Sorting Algorithms

Sorting

- Order a set of elements so that they satisfy a given condition.
- For example, rearrange n elements into ascending order.
- Example: 9, 5, 6, 4, 2 → 2, 4, 5, 6, 9

Selection Sort

1. Find the smallest number in the list and places it first.
2. Then it finds the smallest remaining number and places it next to first.
3. Repeat until the list contains only a single number.

element	2	7	4	1	5	8
index	0	1	2	3	4	5

i = 0
Loop
    The remaining elements: from index i to (n-1)
    Find the smallest number from remaining elements
    Swap the smallest number to the position i
    i <- i + 1
Repeat if i != (n-1)

Selection Sort

1. Find the smallest number in the list and places it first.
2. Then it finds the smallest remaining number and places it next to first.
3. Repeat until the list contains only a single number.

element	2	7	4	1	5	8
index	0	1	2	3	4	5

i = 0

Loop

    The remaining elements: from index i to (n-1)

    Find the smallest number from remaining elements

    Swap the smallest number to the position i

    i <- i + 1

Repeat if i != (n-1)

Selection Sort

1. Find the smallest number in the list and places it first.
2. Then it finds the smallest remaining number and places it next to first.
3. Repeat until the list contains only a single number.

element	1	7	4	2	5	8
index	0	1	2	3	4	5

i = 0

Loop

The remaining elements: from index i to (NUM-1)

Find the smallest number from remaining elements

Swap the smallest number to the position i

i <- i + 1

Repeat if i != (NUM-1)

Selection Sort

1. Find the smallest number in the list and places it first.
2. Then it finds the smallest remaining number and places it next to first.
3. Repeat until the list contains only a single number.

element	1	7	4	2	5	8
index	0	1	2	3	4	5

i = 0

Loop

The remaining elements: from index i to (n-1)

Find the smallest number from remaining elements

Swap the smallest number to the position i

$i \leftarrow i + 1$

Repeat if i != (n-1)

Time complexity

$O(n) + O(n-1) + \ldots + O(1)$

$= O(n^2)$

Insertion sort

1. Sort a list of values by repeatedly inserting an unsorted element into a sorted sublist.
2. Repeat the process until the whole list is sorted.

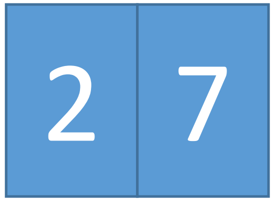
2	7	4	1	5	8

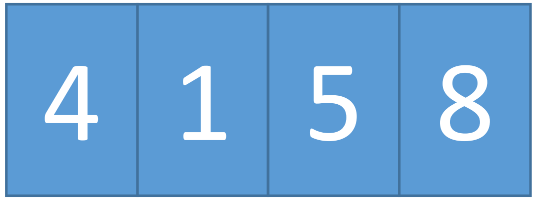
Insertion sort

1. Sort a list of values by repeatedly inserting an unsorted element into a sorted sublist.
2. Repeat the process until the whole list is sorted.

Insertion sort

1. Sort a list of values by repeatedly inserting an unsorted element into a sorted sublist.
2. Repeat the process until the whole list is sorted.





Insertion sort

1. Sort a list of values by repeatedly inserting an unsorted element into a sorted sublist.
2. Repeat the process until the whole list is sorted.

Insertion sort

1. Sort a list of values by repeatedly inserting an unsorted element into a sorted sublist.
2. Repeat the process until the whole list is sorted.

Insertion sort

1. Sort a list of values by repeatedly inserting an unsorted element into a sorted sublist.
2. Repeat the process until the whole list is sorted.

Insertion sort

1. Sort a list of values by repeatedly inserting an unsorted element into a sorted sublist.
2. Repeat the process until the whole list is sorted.

Insertion sort

1. Sort a list of values by repeatedly inserting an unsorted element into a sorted sublist.
2. Repeat the process until the whole list is sorted.

1	2	4	7

5	8

Time complexity
$O(n) + O(n-1) + \ldots + O(1)$
$= O(n^2)$

Quick Sort

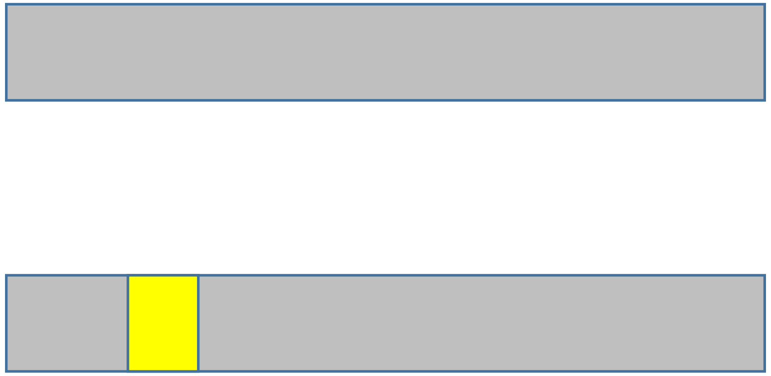
1. When n <= 1, the list is sorted.
2. When n > 1, select a pivot element.
3. Partition the n elements into 3 sets: left, middle and right.
4. The middle set contains only the pivot element.
5. All elements in the left set are <= pivot.
6. All elements in the right set are >= pivot.
7. Sort left and right sets recursively.

Quick Sort

1. When n <= 1, the list is sorted.
2. When n > 1, select a pivot element.
3. Partition the n elements into 3 sets: left, middle and right.
4. The middle set contains only the pivot element.
5. All elements in the left set are <= pivot.
6. All elements in the right set are >= pivot.
7. Sort left and right sets recursively.

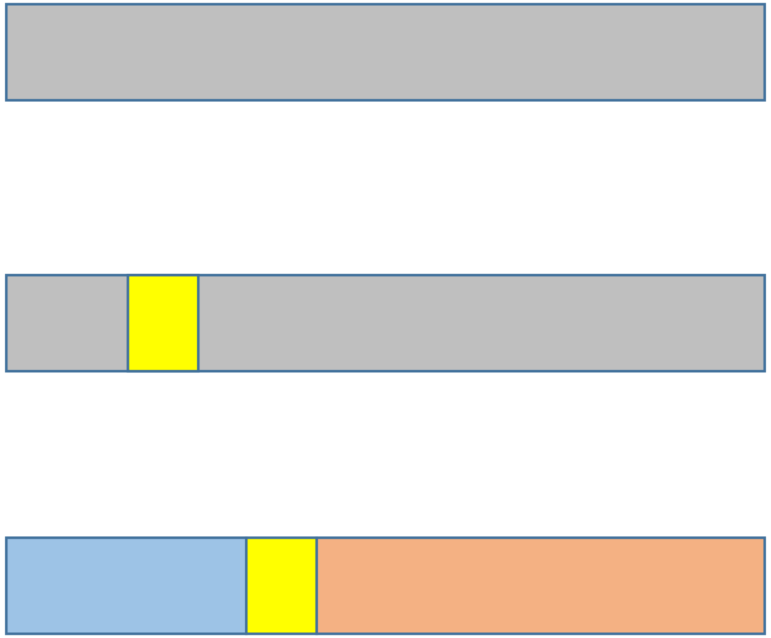
Quick Sort

1. When n <= 1, the list is sorted.
2. When n > 1, select a pivot element.
3. Partition the n elements into 3 sets: left, middle and right.
4. The middle set contains only the pivot element.
5. All elements in the left set are <= pivot.
6. All elements in the right set are >= pivot.
7. Sort left and right sets recursively.



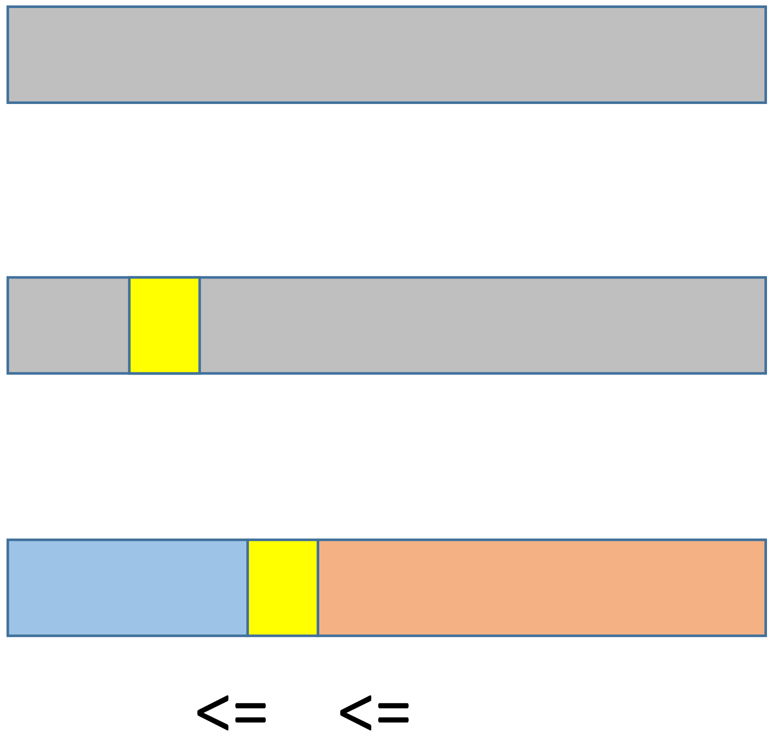
Quick Sort

1. When n <= 1, the list is sorted.
2. When n > 1, select a pivot element.
3. Partition the n elements into 3 sets: left, middle and right.
4. The middle set contains only the pivot element.
5. All elements in the left set are <= pivot.
6. All elements in the right set are >= pivot.
7. Sort left and right sets recursively.



