”])            Error

print(users.get(“frank”))        # prints None

print(users.get(“frank”, 0))     # prints 0
```

Example: Find most frequent element

Input: a list of integers

Output: the most frequent element in the list



Exercise: Write the code.