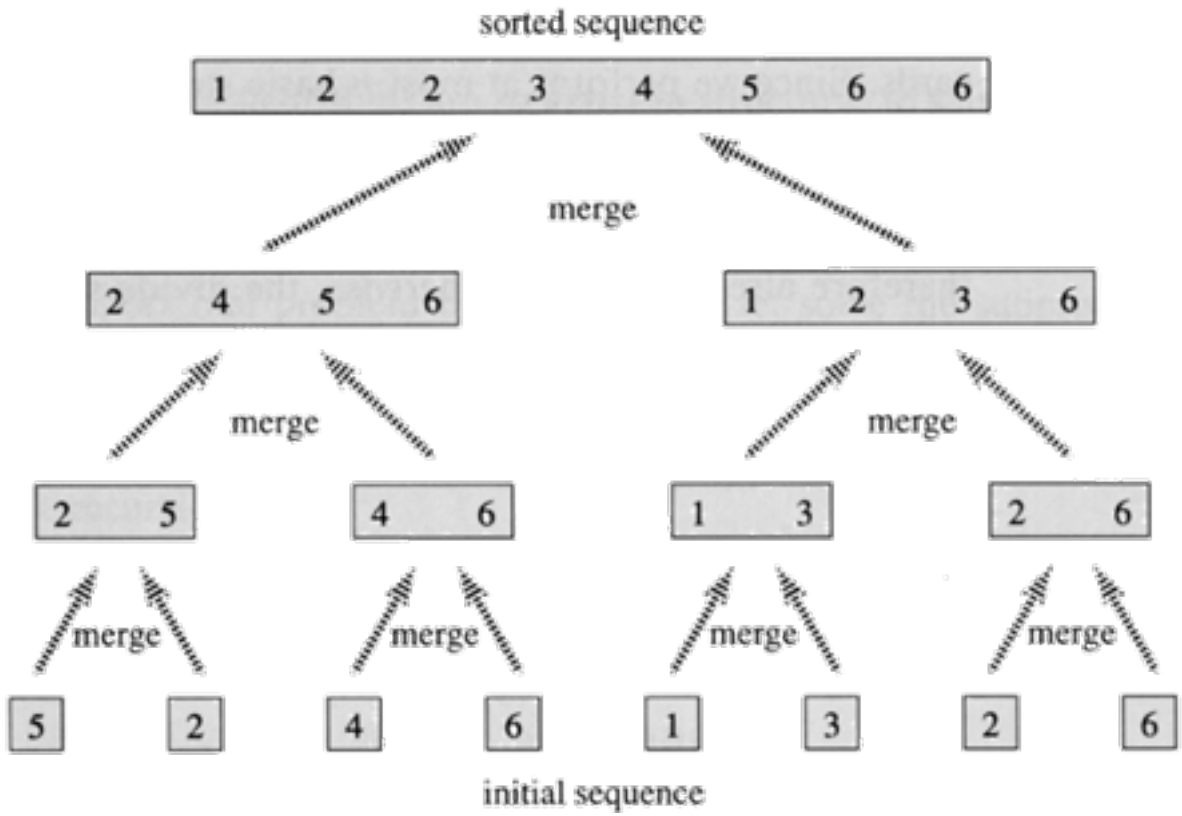
15-112
Fundamentals of Programming

Week 5 - Lecture 2:
Efficiency continued. Merge sort. Sets. Dictionaries.



February 11, 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 three definitions of $\log_2 n$

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

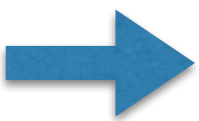
The number k that satisfies $2^k = n$

The inverse of the function $\exp_2(n) = 2^n$

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

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

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

Does the base of the log matter?

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

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

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:

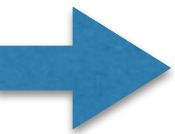
- sort the array, then do binary search, or
- do a linear search?

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 not a list

> 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

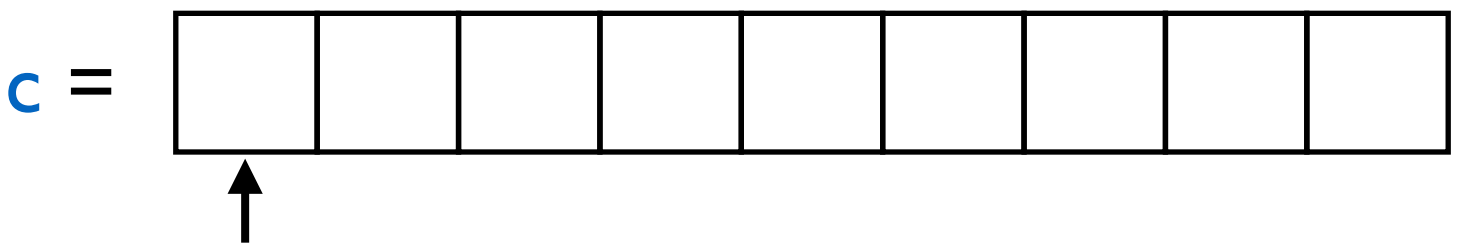
Output: a and b merged into a single list, all sorted.

Turns out we can do this pretty efficiently.

And that turns out to be quite useful!

