Lecture 18
Sorting

Slide sources: Leiserson et. al., Introduction to Algorithms, Goodrich et al., Data Structures and Algorithms in C++.

Why sorting?

- ~90% of the times computers just sort
- element uniqueness
- databases
- computer graphics

Sorting Algorithm Characterization

- Memory: In-place
- Complexity: $O(n^2)$ and $O(n \log n)$
- Original order: stable/unstable

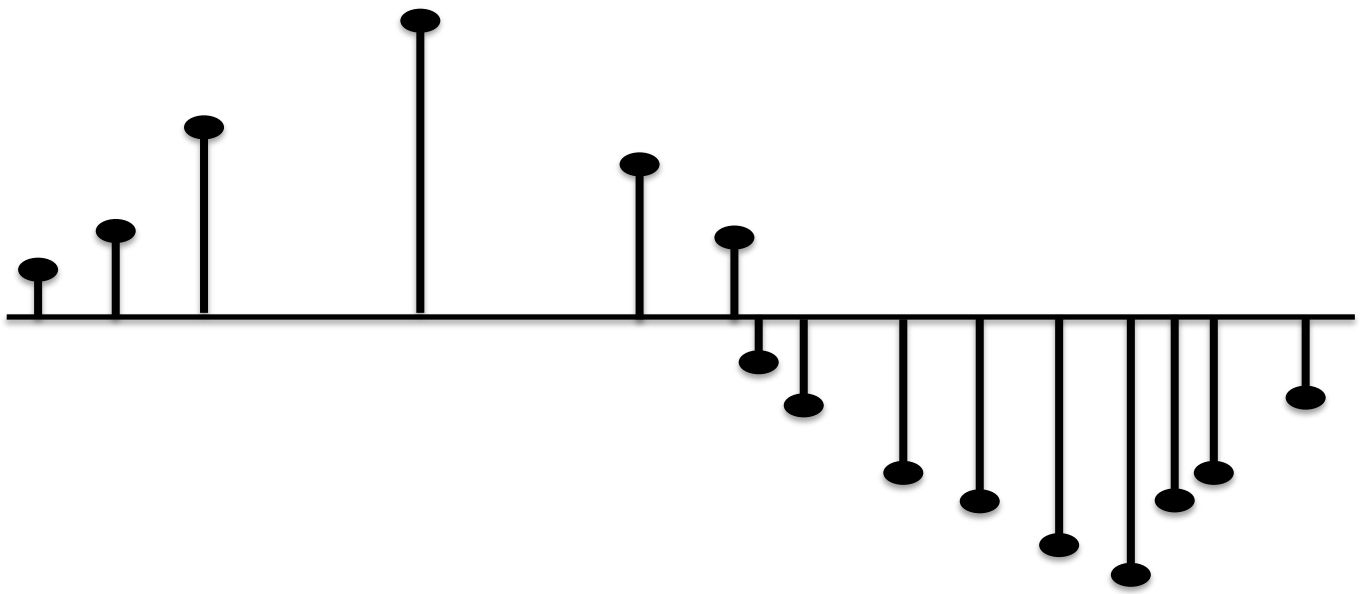
Is insertion sort stable?

Is heap sort stable?

Is selection sort stable?

What about in-place selection sort?

A sine wave was randomly sampled n times during one cycle, find max and min of the samples!



Merge-Sort

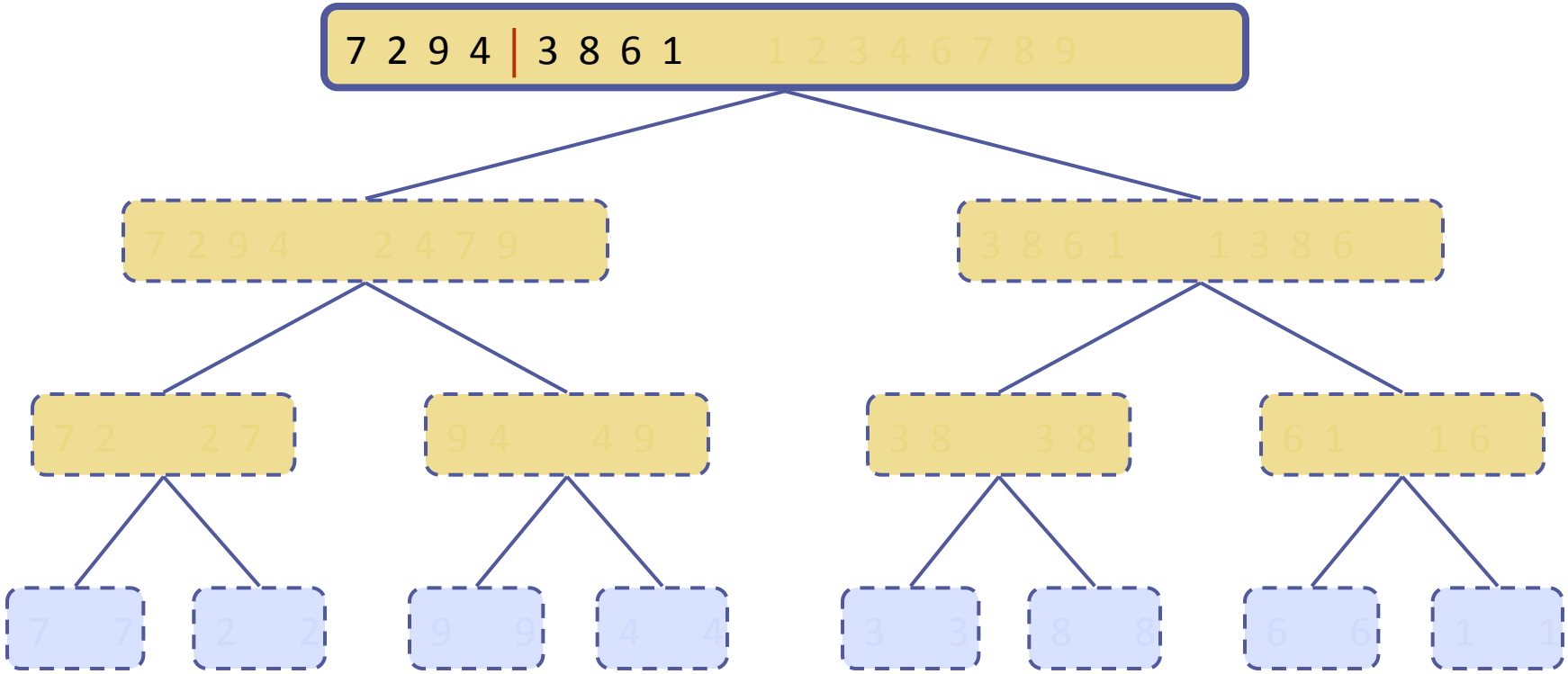
- Divide
- Recur
- Conquer

```
Algorithm mergeSort(S, C)
    Input sequence S with n
        elements, comparator C
    Output sequence S sorted
        according to C
    if S.size() > 1
        (S1, S2) ← partition(S, n/2)
        mergeSort(S1, C)
        mergeSort(S2, C)
    S ← merge(S1, S2)
```

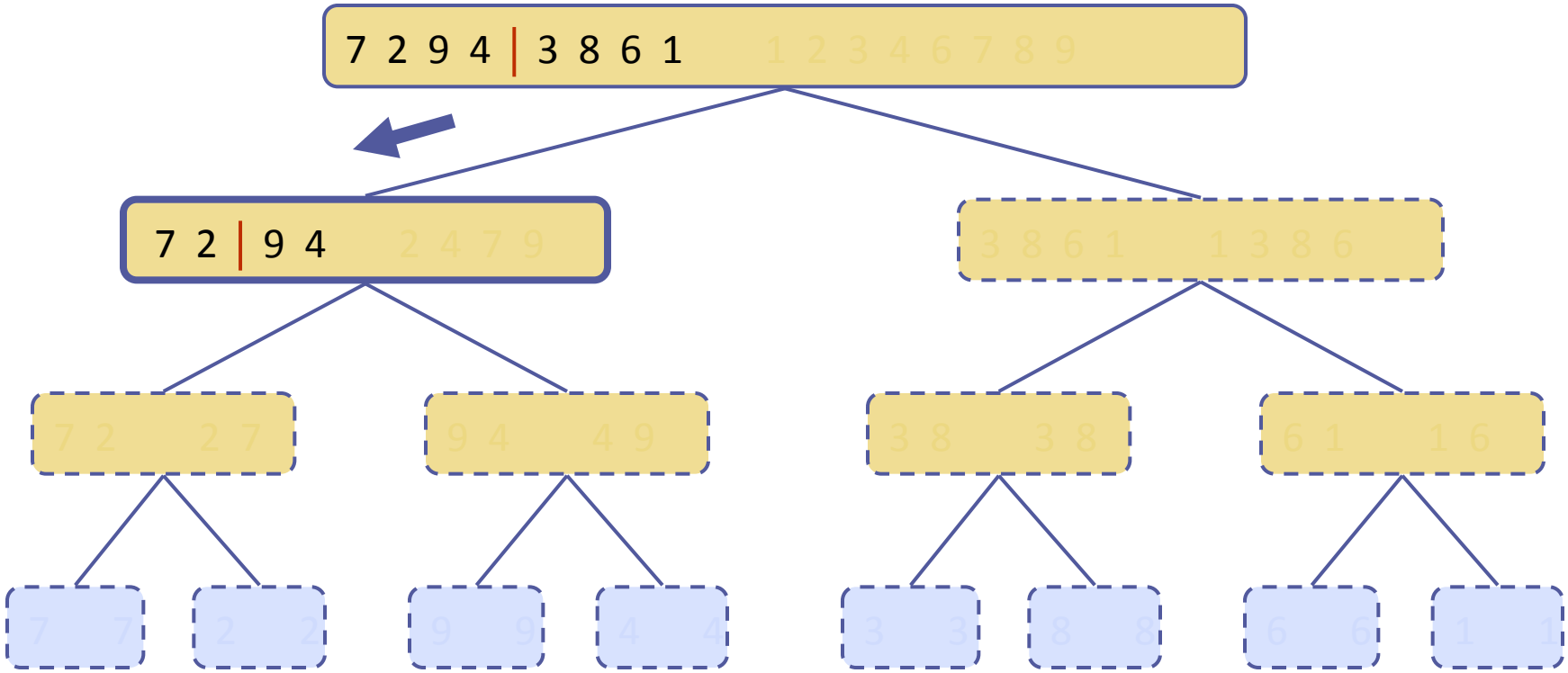
```
Algorithm merge(A, B)
    Input sequences A and B with
        n/2 elements each
    Output sorted sequence of A ∪ B

    S ← empty sequence
    while !A.empty()  ∧ !B.empty()
        if A.front() < B.front()
            S.addBack(A.front()); A.eraseFront();
        else
            S.addBack(B.front()); B.eraseFront();
    while ¬A.empty()
        S.addBack(A.front()); A.eraseFront();
    while ¬B.empty()
        S.addBack(B.front()); B.eraseFront();
    return S
```