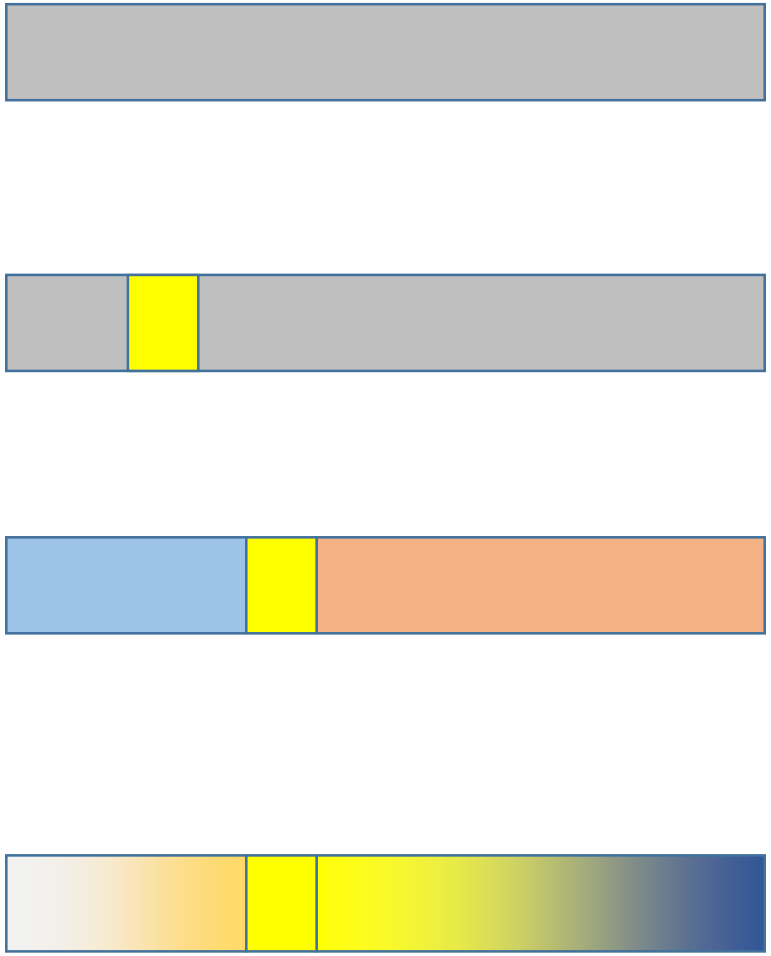
Quick Sort

1. When n <= 1, the list is sorted.
2. When n > 1, select a pivot element.
3. Partition the n elements into 3 sets: left, middle and right.
4. The middle set contains only the pivot element.
5. All elements in the left set are <= pivot.
6. All elements in the right set are >= pivot.
7. Sort left and right sets recursively.



Quick Sort

1. When n <= 1, the list is sorted.
2. When n > 1, select a pivot element.
3. Partition the n elements into 3 sets: left, middle and right.
4. The middle set contains only the pivot element.
5. All elements in the left set are <= pivot.
6. All elements in the right set are >= pivot.
7. Sort left and right sets recursively.



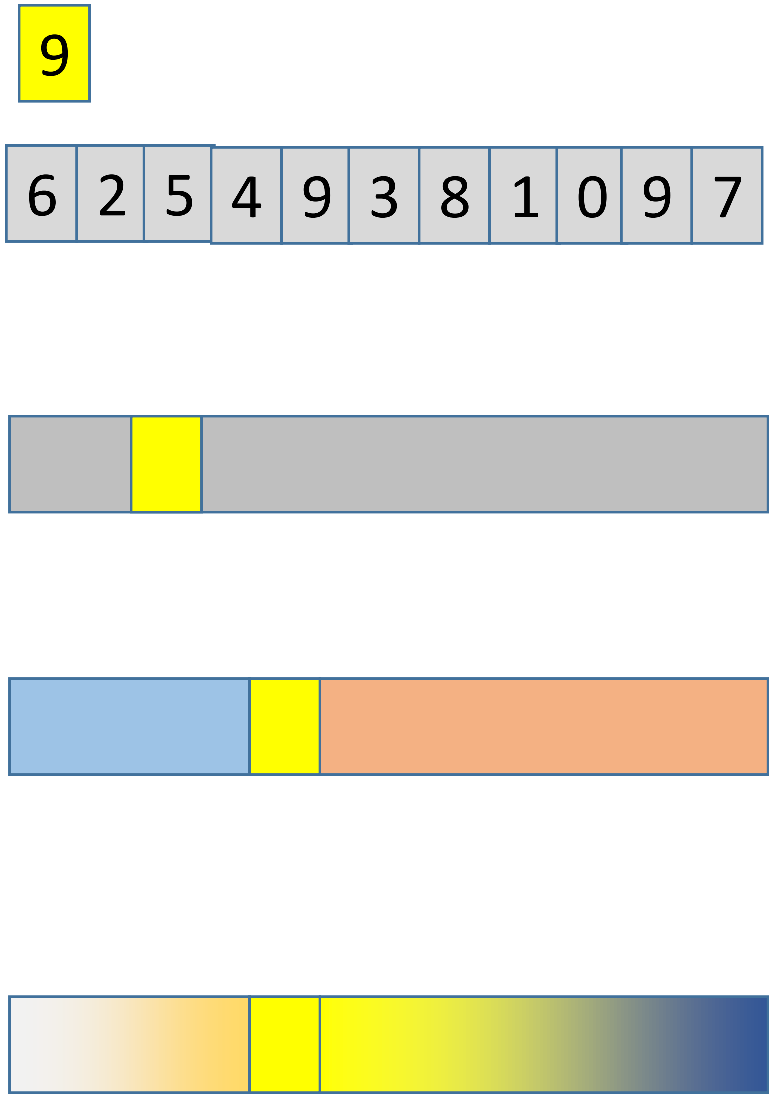
Quick Sort: Example

9

1. When n <= 1, the list is sorted.
2. When n > 1, select a pivot element.
3. Partition the n elements into 3 sets: left, middle and right.
4. The middle set contains only the pivot element.
5. All elements in the left set are <= pivot.
6. All elements in the right set are >= pivot.
7. Sort left and right sets recursively.

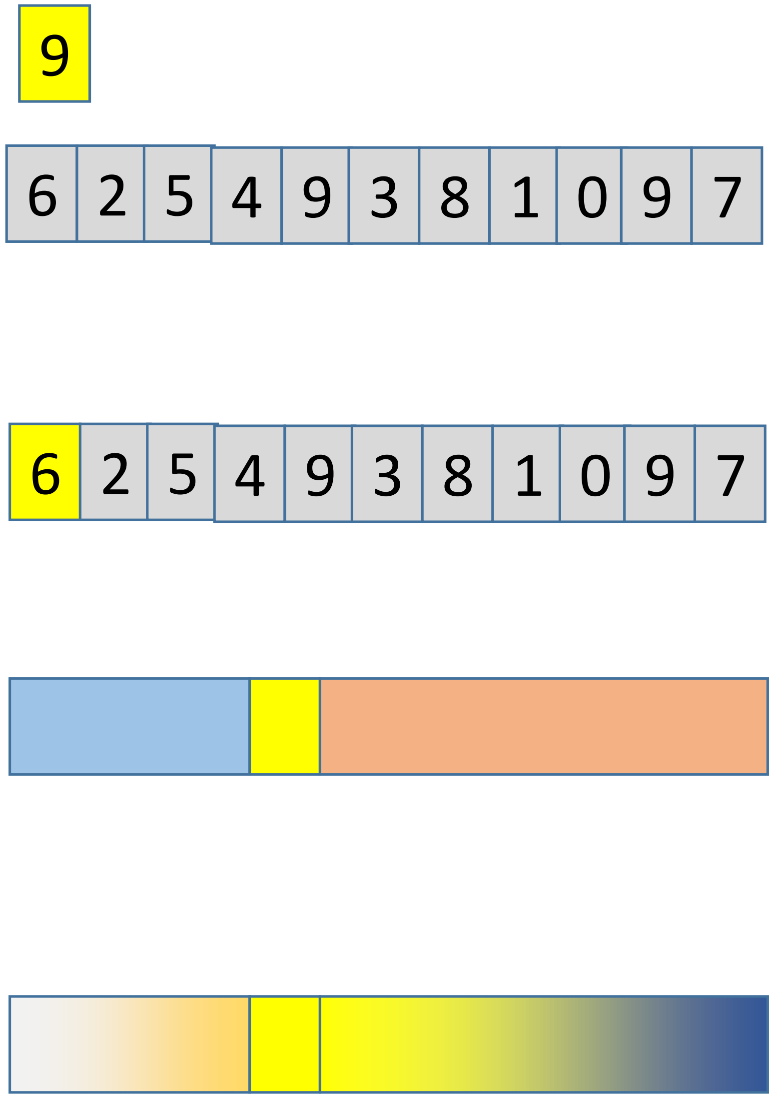
Quick Sort: Example

1. When n <= 1, the list is sorted.
2. When n > 1, select a pivot element.
3. Partition the n elements into 3 sets: left, middle and right.
4. The middle set contains only the pivot element.
5. All elements in the left set are <= pivot.
6. All elements in the right set are >= pivot.
7. Sort left and right sets recursively.



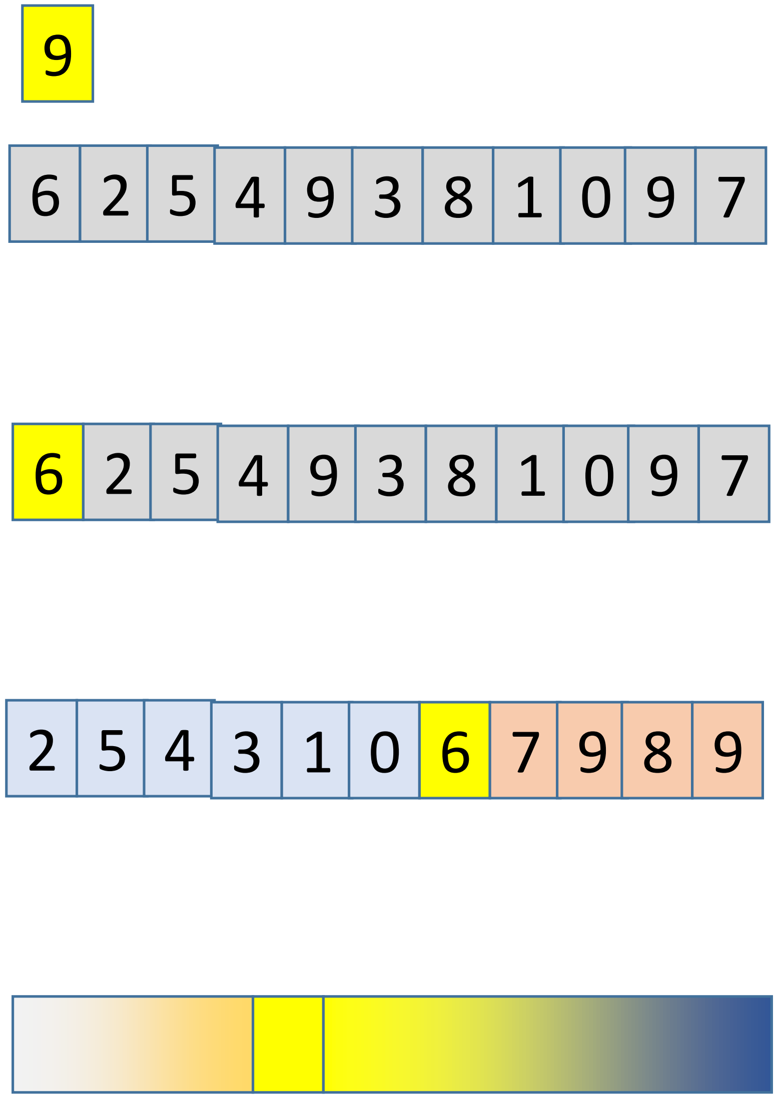
Quick Sort: Example

1. When n <= 1, the list is sorted.
2. When n > 1, select a pivot element.
3. Partition the n elements into 3 sets: left, middle and right.
4. The middle set contains only the pivot element.
5. All elements in the left set are <= pivot.
6. All elements in the right set are >= pivot.
7. Sort left and right sets recursively.