Merge Sort: Merge Algorithm

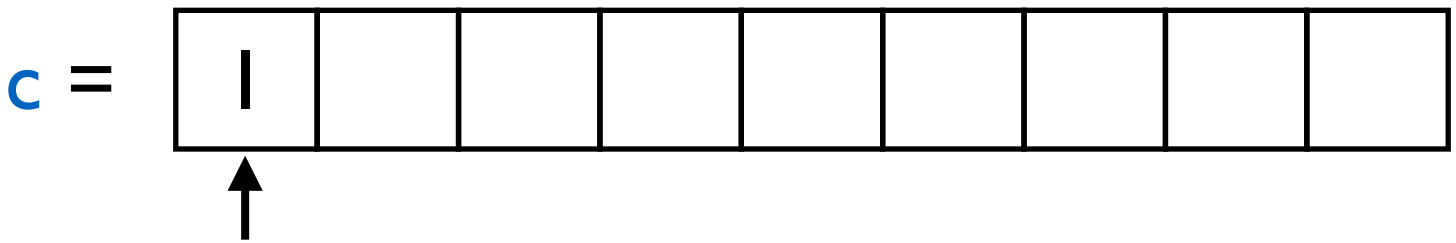
Merge



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

Merge Sort: Merge Algorithm

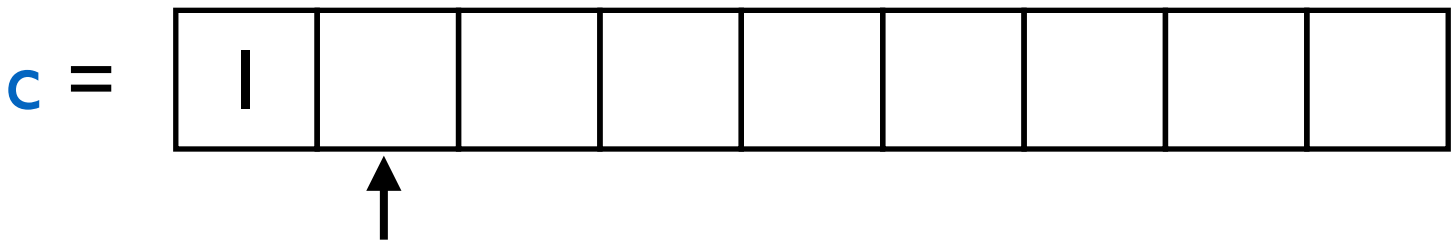
Merge



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

Merge Sort: Merge Algorithm

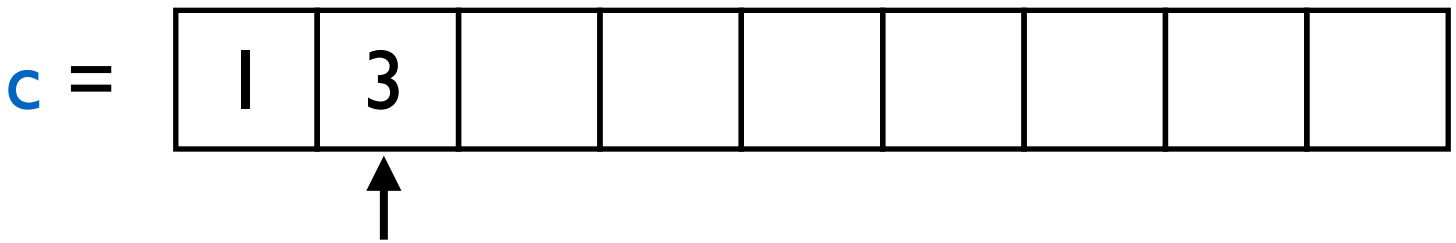
Merge



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

Merge Sort: Merge Algorithm

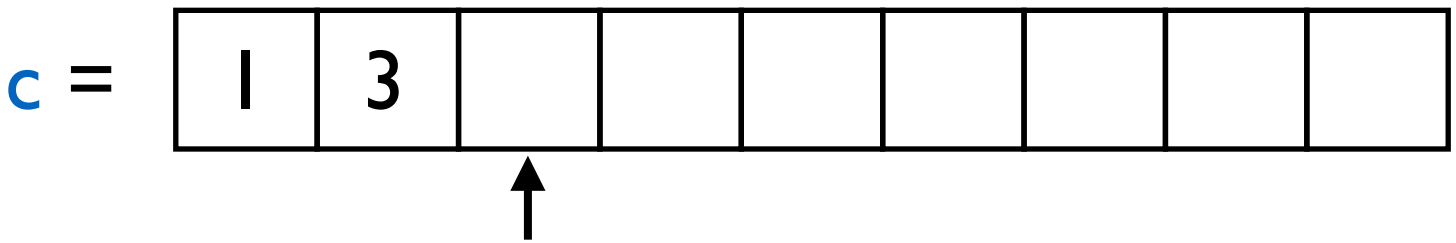
Merge



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

Merge Sort: Merge Algorithm

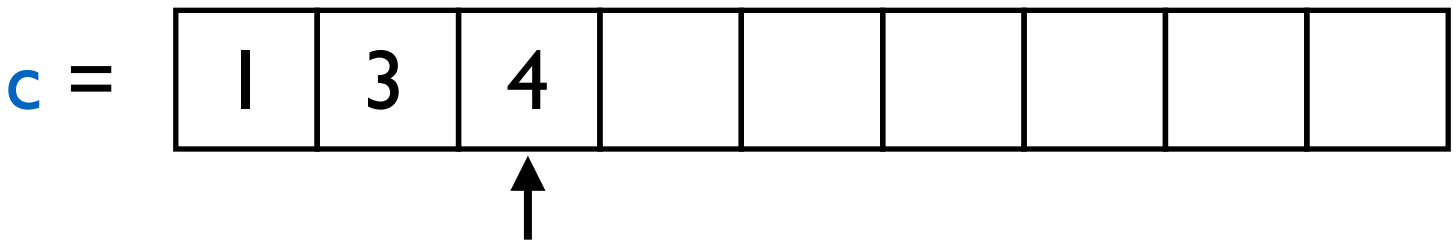
Merge



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

Merge Sort: Merge Algorithm

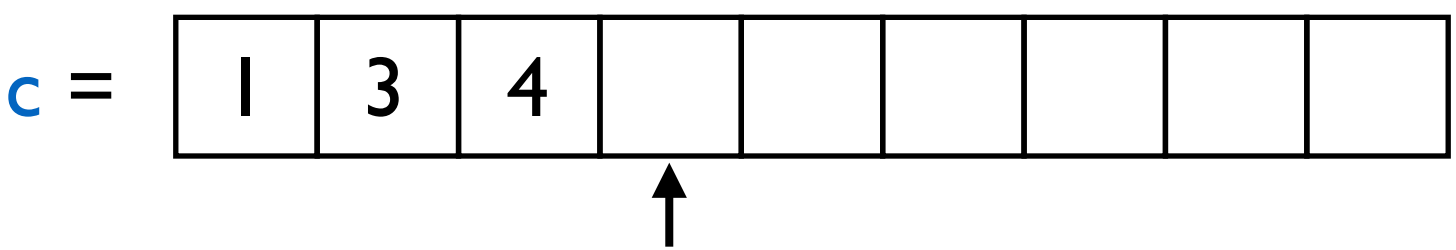
Merge



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

Merge Sort: Merge Algorithm

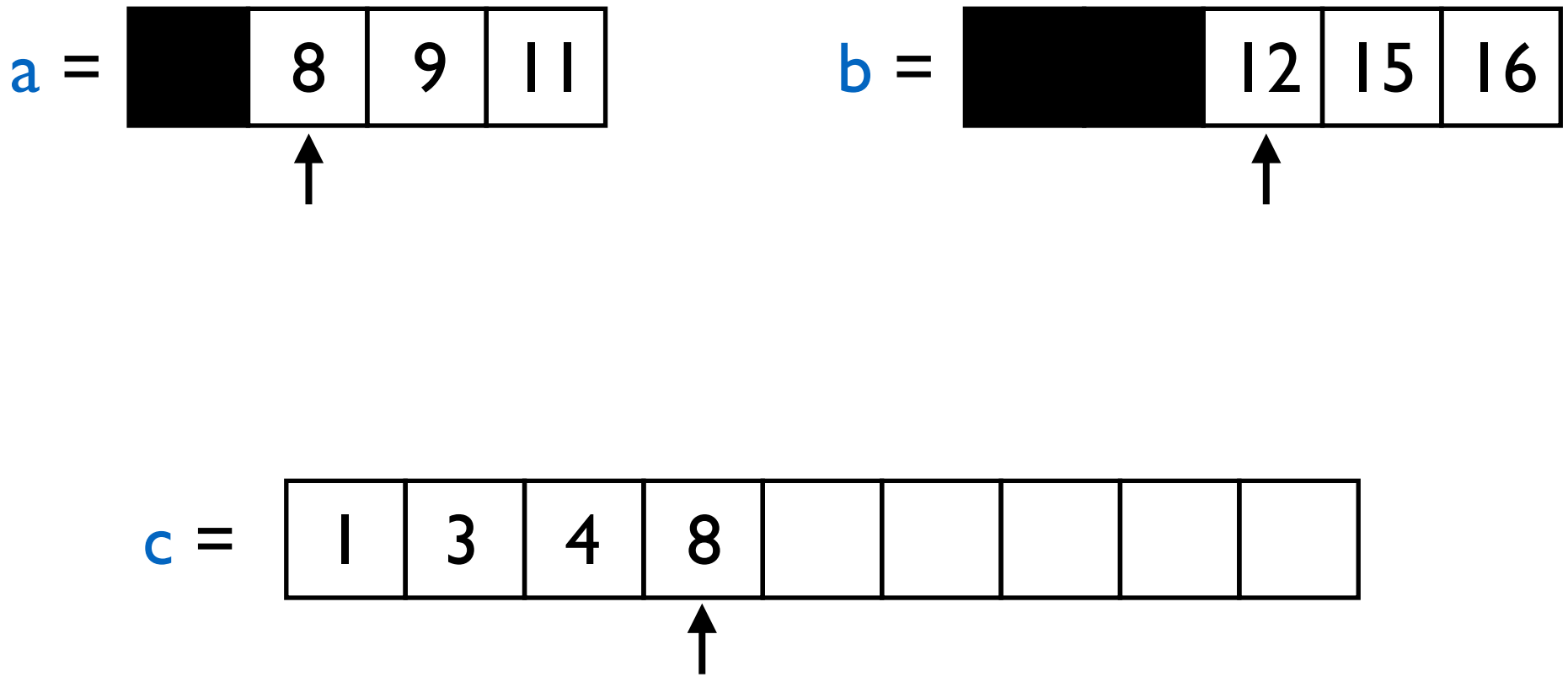
Merge



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

Merge Sort: Merge Algorithm

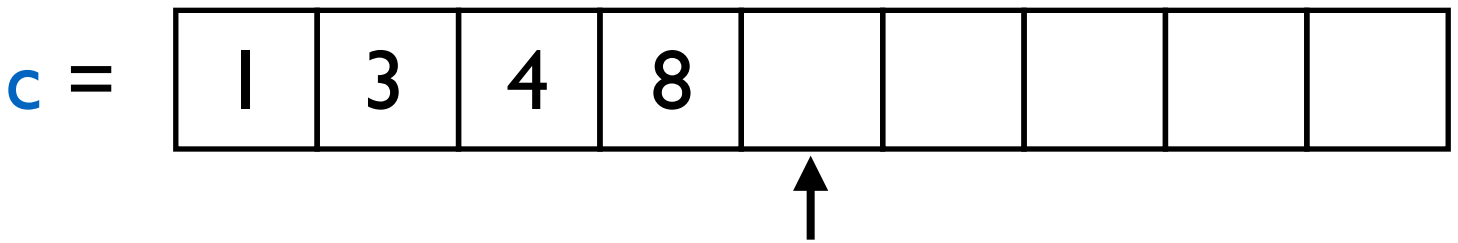
Merge



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

Merge Sort: Merge Algorithm

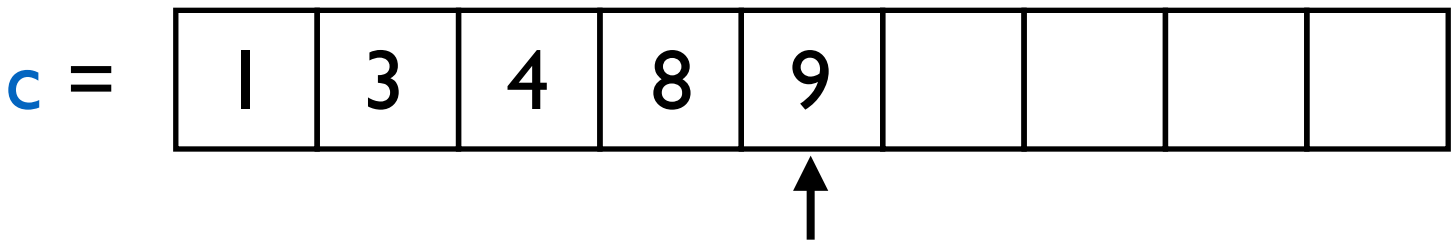
Merge



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

Merge Sort: Merge Algorithm

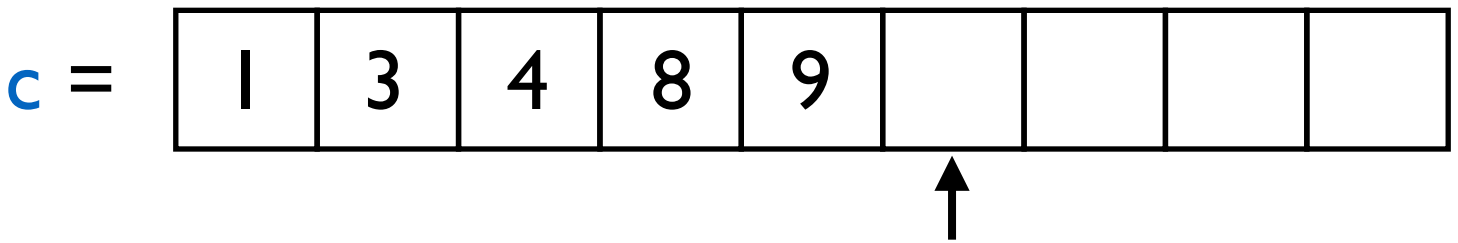
Merge



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

Merge Sort: Merge Algorithm

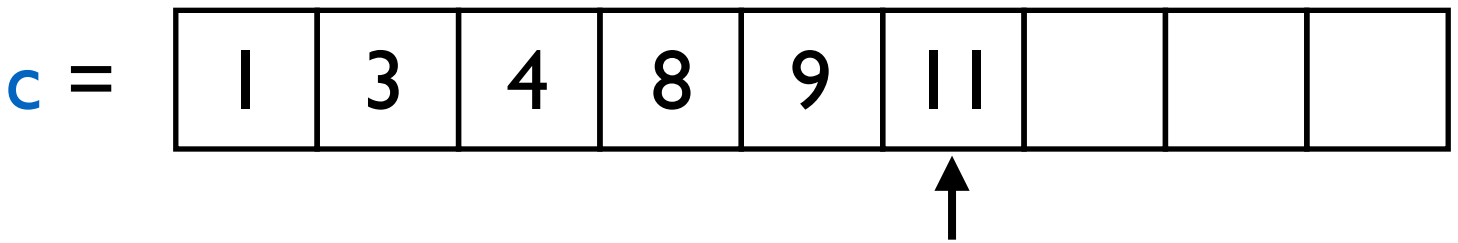
Merge



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

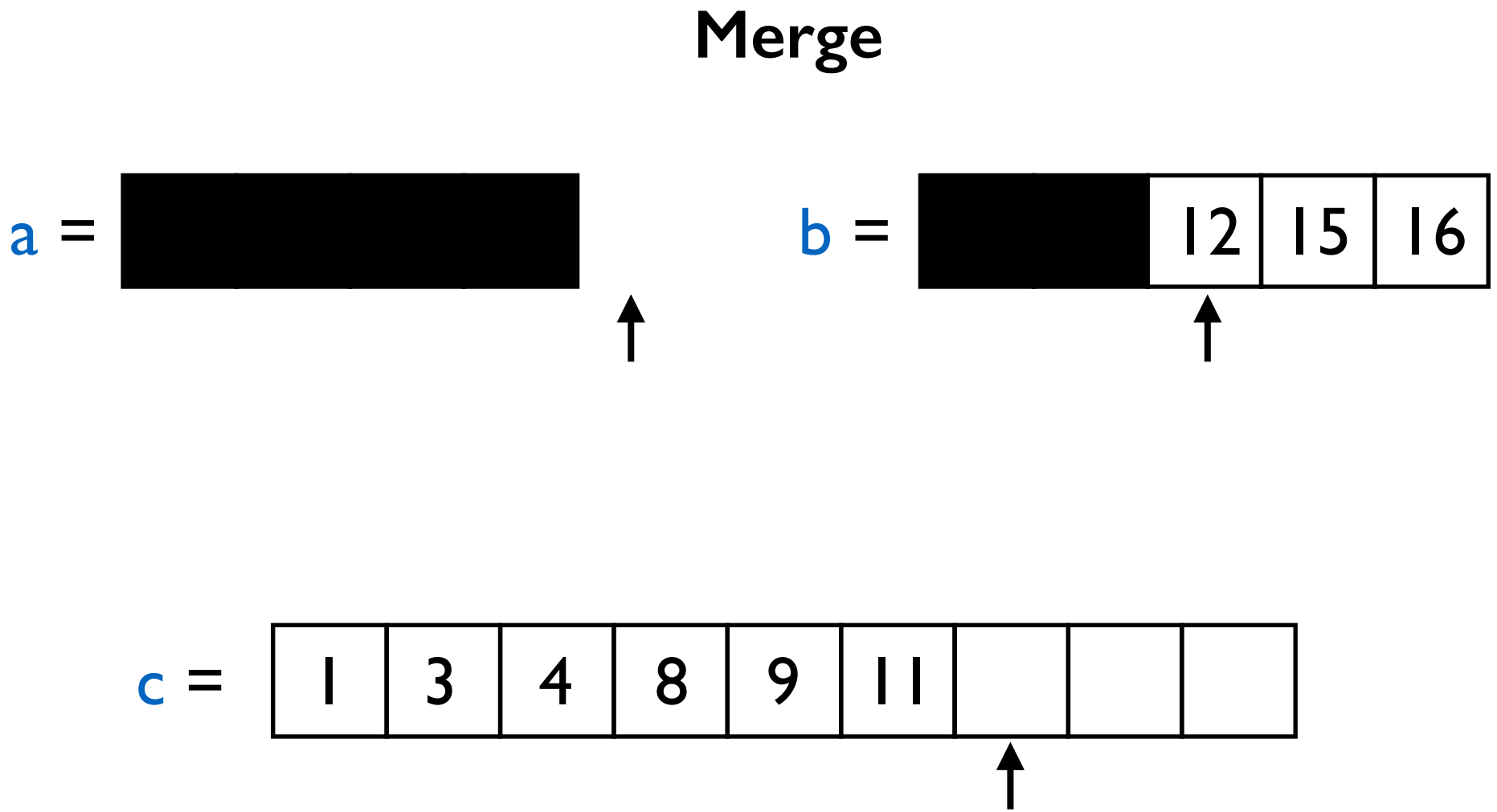
Merge Sort: Merge Algorithm

Merge



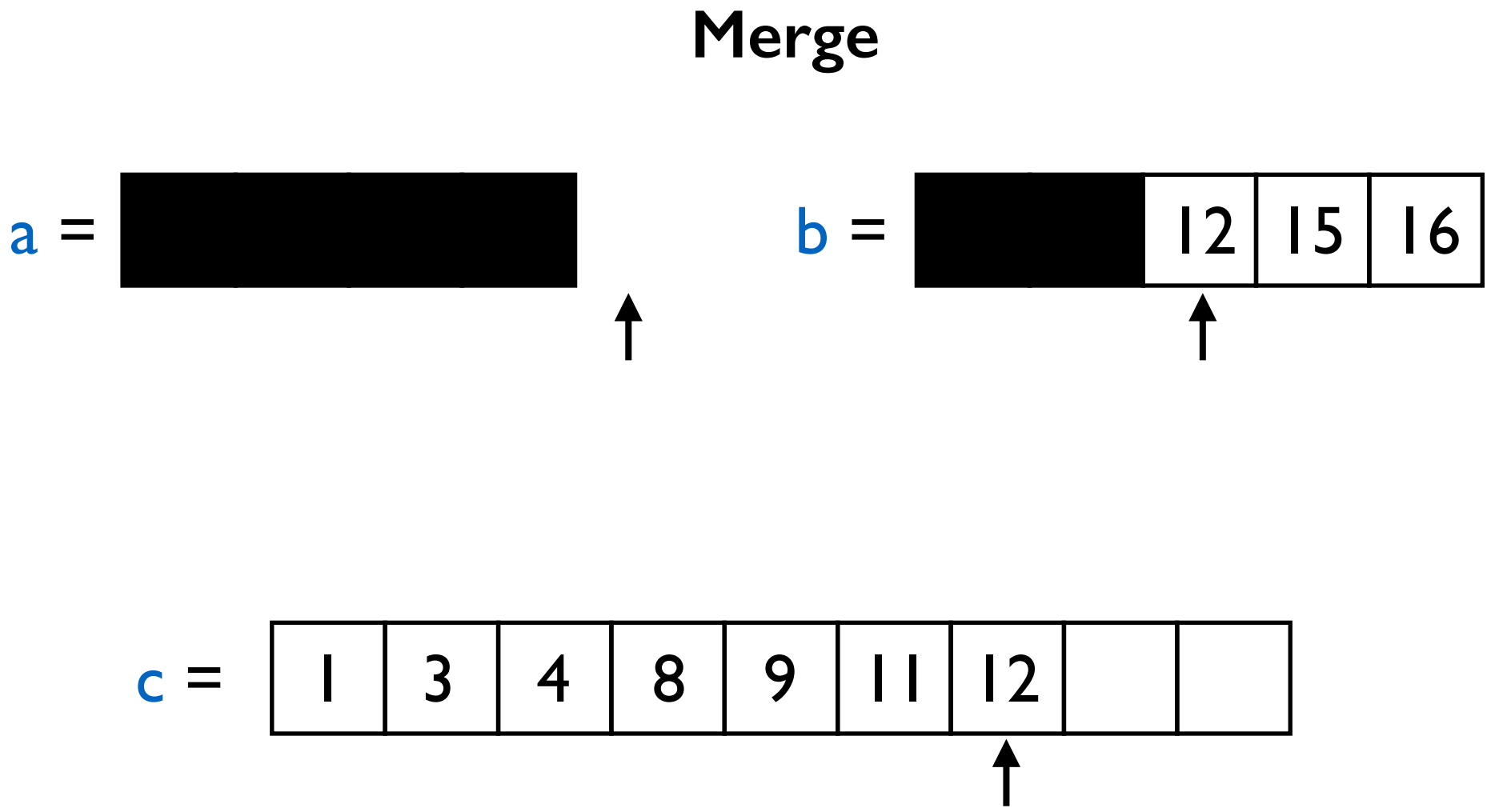
Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm



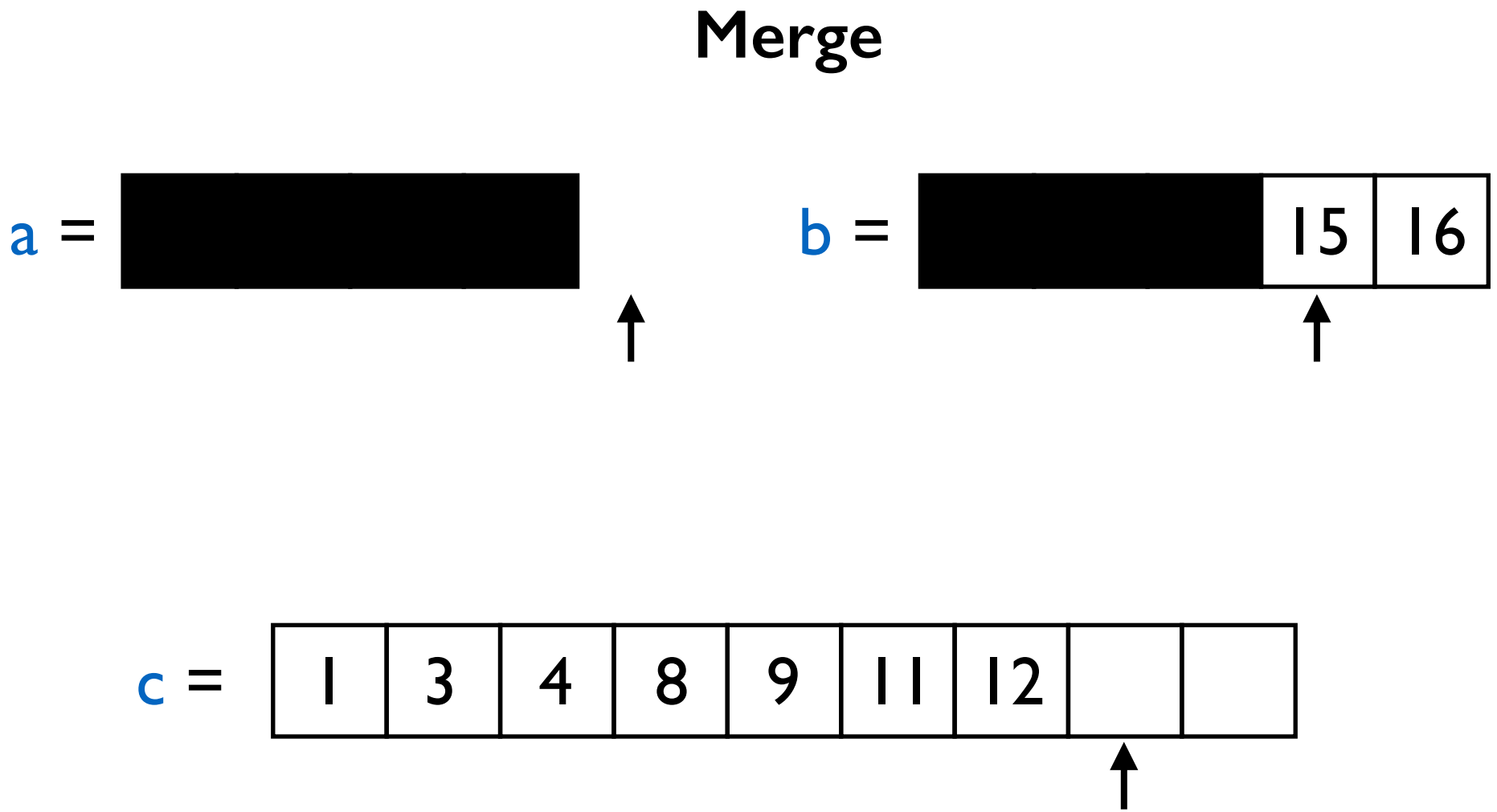
Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm



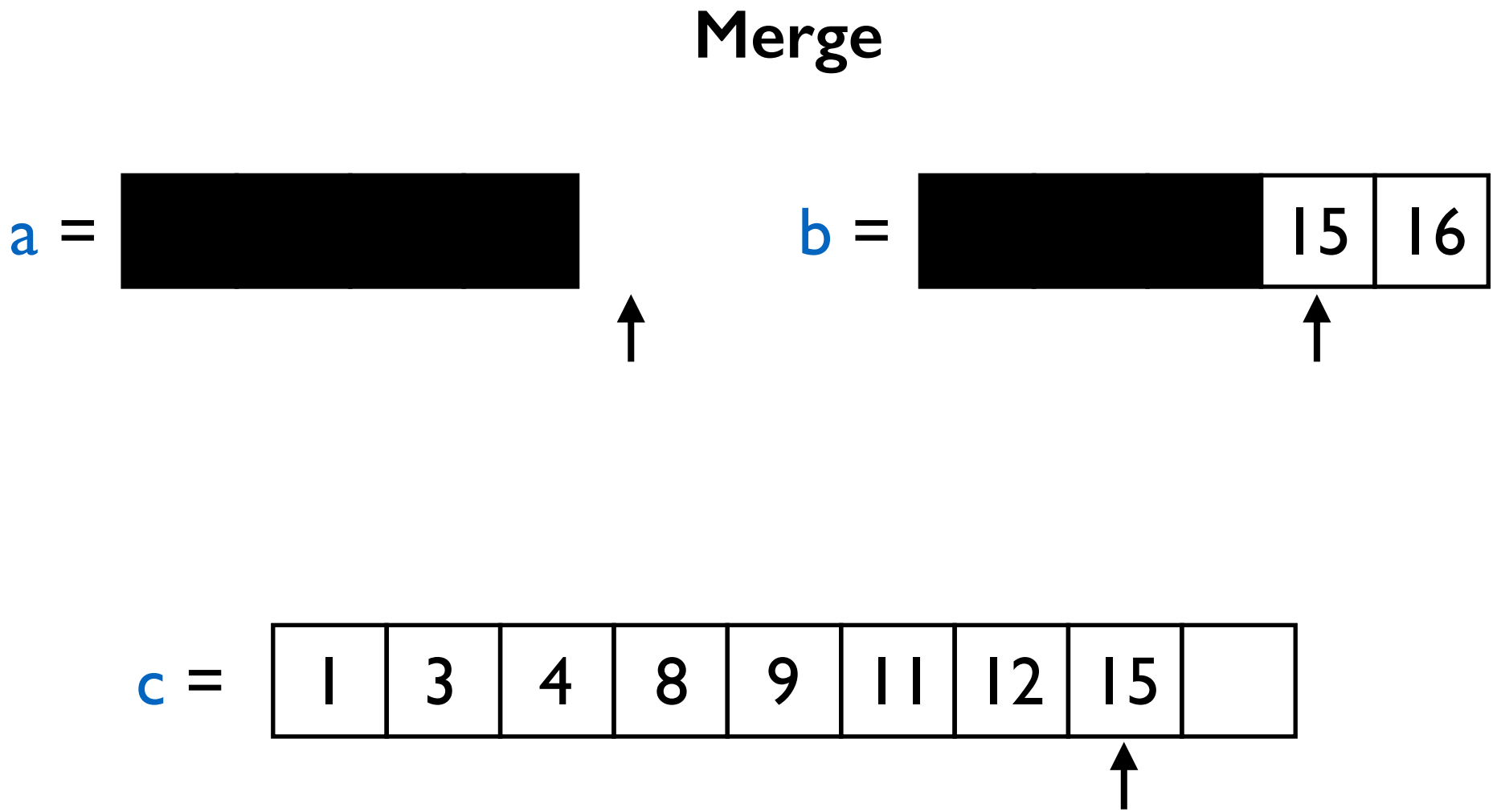
Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm