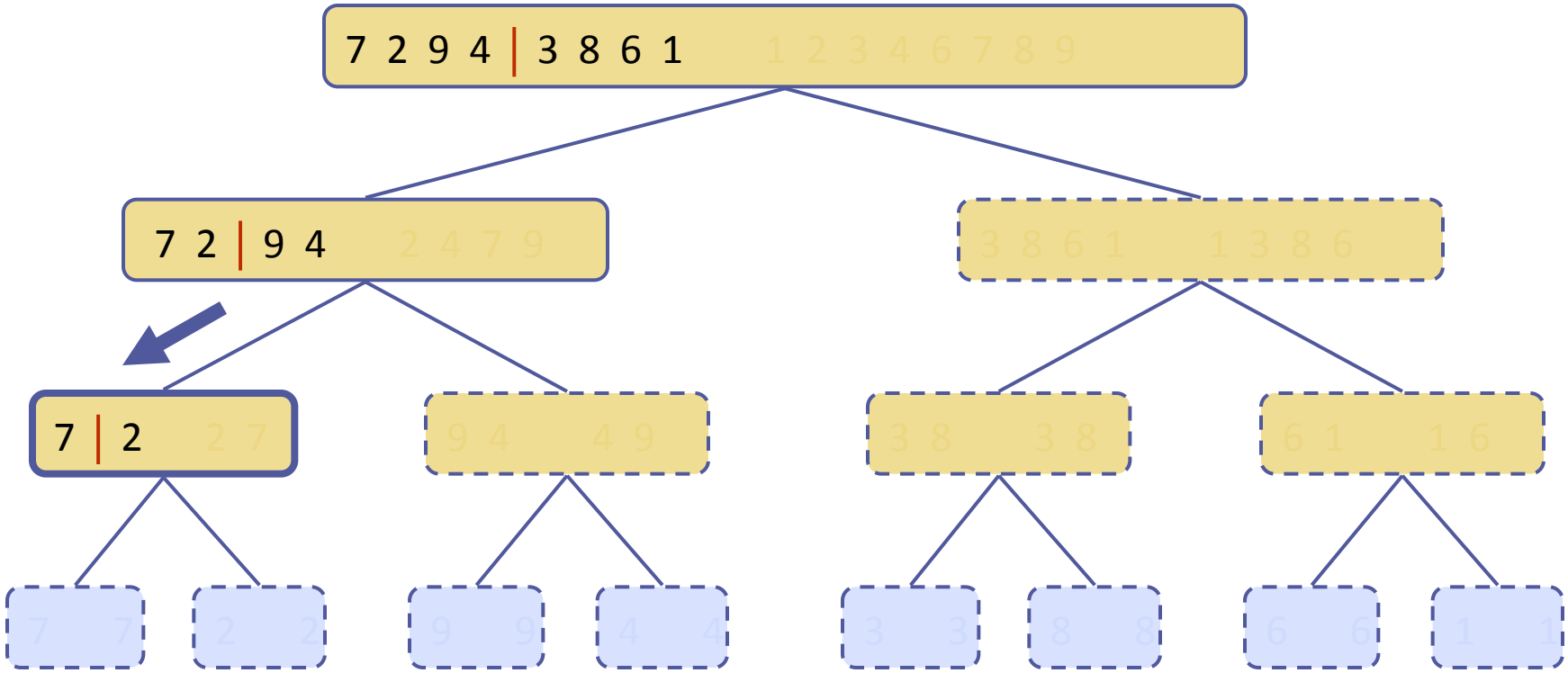
Example


7 2 9 4 | 3 8 6 1 1 2 3 4 6 7 8 9
7 2 9 4 2 4 7 9
3 8 6 1 1 3 8 6
7 2 2 7
9 4 4 9
3 8 3 8
6 1 1 6
7 7
2 2
9 9
4 4
3 3
8 8
6 6
1 1

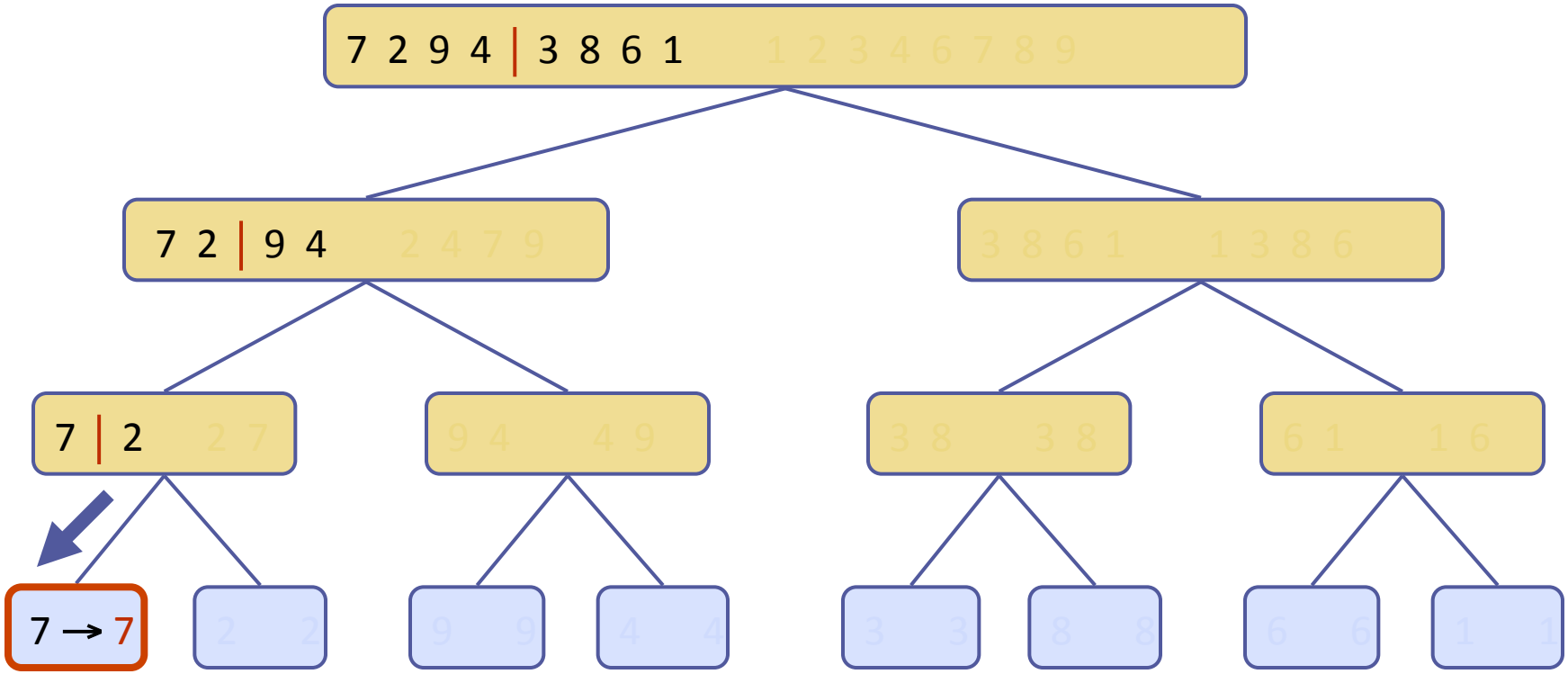
Example


7 2 9 4 | 3 8 6 1
1 2 3 4 6 7 8 9
7 2 | 9 4
2 4 7 9
3 8 6 1
1 3 8 6
7 2
2 7
9 4
4 9
3 8
3 8
6 1
1 6
7
7
2
2
9
9
4
4
3
3
8
8
6
6
1
1

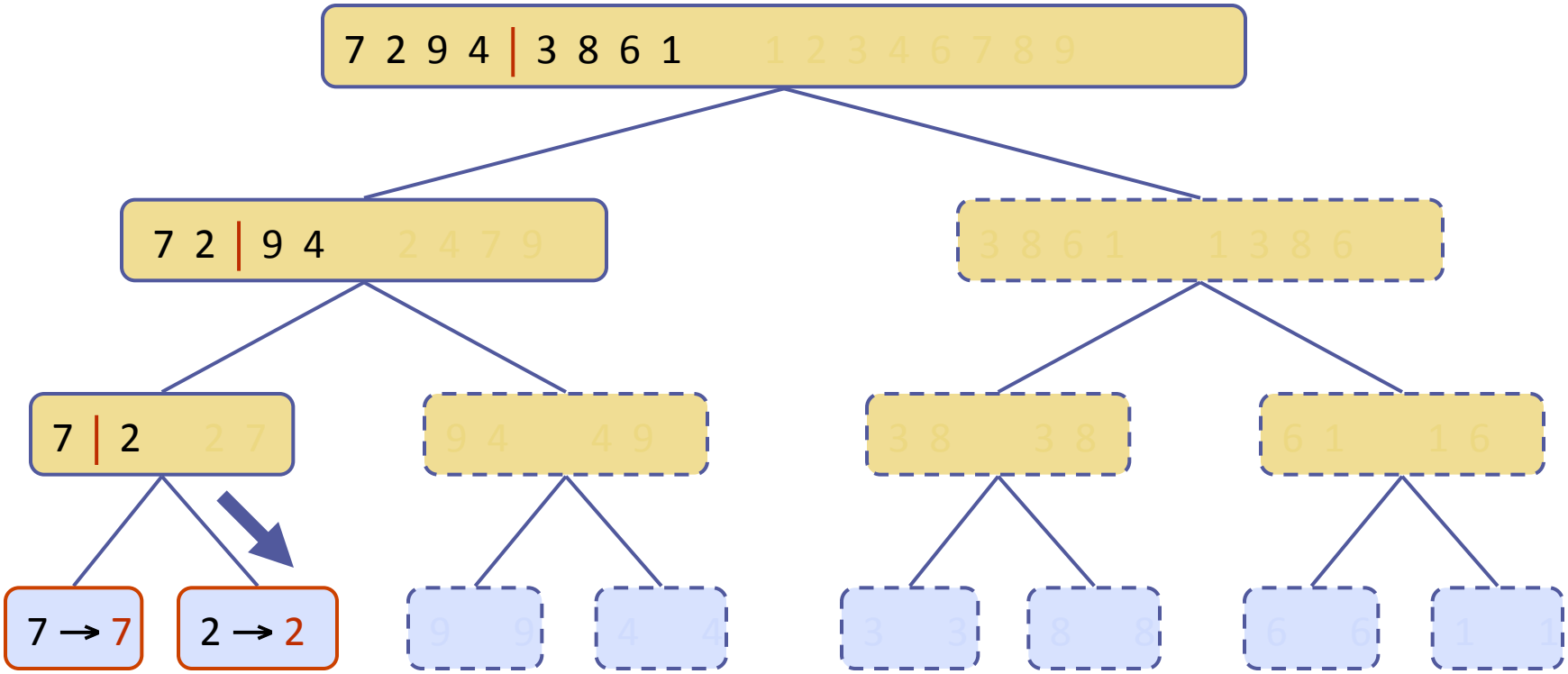
Example


7 2 9 4 | 3 8 6 1 1 2 3 4 6 7 8 9
7 2 | 9 4 2 4 7 9
3 8 6 1 1 3 8 6
7 | 2 2 7
9 4 4 9
3 8 3 8
6 1 1 6
7 7
2 2
9 9
4 4
3 3
8 8
6 6
1 1