Quick Sort: Example

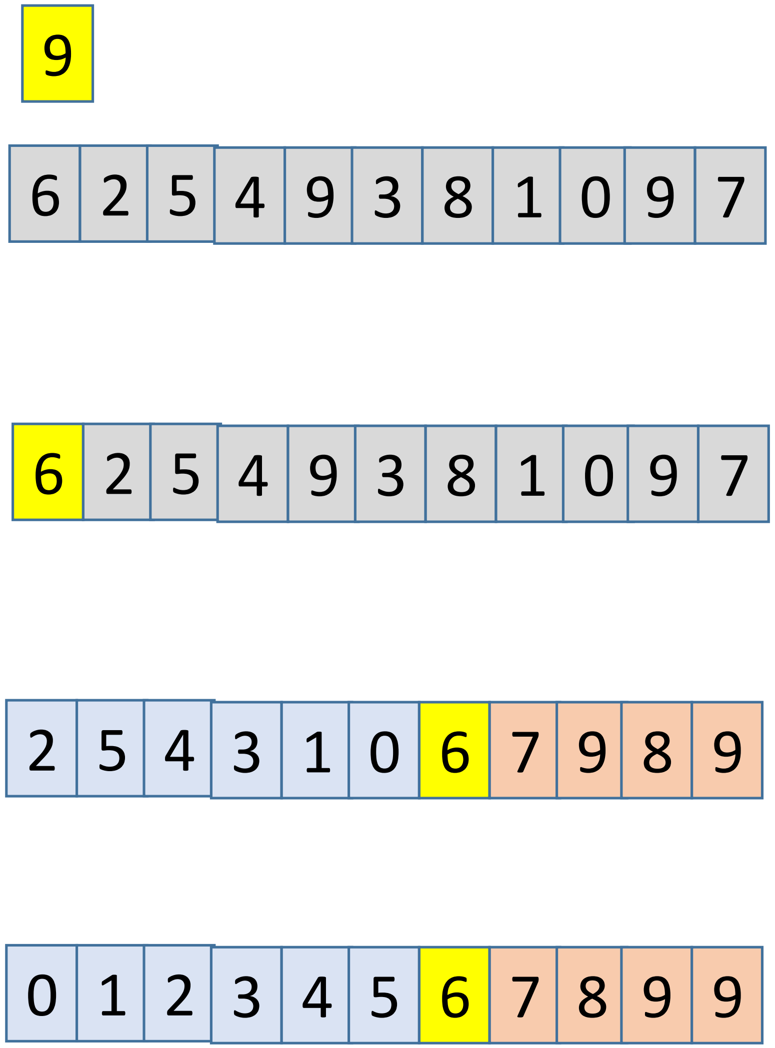
1. When n <= 1, the list is sorted.
2. When n > 1, select a pivot element.
3. Partition the n elements into 3 sets: left, middle and right.
4. The middle set contains only the pivot element.
5. All elements in the left set are <= pivot.
6. All elements in the right set are >= pivot.
7. Sort left and right sets recursively.



Quick Sort: Example

1. When n <= 1, the list is sorted.
2. When n > 1, select a pivot element.
3. Partition the n elements into 3 sets: left, middle and right.
4. The middle set contains only the pivot element.
5. All elements in the left set are <= pivot.
6. All elements in the right set are >= pivot.
7. Sort left and right sets recursively.

   quicksort(Left), quicksort(Right)



Choice Of Pivot

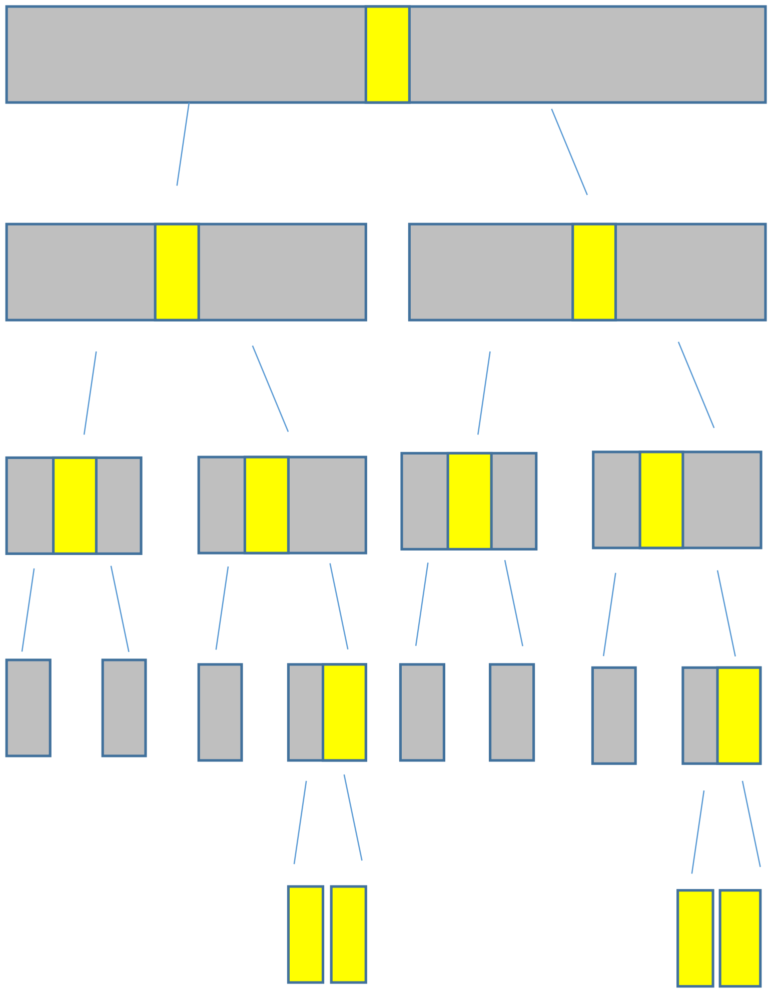
- The leftmost element in list that is to be sorted.
- The rightmost element in list that is to be sorted.
- The middle element in list that is to be sorted.
- Randomly select one of the elements to be sorted as the pivot.
- Median-of-Three rule. From the leftmost, middle, and rightmost elements of the list to be sorted, select the one with median key as the pivot.