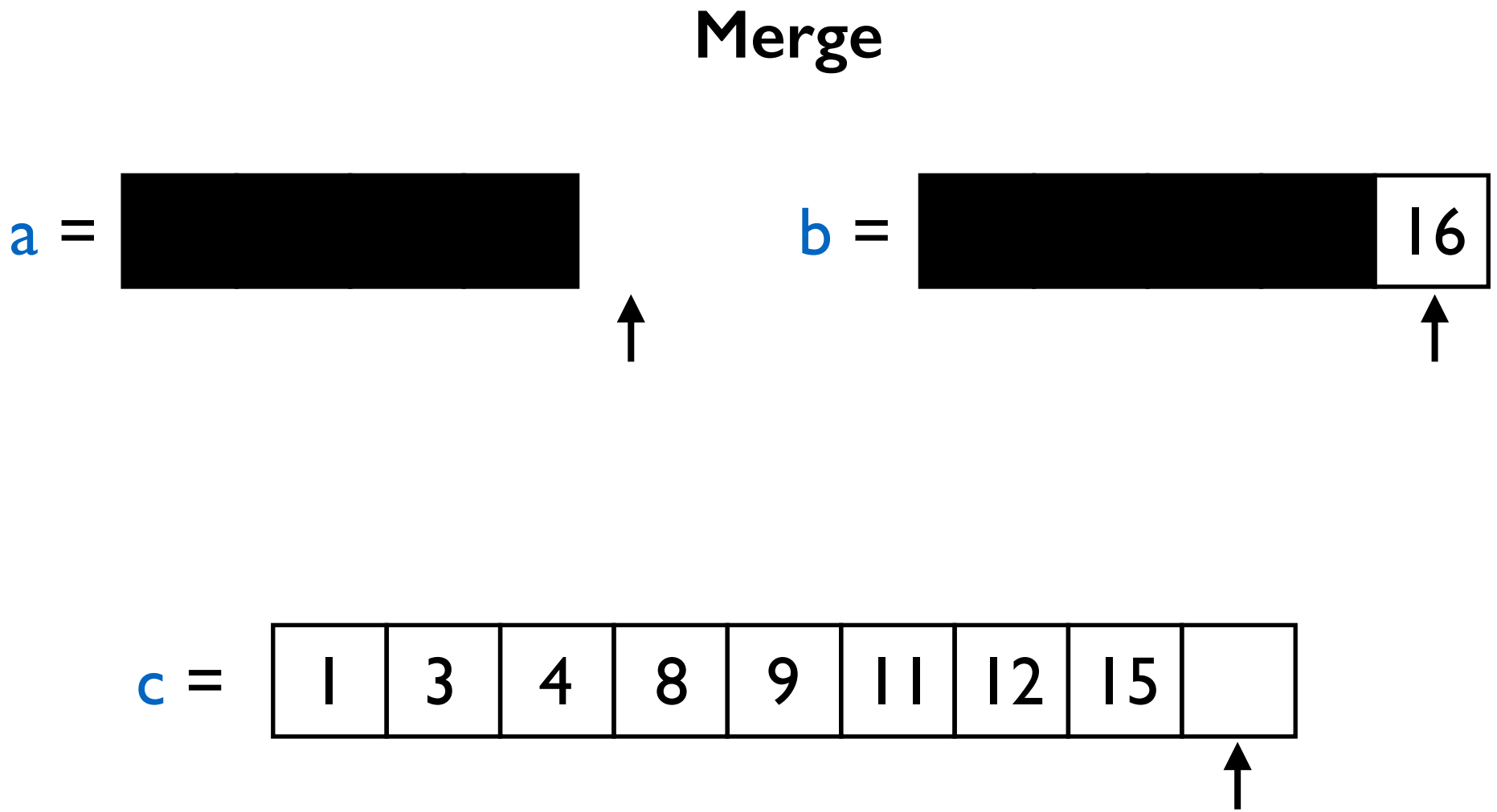
Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm



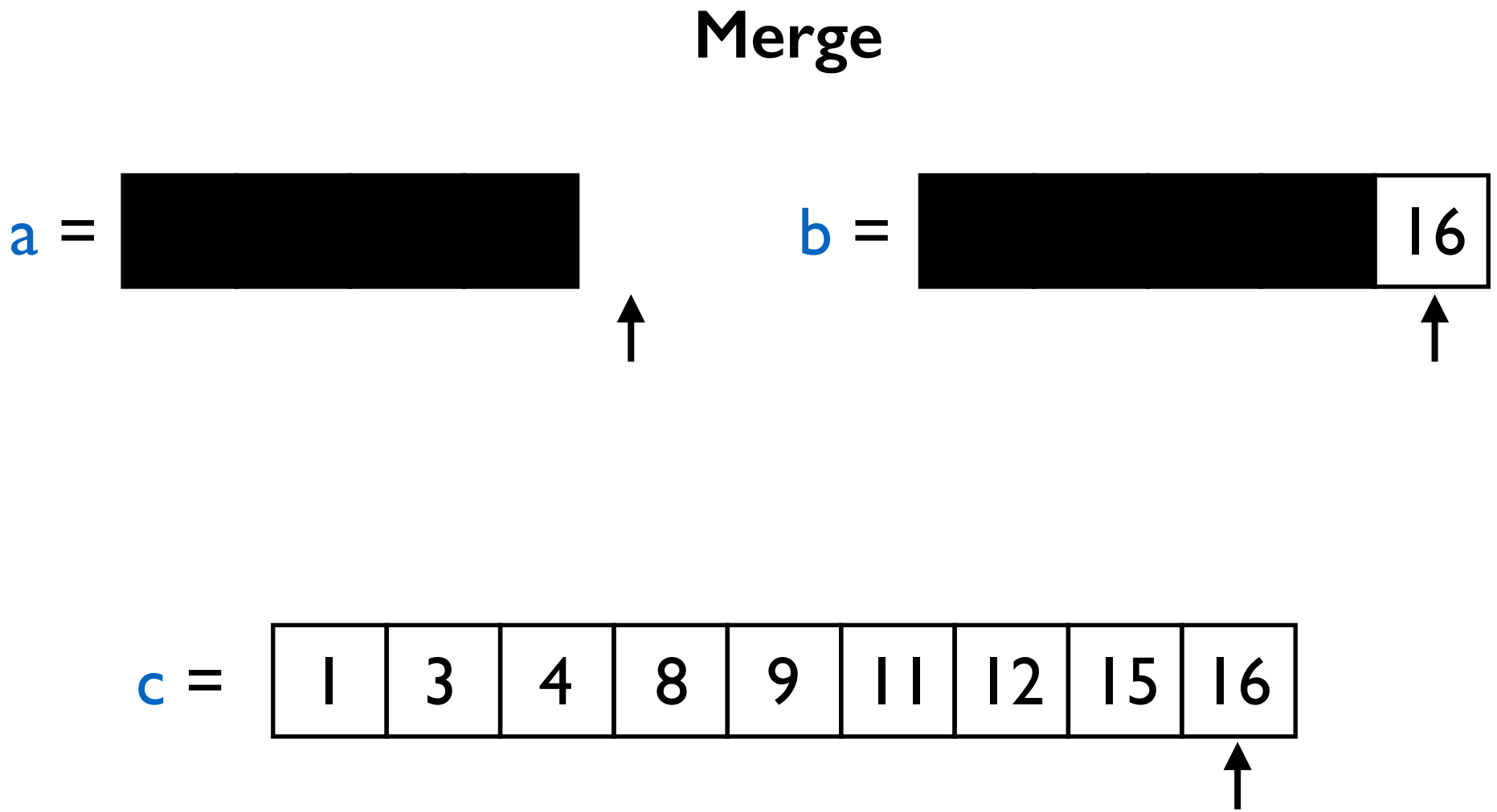
Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm



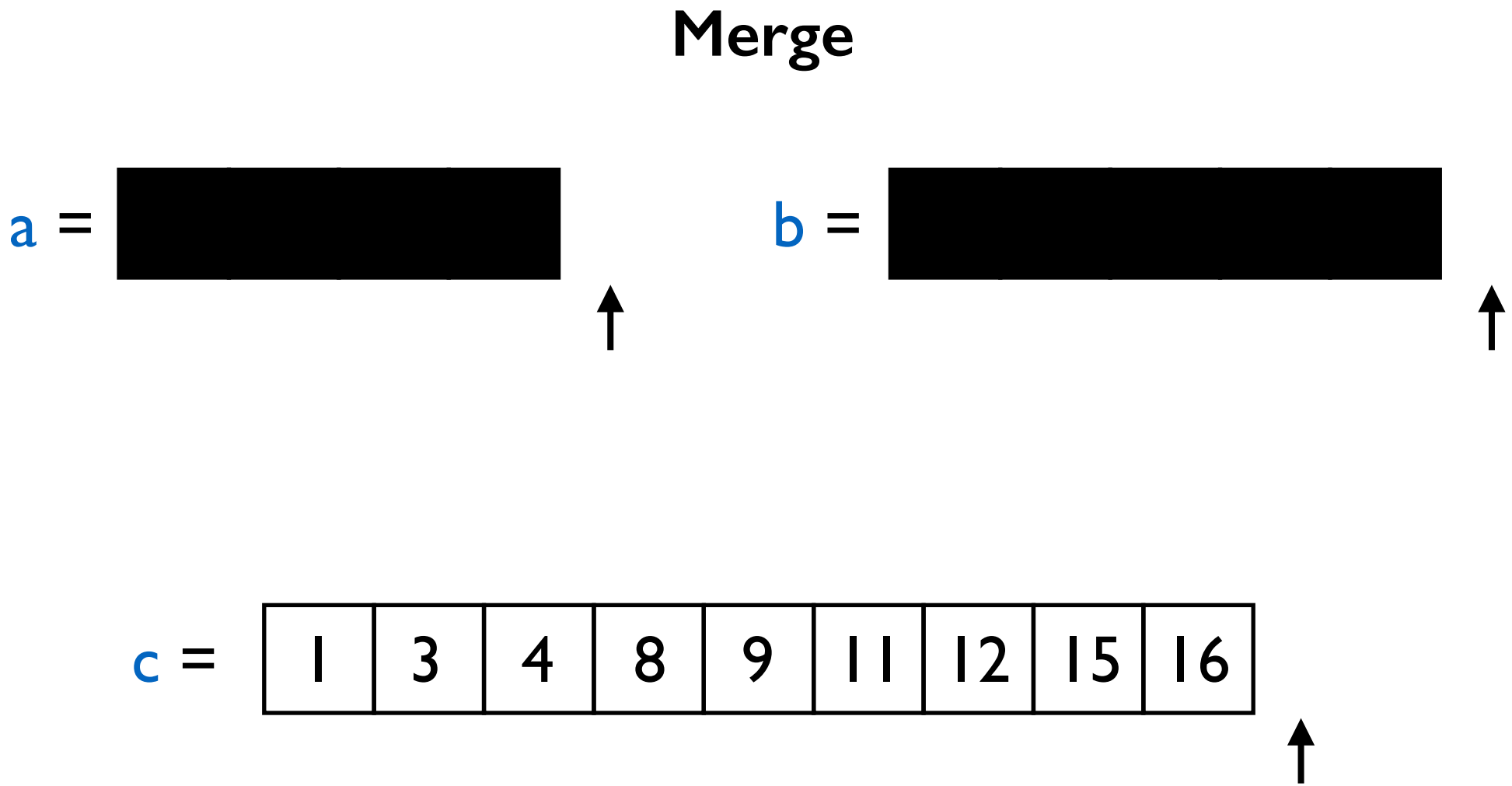
Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm



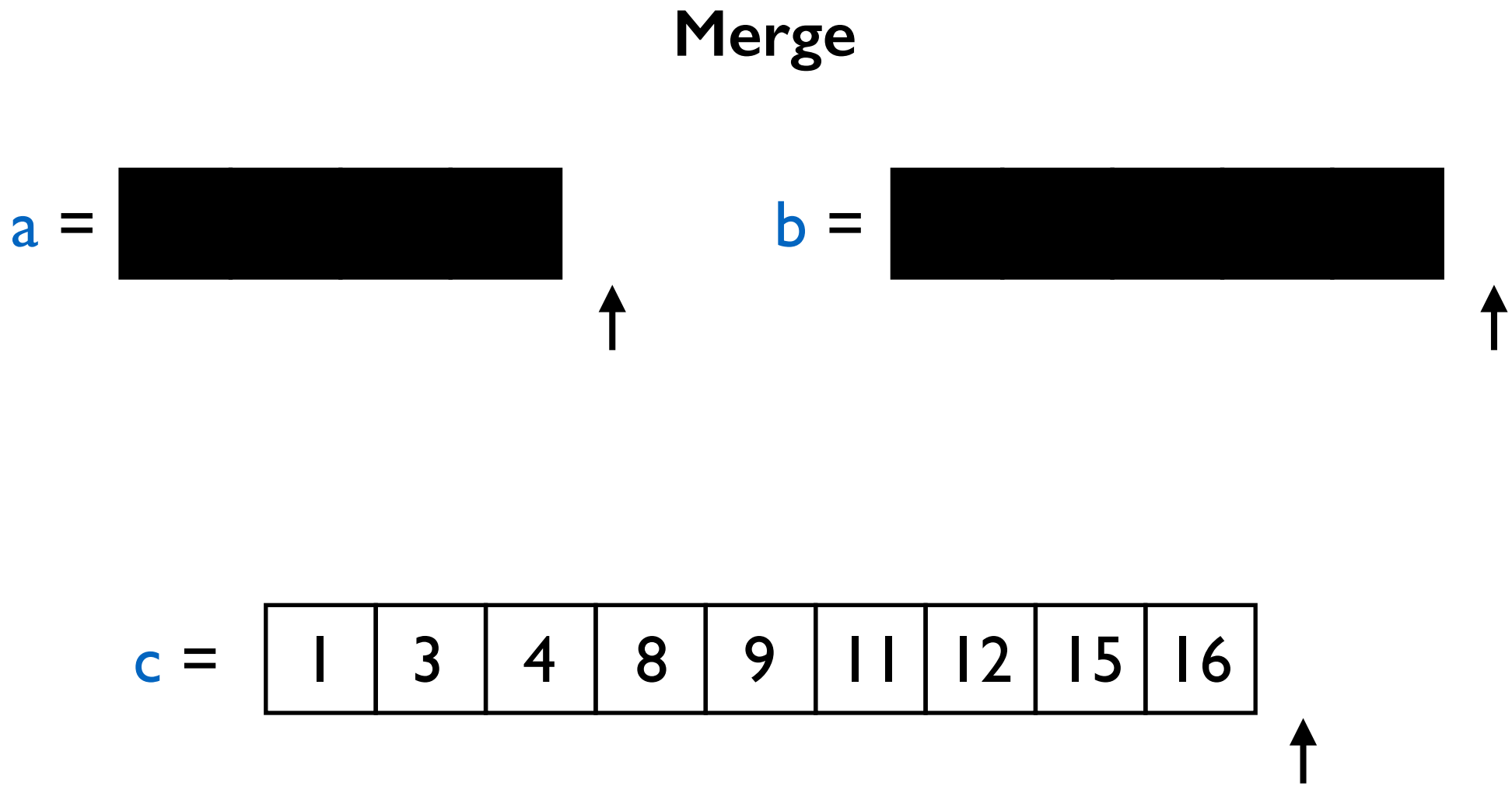
Main idea: min(c) = min(min(a), min(b))

