Sorting lists — work and span revisited

15-150

Lecture 7: September 17, 2024

Stephanie Balzer
Carnegie Mellon University

Announcement: midterm I

Announcement: midterm I

When and where:

- Thursday, **September 26**, **11:00am—12:20pm**.
- **PH 100** (we may get a second room for more space, stay tuned).

Announcement: midterm I

Be on time; next lecture starts at 12:30pm!

When and where:

- Thursday, **September 26**, **11:00am—12:20pm**.
- **PH 100** (we may get a second room for more space, stay tuned).

Announcement: midterm I

Be on time; next lecture starts at 12:30pm!

When and where:

- Thursday, **September 26**, **11:00am—12:20pm**.
- **PH 100** (we may get a second room for more space, stay tuned).

Scope:

- Lectures: 1—8.
- Labs: 1—4 and midterm review section of Lab 5.
- Assignments: Basics, Induction, and Datatypes.

Announcement: midterm I

Be on time; next lecture starts at 12:30pm!

When and where:

- Thursday, **September 26**, **11:00am—12:20pm**.
- **PH 100** (we may get a second room for more space, stay tuned).

Scope:

- Lectures: 1—8.
- Labs: 1—4 and midterm review section of Lab 5.
- Assignments: Basics, Induction, and Datatypes.

What you may have on your desk:

- Writing utensils, we provide paper, something to drink/eat, tissues.
- 8.5” x 11” cheatsheet (back and front), handwritten or typeset.
- No cell phones, laptops, or any other smart devices.

Last week

Last week

In week 2 we discovered (and exploited) the correspondence between programs and **proofs**.

Last week

In week 2 we discovered (and exploited) the correspondence between programs and **proofs**.

recursive call corresponds to inductive hypothesis

Last week

In week 2 we discovered (and exploited) the correspondence between programs and **proofs**.

recursive call corresponds to inductive hypothesis

"programs as proofs"!

Last week

In week 2 we discovered (and exploited) the correspondence between programs and **proofs**.

recursive call corresponds to inductive hypothesis

Last week

In week 2 we discovered (and exploited) the correspondence between programs and **proofs**.

recursive call corresponds to inductive hypothesis

In week 3 we discovered (and exploited) the correspondence between programs and **asymptotic analysis**.

Last week

- In week 2 we discovered (and exploited) the correspondence between programs and **proofs**.
  - recursive call corresponds to inductive hypothesis
- In week 3 we discovered (and exploited) the correspondence between programs and **asymptotic analysis**.
  - recursive calls give rise to recurrence

Last week

- In week 2 we discovered (and exploited) the correspondence between programs and **proofs**.
  - recursive call corresponds to inductive hypothesis
- In week 3 we discovered (and exploited) the correspondence between programs and **asymptotic analysis**.
  - recursive calls give rise to recurrence
  - closed form solutions of recurrences for **work** and **span**

Last week

In week 2 we discovered (and exploited) the correspondence between programs and **proofs**.

- recursive call corresponds to inductive hypothesis

In week 3 we discovered (and exploited) the correspondence between programs and **asymptotic analysis**.

- recursive calls give rise to recurrence
- closed form solutions of recurrences for **work** and **span**

“programs as recurrences”!

Last week

- In week 2 we discovered (and exploited) the correspondence between programs and **proofs**.
  - recursive call corresponds to inductive hypothesis
- In week 3 we discovered (and exploited) the correspondence between programs and **asymptotic analysis**.
  - recursive calls give rise to recurrence
  - closed form solutions of recurrences for **work** and **span**

Last week

- In week 2 we discovered (and exploited) the correspondence between programs and **proofs**.
  - recursive call corresponds to inductive hypothesis
- In week 3 we discovered (and exploited) the correspondence between programs and **asymptotic analysis**.
  - recursive calls give rise to recurrence
  - closed form solutions of recurrences for **work** and **span**
- This week, we revisit work and span analysis for **sorting**!