6	2	5	4	9	3	8	1	0	9	7

{3, 6, 7}

Time Complexity

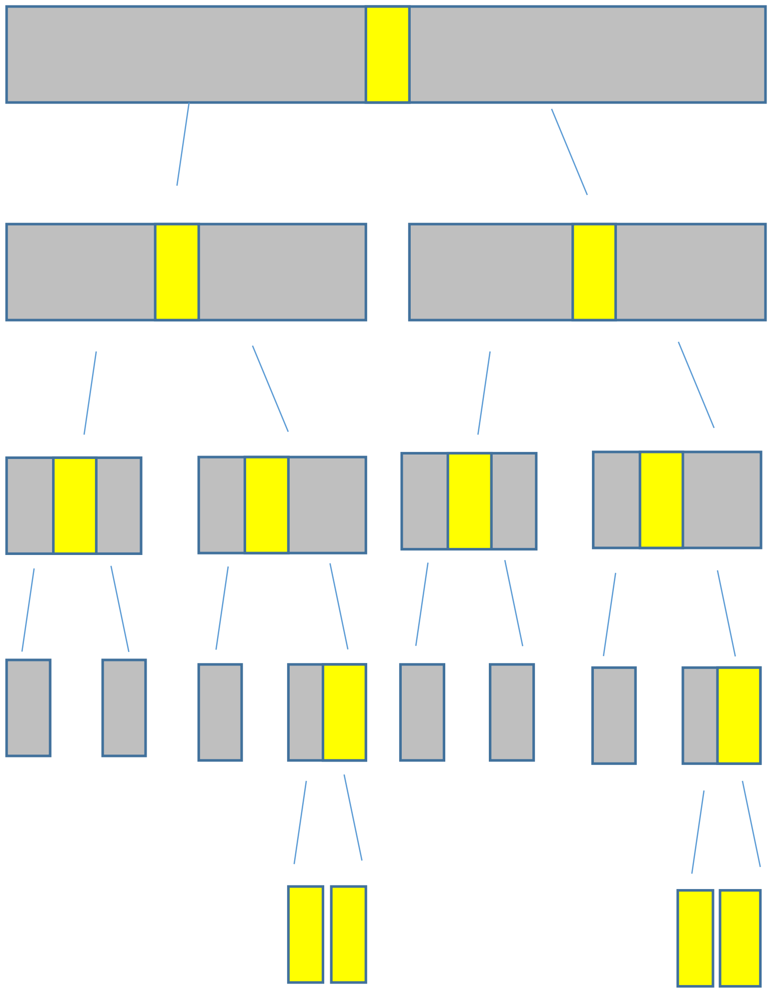
n = 16



Depth of the call stack: 1 + log n

Time Complexity

n = 16

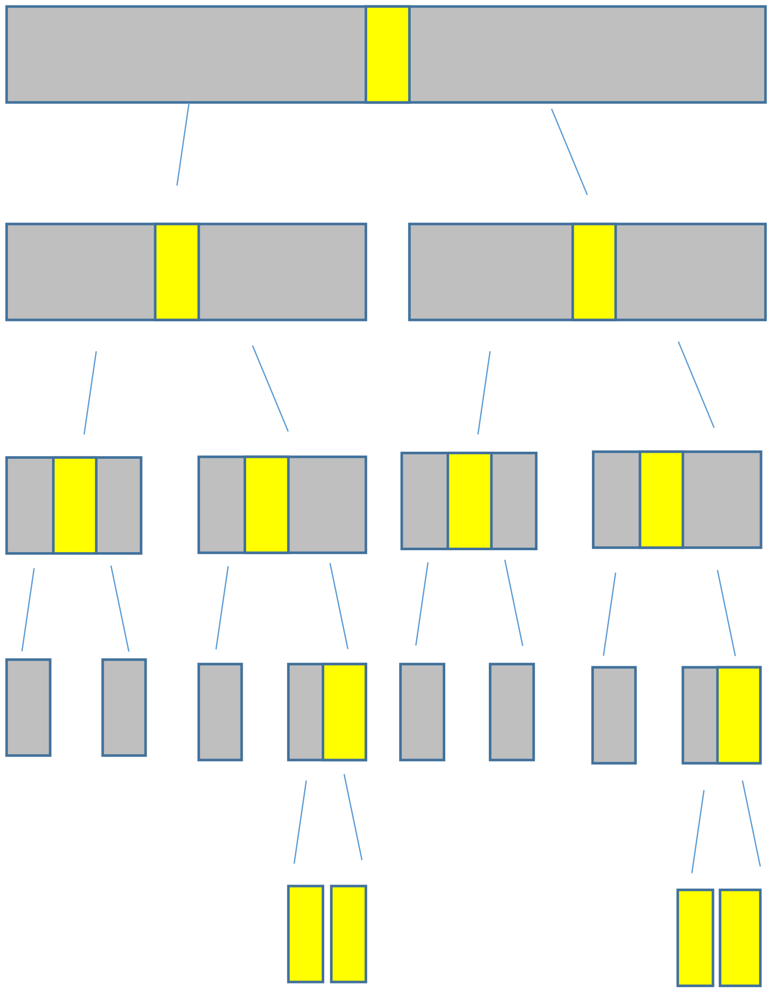


Depth of the call stack: 1 + log n

Time Complexity

n = 16

- Partition an array of n elements: O(n)
- t(n): time needed to sort n elements.
- t(0) = t(1) = c, where c is a constant.
  - t(n) = t(|left|) + t(|right|) + dn,
  - where d is a constant, for n >=2.
- t(n) is maximum when either |left| = 0 or |right| = 0 after each partitioning step.
- The best case, | t(|left|) - t(|right|)| <= 1.
- Best case performance O(n log n).
- Average case performance O(n log n).
- Memory space: O(n), using in-place partitioning.



Depth of the call stack: 1 + log n

Complexity Of Quick Sort

- Best case performance O(n log n).
- Average case performance O(n log n).

- Stop recursion when a set of number of elements <= m, e.g., m = 15. Sort the set using insertion sort.

Exercise: Implement Quick Sort

```
void quickSort(int n, int *a) {
   if n <= 1  return;
   pivot = computePivot(n, a);
   int Ln = split(n, a, pivot);
   quickSort(Ln, a);
   quicksort(n-Ln-1, &a[Ln+1]);
}
```

```
void quickSort(int n, int *a)
{
    if (n <= 1)   return;
    int pIndex = n/2;
    int pivot = a[pIndex]; // middle
    int Ln = split(n, a, pivot, pIndex);
    quickSort(Ln, a);
    quickSort(n-Ln-1, &a[Ln+1]);
}
//Ln : number of elements in the left part.
```

n-Ln-1 = 11-6-1 = 4

1	2	5	4	0	3	6	8	9	9	7

Ln=6

a[Ln+1]

Merge Sort

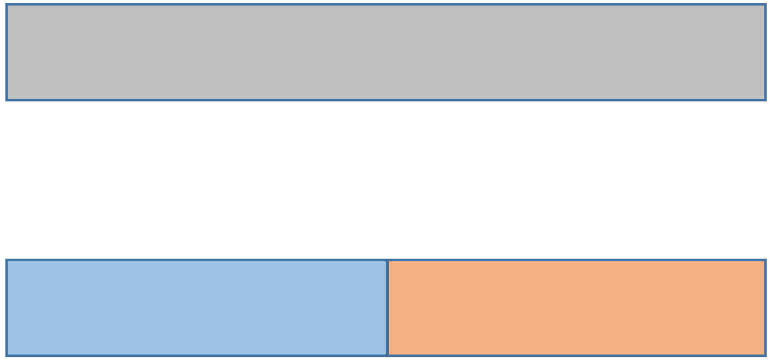
- Partition the n > 1 elements into two near equal smaller sets.
- Example: m = (n-1)/2. {0…m}, {m+1,…,n-1}.
- Each of the two smaller parts is sorted recursively.
- The sorted smaller parts are combined, called merging.
- Complexity is O(n log n).
- Can be implemented in a non-recursive manner.

Merge Sort

- Partition the $n > 1$ elements into two near equal smaller sets.
- Example: $m = (n-1)/2$. $\{0…m\}$, $\{m+1,…,n-1\}$.
- Each of the two smaller parts is sorted recursively.
- The sorted smaller parts are combined, called merging.
- Complexity is $O(n \log n)$.
- Can be implemented in a non-recursive manner.

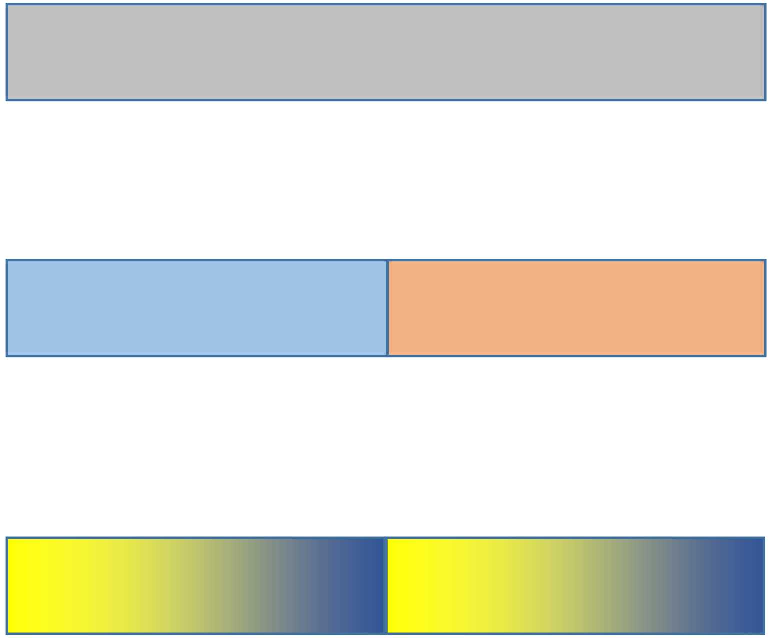
Merge Sort

- Partition the n > 1 elements into two near equal smaller sets.
- Example: m = (n-1)/2. {0…m}, {m+1,…,n-1}.
- Each of the two smaller parts is sorted recursively.
- The sorted smaller parts are combined, called merging.
- Complexity is O(n log n).
- Can be implemented in a non-recursive manner.



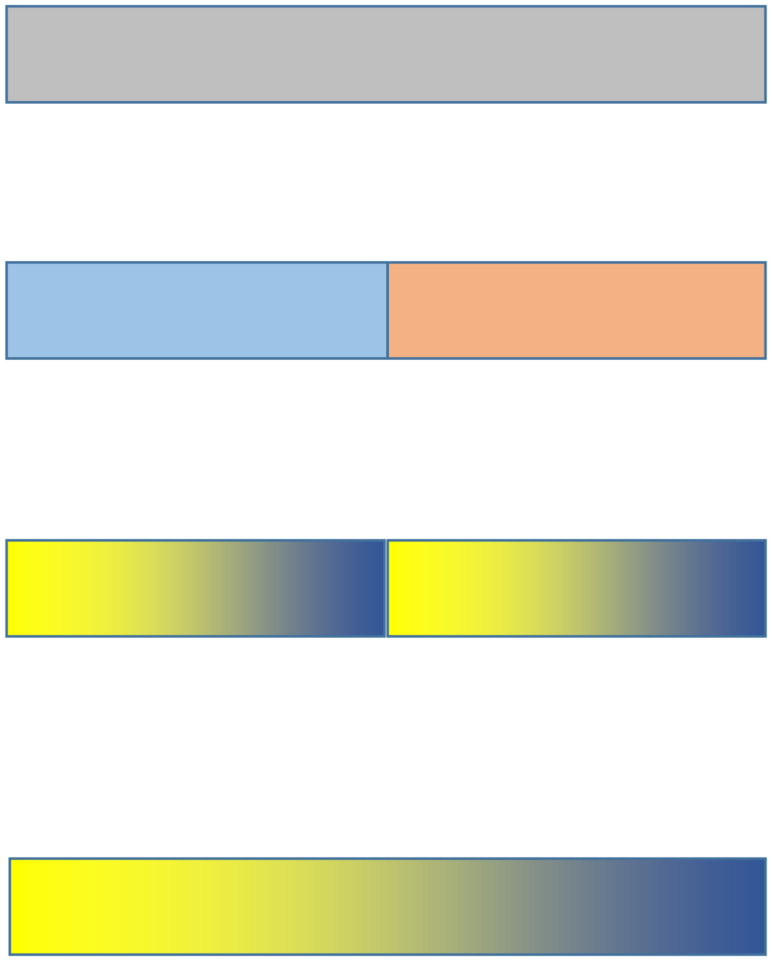
Merge Sort

- Partition the n > 1 elements into two near equal smaller sets.
- Example: m = (n-1)/2. {0…m}, {m+1,…,n-1}.
- Each of the two smaller parts is sorted recursively.
- The sorted smaller parts are combined, called merging.
- Complexity is O(n log n).
- Can be implemented in a non-recursive manner.



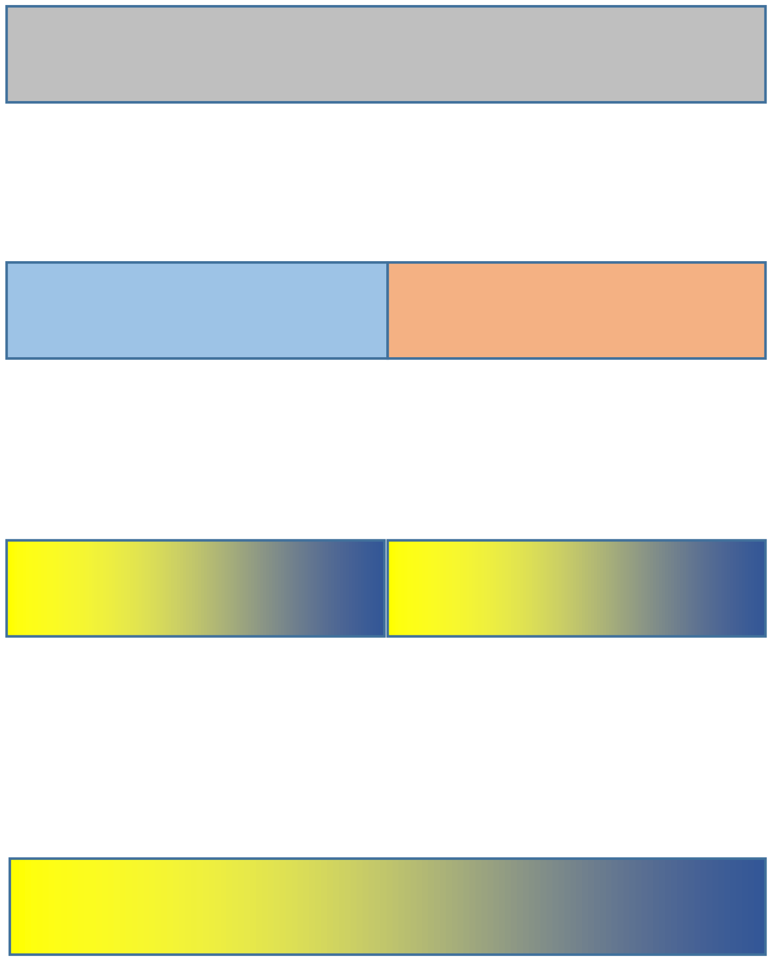
Merge Sort

- Partition the n > 1 elements into two near equal smaller sets.
- Example: m = (n-1)/2. {0…m}, {m+1,…,n-1}.
- Each of the two smaller parts is sorted recursively.
- The sorted smaller parts are combined, called merging.
- Complexity is O(n log n).
- Can be implemented in a non-recursive manner.