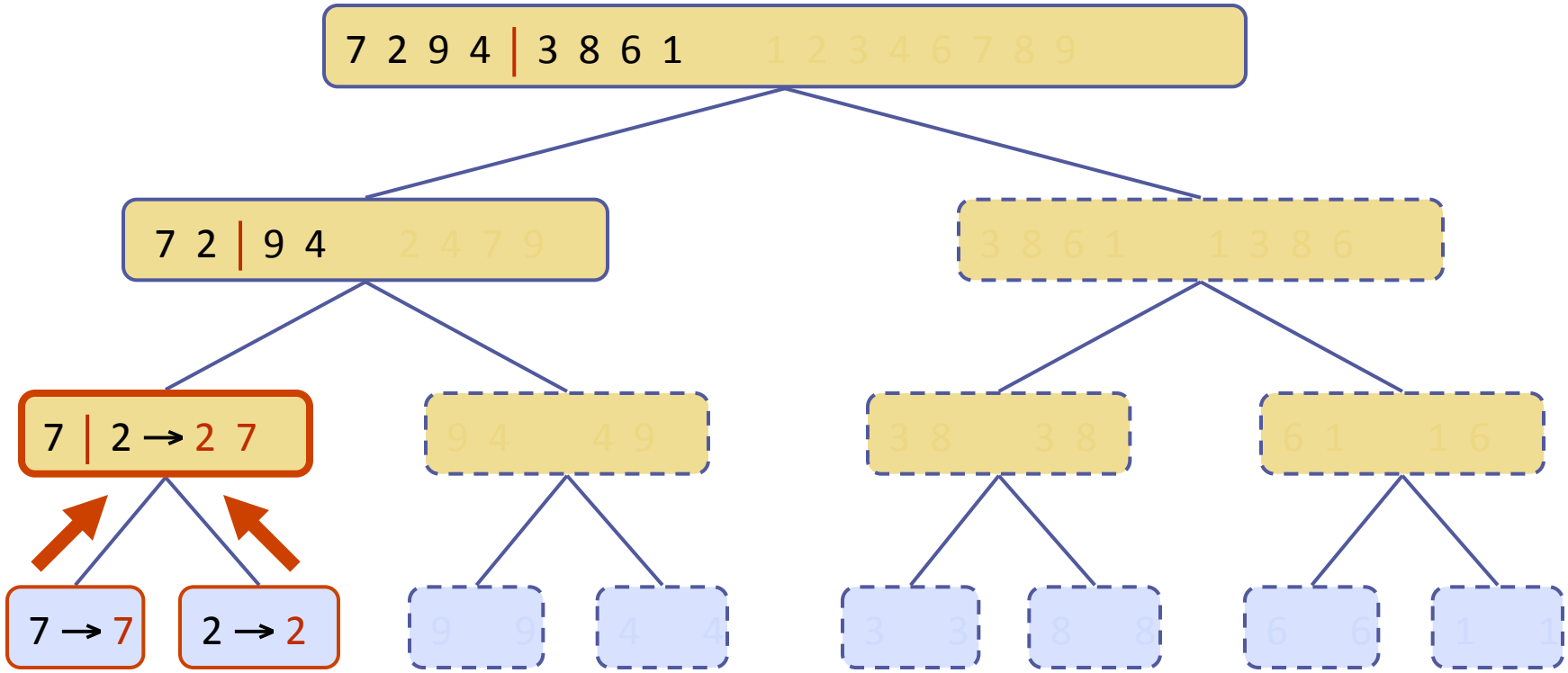
Example


7 2 9 4 | 3 8 6 1
1 2 3 4 6 7 8 9
7 2 | 9 4
2 4 7 9
3 8 6 1
1 3 8 6
7 | 2
2 7
9 4
4 9
3 8
3 8
6 1
1 6
7 → 7
2 2
9 9
4 4
3 3
8 8
6 6
1 1

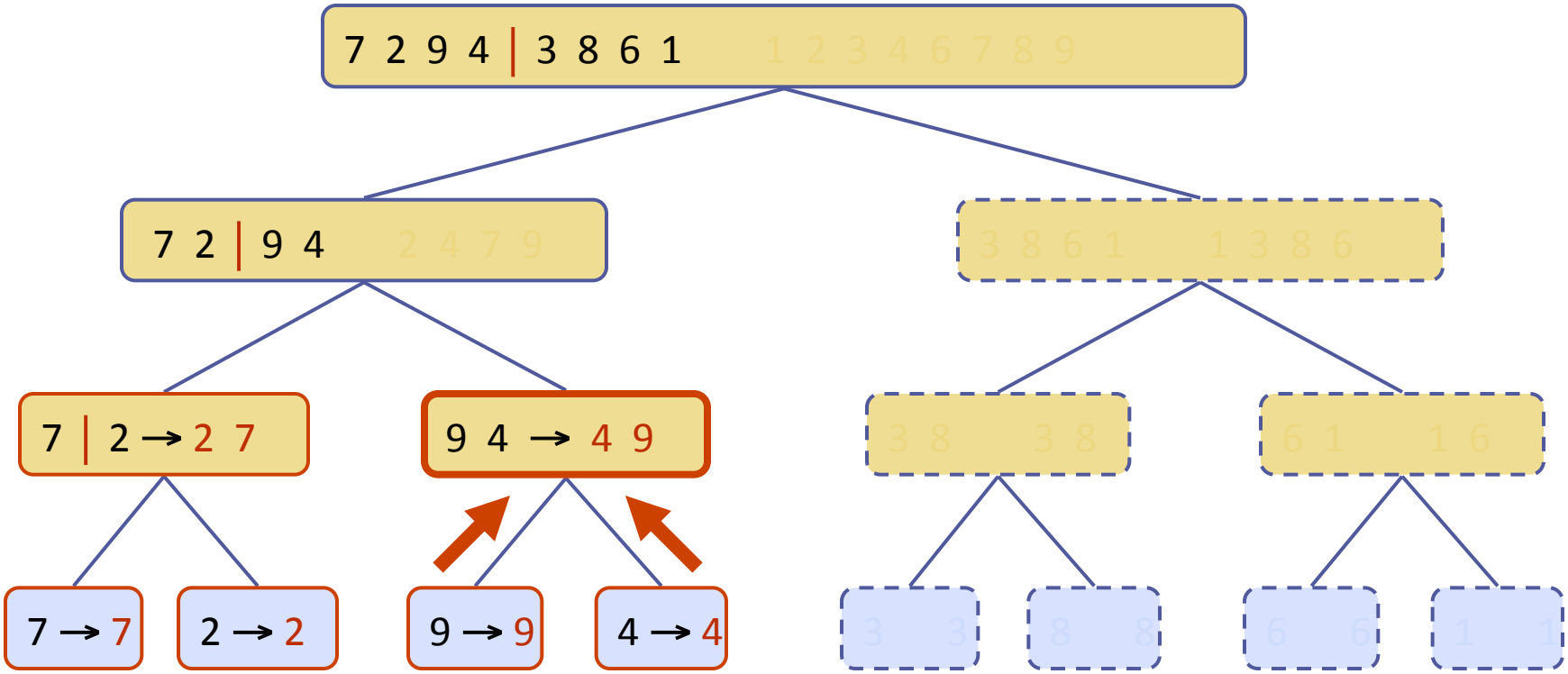
Example


7 2 9 4 | 3 8 6 1
1 2 3 4 6 7 8 9
7 2 | 9 4
2 4 7 9
3 8 6 1
1 3 8 6
7 | 2
2 7
9 4
4 9
3 8
3 8
6 1
1 6
7 → 7
2 → 2
9 9
4 4
3 3
8 8
6 6
1 1

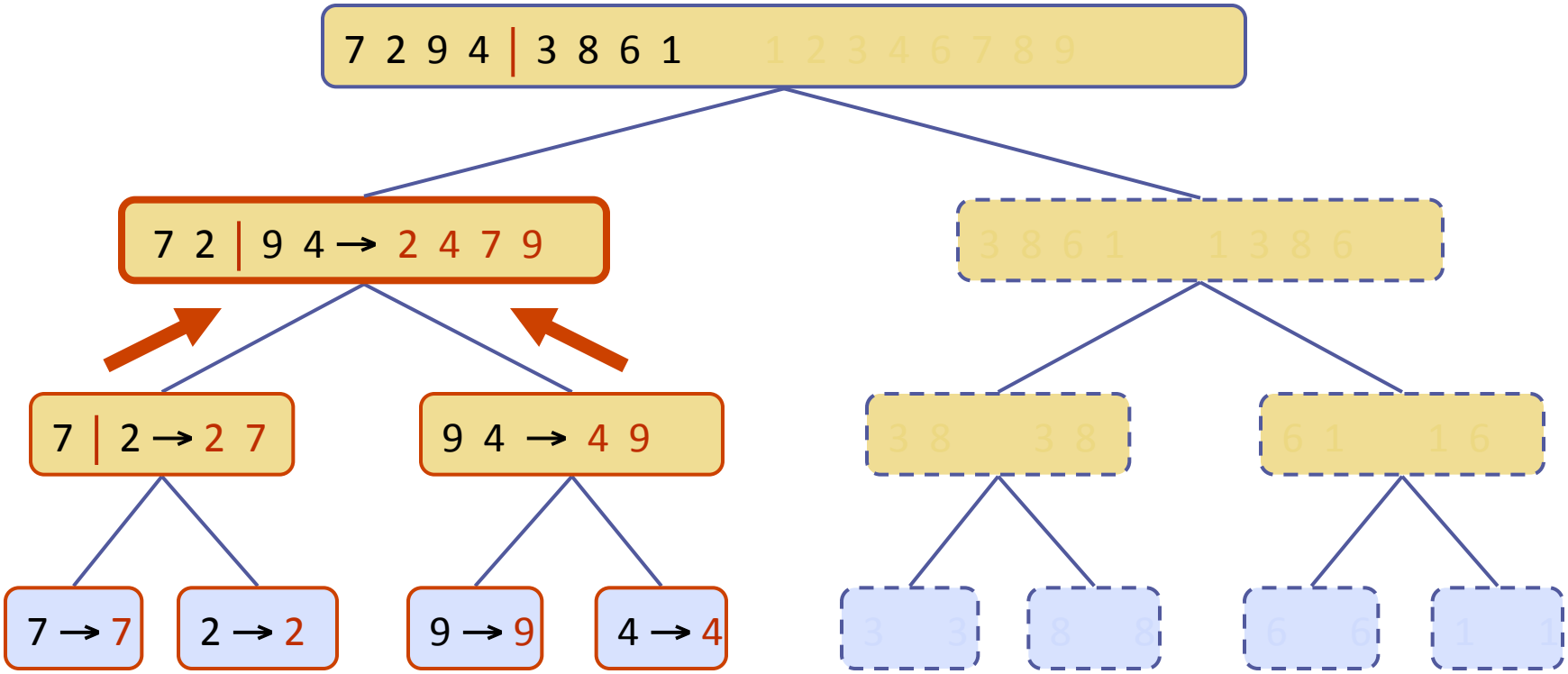
Example


7 2 9 4 | 3 8 6 1
1 2 3 4 6 7 8 9
7 2 | 9 4
2 4 7 9
3 8 6 1
1 3 8 6
7 | 2 → 2 7
9 4
4 9
3 8
3 8
6 1
1 6
7 → 7
2 → 2
9
9
4
4
3
3
8
8
6
6
1
1