Merge Sort: Merge Algorithm



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

Merge Sort: Merge Running Time

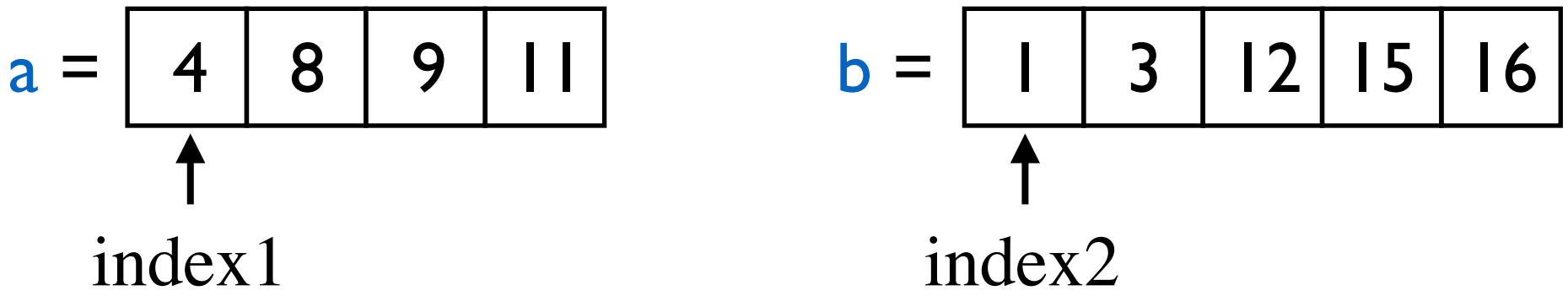


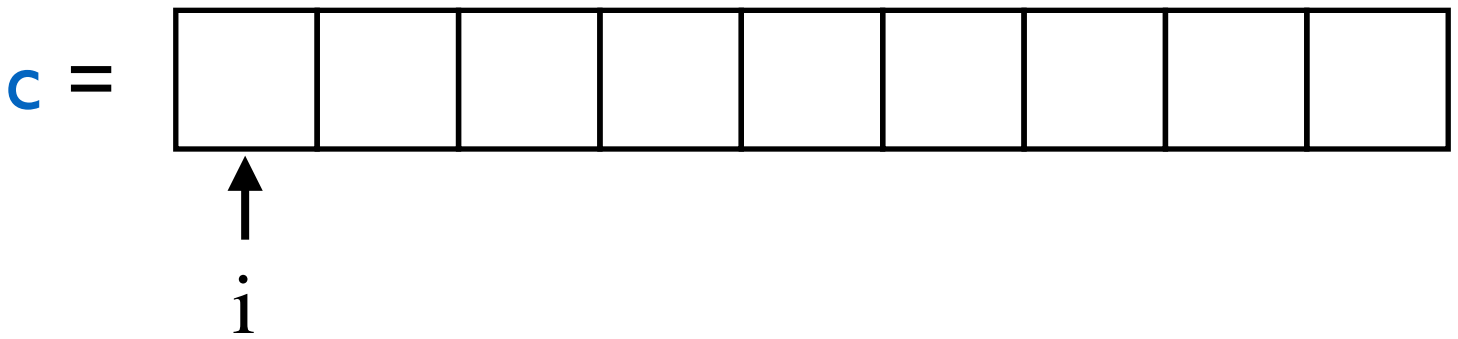
Running time? $N = \text{len}(a) + \text{len}(b)$

steps: $O(N)$

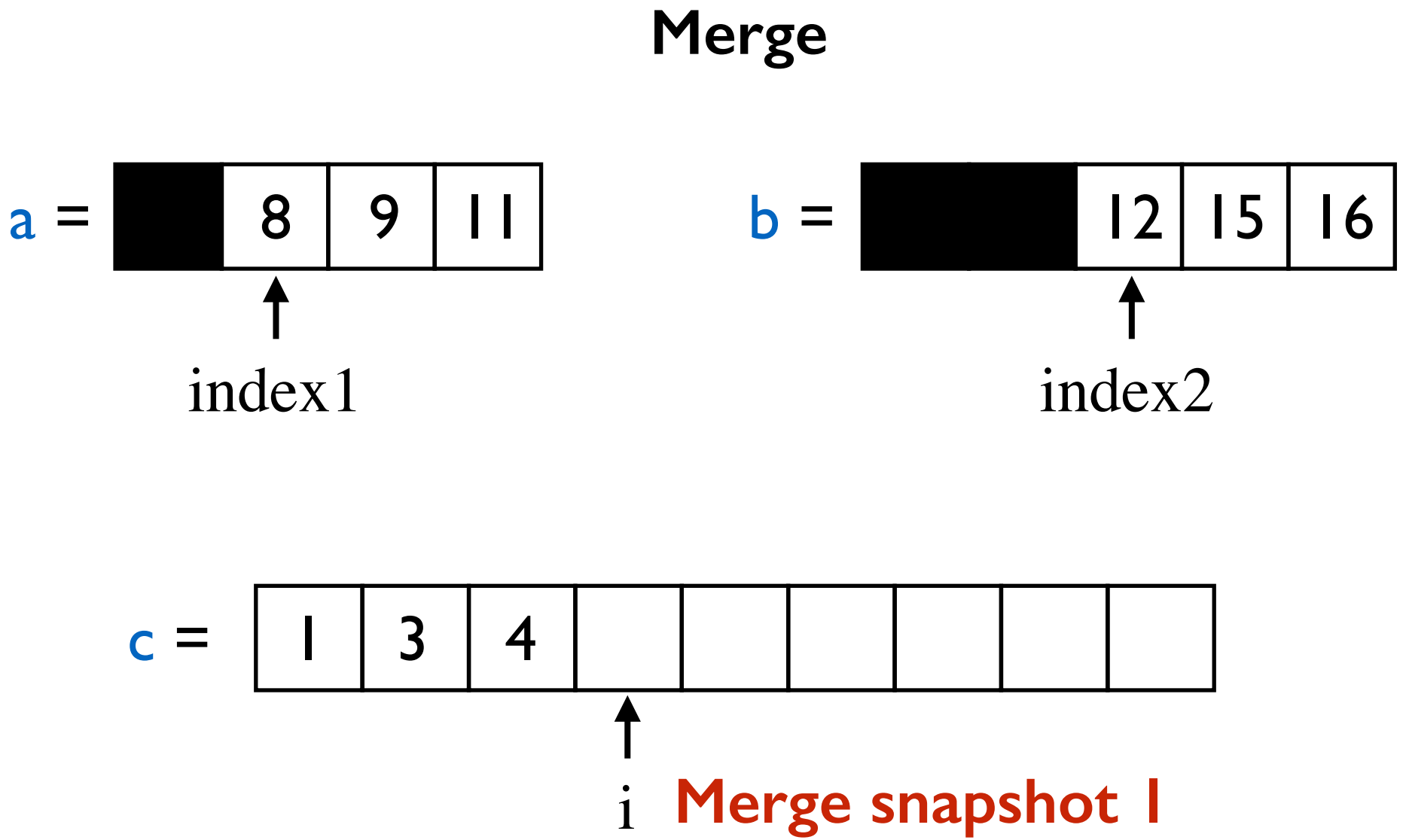
Merge Sort: Merge Code

Merge

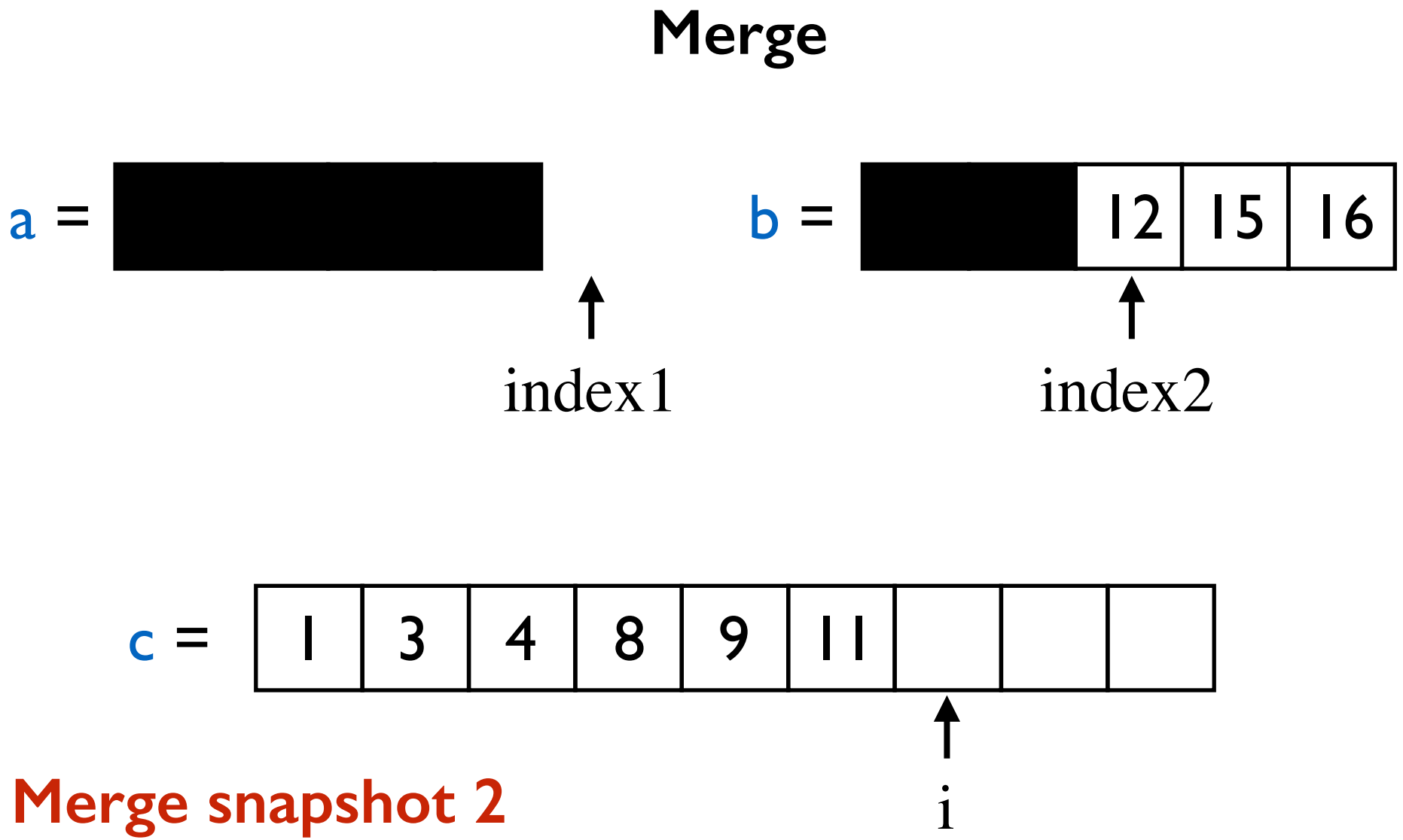




Merge Sort: Merge Code



Merge Sort: Merge Code



Merge snapshot 2

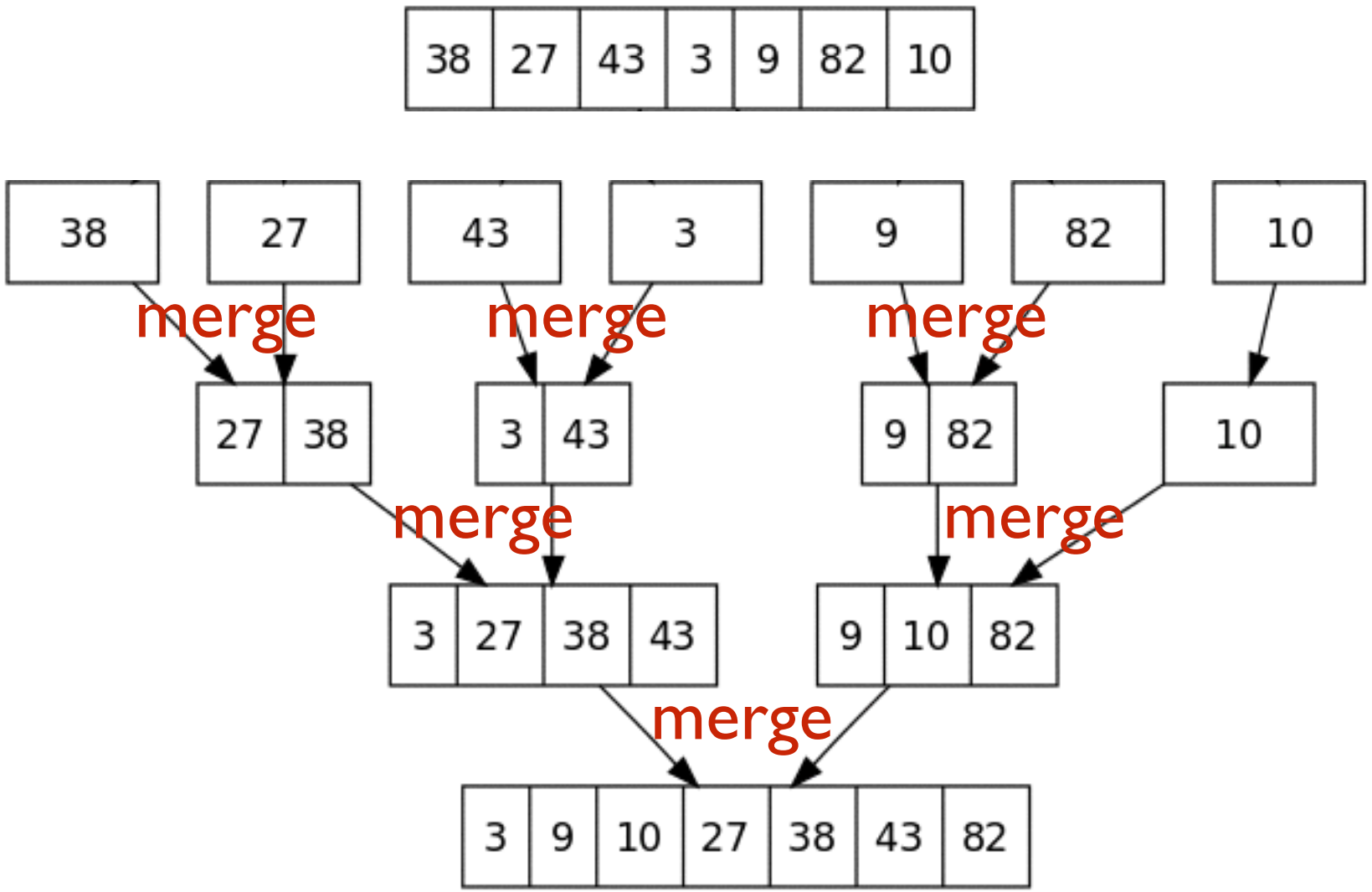
- one of the indices is out of range
- use other list and index to populate c