Merge Sort

- Partition the n > 1 elements into two near equal smaller sets.
- Example: m = (n-1)/2. {0…m}, {m+1,…,n-1}.
- Each of the two smaller parts is sorted recursively.
- The sorted smaller parts are combined, called merging.
- Complexity is O(n log n).
- Can be implemented in a non-recursive manner.



Merging Two Sorted Lists

- C = { }. C stores the final sorted elements.
- Given two sorted lists, scan the elements of A and B from the smallest to the largest. Place the smaller one into C each time.

Example:

A = { 1, 3, 4, 5, 6, 6 , 7 }
B = { 1, 2, 7, 9 }
C = { }

Merging Two Sorted Lists

- C = { }. C stores the final sorted elements.
- Given two sorted lists, scan the elements of A and B from the smallest to the largest. Place the smaller one into C each time.
- Use three counters: a, b, and c.

If A(a) <= B(b), C(c) = A(a).
Otherwise C(c) = B(b);

Example:

A = { 1, 3, 4, 5, 6, 6 , 7 }
B = { 1, 2, 7, 9 }
C = { }

Merging Two Sorted Lists

- C = { }. C stores the final sorted elements.
- Given two sorted lists, scan the elements of A and B from the smallest to the largest. Place the smaller one into C each time.
- Use three counters: a, b, and c.

If A(a) <= B(b), C(c) = A(a).
Otherwise C(c) = B(b);

Example:

A = { 1, 3, 4, 5, 6, 6 , 7 }, a = 0
B = { 1, 2, 7, 9 }, b = 0
C = { }, c = 0

Merging Two Sorted Lists

- C = { }. C stores the final sorted elements.
- Given two sorted lists, scan the elements of A and B from the smallest to the largest. Place the smaller one into C each time.
- Use three counters: a, b, and c.

If A(a) <= B(b), C(c) = A(a).
Otherwise C(c) = B(b);

Example:

A = { 1, 3, 4, 5, 6, 6 , 7 }, a = 0 => a = 1

B = { 1, 2, 7, 9 }, b = 0

C = { 1 }, c = 0 => c = 1

Merging Two Sorted Lists

- C = { }. C stores the final sorted elements.
- Given two sorted lists, scan the elements of A and B from the smallest to the largest. Place the smaller one into C each time.
- Use three counters: a, b, and c.

If A(a) <= B(b), C(c) = A(a).
Otherwise C(c) = B(b);

Example:

A = { 1, 3, 4, 5, 6, 6 , 7 }, a = 1

B = { 1, 2, 7, 9 }, b = 0

C = { 1 }, c = 0 => c = 1

Merging Two Sorted Lists

- C = { }. C stores the final sorted elements.
- Given two sorted lists, scan the elements of A and B from the smallest to the largest. Place the smaller one into C each time.
- Use three counters: a, b, and c.

If A(a) <= B(b), C(c) = A(a).
Otherwise C(c) = B(b);

Example:

A = { 1, 3, 4, 5, 6, 6 , 7 }, a = 1

B = { 1, 2, 7, 9 }, b = 0 => b = 1

C = { 1, 1 }, c = 1 => c = 2

Merging Two Sorted Lists

- C = { }. C stores the final sorted elements.
- Given two sorted lists, scan the elements of A and B from the smallest to the largest. Place the smaller one into C each time.
- Use three counters: a, b, and c.

If A(a) <= B(b), C(c) = A(a).
Otherwise C(c) = B(b);

Example:

A = { 1, 3, 4, 5, 6, 6 , 7 }, a = 1

B = { 1, 2, 7, 9 }, b = 1

C = { 1, 1 }, c = 2

Merging Two Sorted Lists

- C = { }. C stores the final sorted elements.
- Given two sorted lists, scan the elements of A and B from the smallest to the largest. Place the smaller one into C each time.
- Use three counters: a, b, and c.

If A(a) <= B(b), C(c) = A(a).
Otherwise C(c) = B(b);

Example:

A = { 1, 3, 4, 5, 6, 6 , 7 }, a = 1

B = { 1, 2, 7, 9 }, b = 1 => b = 2

C = { 1, 1, 2 }, c = 2 => c = 3;

Merging Two Sorted Lists

- C = { }. C stores the final sorted elements.
- Given two sorted lists, scan the elements of A and B from the smallest to the largest. Place the smaller one into C each time.
- Use three counters: a, b, and c.

If A(a) <= B(b), C(c) = A(a).
Otherwise C(c) = B(b);

Example:

A = { 1, 3, 4, 5, 6, 6 , 7 }, a = 1

B = { 1, 2, 7, 9 }, b = 2

C = { 1, 1, 2 }, c = 3;

Merging Two Sorted Lists

- C = { }. C stores the final sorted elements.
- Given two sorted lists, scan the elements of A and B from the smallest to the largest. Place the smaller one into C each time.
- Use three counters: a, b, and c.

If A(a) <= B(b), C(c) = A(a).
Otherwise C(c) = B(b);

Example:

A = { 1, 3, 4, 5, 6, 6 , 7 }, a = 1 => a = 2

B = { 1, 2, 7, 9 }, b = 2

C = { 1, 1, 2, 3 }, c = 3 => c = 4;

Merging Two Sorted Lists

- C = { }. C stores the final sorted elements.
- Given two sorted lists, scan the elements of A and B from the smallest to the largest. Place the smaller one into C each time.
- Use three counters: a, b, and c.

If A(a) <= B(b), C(c) = A(a).
Otherwise C(c) = B(b);

Example:

A = { 1, 3, 4, 5, 6, 6 , 7 }, a = 2
B = { 1, 2, 7, 9 }, b = 2
C = { 1, 1, 2, 3 }, c = 4;

Merging Two Sorted Lists

- C = { }. C stores the final sorted elements.
- Given two sorted lists, scan the elements of A and B from the smallest to the largest. Place the smaller one into C each time.
- Use three counters: a, b, and c.

If A(a) <= B(b), C(c) = A(a).
Otherwise C(c) = B(b);

Example:

A = { 1, 3, 4, 5, 6, 6 , 7 }, a = 2 => a = 3
B = { 1, 2, 7, 9 }, b = 2
C = { 1, 1, 2, 3, 4 }, c = 4 => c = 5;

Merging Two Sorted Lists

- C = { }. C stores the final sorted elements.
- Given two sorted lists, scan the elements of A and B from the smallest to the largest. Place the smaller one into C each time.
- Use three counters: a, b, and c.

If A(a) <= B(b), C(c) = A(a).
Otherwise C(c) = B(b);

Example:

A = { }

B = { 1, 2, 7, 9 }

C = { }

If a list is empty, copy another list to C.

Merge to sorted lists

```
void merge(const vector &A, int ia, const vector &B, int ib, vector &C) {

}
```

A = { }

B = { 1, 2, 7, 9 }

C = { }

Merge to sorted lists

```
void merge(const vector &A, int ia, const vector &B, int ib, vector &C) {
        if (ia >= A.size()) { append(B, ib, C); return;}
        if (ib >= B.size()) { append(A, ia, C); return;}

        int a = A[ia];
        int b = B[ib];
        if (a>b) {
                C.push_back(b);
                ib++;
                merge(A, ia, B, ib, C);
        } else
}
```