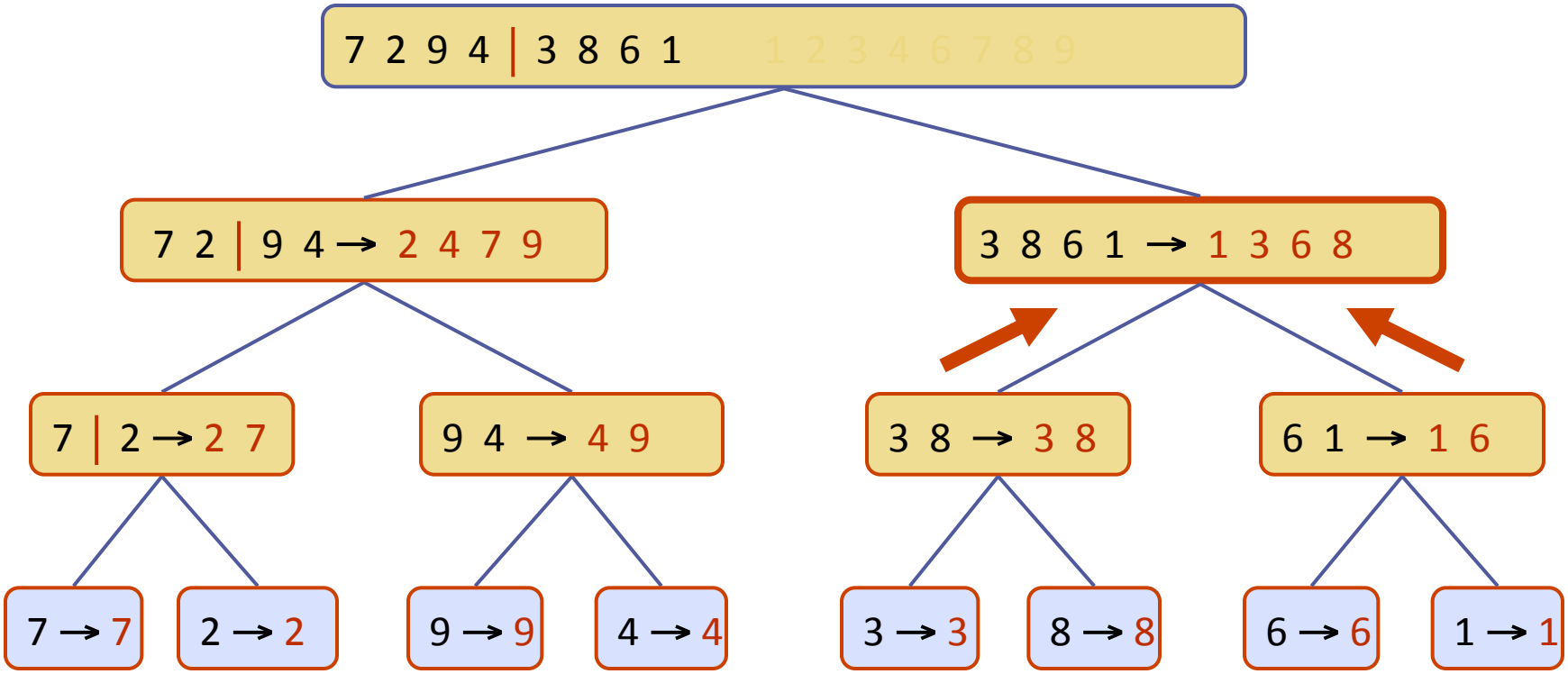
Example


7 2 9 4 | 3 8 6 1
1 2 3 4 6 7 8 9
7 2 | 9 4
2 4 7 9
3 8 6 1
1 3 8 6
7 | 2 → 2 7
9 4 → 4 9
3 8
3 8
6 1
1 6
7 → 7
2 → 2
9 → 9
4 → 4
3
3
8
8
6
6
1
1

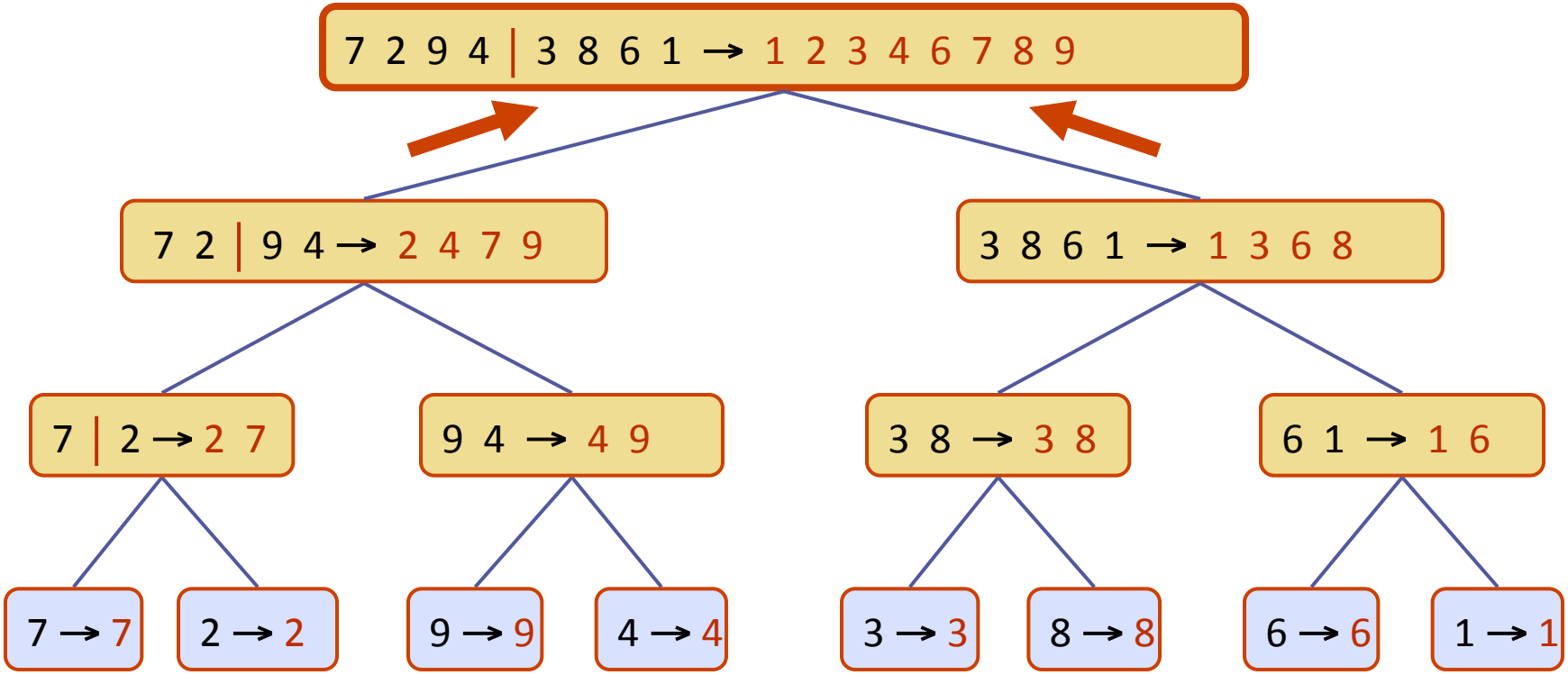
Example


7 2 9 4 | 3 8 6 1
1 2 3 4 6 7 8 9
7 2 | 9 4 → 2 4 7 9
3 8 6 1
1 3 8 6
7 | 2 → 2 7
9 4 → 4 9
3 8
3 8
6 1
1 6
7 → 7
2 → 2
9 → 9
4 → 4
3
3
8
8
6
6
1
1

Example


7 2 9 4 | 3 8 6 1 1 2 3 4 6 7 8 9
7 2 | 9 4 → 2 4 7 9
3 8 6 1 → 1 3 6 8
7 | 2 → 2 7
9 4 → 4 9
3 8 → 3 8
6 1 → 1 6
7 → 7
2 → 2
9 → 9
4 → 4
3 → 3
8 → 8
6 → 6
1 → 1

Example

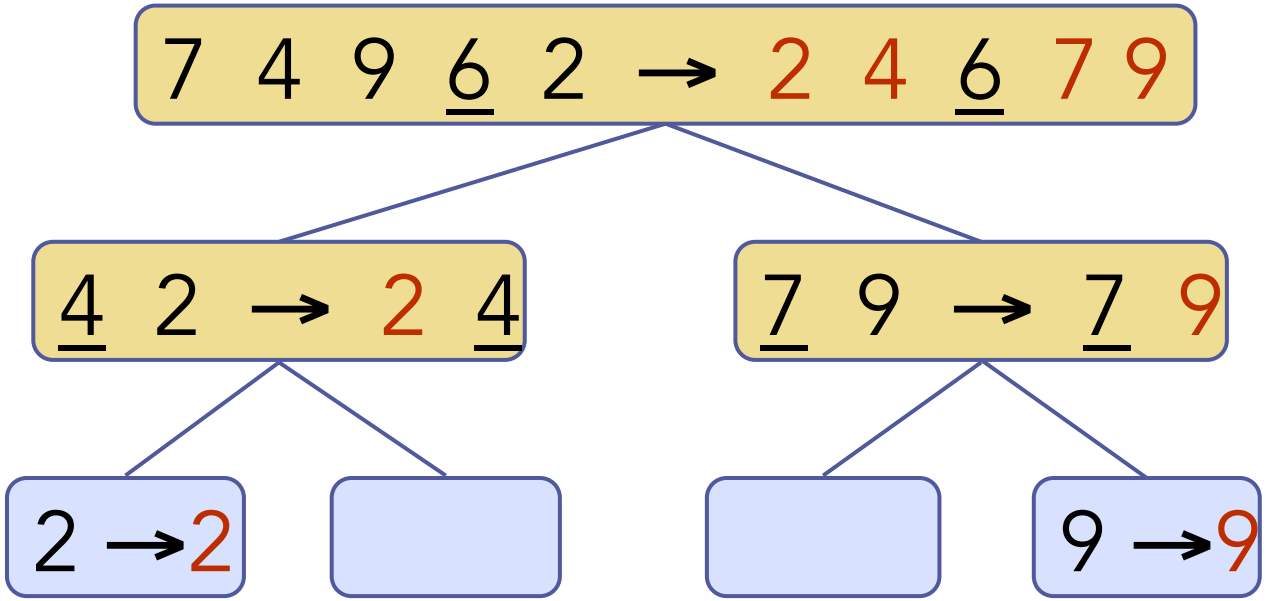

7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9
7 2 | 9 4 → 2 4 7 9
3 8 6 1 → 1 3 6 8
7 | 2 → 2 7
9 4 → 4 9
3 8 → 3 8
6 1 → 1 6
7 → 7
2 → 2
9 → 9
4 → 4
3 → 3
8 → 8
6 → 6
1 → 1

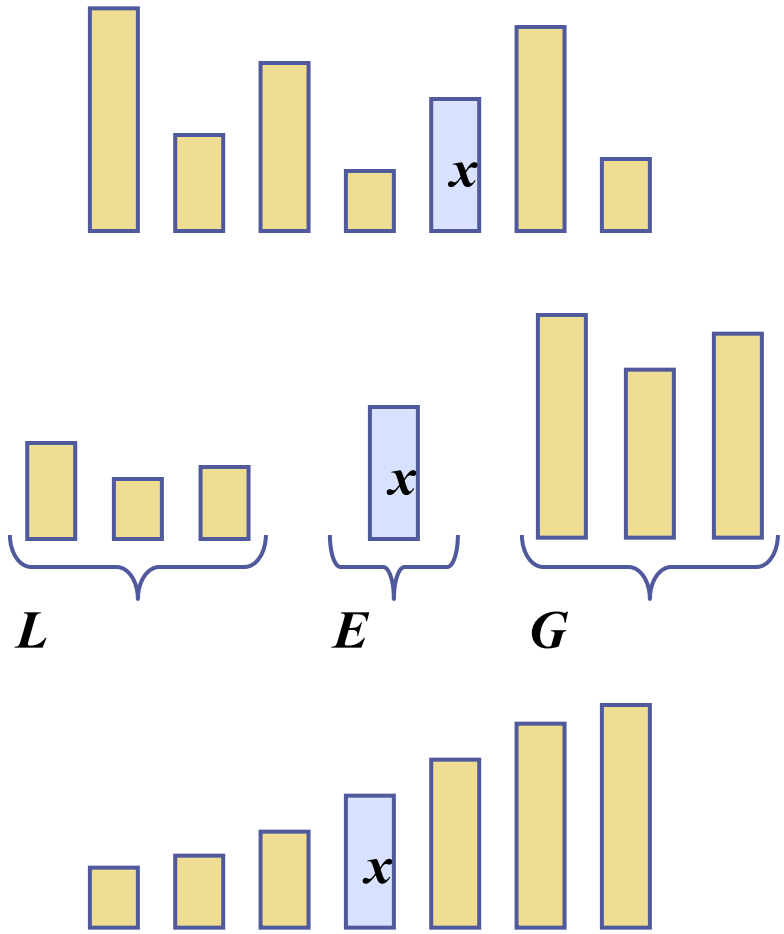
Time complexity?

$$T(n)=2T(n/2) + cn$$

$$O(n \log n)$$

Quick-Sort




x
x
L
E
G
x

Quicksort

```
QuickSort(A, p, r)
if p<r
    q = Partition(A, p, r)
    QuickSort(A, p, q-1)
    QuickSort(A, q+1, r)
```

How will you partition an array?