Last week

- In week 2 we discovered (and exploited) the correspondence between programs and **proofs**.
  - recursive call corresponds to inductive hypothesis
- In week 3 we discovered (and exploited) the correspondence between programs and **asymptotic analysis**.
  - recursive calls give rise to recurrence
  - closed form solutions of recurrences for **work** and **span**
- This week, we revisit work and span analysis for **sorting**!
  - today: sorting **lists**

Last week

- In week 2 we discovered (and exploited) the correspondence between programs and **proofs**.
  - recursive call corresponds to inductive hypothesis
- In week 3 we discovered (and exploited) the correspondence between programs and **asymptotic analysis**.
  - recursive calls give rise to recurrence
  - closed form solutions of recurrences for **work** and **span**
- This week, we revisit work and span analysis for **sorting**!
  - today: sorting **lists**
  - Thursday: sorting **binary trees**

Last week

- In week 2 we discovered (and exploited) the correspondence between programs and **proofs**.
  - recursive call corresponds to inductive hypothesis
- In week 3 we discovered (and exploited) the correspondence between programs and **asymptotic analysis**.
  - recursive calls give rise to recurrence
  - closed form solutions of recurrences for **work** and **span**
- This week, we revisit work and span analysis for **sorting**!
  - today: sorting **lists**
  - Thursday: sorting **binary trees**

mergesort

Sorting

Sorting

Useful datatype:

```
datatype order = LESS | EQUAL | GREATER
```

Sorting

Useful datatype:

```
datatype order = LESS | EQUAL | GREATER
```

Eg:

```
Int.compare : int * int -> order
String.compare : string * string -> order
```

Sorting

Useful datatype:

```
datatype order = LESS | EQUAL | GREATER
```

Eg:

```
Int.compare : int * int -> order
String.compare : string * string -> order
```

More generally, what we would like, for some type t:

```
compare : t * t -> order
```

Sorting

Useful datatype:

```
datatype order = LESS | EQUAL | GREATER
```

Eg:

```
Int.compare : int * int -> order
String.compare : string * string -> order
```

More generally, what we would like, for some type t:

```
compare : t * t -> order
```

→ But let's focus on comparing integers for now.

Sorting

Useful datatype:

```
datatype order = LESS | EQUAL | GREATER
```

Sorting

Useful datatype:

```
datatype order = LESS | EQUAL | GREATER
```

➡ What does it mean to be sorted?

Sorting

Useful datatype:

```
datatype order = LESS | EQUAL | GREATER
```

What does it mean to be sorted?

Eg, for lists of integers:

Sorting

Useful datatype:

```
datatype order = LESS | EQUAL | GREATER
```

What does it mean to be sorted?

Eg, for lists of integers:

A list of integers is **sorted** iff each integer in the list is **no greater** than all integers that occur to its **right**.

Sorting

Useful datatype:

```
datatype order = LESS | EQUAL | GREATER
```

What does it mean to be sorted?

Eg, for lists of integers:

A list of integers is **sorted** iff each integer in the list is **no greater** than all integers that occur to its **right**.

```
[..., x,..., y,...]
```

Sorting

Useful datatype:

```
datatype order = LESS | EQUAL | GREATER
```

What does it mean to be sorted?

Eg, for lists of integers:

A list of integers is **sorted** iff each integer in the list is **no greater** than all integers that occur to its **right**.

```
       LESS | EQUAL
[..., x,..., y,...]
```

Warm-up: insertion sort for int lists

Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)
```

Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list =
  |                 =
```

Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  |                    =
```

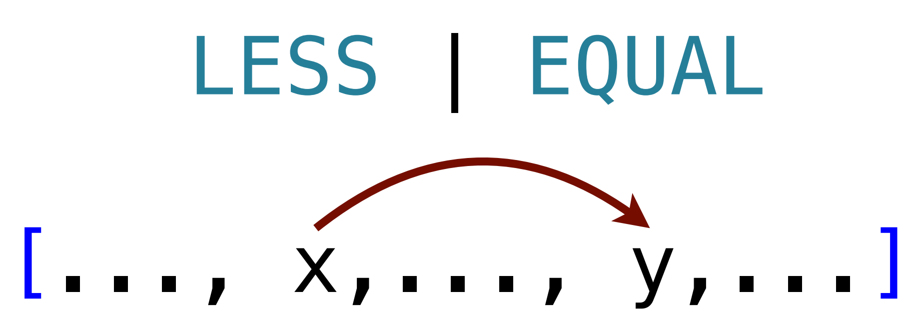
Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) =
```

Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) =
```

Remember our definition of a sorted list:



Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case                of


                                                     )
```

Remember our definition of a sorted list:

```
             LESS | EQUAL
[..., x,..., y,...]
```

Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of


                                                  )
```

Remember our definition of a sorted list:

```
                    LESS | EQUAL
              [..., x,..., y,...]
```

Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                        GREATER =>

                                                      )
```