Merge Sort: Merge Code

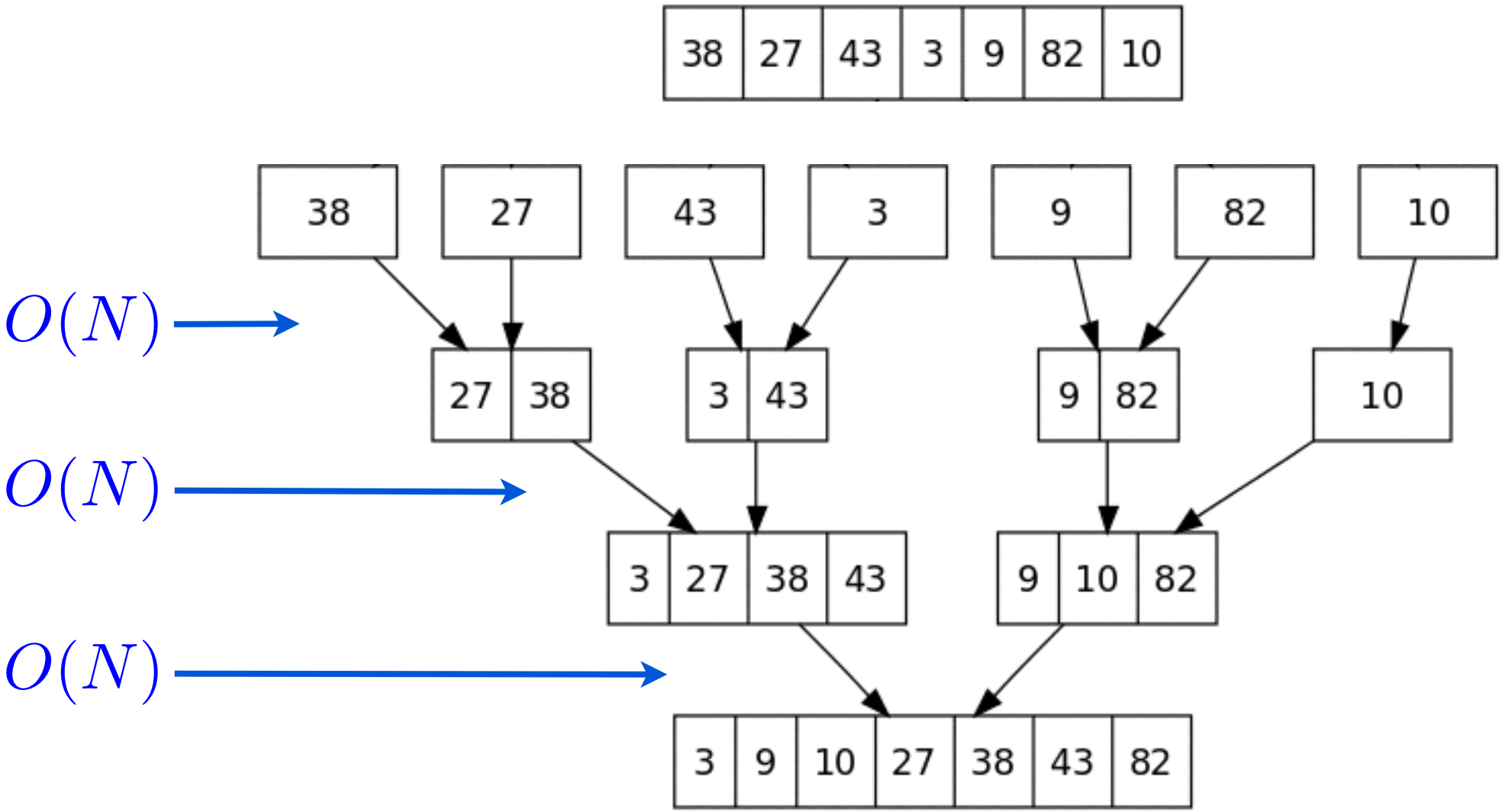
```
def merge(a, b):
    (index1, index2) = (0, 0)
    length = len(a) + len(b)
    c = [None] * length
    for i in range(length):
        if (index1 == len(a)):
            c[i] = b[index2]
            index2 += 1
        elif (index2 == len(b)):
            c[i] = a[index1]
            index1 += 1
        elif (a[index1] < b[index2]):
            c[i] = a[index1]
            index1 += 1
        else:
            c[i] = b[index2]
            index2 += 1
    return c
```

Merge Sort: Algorithm

Merge Sort



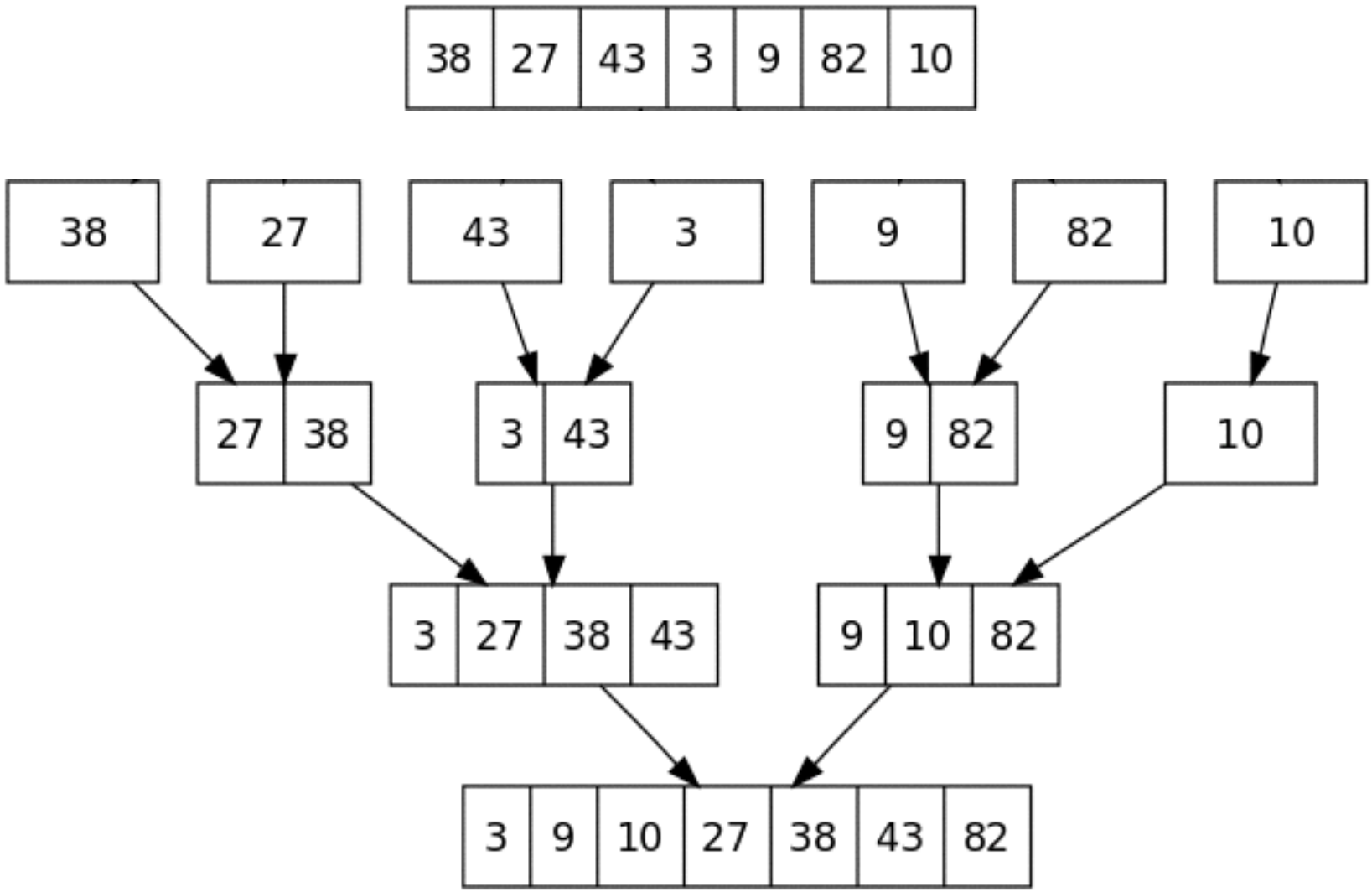
Merge Sort: Running Time



$O(\log N)$ levels

Total: $O(N \log N)$

Merge Sort: Code

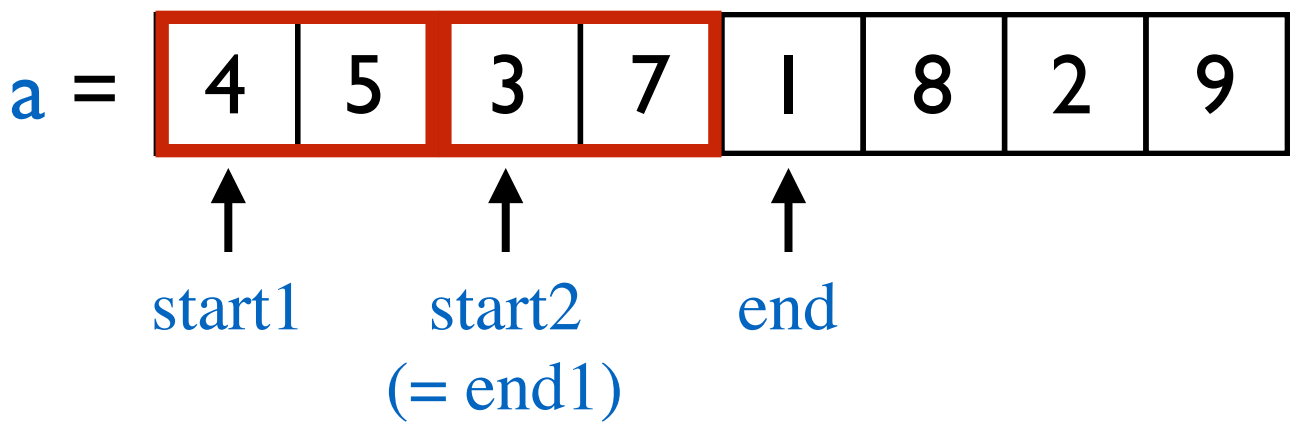


Every time we slice and merge, we create new lists.

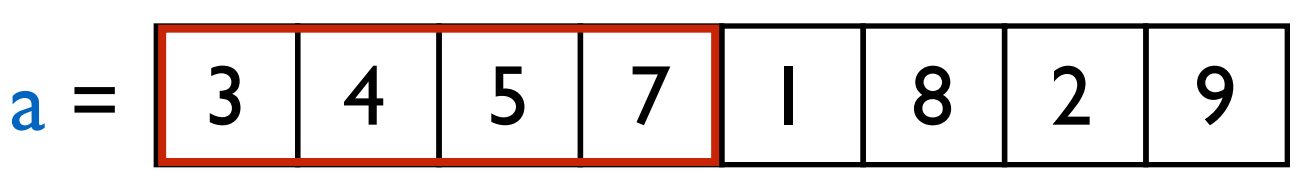
Better to do the merge operations “in-place”.

In-place Merge Code

```
def merge(a, start1, start2, end):
    (index1, index2) = (start1, start2)
    length = end - start1
    c = [None] * length
    for i in range(length):
        if (index1 == start2):
            c[i] = a[index2]
            index2 += 1
        elif (index2 == end):
            c[i] = a[index1]
            index1 += 1
        elif (a[index1] < a[index2]):
            c[i] = a[index1]
            index1 += 1
        else:
            c[i] = a[index2]
            index2 += 1
    for i in range(start1, end):
        a[i] = c[i - start1]
```

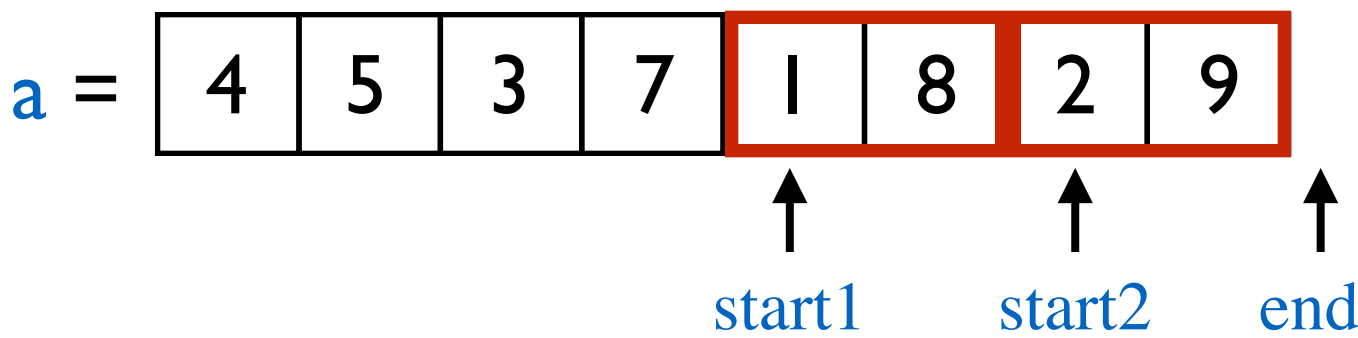


merge(a, 0, 2, 4)

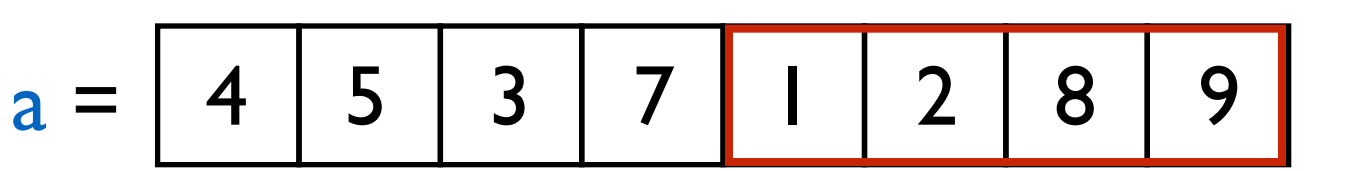


In-place Merge Code

```
def merge(a, start1, start2, end):
    (index1, index2) = (start1, start2)
    length = end - start1
    c = [None] * length
    for i in range(length):
        if (index1 == start2):
            c[i] = a[index2]
            index2 += 1
        elif (index2 == end):
            c[i] = a[index1]
            index1 += 1
        elif (a[index1] < a[index2]):
            c[i] = a[index1]
            index1 += 1
        else:
            c[i] = a[index2]
            index2 += 1
    for i in range(start1, end):
        a[i] = c[i - start1]
```

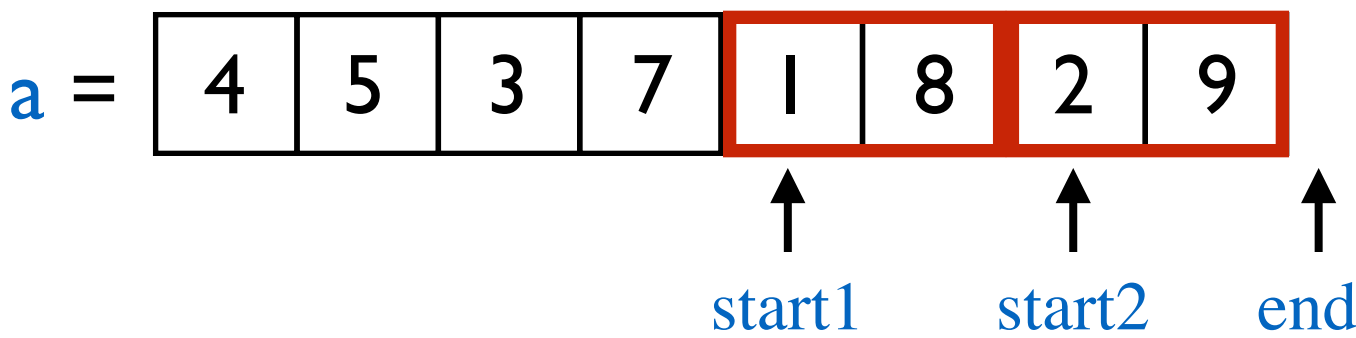


merge(a, 4, 6, 8)

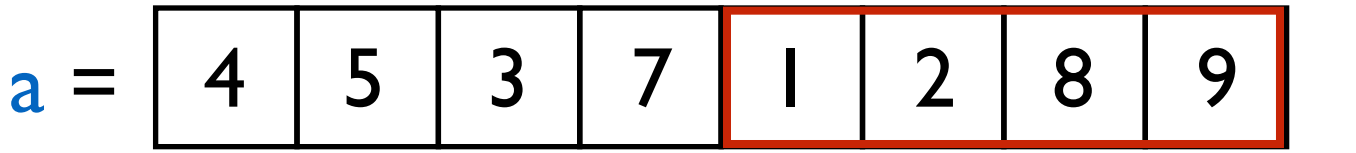


In-place Merge Code

```
def merge(a, start1, start2, end):
    (index1, index2) = (start1, start2)
    length = end - start1
    aux = [None] * length
    for i in range(length):
        if (index1 == start2):
            aux[i] = a[index2]
            index2 += 1
        elif (index2 == end):
            aux[i] = a[index1]
            index1 += 1
        elif (a[index1] < a[index2]):
            aux[i] = a[index1]
            index1 += 1
        else:
            aux[i] = a[index2]
            index2 += 1
    for i in range(start1, end):
        a[i] = aux[i - start1]
```



merge(a, 4, 6, 8)