Partition

```
Partition(A, p, r)
    x=A(r)
    i = p-1
    for j = p to r-1
        if A(j) ≤ x
            i = i+1
            swap(A[i], A[j])
    swap(A[i+1], A[r])
    return i+1
```

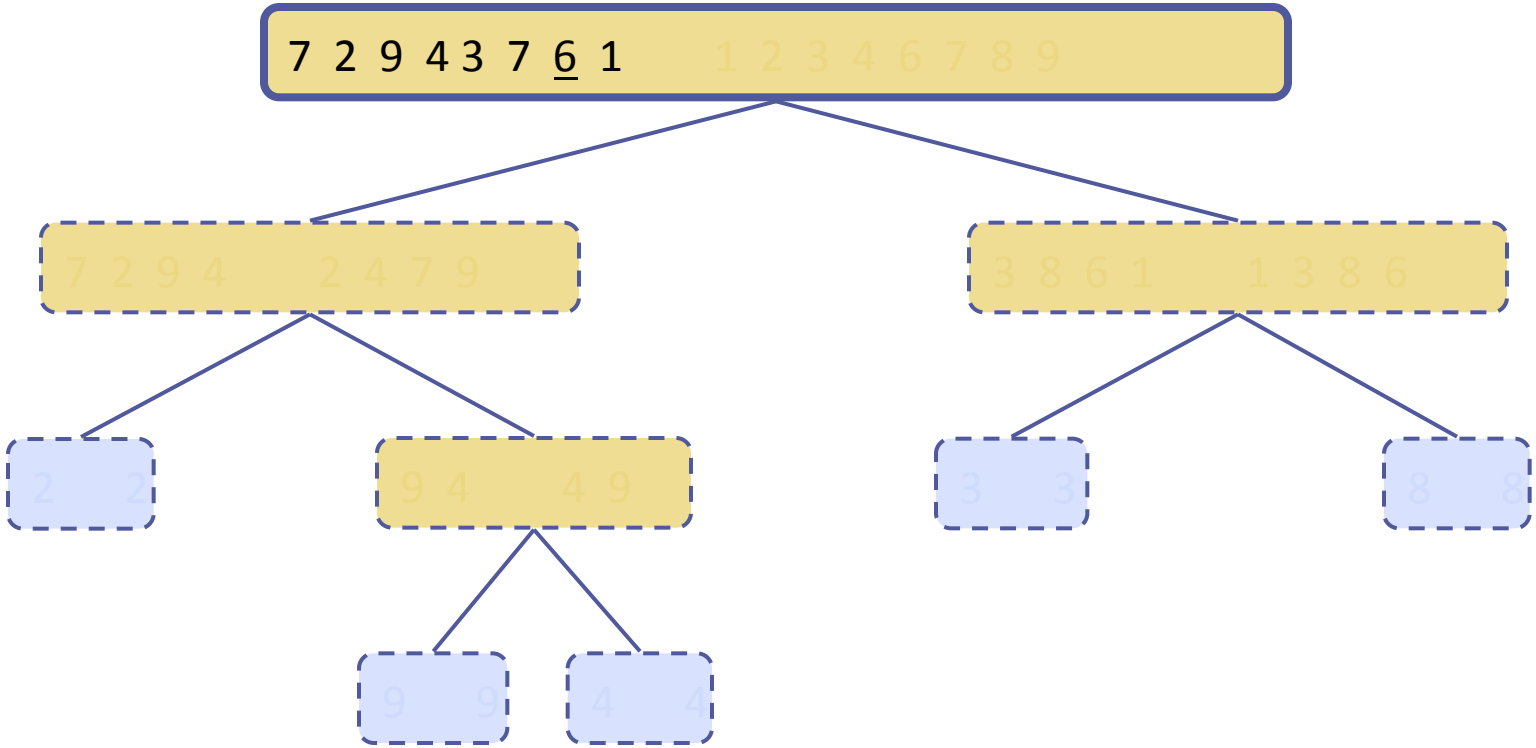
Partition – Linked List

```
Algorithm partition(S, p)
    L, E, G ← empty sequences
    x ← S.remove(p)
    while ¬S.empty()
        y ← S.removeFront()
        if y < x
            L.insertBack(y)
        else if y = x
            E.insertBack(y)
        else { y > x }
            G.insertBack(y)
    return L, E, G
```