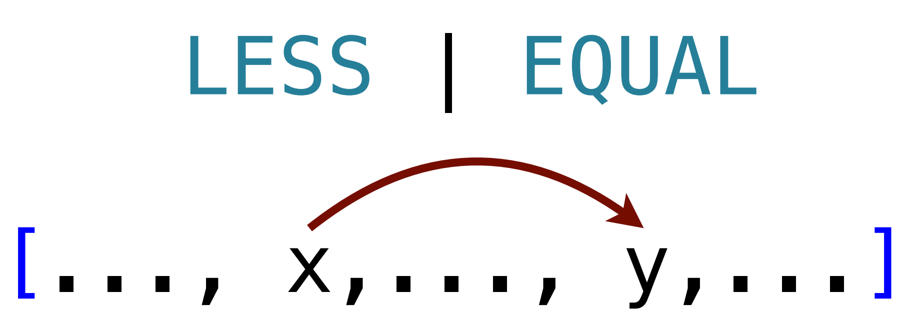
Remember our definition of a sorted list:

```
             LESS | EQUAL
        [..., x,..., y,...]
```

Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                                                    )
```

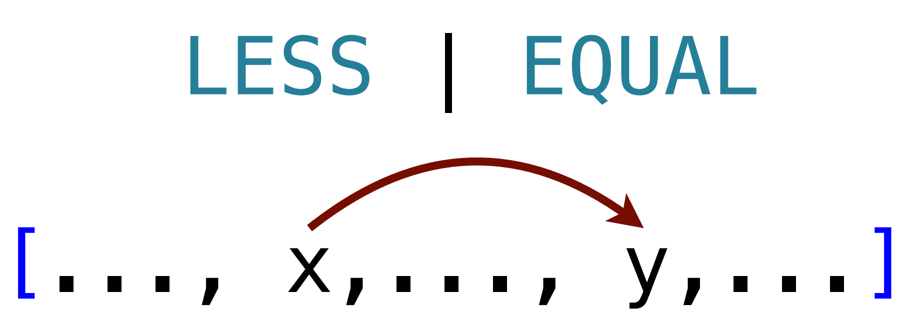
Remember our definition of a sorted list:



Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                        |   _     =>                )
```

Remember our definition of a sorted list:



Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                         |  _     => x::y::L       )
```

Remember our definition of a sorted list:

```
             LESS | EQUAL
[..., x,..., y,...]
```

Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                        |  _      => x::y::L      )
```

Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                         GREATER => y::ins(x, L)
                       |  _      => x::y::L      )

(* isort : int list -> int list
   REQUIRES: true
   ENSURES: isort(L) evaluates to a sorted permutation of L
*)
```

Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                        |  _      => x::y::L       )

(* isort : int list -> int list
   REQUIRES: true
   ENSURES: isort(L) evaluates to a sorted permutation of L
*)

fun isort ([ ] : int list) : int list =
  |
```

Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                        GREATER => y::ins(x, L)
                      |  _      => x::y::L      )

(* isort : int list -> int list
   REQUIRES: true
   ENSURES: isort(L) evaluates to a sorted permutation of L
*)

fun isort ([ ] : int list) : int list = [ ]
  |
```

Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                         GREATER => y::ins(x, L)
                       |  _      => x::y::L       )

(* isort : int list -> int list
   REQUIRES: true
   ENSURES: isort(L) evaluates to a sorted permutation of L
*)

fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) =
```

Warm-up: insertion sort for int lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x, L) evaluates to sorted permutation of x::L
*)

fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                        |  _      => x::y::L      )

(* isort : int list -> int list
   REQUIRES: true
   ENSURES: isort(L) evaluates to a sorted permutation of L
*)

fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work for insertion sort

Work for insertion sort

```
fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                          |  _    => x::y::L      )
```

Work for insertion sort

```
fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                        |  _      => x::y::L      )
```

Work: $W_{ins}(n)$ with n the list length.

Work for insertion sort

```
fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                        |  _      => x::y::L       )
```

Work: $W_{ins}(n)$ with n the list length.

Equations:

Work for insertion sort

```
fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                        |  _      => x::y::L        )
```

Work: $W_{ins}(n)$ with n the list length.

Equations:

$W_{ins}(0) =$

Work for insertion sort

```
fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                           GREATER => y::ins(x, L)
                         |  _      => x::y::L      )
```

Work: $W_{ins}(n)$ with n the list length.

Equations:

$W_{ins}(0) = c_0$

Work for insertion sort

```
fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                        |  _      => x::y::L      )
```

Work: $W_{ins}(n)$ with n the list length.

Equations:

$W_{ins}(0) = c_0$
$W_{ins}(n) =$

Work for insertion sort

```
fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                          |  _    => x::y::L       )
```

Work: $W_{ins}(n)$ with n the list length.