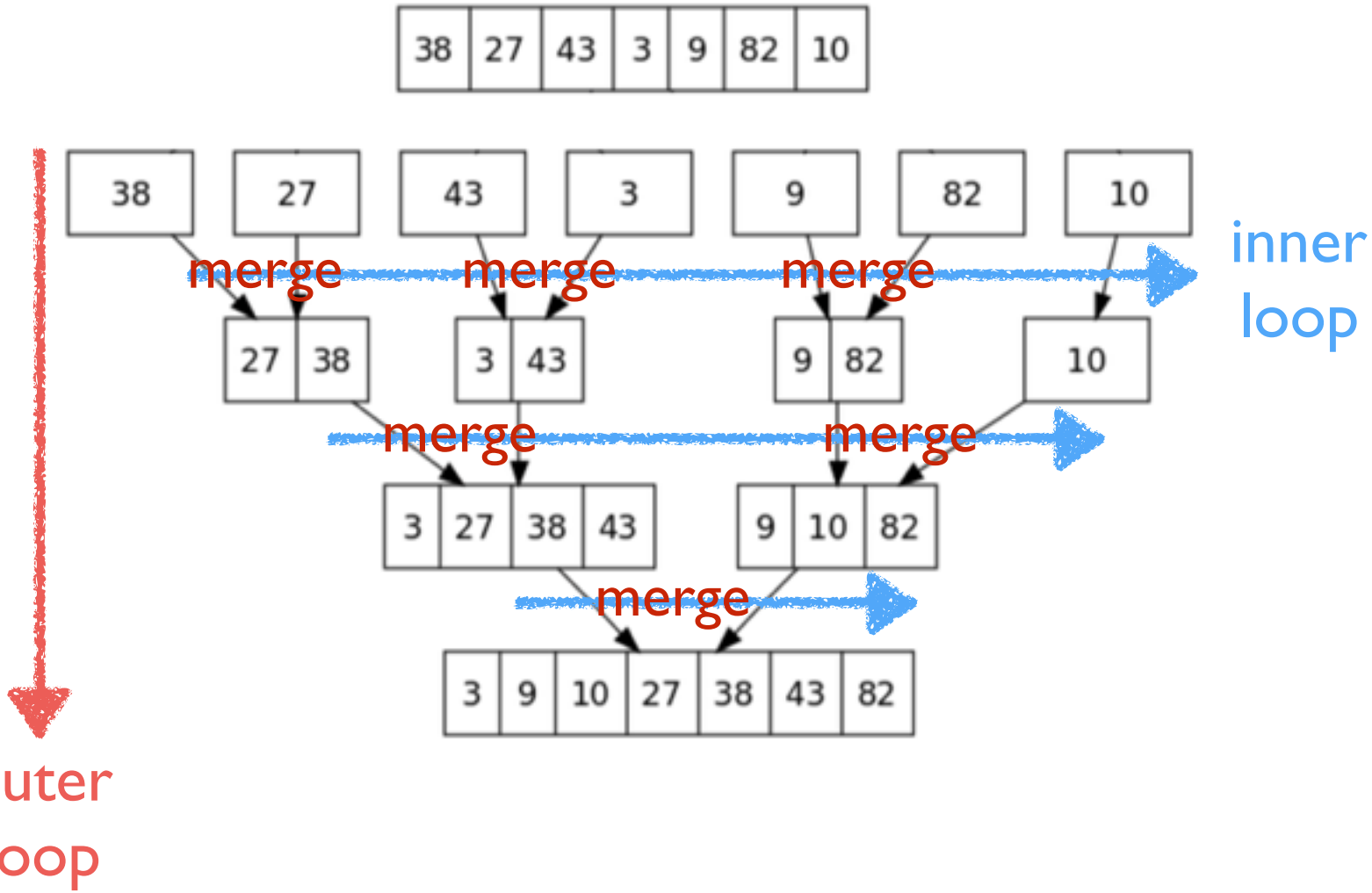
Merge Sort Code

step = 1

step = 2

step = 4

step = 8



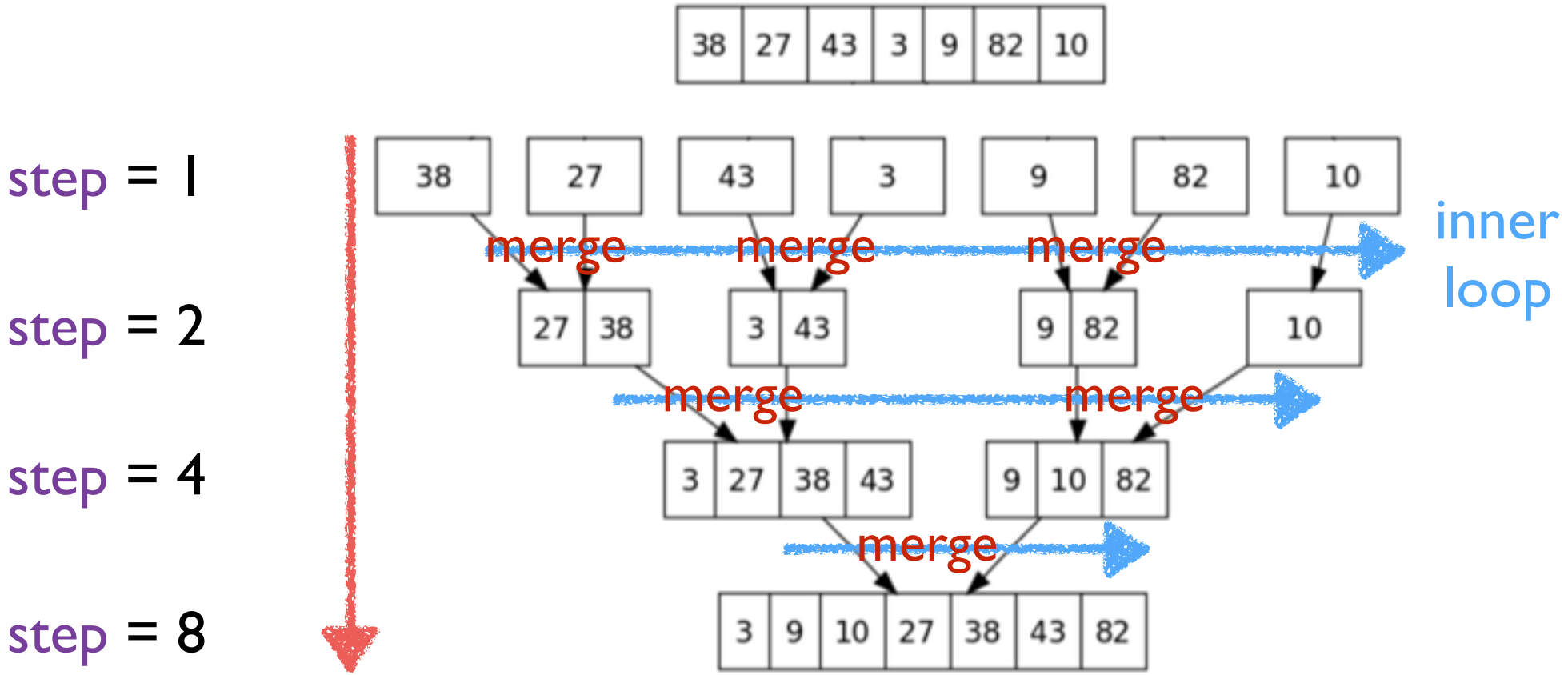
step doubles in each iteration of outer loop

step corresponds to the length of sublists being considered.

Merge Sort Code

```
def mergeSort(a):
    N = len(a)
    step = 1
    while (step < N):
        # Inner loop will go here
        # It will repeatedly call merge for the corresponding step


        step *= 2
```

Merge Sort Code

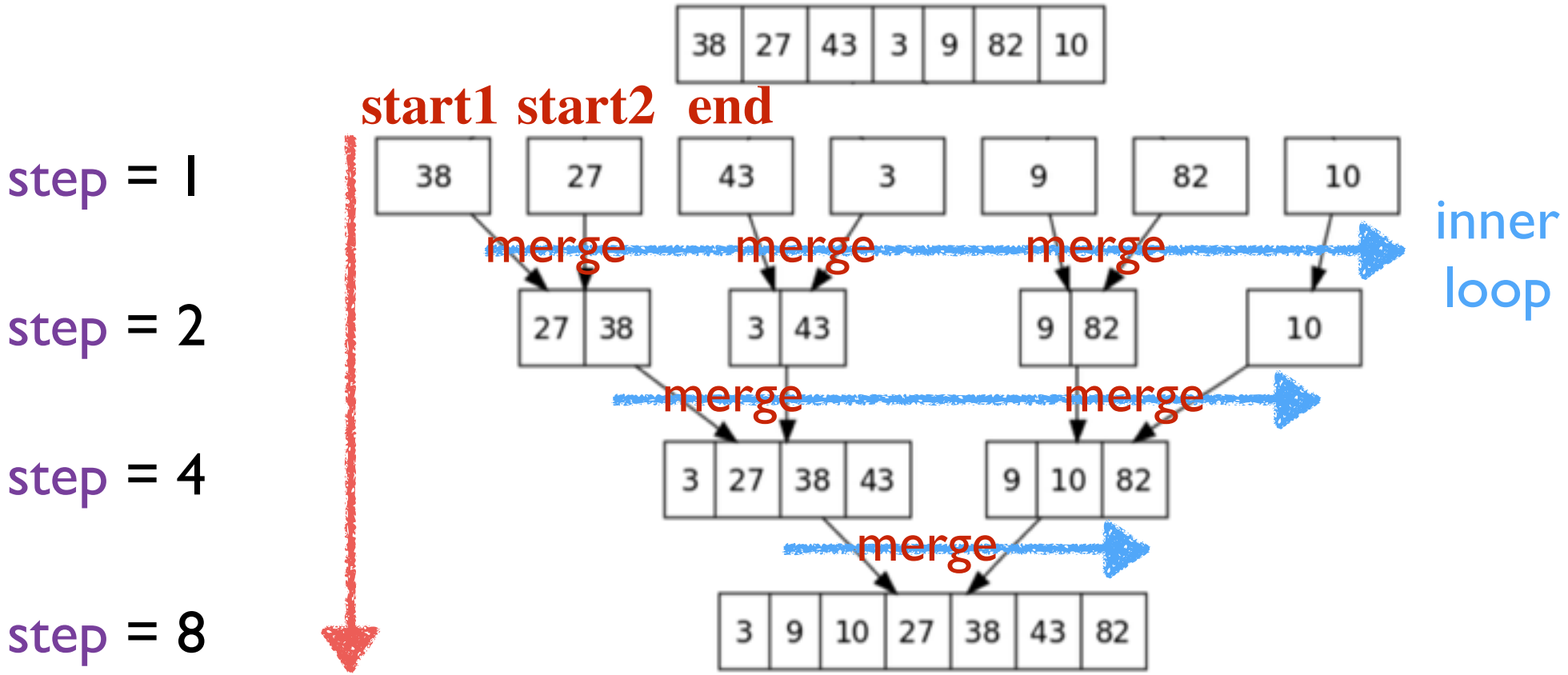



```
def merge(a, start1, start2, end):
    …
```

inner loop:

```
for bla in range(bla, bla, bla):
    # set appropriate values for
    # start1, start2, end
    merge(a, start1, start2, end)
```

Merge Sort Code

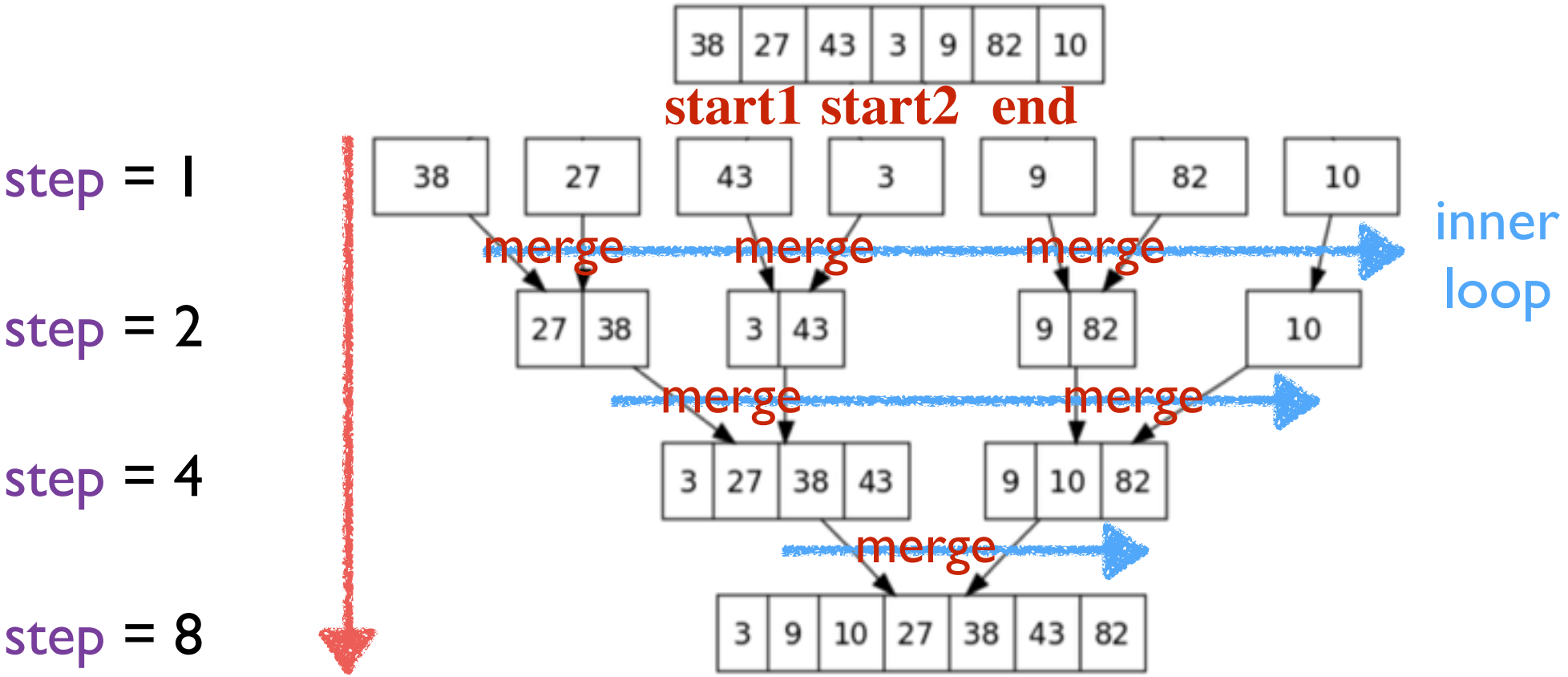



```
def merge(a, start1, start2, end):
    …
```

inner loop:

```
for bla in range(bla, bla, bla):
    # set appropriate values for
    # start1, start2, end
    merge(a, start1, start2, end)
```

Merge Sort Code

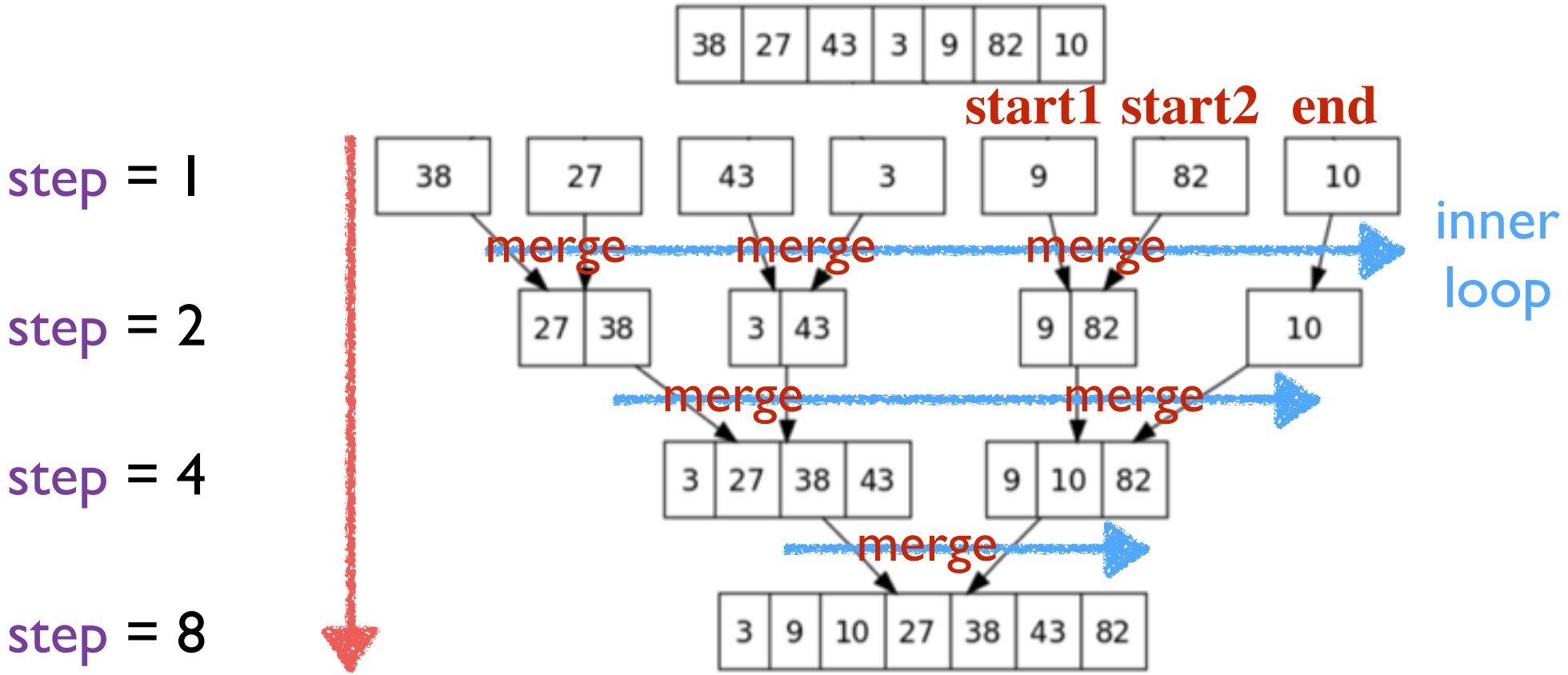



```
def merge(a, start1, start2, end):
    …
```

inner loop:

```
for bla in range(bla, bla, bla):
    # set appropriate values for
    # start1, start2, end
    merge(a, start1, start2, end)
```