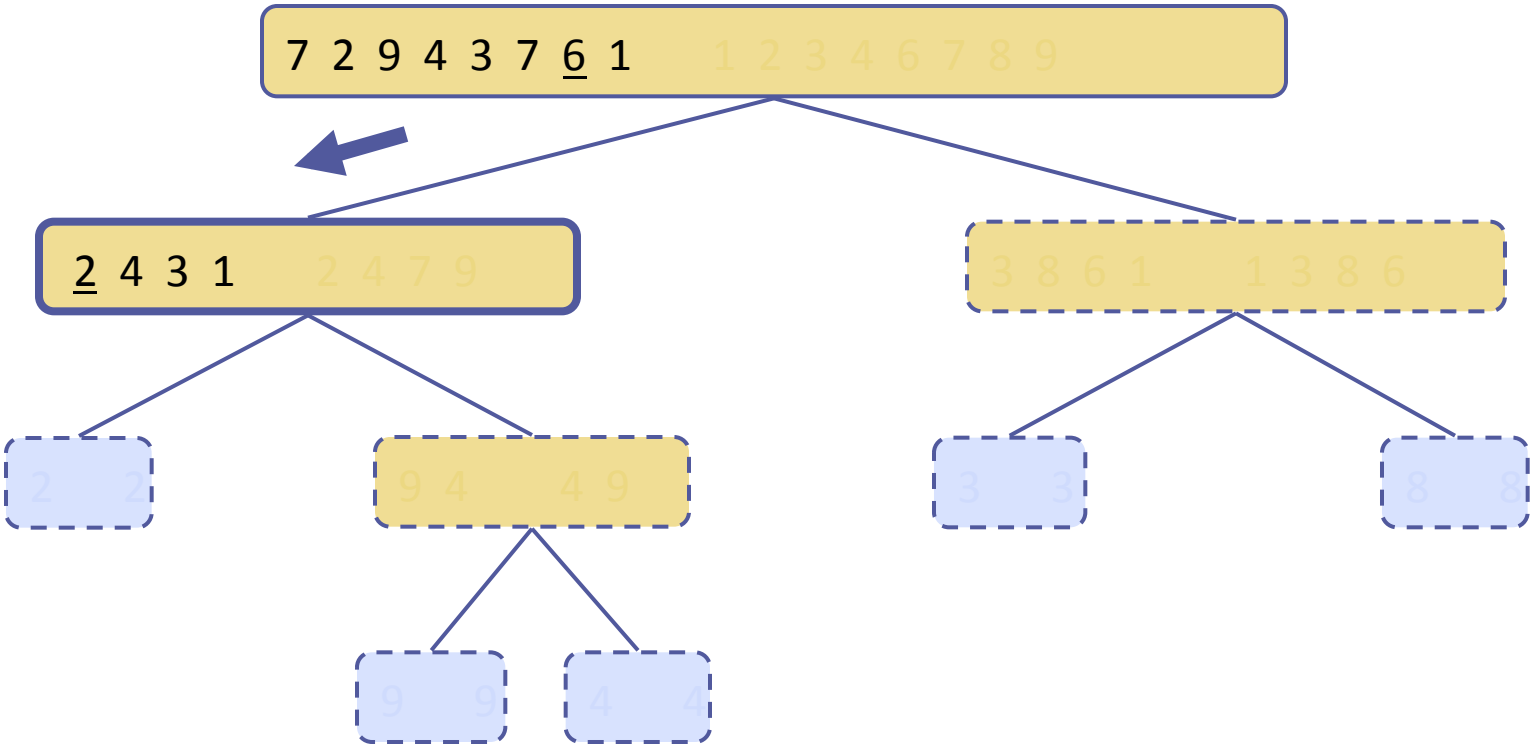
Merge

{L} U {E} U{G}

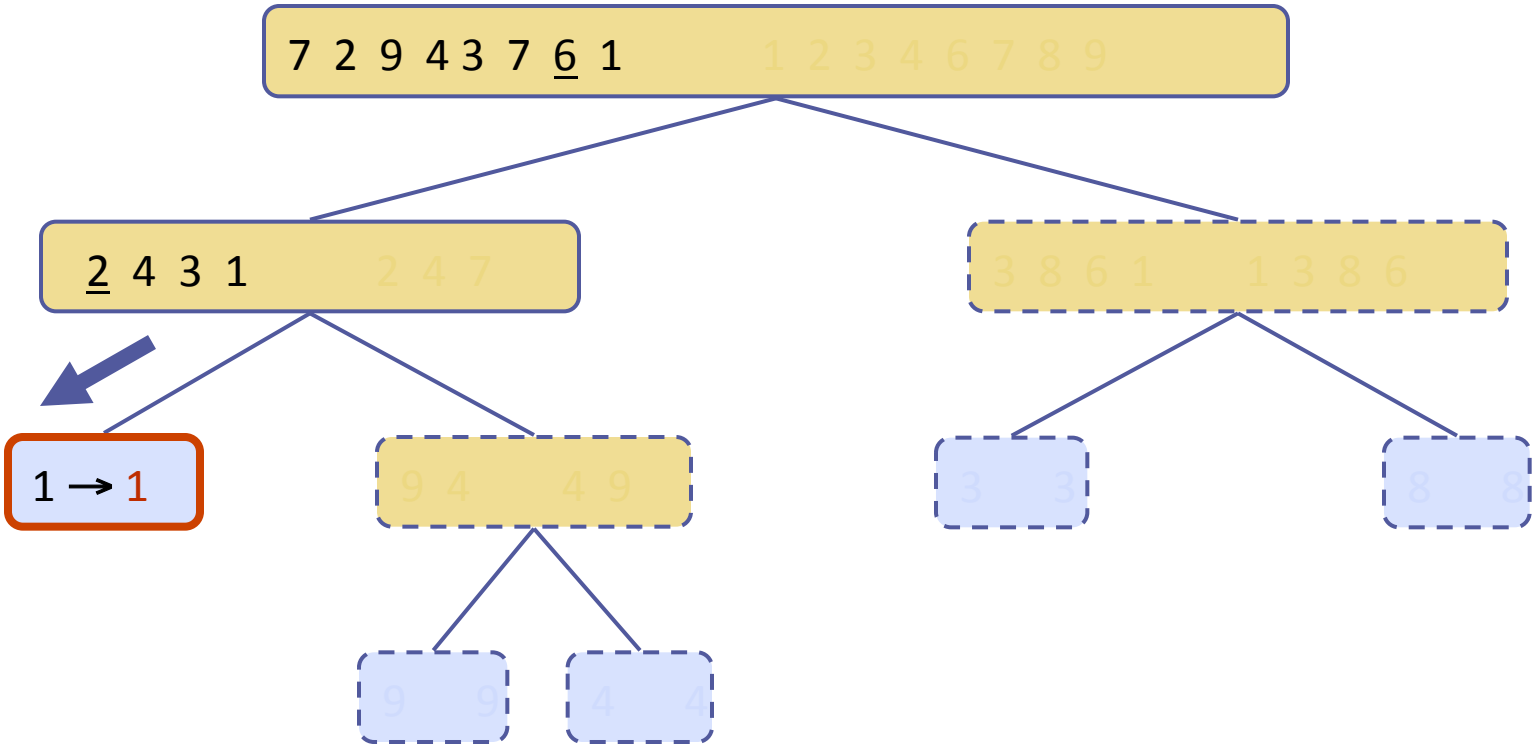
Example



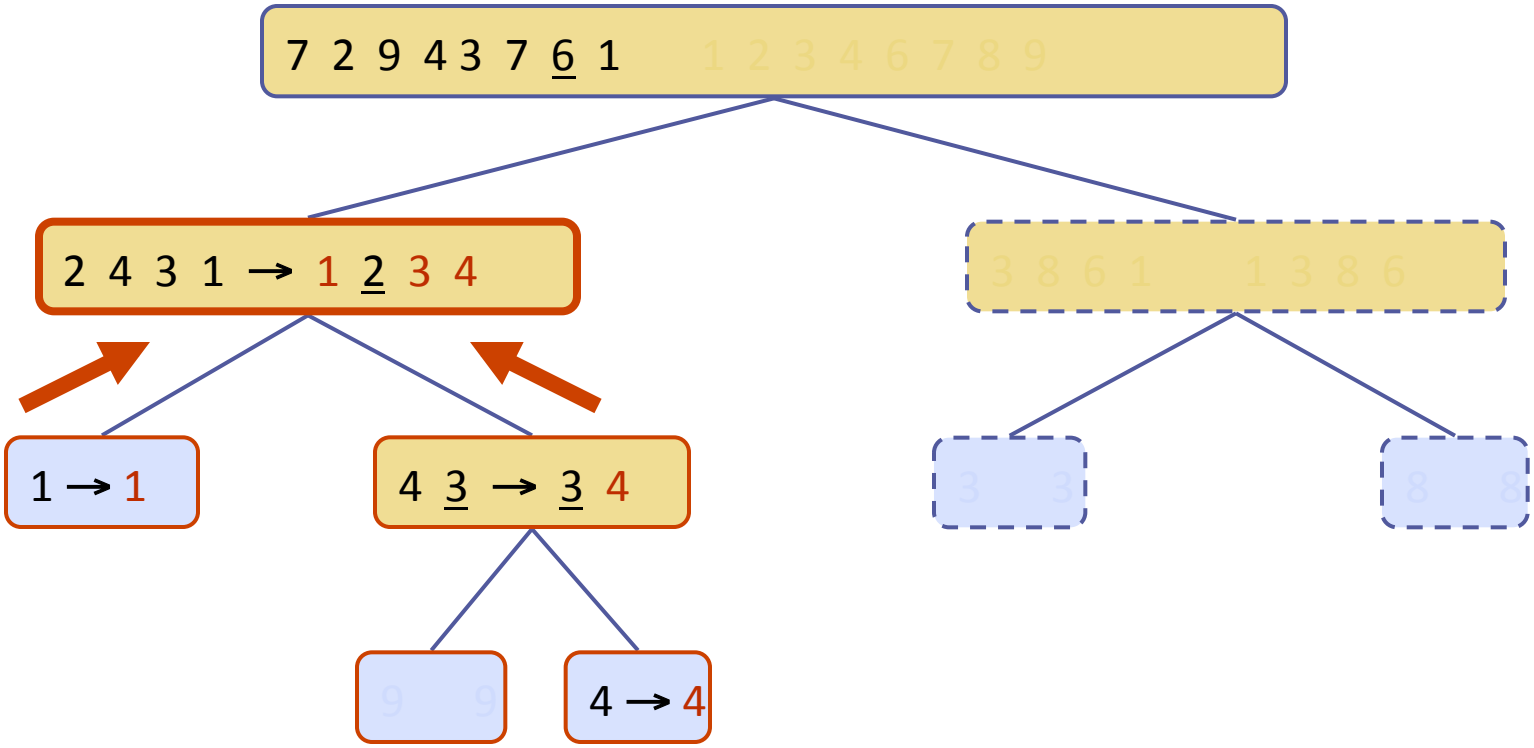
Example



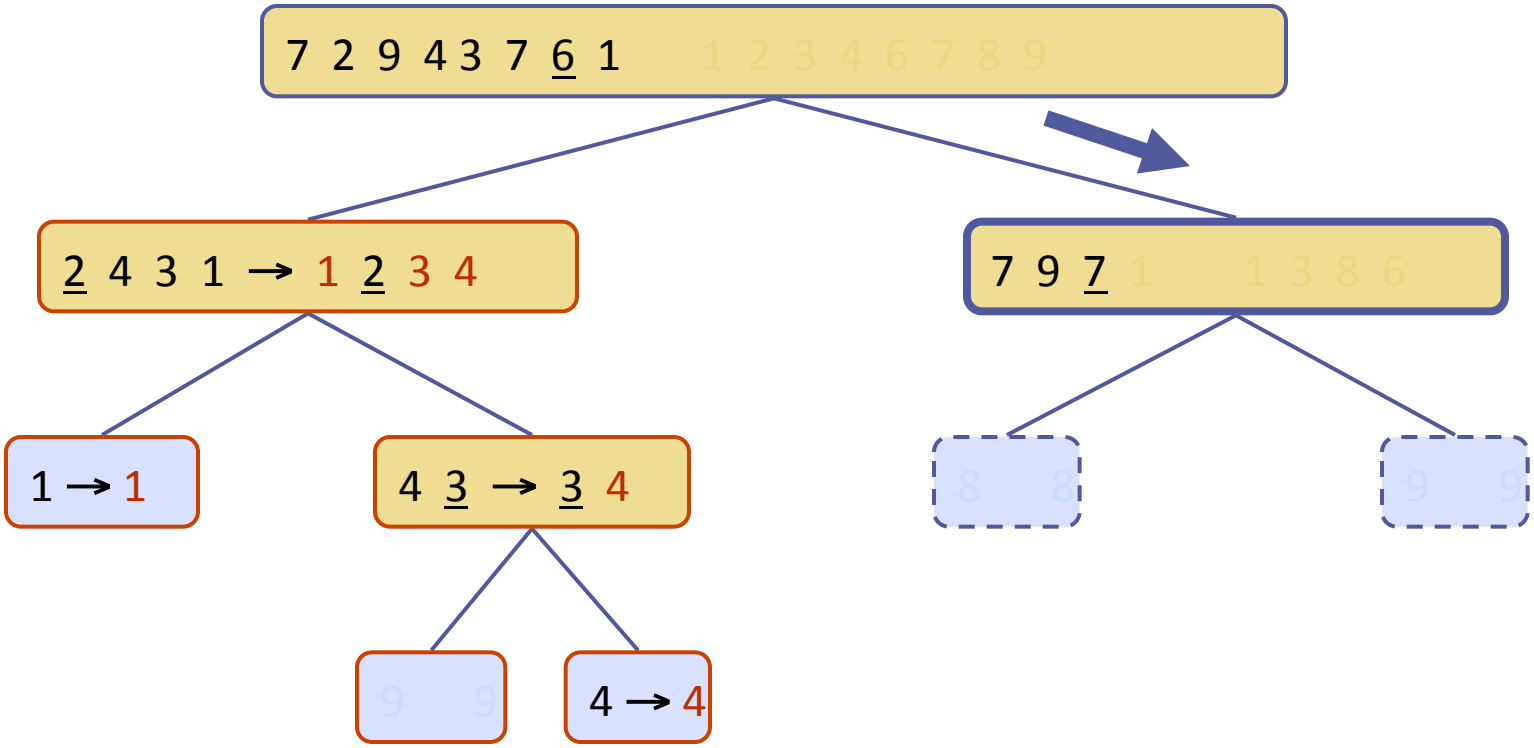
Example



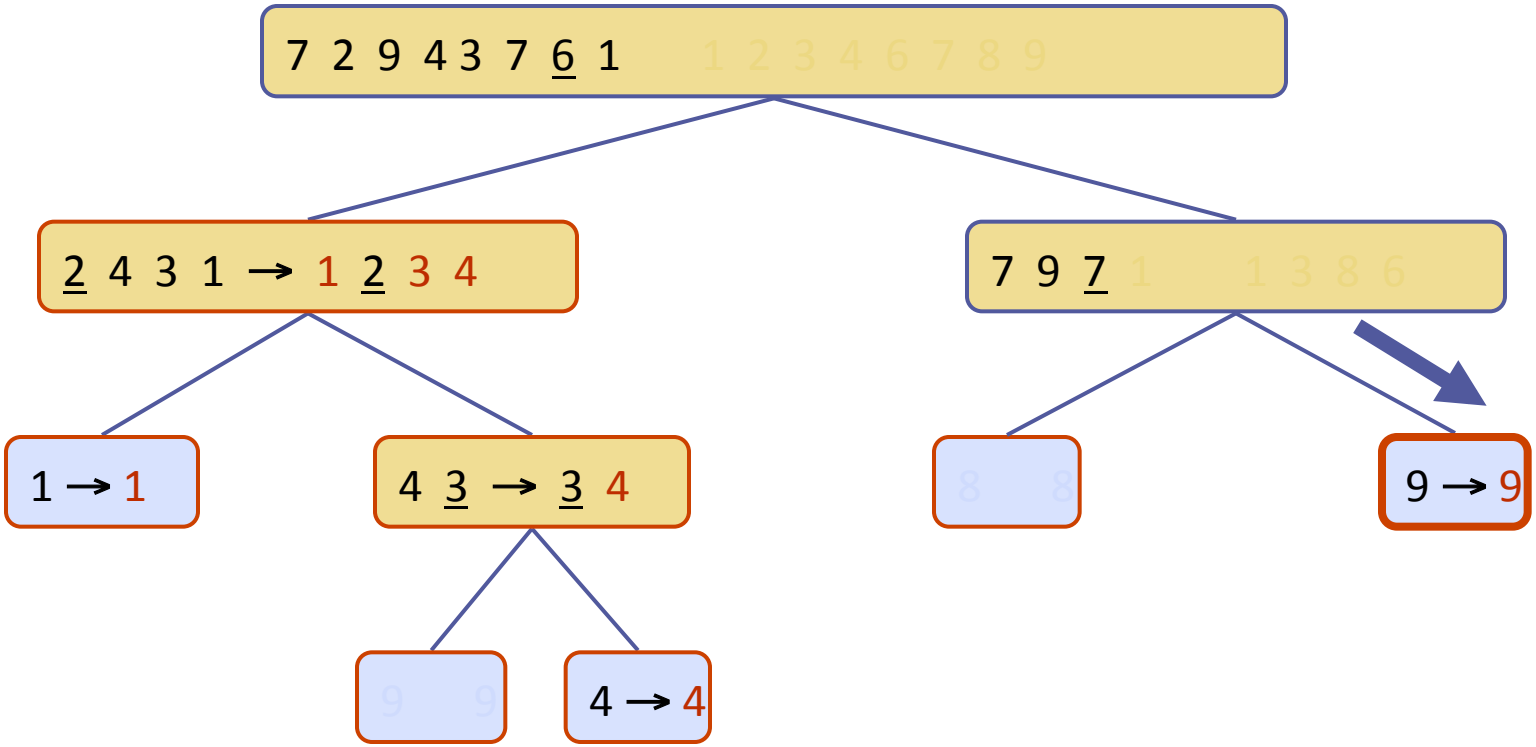
Example



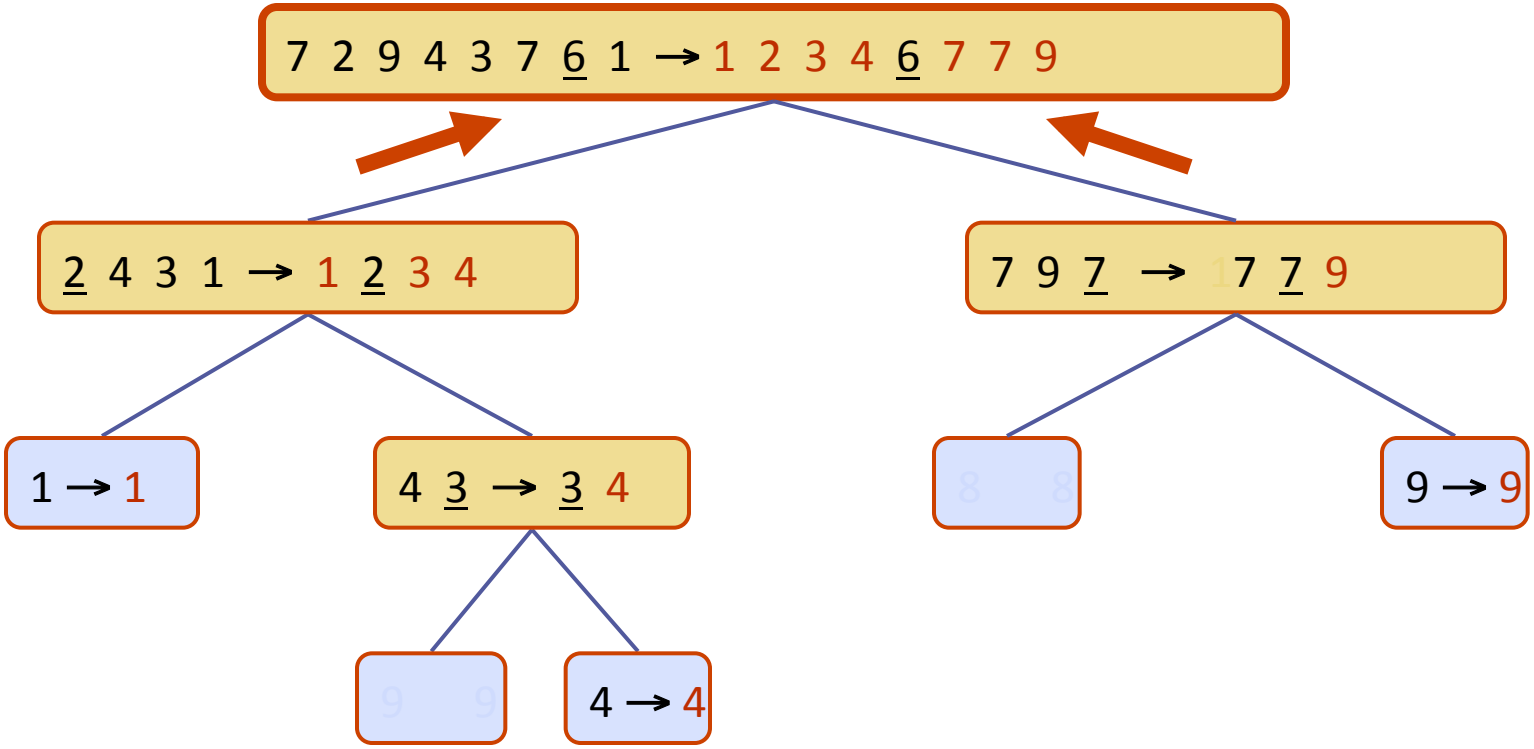
Example



Example



Example



Time Complexity of Quick Sort

What is the worst case for the quick sort?

$$T(n) = T(n-1) + T(1) + cn$$

$$O(n^2)$$

What is the best case?

$$T(n) = 2T(n/2) + cn$$

$$O(n \log n)$$

Average Case

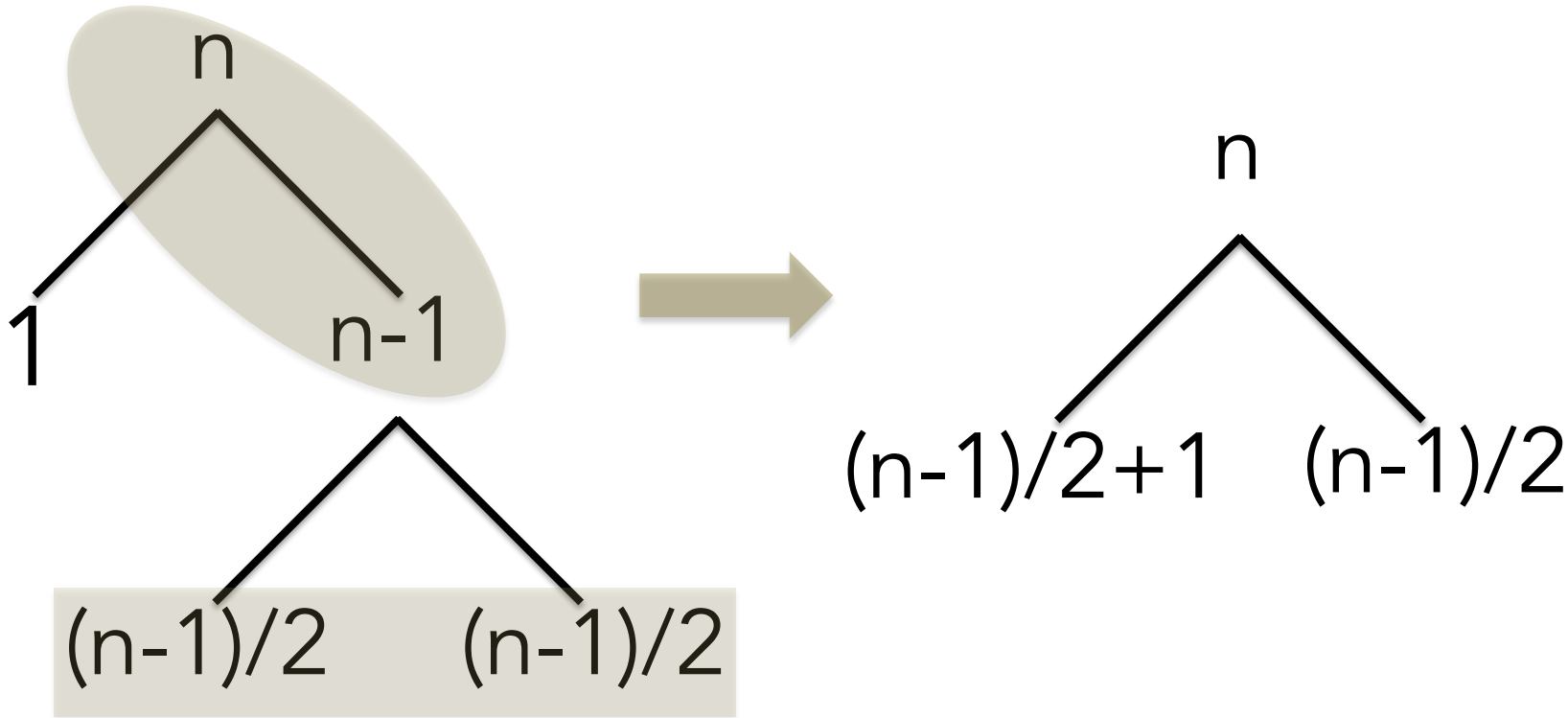
Suppose pivot always produces 9-1 split!

$$T(n) = T(9n/10) + T(n/10) + cn$$

What is complexity?

In case of quick sort, best case is close to the average case!

Let's have best case and worst case alternatively!



How do we make sure we are lucky?

- Partition around middle element?
- Randomize the input array?
- Choose pivot randomly?

Randomized Quicksort!

What is the expected running time of randomized Quicksort?

Expected Value of x

$$E(X) = \sum_{\forall i} x_i p_i$$

Example:

4 7 2 10 6 5 1 8 9 3

How many times is the partition method called?

```
QuickSort(A, p, r)
if p<r
    q = Partition(A, p, r)
    QuickSort(A, p, q-1)
    QuickSort(A, q+1, r)
```

How many comparisons are we doing in each partition?

Count total number of comparisons in all calls of partition!

Are we comparing each element with each other element?

Total number of comparisons!

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

$X_{ij} = 1$ if z_i is compared with z_j, else 0

Expectation

$$E(X) = E\left(\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right) = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E(X_{ij})$$

$$E(X) = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr(z_i \text{ is compared with } z_j)$$

$E(X)$ is O(n log n)

Time Complexity of Quicksort

- Partitions + Comparisons
  - O(n) + O(n log n)
    - O(n log n)

Quick-Sort Vs Merge-Sort

- The merge step in quick-sort does not need any comparisons!
- Typically, merge-sort is stable while quick-sort isn't!
- Typically, quick-sort is in-place, while merge-sort isn't!

Assumption of random input is not valid in many cases, so we drop this assumption and randomize the algorithm instead!

Randomization

- Randomize the input
- Randomize the pivot selection, random sampling!

Sorting algorithms

1. Insertion sort
2. Selection sort
3. Bubble sort
4. Heap sort
5. Merge sort
6. Quicksort

Sort integers that are in range 0 to k, where k is O(n) in linear time!

Counting Sort

- Suppose range(A[]) = k
- take an array C of size k
- use A[i] as index in C to COUNT
- read back the sorted list in B

```
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to n