A = { }

B = { 1, 2, 7, 9 }

C = { }

Merge Sort

{7, 5, 18, 5, 1, 13, 7, 11, 15, 3, 8, 6, 1}

Merge Sort: Top down: Splitting

{7, 5, 18, 5, 1, 13, 7, 11, 15, 3, 8, 6, 1}

{7, 5, 18, 5, 1, 13, 7}

{11, 15, 3, 8, 6, 1}

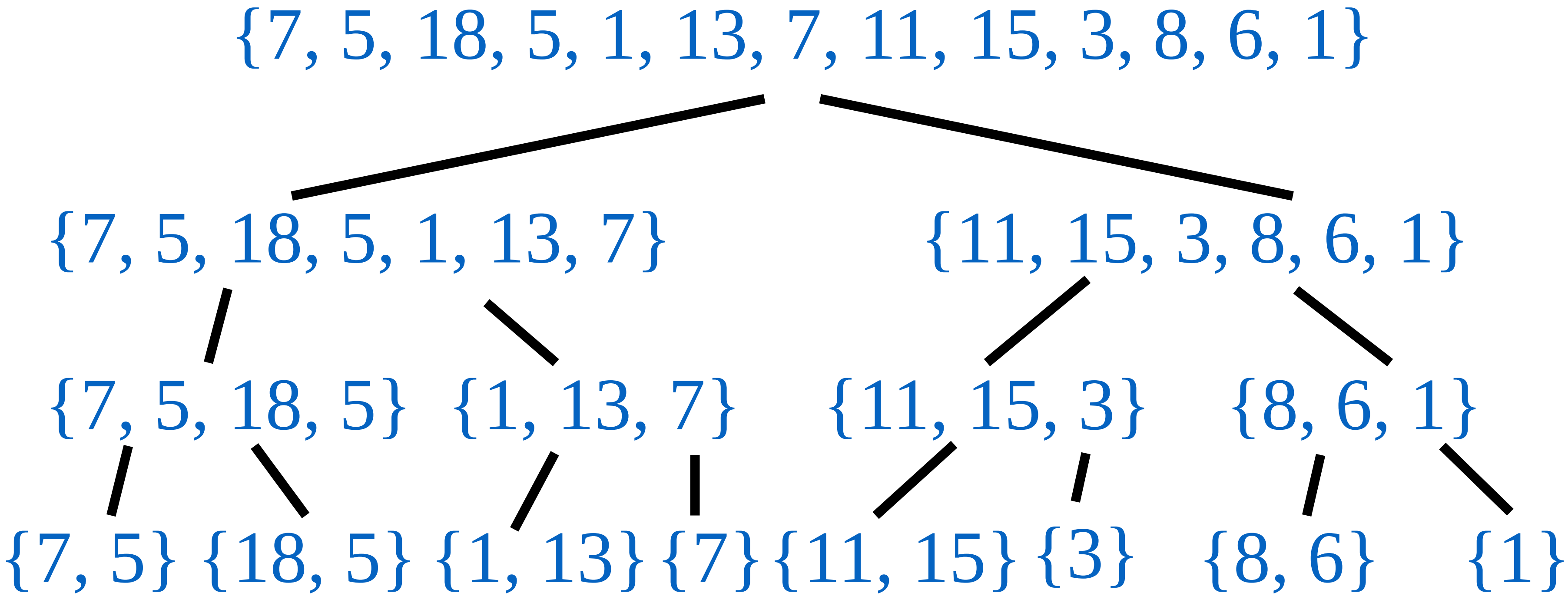
Merge Sort: Top down: Splitting

{7, 5, 18, 5, 1, 13, 7, 11, 15, 3, 8, 6, 1}

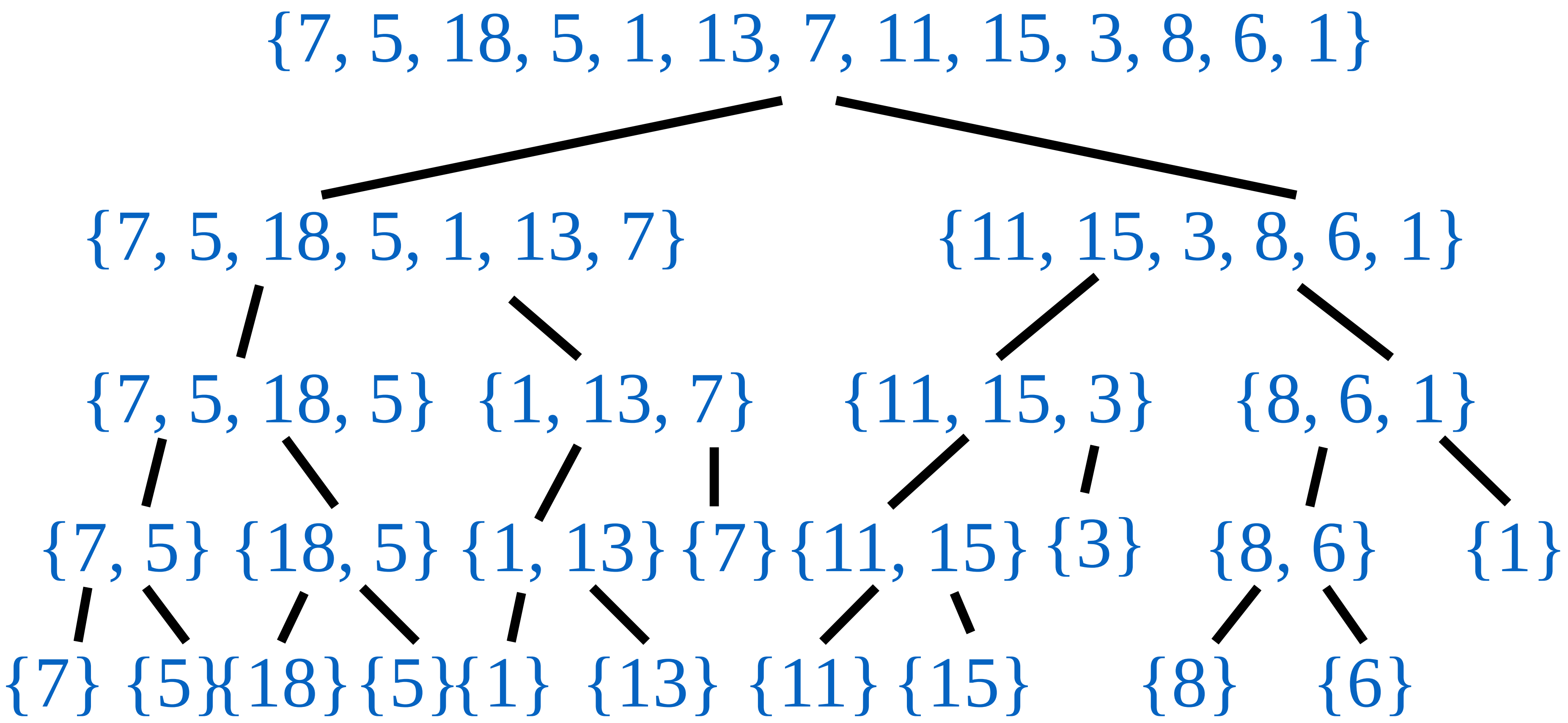
{7, 5, 18, 5, 1, 13, 7} {11, 15, 3, 8, 6, 1}