Merge Sort Code

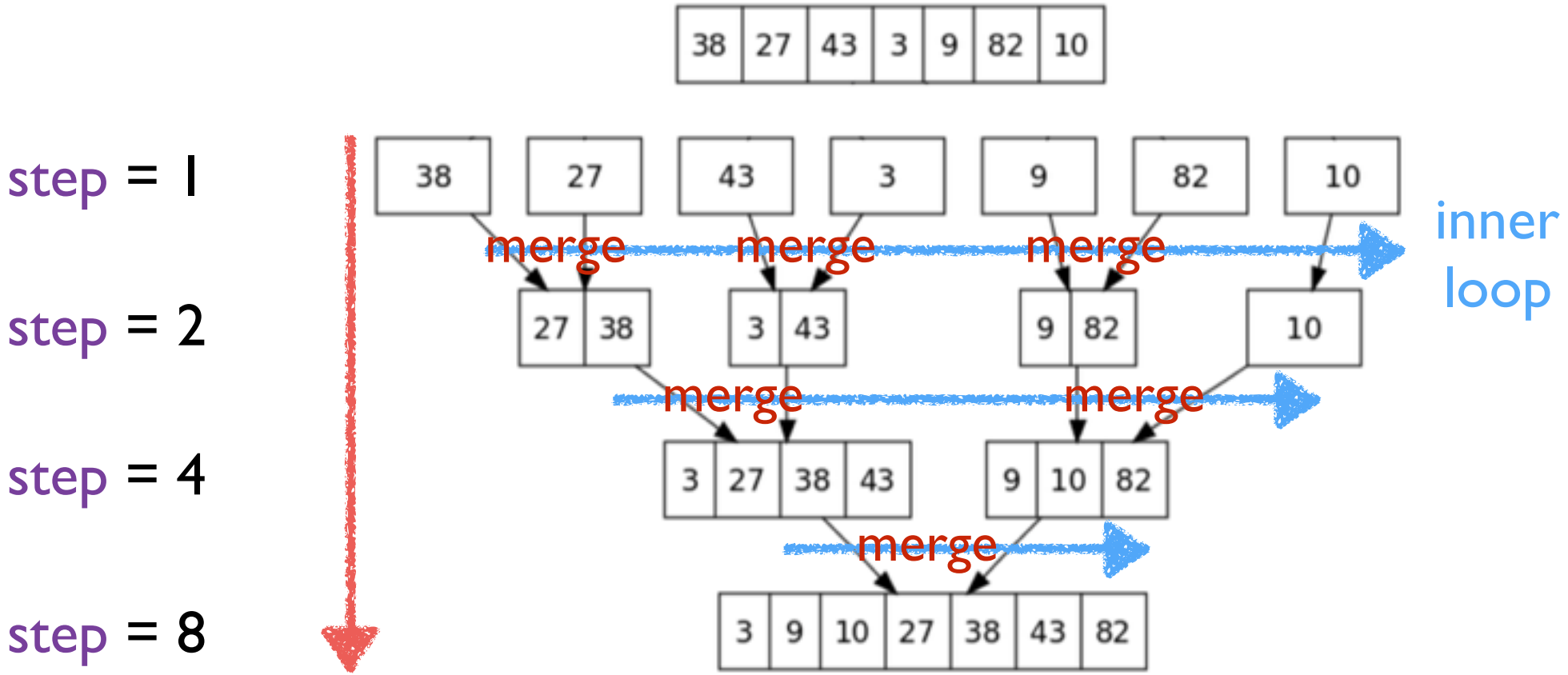



```
def merge(a, start1, start2, end):
    …
```

inner loop:

```
for bla in range(bla, bla, bla):
    # set appropriate values for
    # start1, start2, end
    merge(a, start1, start2, end)
```

Merge Sort Code

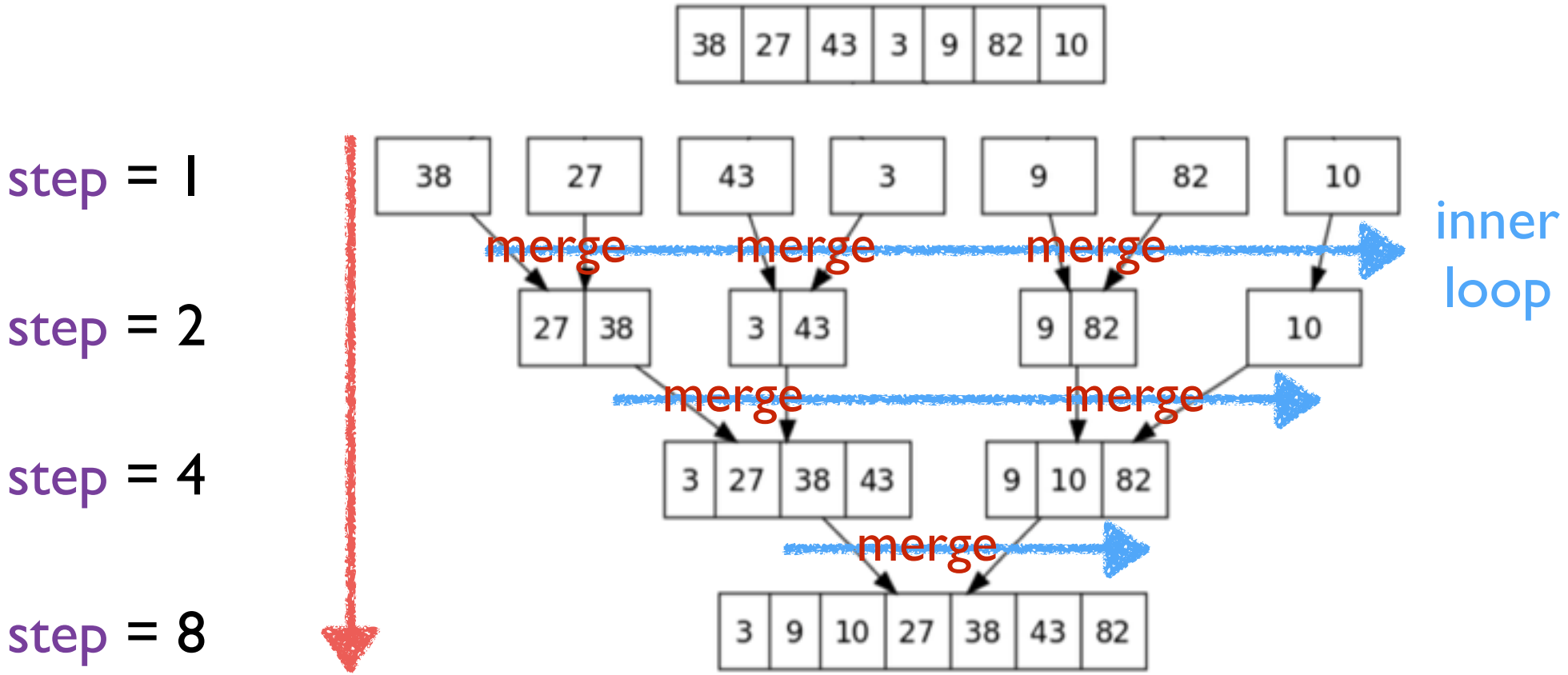



```
def merge(a, start1, start2, end):
    …
```

inner loop:

```
for start1 in range(bla, bla, bla):
    # set appropriate values for
    # start2, end
    merge(a, start1, start2, end)
```

Merge Sort Code

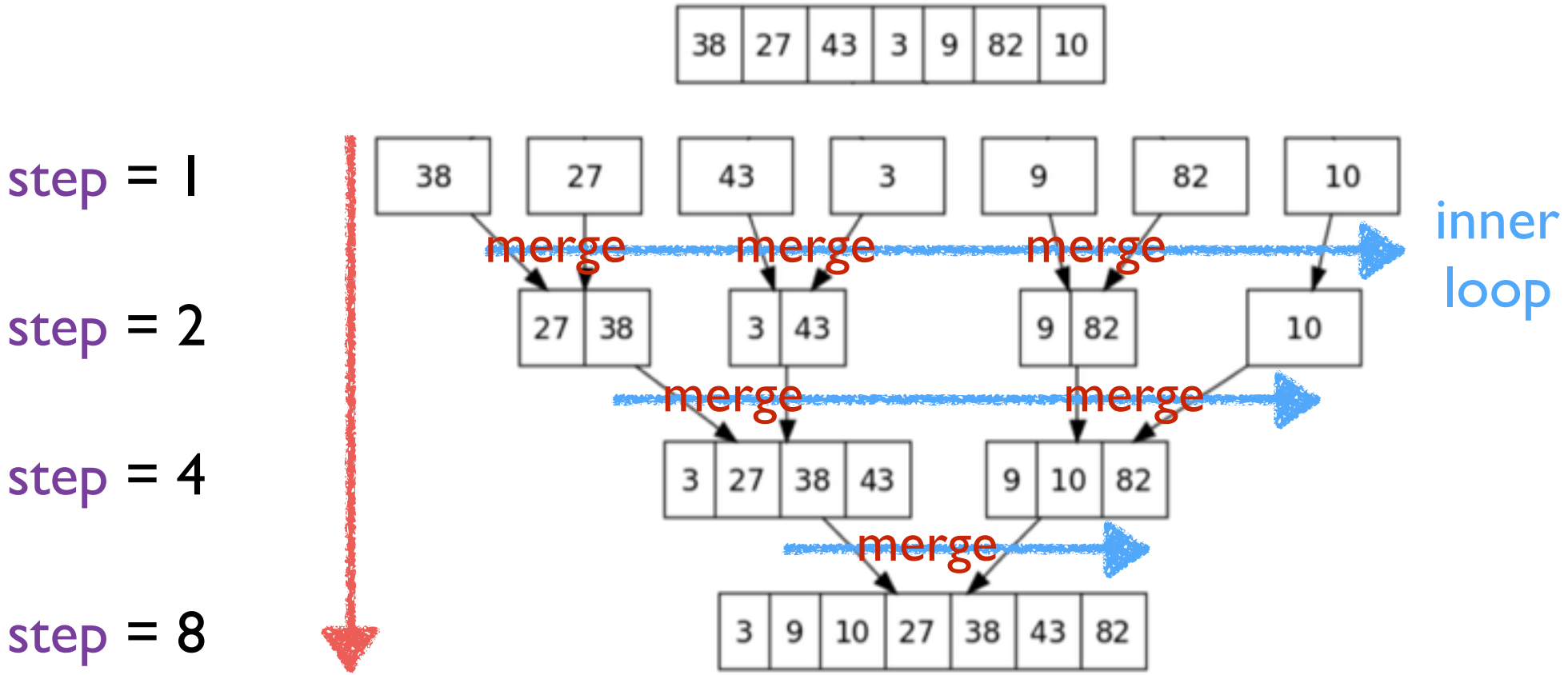



```
def merge(a, start1, start2, end):
    …
```

inner loop:

```
for start1 in range(0, N, 2*step):
    # set appropriate values for
    # start2, end
    merge(a, start1, start2, end)
```

Merge Sort Code

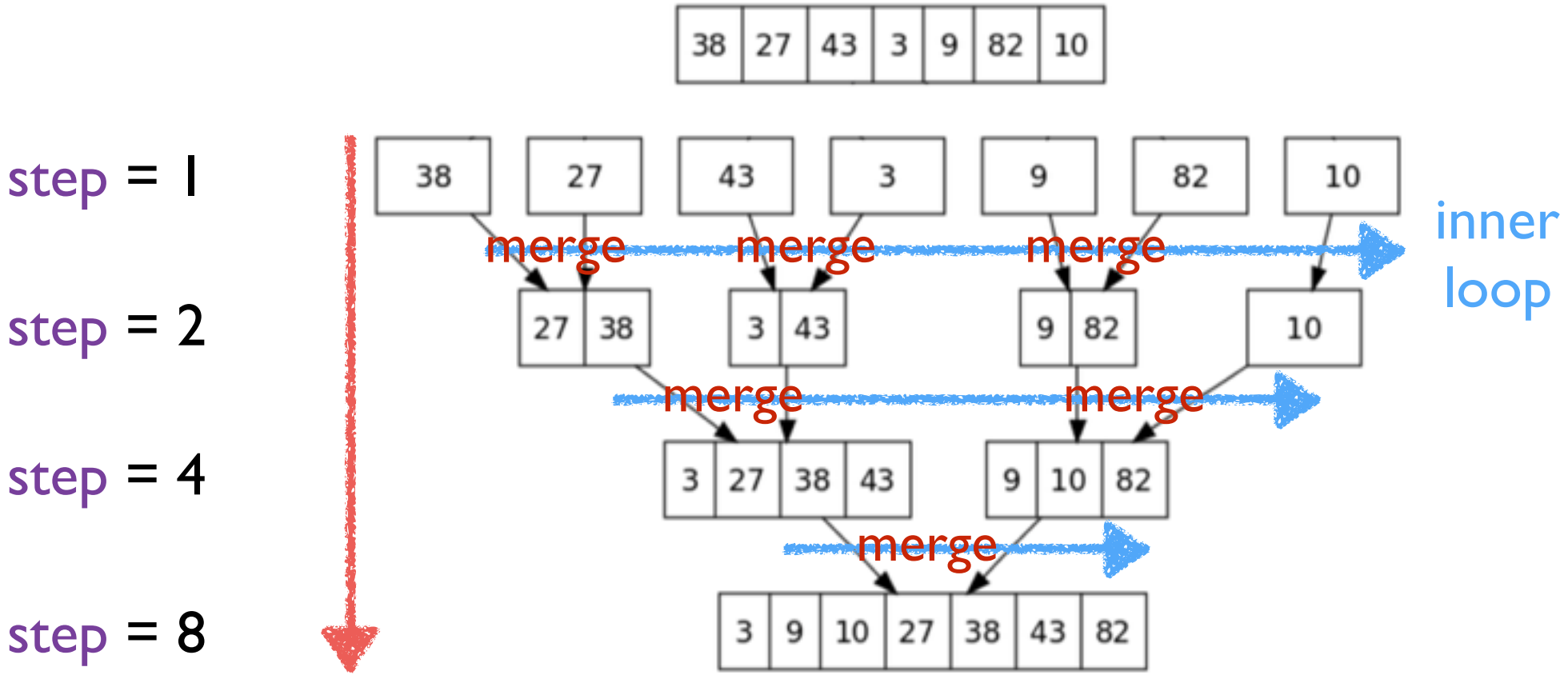



```
def merge(a, start1, start2, end):
    …
```

inner loop:

```
for start1 in range(0, N, 2*step):
    start2 = start1 + step

    merge(a, start1, start2, end)
```

Merge Sort Code

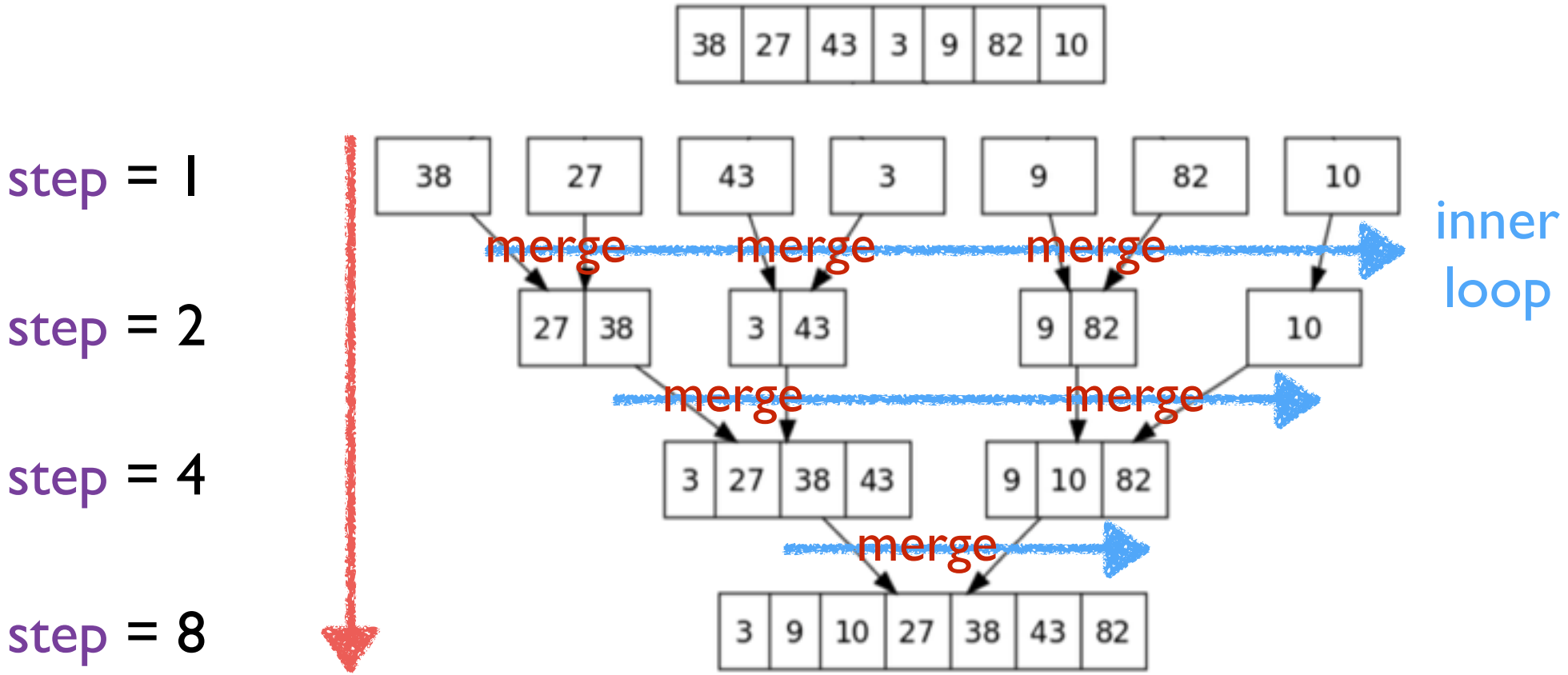



```
def merge(a, start1, start2, end):
    …
```

inner loop:

```
for start1 in range(0, N, 2*step):
    start2 = start1 + step
    end = start1 + 2*step
    merge(a, start1, start2, end)
```