    do C[A[ j]] ← C[A[ j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i–1]
for j ← n downto 1
    do
        B[C[A[ j]]] ← A[ j]
        C[A[ j]] ← C[A[ j]] – 1
```

Counting Sort Example

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:				

B:					

Loop 1

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	0

B:					

for $i \leftarrow 1$ **to** k

  do $C[i] \leftarrow 0$

Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	1

B:					

for $j \leftarrow 1$ **to** n

  do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	0	1

B:					

for $j \leftarrow 1$ **to** n

  do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	1	1

B:					

for $j \leftarrow 1$ **to** n

  do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	1	2

B:					

for $j \leftarrow 1$ **to** n

  do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	2	2

B:					

for $j \leftarrow 1$ **to** n

  do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Loop 3

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	2	2

for $i \leftarrow 2$ **to** k

  do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \le i\}|$

Loop 3

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	2

for $i \leftarrow 2$ **to** k

  do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \le i\}|$

Loop 3

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	5

for $i \leftarrow 2$ **to** k

  do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

Loop 4

	1	2	3	4	5
A:	4	1	3	4	3
B:			3		

	1	2	3	4
C:	1	1	3	5
C':	1	1	2	5

for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow A[j]$

    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

	1	2	3	4	5
A:	4	1	3	4	3
B:			3		4

	1	2	3	4
C:	1	1	2	5
C':	1	1	2	4

for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow A[j]$

    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

	1	2	3	4	5
A:	4	1	3	4	3
B:		3	3		4

	1	2	3	4
C:	1	1	2	4
C':	1	1	1	4

for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow A[j]$

    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

	1	2	3	4	5
A:	4	1	3	4	3
B:	1	3	3		4

	1	2	3	4
C:	1	1	1	4
C':	0	1	1	4

for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow A[j]$

    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

	1	2	3	4	5
A:	4	1	3	4	3
B:	1	3	3	4	4

	1	2	3	4
C:	0	1	1	4
C':	0	1	1	3

for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow A[j]$

    $C[A[j]] \leftarrow C[A[j]] - 1$

Analysis

$O(k)$

```
for i ← 1 to k
    do C[i] ← 0
```

$O(n)$

```
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1
```

$O(k)$

```
for i ← 2 to k
    do C[i] ← C[i] + C[i–1]
```

$O(n)$

```
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] – 1
```

$O(n + k)$

Time complexity

Since $k = O(n)$, counting sort is $O(n)$!