{7, 5, 18, 5} {1, 13, 7} {11, 15, 3} {8, 6, 1}

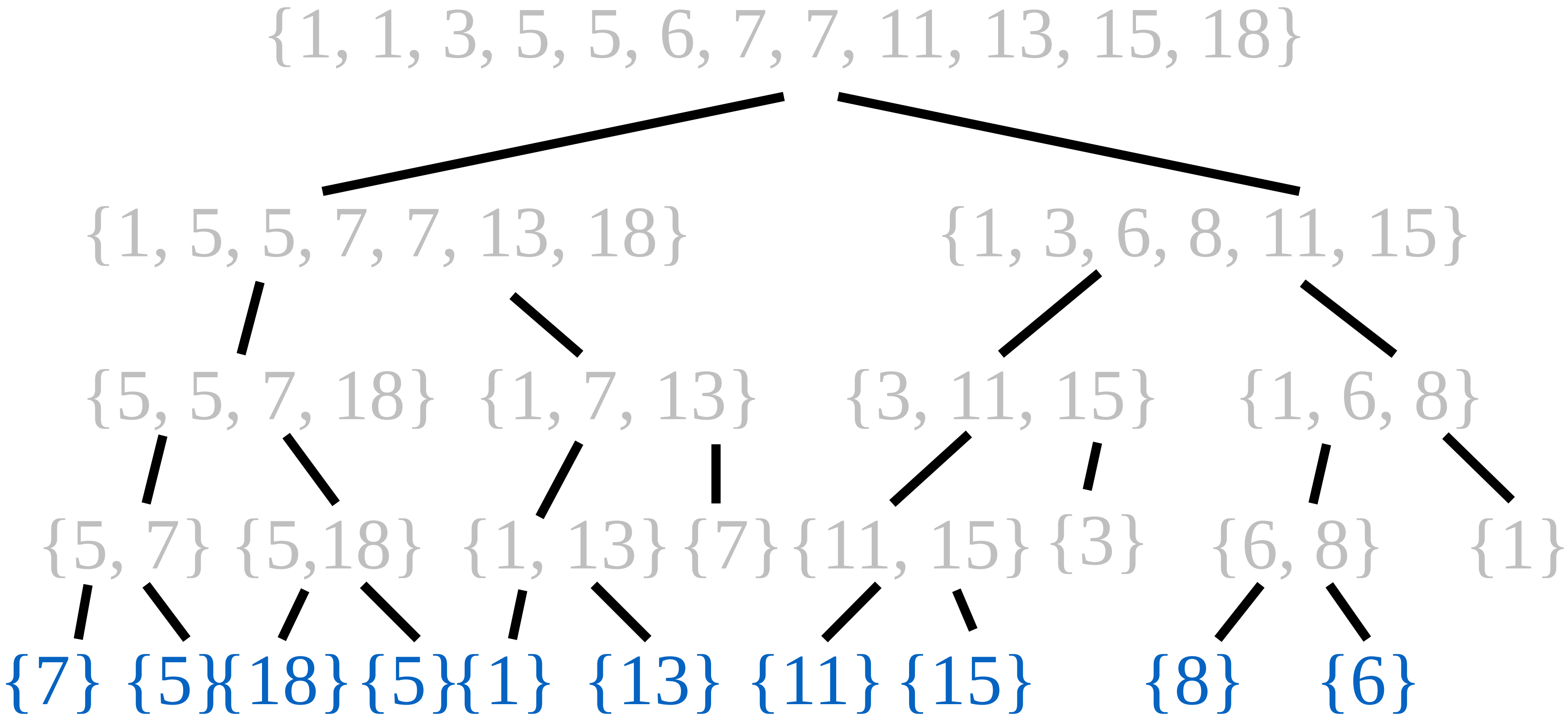
Merge Sort: Top down: Splitting



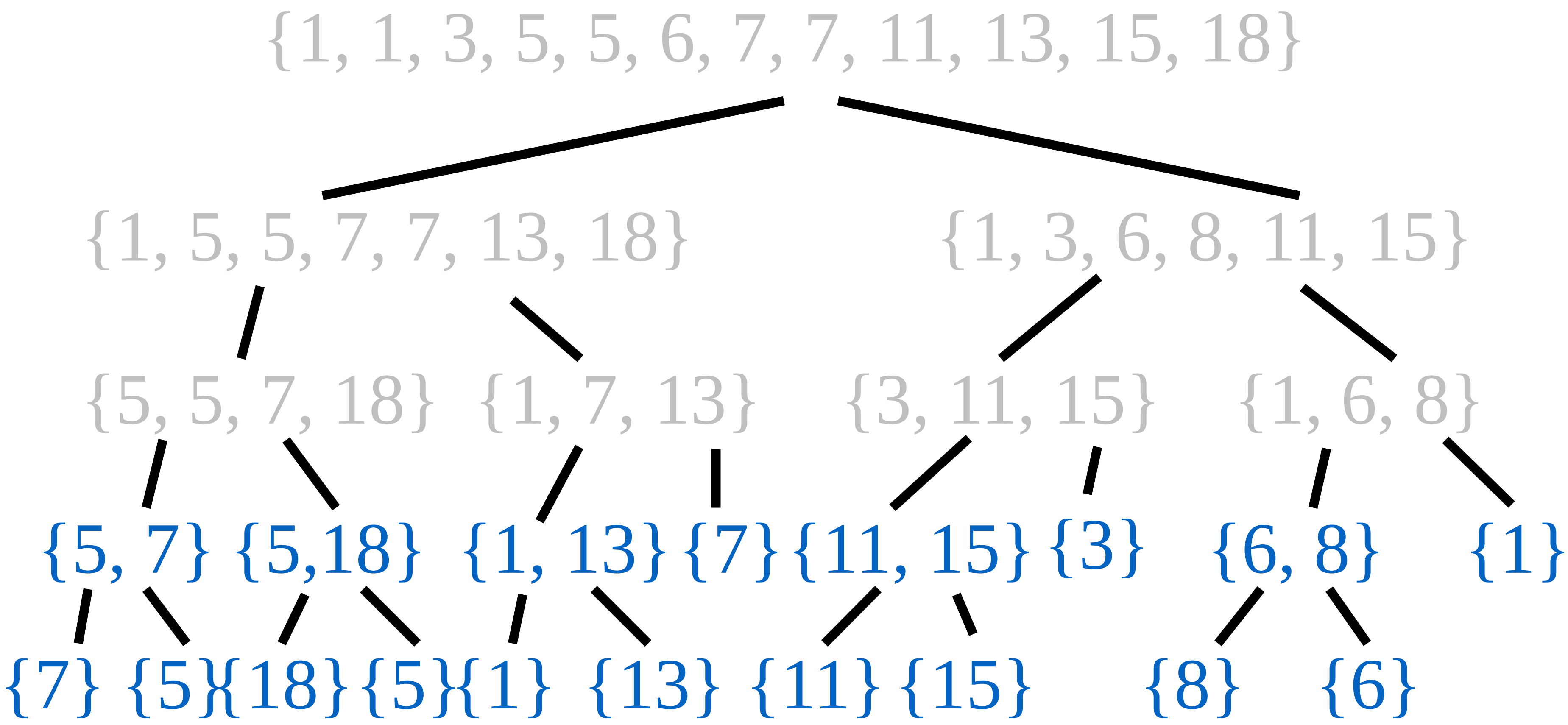
Merge Sort: Top down: Splitting



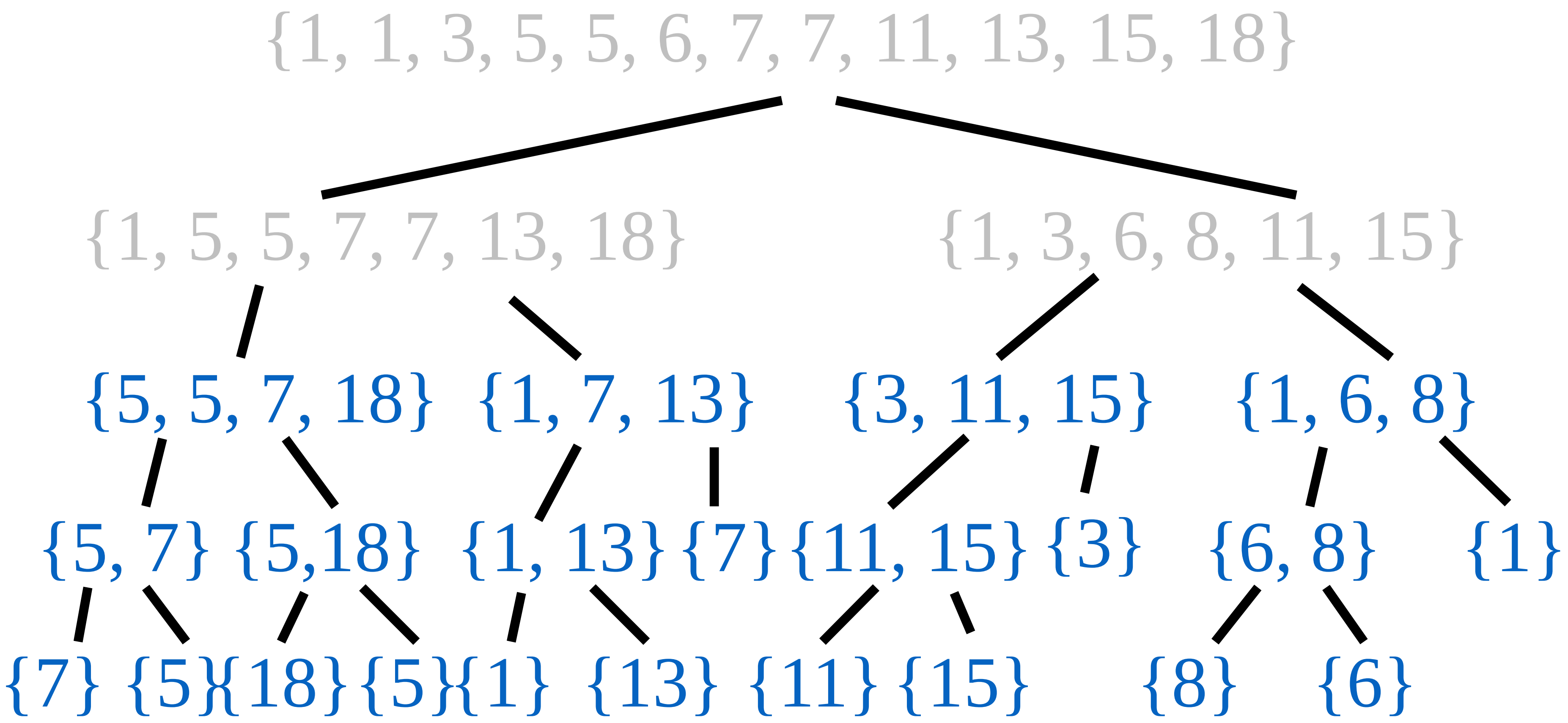
Merge Sort: Bottom-up: Merging



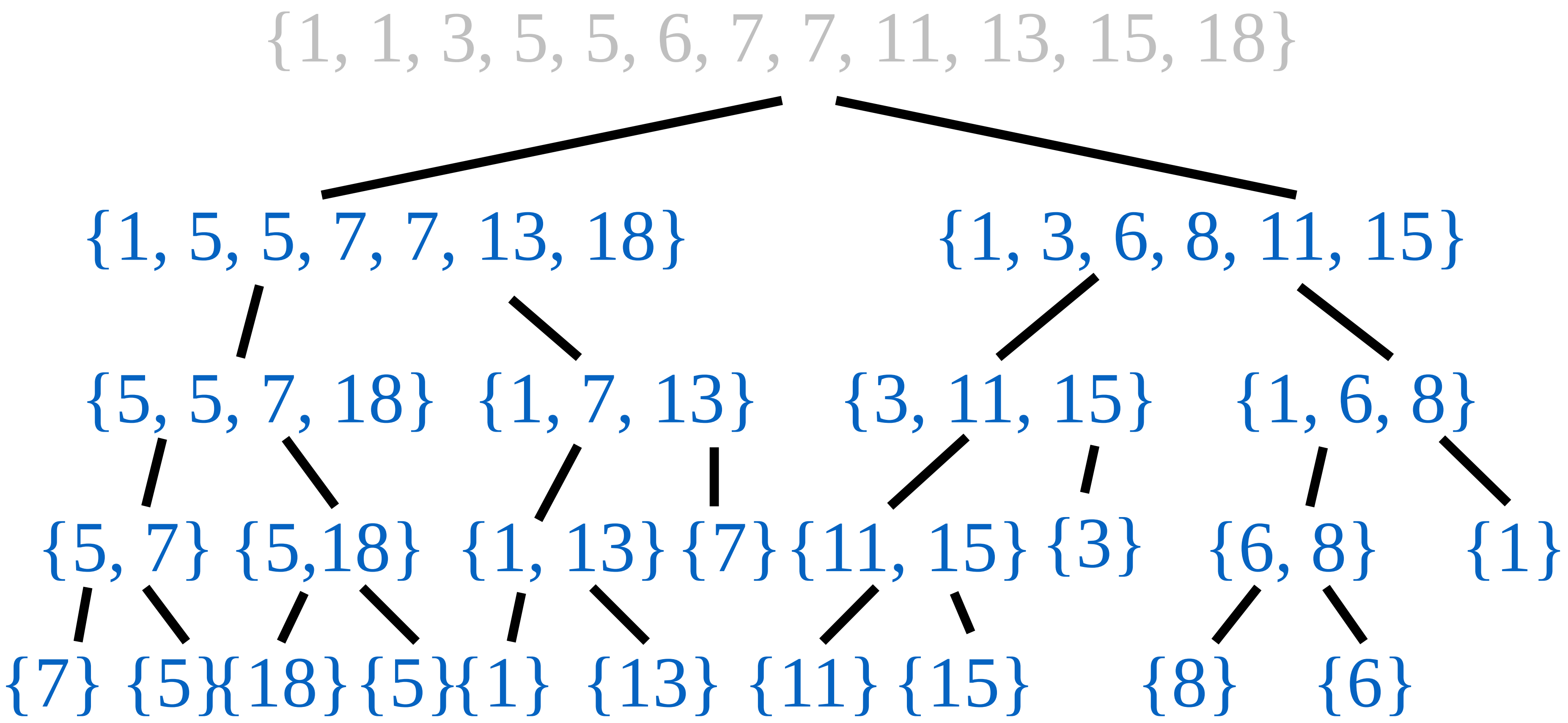
Merge Sort: Bottom-up: Merging



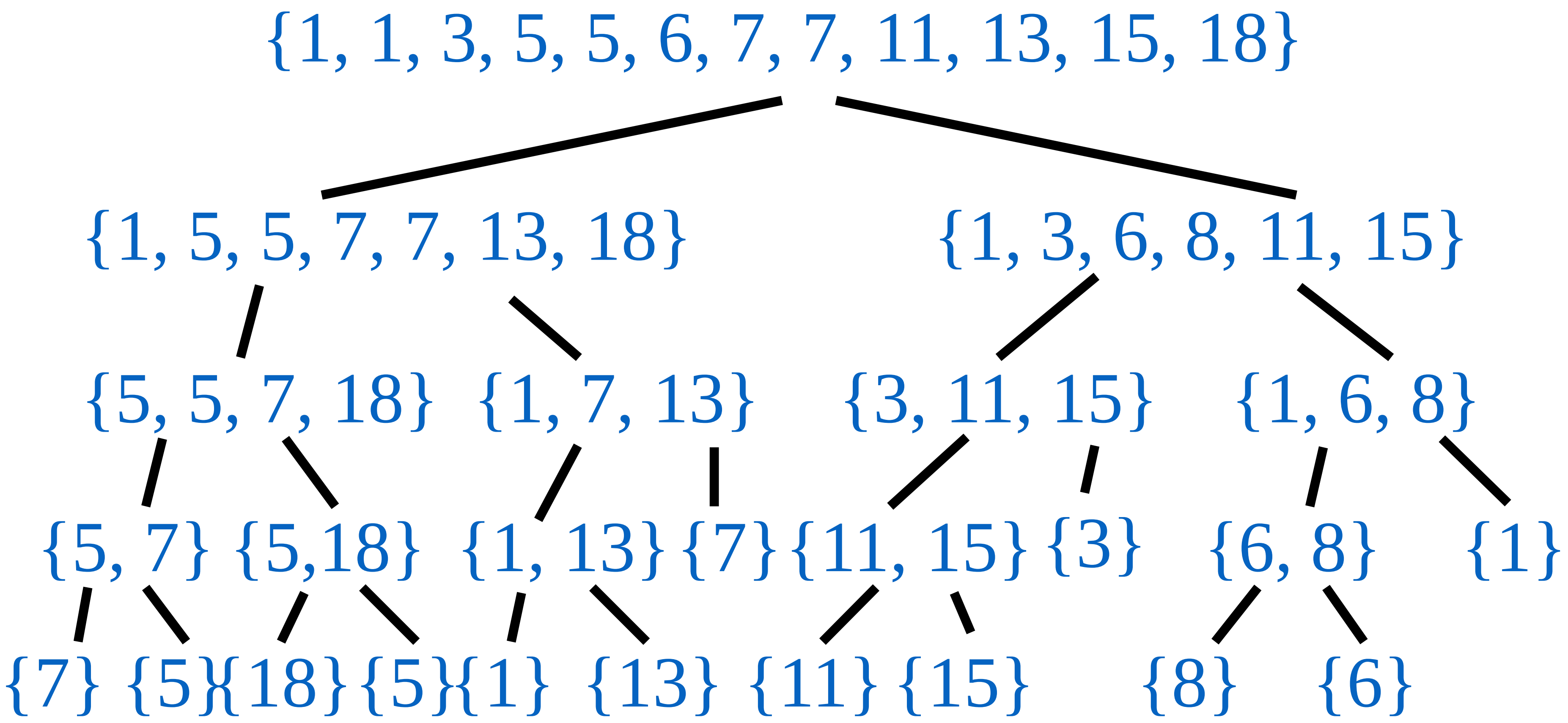
Merge Sort: Bottom-up: Merging



Merge Sort: Bottom-up: Merging



Merge Sort: Bottom-up: Merging



Nonrecursive Version

- Eliminate downward pass. The download pass only computes the required indices for splitting the elements. If we can compute these required indices beforehand, we can skip the downward pass.
- Start with sorted lists of size one.
- Do pairwise merging of the sorted lists as in the upward pass.

Nonrecursive Merge Sort

{7} {5} {18} {5} {1} {13} {7} {11} {15} {3} {8} {6} {1}