Merge Sort Code




```
def merge(a, start1, start2, end):
    …
```

inner loop:

```
for start1 in range(0, N, 2*step):
    start2 = min(start1 + step, N)
    end = min(start1 + 2*step, N)
    merge(a, start1, start2, end)
```

Merge Sort Code

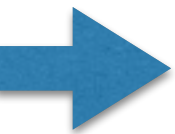
```
def mergeSort(a):
    N = len(a)
    step = 1
    while (step < N):
        # Inner loop will go here
        # It will repeatedly call merge for the corresponding step


        step *= 2
```

Merge Sort Code

```
def mergeSort(a):
    N = len(a)
    step = 1
    while (step < N):
        for start1 in range(0, N, 2*step):
            start2 = min(start1 + step, N)
            end = min(start1 + 2*step, N)
            merge(a, start1, start2, end)
        step *= 2
```

The Plan

> Merge sort



> Measuring running time when the input is not a list

> Efficient data structures: sets and dictionaries

String inputs

Write a function isSubString(a, b) that checks whether the string b is a substring of the string a.

```
def isSubString(a, b):
    for i in range(len(a) - len(b) + 1):
        substring = True
        for j in range(len(b)):
            if(a[i+j] != b[j]):
                substring = False
                break
        if(substring):
            return True
    return False
```

Input length: N = len(a)+len(b)

Running time: $O(N^2)$

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

~ $\log_{10} n$

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

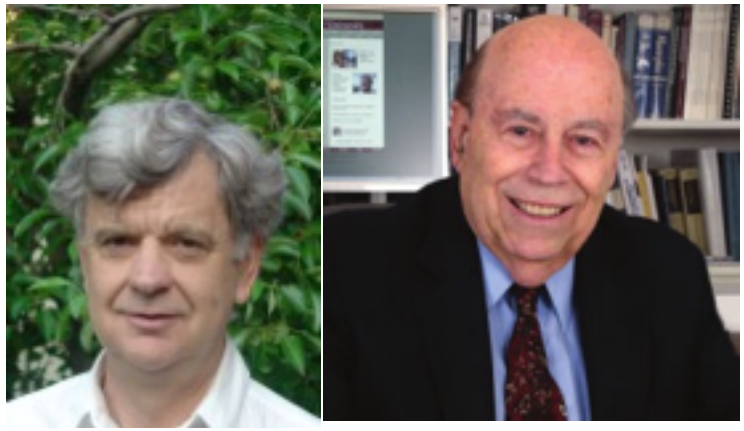
However, best known implementation is ~ $O(N^6)$ time.

Not feasible when $N = 2048$.

isPrime

So that's not what we use in practice.

Everyone uses the Miller-Rabin algorithm (1975).



CMU
Professor

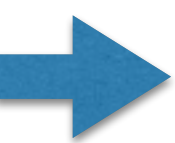
The running time is ~ $O(N^2)$.

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 not a list



> 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( ) $O(N)$

…

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

Motivating example: A time/space tradeoff

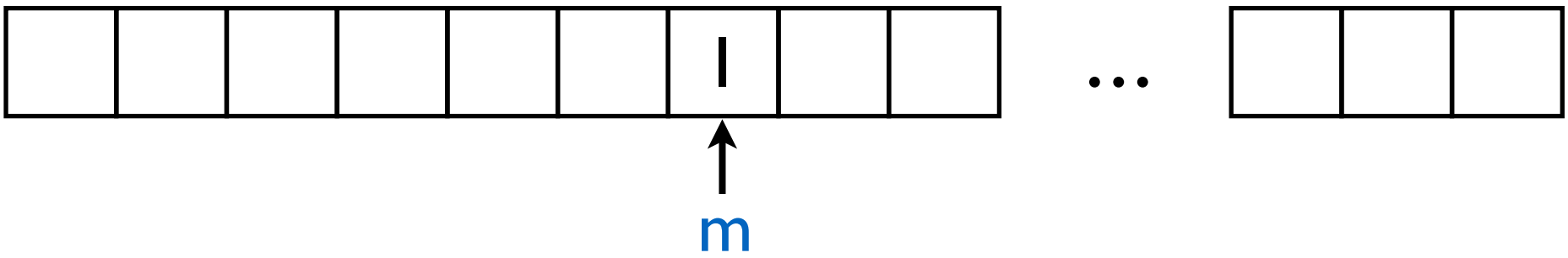
Sorting a list of numbers.

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

Solution:

Create a list of size 1 million.

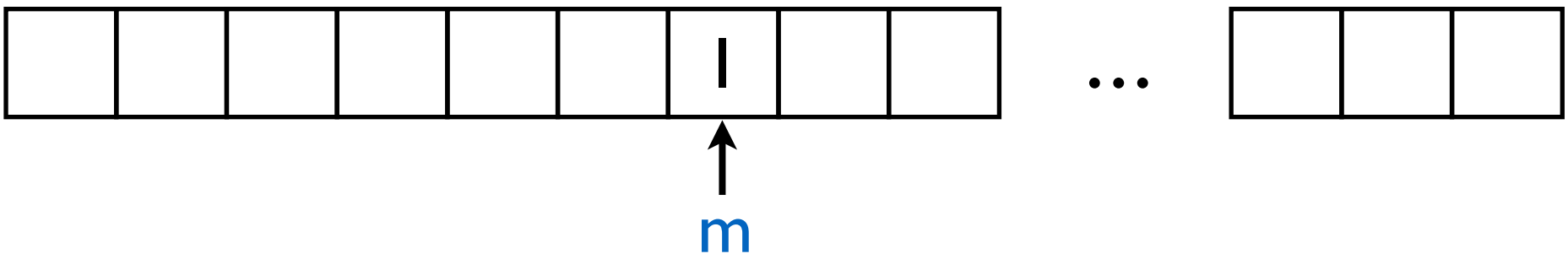
Put number m at index m.



What is the running time for searching for an element?

$O(1)$

Motivating example: A time/space tradeoff



The sweet idea:

Connecting value to index.

Motivating example: A time/space tradeoff

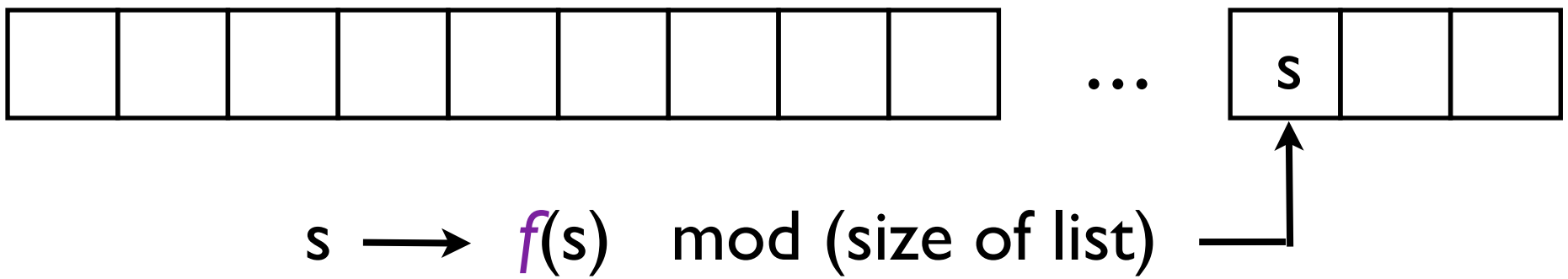
Questions

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?



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

Pick a function *f* that maps strings to numbers.

Store s at index *f*(s) mod (size of list)

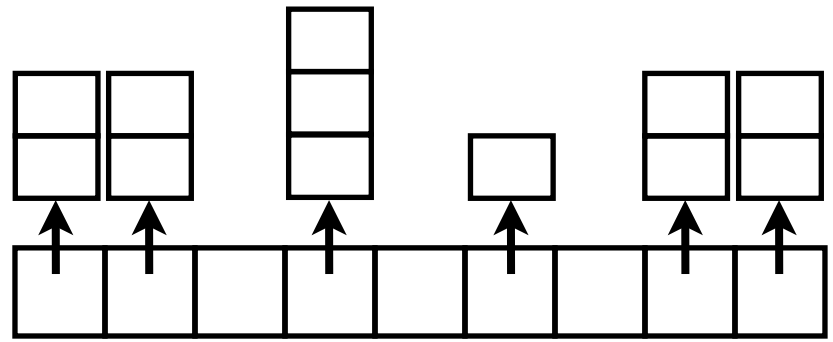
f is called a **hash function**.

Extending the sweet idea

Potential Problems

Collision: two strings map to the same index

List fills up



Fixes

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

Lists:

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

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.)

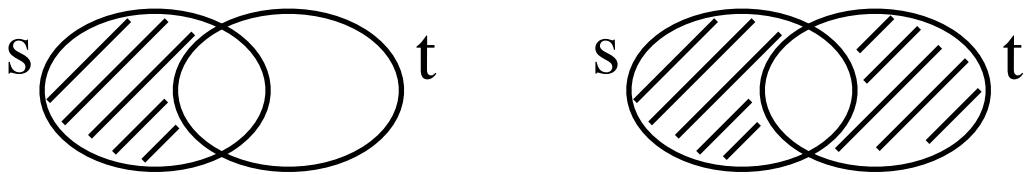
```
s = set("hello")                    # {'e', 'h', 'l', 'o'}
s = set((2, 4, 8))                  # {8, 2, 4}
s = set(range(10))                  # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

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):

Can raise "KeyError"

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)

Set methods

Shortcuts for 2-set methods.

s.issubset(t)	s <= t
s.issuperset(t)	s >= t
s.union(t)	s | t
s.intersection(t)	s & t
s.difference(t)	s - t
s.symmetric_difference(t)	s ^ t
s.update(t)	*s* |= *t*
s.intersection_update(t)	*s* &= *t*
s.difference_update(t)	*s* -= *t*
s.symmetric_difference_update(t)	*s* ^= *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)

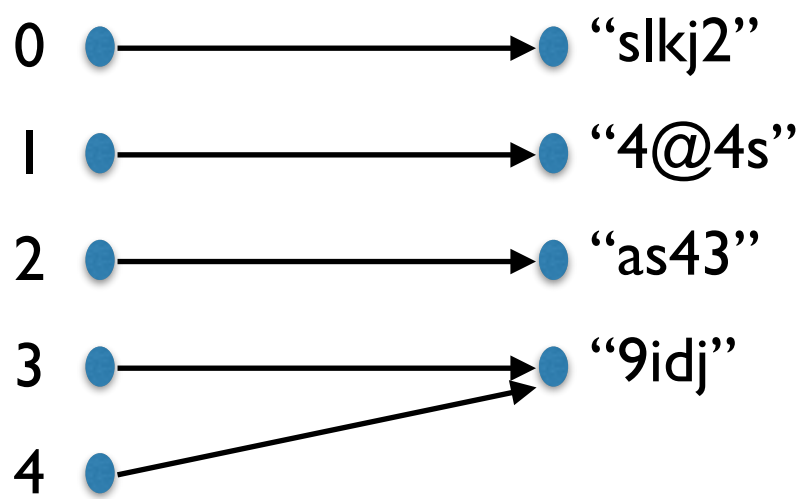
Dictionaries:

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

Dictionaries / maps

a = list()
a[0] = “slkj2”
a[1] = “4@4s”
a[2] = “as43”
a[3] = “9idj”
a[4] = “9idj”

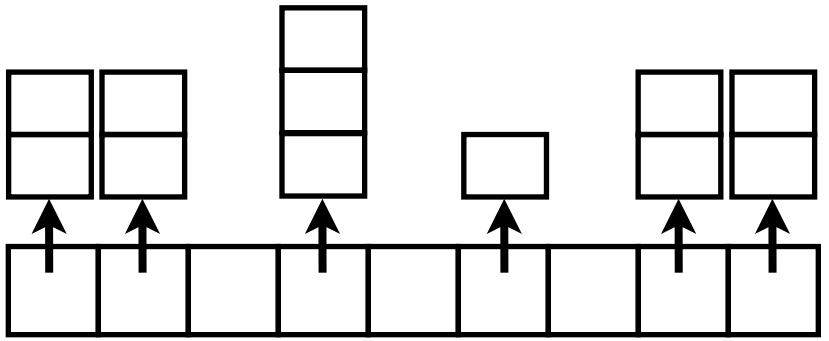
List



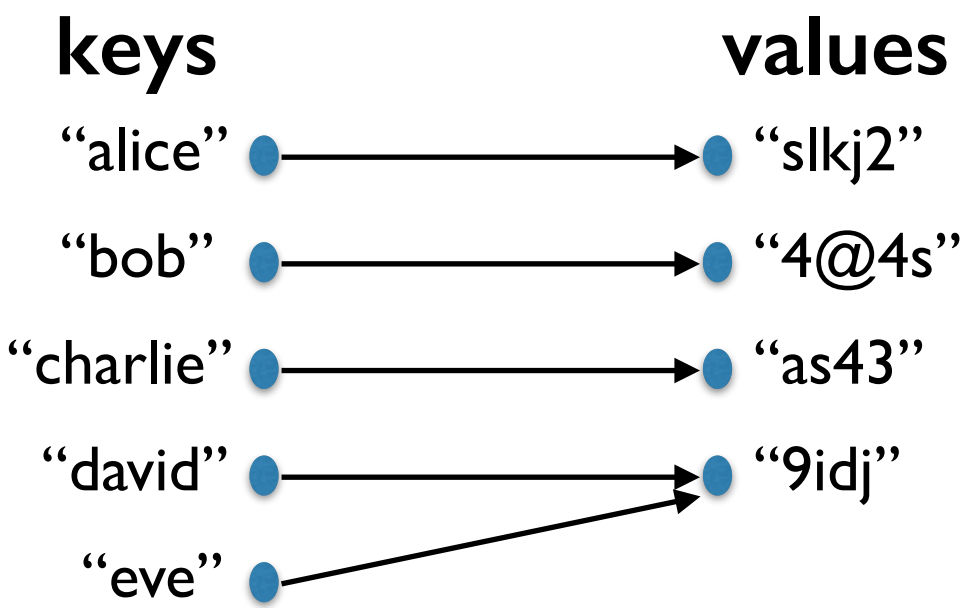
Dictionaries / maps

```
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”}

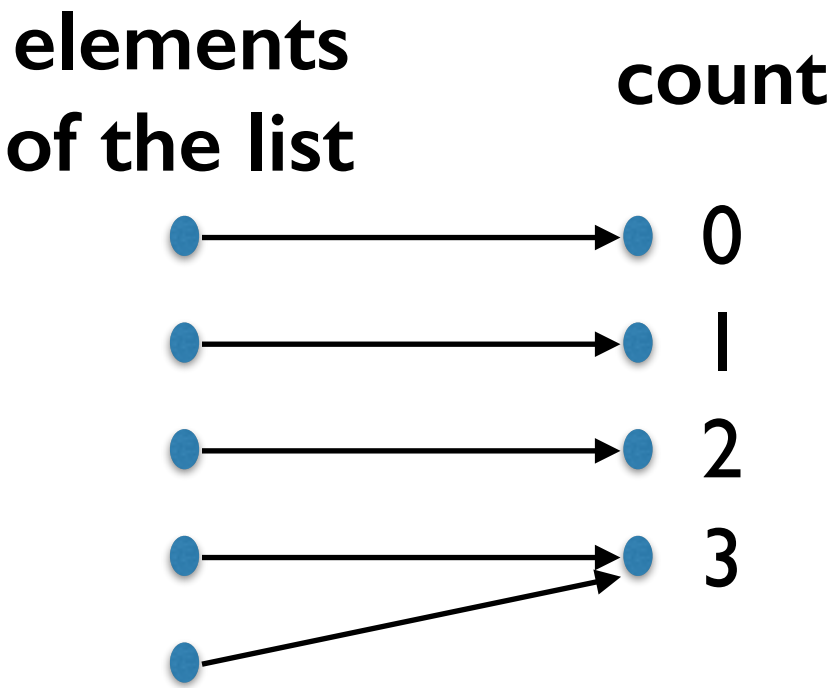
```
for key in users:
    print(key, d[key])
```

```
print(users[“rudina”])          Error

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

print(users.get(“rudina”, 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.