How is it different from other sorting algorithms?

Comparison based sorting algorithms cannot be faster than O(n log n)!

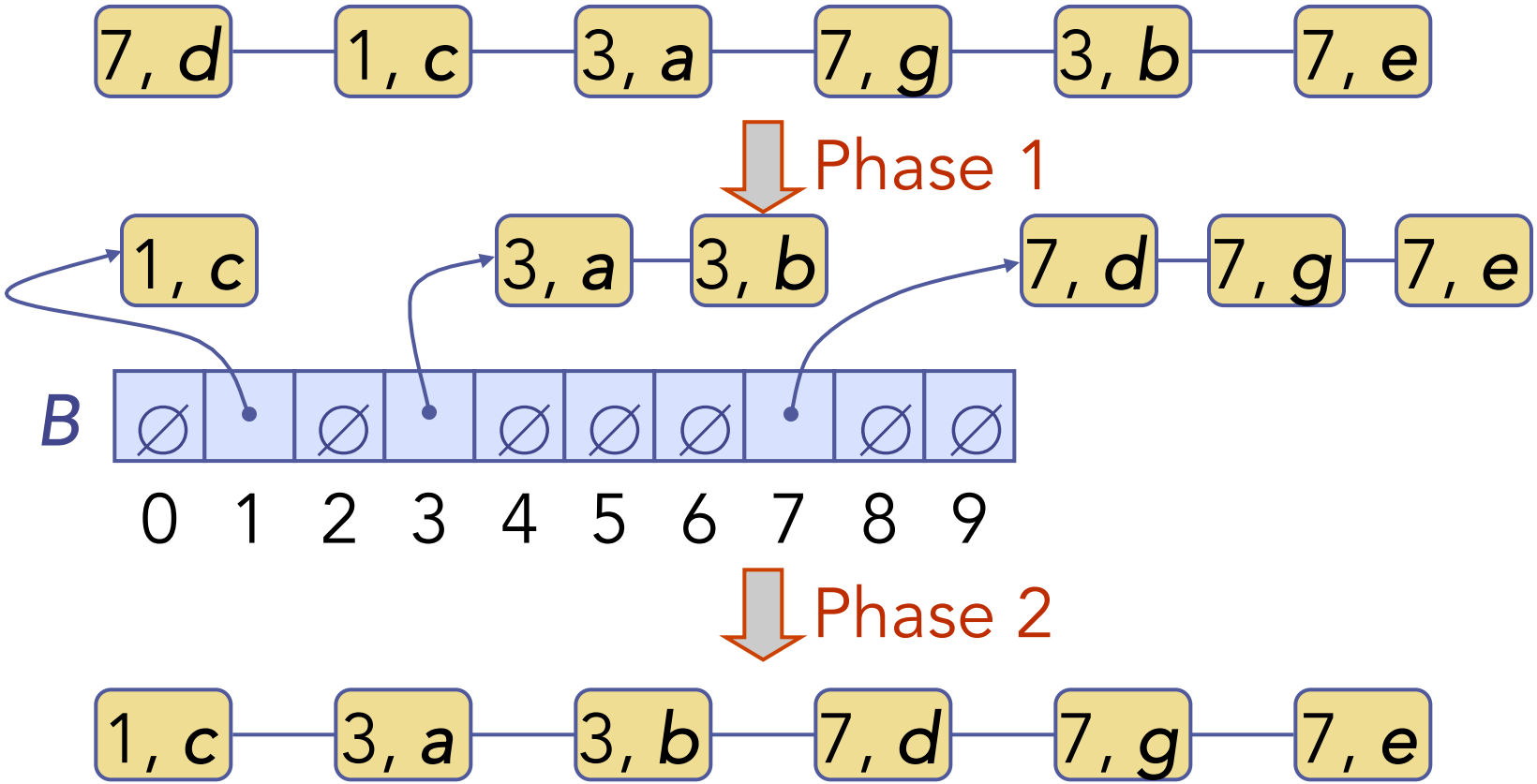
What if k is NOT O(n)!

But numbers are uniformly distributed!

Bucket Sort

```
Let B[0..n-1] be a new array
for i ← 0 to n-1
    make B[i] an empty list
for i ← 0 to n-1
    insert A[i] into list B[⌊nA[i]⌋]
for i ← 0 to n-1
    sort list B[i] with insertion sort
concatenate B[0], B[1], …B[n-1]
```

Example


7, d
1, c
3, a
7, g
3, b
7, e
Phase 1
1, c
3, a
3, b
7, d
7, g
7, e
B
∅
∅
∅
∅
∅
∅
0 1 2 3 4 5 6 7 8 9
Phase 2
1, c
3, a
3, b
7, d
7, g
7, e

Time complexity

$$T(n) = O(n) + \Sigma\, O(n_i^2)$$

$$\mathbf{E}(n_i^2) = 2 - 1/n$$

$$\Sigma\, O(n_i^2) = n.O(2-1/n) = O(n)$$

Hence, bucket sort is O(n)

Radix Sort

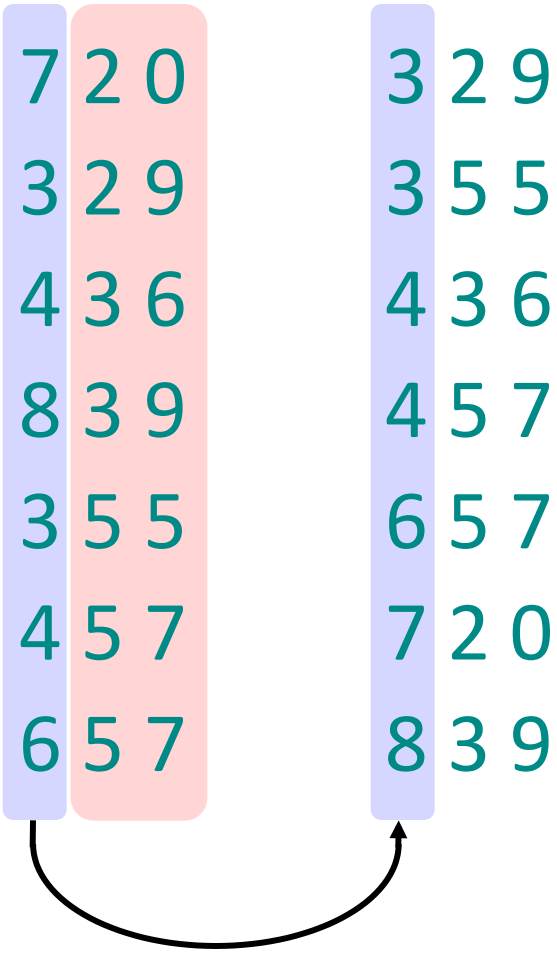
Operation of radix sort

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

Correctness of radix sort

Induction on digit position

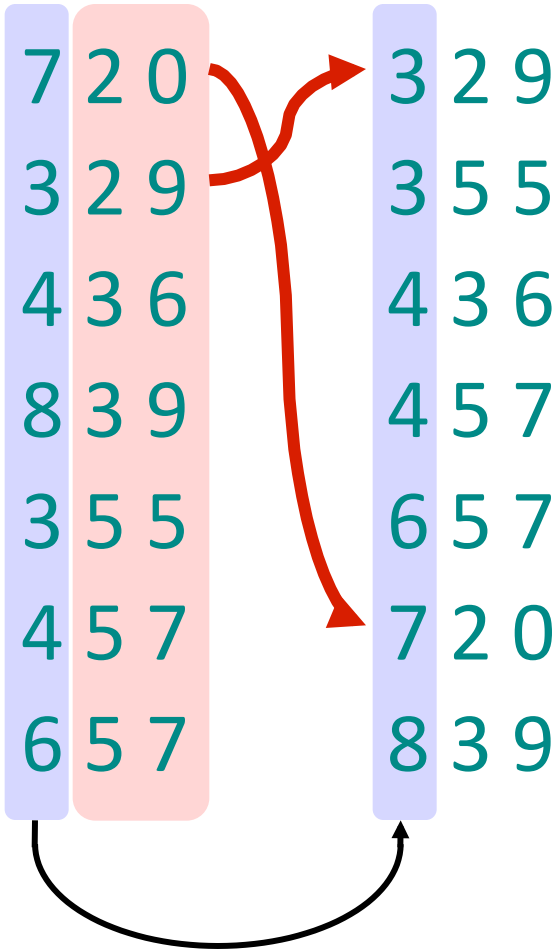
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t



Correctness of radix sort

Induction on digit position

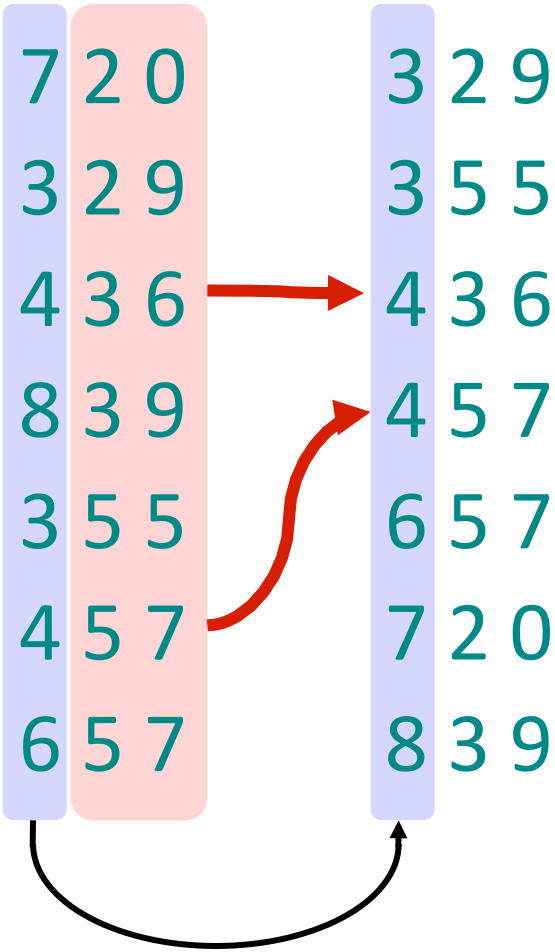
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.



Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input $\Rightarrow$ correct order.



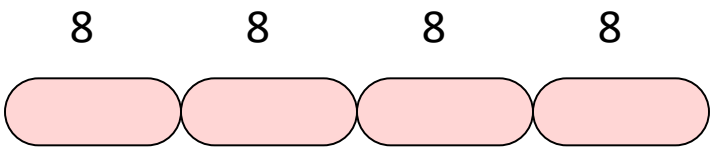
Time complexity of radix sort

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

Analysis of radix sort with b bit words

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.

Example: 32-bit word



$r = 8 \Rightarrow b/r = 4$ passes of counting sort on base-2^8 digits; or $r = 16 \Rightarrow b/r = 2$ passes of counting sort on base-2^{16} digits.

How many passes should we make?

Analysis (continued)

Recall: Counting sort takes $O(n + k)$ time to sort n numbers in the range from 0 to $k - 1$.

If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $O(n + 2^r)$ time. Since there are b/r passes, we have

Time complexity =is $O(b/r(n + 2^r))$

Choose r to minimize $T(n, b)$:

- Increasing r means fewer passes, but as $r >> \lg n$, the time grows exponentially.

Choosing *r*

Choosing $r = \lg n$ implies
time complexity is $O(n)$

Can we implement counting sort in-place?

Sorting algorithms

1. Insertion sort
2. Selection sort
3. Bubble sort
4. Merge sort
5. Quicksort
6. Heap sort
7. Counting sort
8. Radix sort
9. Bucket sort