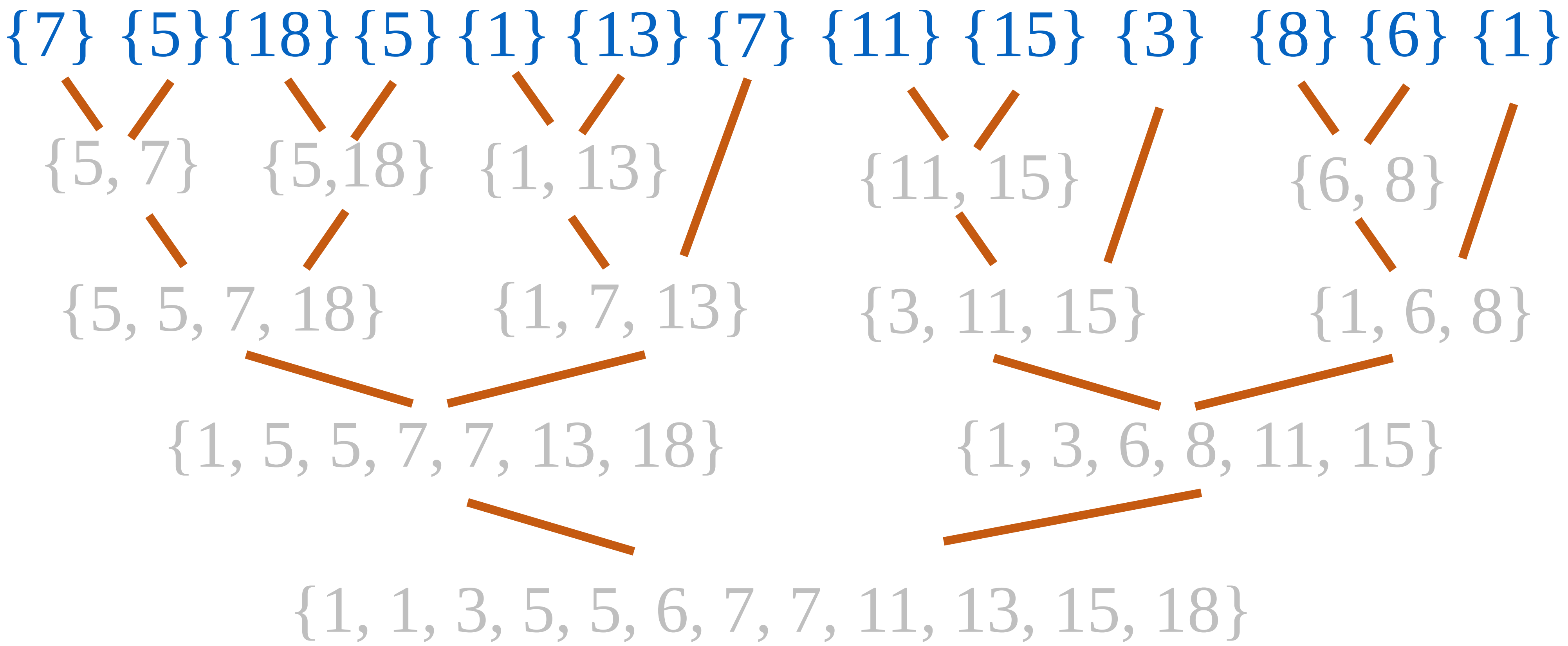
{5, 7} {5,18} {1, 13} {11, 15} {6, 8}

{5, 5, 7, 18} {1, 7, 13} {3, 11, 15} {1, 6, 8}

{1, 5, 5, 7, 7, 13, 18} {1, 3, 6, 8, 11, 15}

{1, 1, 3, 5, 5, 6, 7, 7, 11, 13, 15, 18}

Nonrecursive Merge Sort



Nonrecursive Merge Sort

{7} {5} {18} {5} {1} {13} {7} {11} {15} {3} {8} {6} {1}

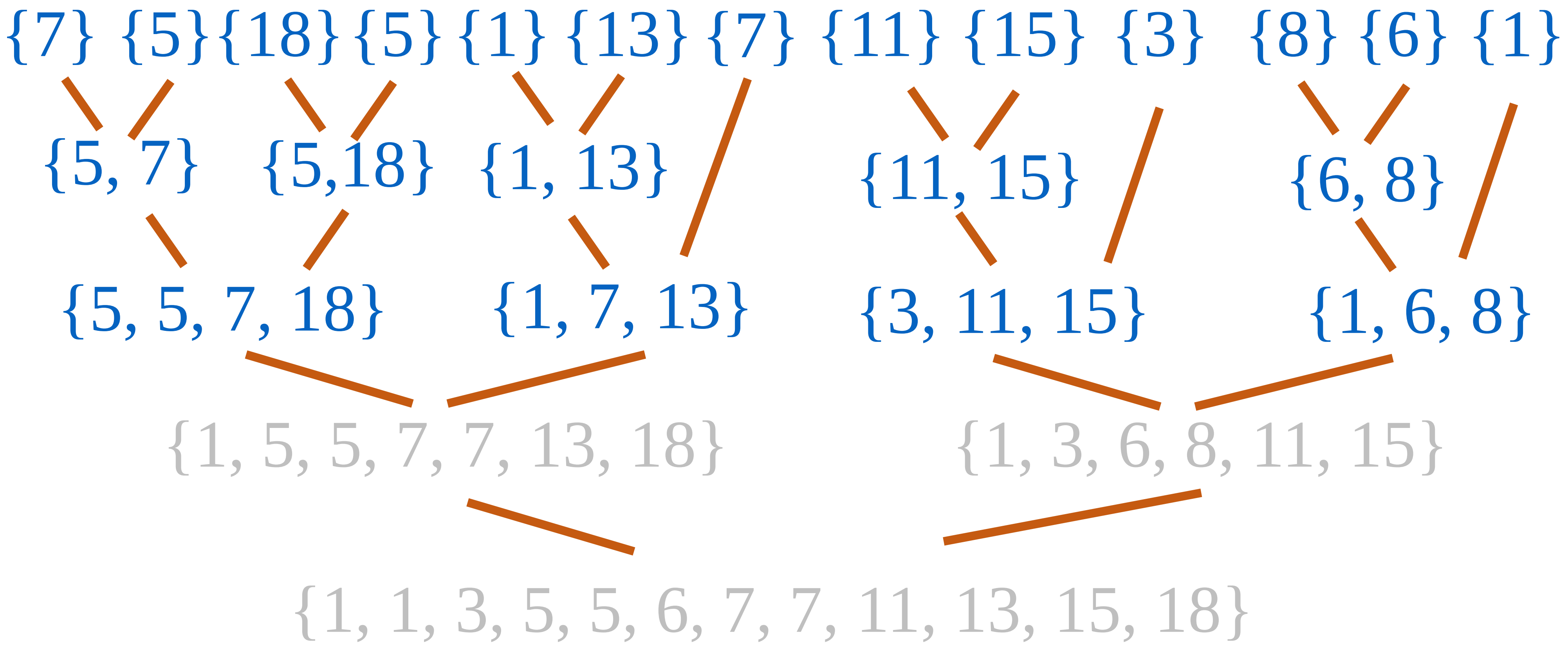
{5, 7} {5,18} {1, 13} {11, 15} {6, 8}

{5, 5, 7, 18} {1, 7, 13} {3, 11, 15} {1, 6, 8}

{1, 5, 5, 7, 7, 13, 18} {1, 3, 6, 8, 11, 15}

{1, 1, 3, 5, 5, 6, 7, 7, 11, 13, 15, 18}

Nonrecursive Merge Sort



Nonrecursive Merge Sort

{7} {5}{18}{5} {1} {13} {7} {11} {15} {3} {8} {6} {1}

{5, 7} {5,18} {1, 13} {11, 15} {6, 8}

{5, 5, 7, 18} {1, 7, 13} {3, 11, 15} {1, 6, 8}

{1, 5, 5, 7, 7, 13, 18} {1, 3, 6, 8, 11, 15}

{1, 1, 3, 5, 5, 6, 7, 7, 11, 13, 15, 18}

Nonrecursive Merge Sort