Equations:

$W_{ins}(0) = c_0$
$W_{ins}(n) = c_1 + W_{ins}(n-1)$, for first `case` clause

Work for insertion sort

```
fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                        |   _     => x::y::L       )
```

Work: $W_{ins}(n)$ with n the list length.

Equations:

$W_{ins}(0) = c_0$
$W_{ins}(n) = c_1 + W_{ins}(n-1)$, for first `case` clause
$W_{ins}(n) =$

Work for insertion sort

```
fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                         GREATER => y::ins(x, L)
                       | _       => x::y::L       )
```

Work: $W_{ins}(n)$ with n the list length.

Equations:

$W_{ins}(0) = c_0$
$W_{ins}(n) = c_1 + W_{ins}(n-1)$, for first `case` clause
$W_{ins}(n) = c_2$, for second `case` clause

Work for insertion sort

```
fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                        | _       => x::y::L       )
```

Work: $W_{ins}(n)$ with n the list length.

Equations:

$W_{ins}(0) = c_0$
$W_{ins}(n) = c_1 + W_{ins}(n-1)$, for first `case` clause
$W_{ins}(n) = c_2$, for second `case` clause

Consequently:

Work for insertion sort

```
fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                         GREATER => y::ins(x, L)
                       |   _     => x::y::L       )
```

Work: $W_{ins}(n)$ with n the list length.

Equations:

$W_{ins}(0) = c_0$
$W_{ins}(n) = c_1 + W_{ins}(n-1)$, for first `case` clause
$W_{ins}(n) = c_2$, for second `case` clause

Consequently: $W_{ins}(n)$ is $O(n)$.

Work for insertion sort

```
fun ins (x : int, [ ] : int list) : int list = [x]
  | ins (x, y::L) = (case compare(x, y) of
                          GREATER => y::ins(x, L)
                        |  _      => x::y::L      )
```

Work: $W_{ins}(n)$ with n the list length.

Equations:

$W_{ins}(0) = c_0$
$W_{ins}(n) = c_1 + W_{ins}(n-1)$, for first `case` clause
$W_{ins}(n) = c_2$, for second `case` clause

Consequently: $W_{ins}(n)$ is $O(n)$.

→ Note: no opportunity for parallelism.

Work for insertion sort

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work: $W_{isort}(n)$ with n the list length.

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work: $W_{isort}(n)$ with n the list length.

Equations:

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work: $W_{isort}(n)$ with n the list length.

Equations:

$W_{isort}(0)$

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work: $W_{isort}(n)$ with n the list length.

Equations:

$W_{isort}(0) = c_0$

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work: $W_{isort}(n)$ with n the list length.

Equations:

$W_{isort}(0) = c_0$

$W_{isort}(n)$

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work: $W_{isort}(n)$ with n the list length.

Equations:

$$W_{isort}(0) = c_0$$
$$W_{isort}(n) = c_1 + W_{isort}(n-1) + W_{ins}(n-1)$$

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work: $W_{isort}(n)$ with n the list length.

Equations:

$W_{isort}(0) = c_0$
$W_{isort}(n) = c_1 + W_{isort}(n-1) + W_{ins}(n-1)$

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work: $W_{isort}(n)$ with n the list length.

Equations:

$$W_{isort}(0) = c_0$$
$$W_{isort}(n) = c_1 + W_{isort}(n-1) + W_{ins}(n-1)$$

b/c spec asserts permutation

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work: $W_{isort}(n)$ with n the list length.

Equations:

$$W_{isort}(0) = c_0$$
$$W_{isort}(n) = c_1 + W_{isort}(n-1) + W_{ins}(n-1)$$

So:

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work: $W_{isort}(n)$ with n the list length.

Equations:

$$W_{isort}(0) = c_0$$
$$W_{isort}(n) = c_1 + W_{isort}(n-1) + W_{ins}(n-1)$$

So: $W_{isort}(n) \leq c_1 + c_2 \cdot n + W_{isort}(n-1)$

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work: $W_{isort}(n)$ with n the list length.

Equations:

$W_{isort}(0) = c_0$
$W_{isort}(n) = c_1 + W_{isort}(n-1) + W_{ins}(n-1)$

So: $W_{isort}(n) \leq c_1 + c_2 \cdot n + W_{isort}(n-1)$

Consequently:

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work: $W_{isort}(n)$ with n the list length.

Equations:

$$W_{isort}(0) = c_0$$
$$W_{isort}(n) = c_1 + W_{isort}(n-1) + W_{ins}(n-1)$$

So: $W_{isort}(n) \leq c_1 + c_2 \cdot n + W_{isort}(n-1)$

Consequently: $W_{isort}(n)$ is $O(n^2)$.

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work: $W_{isort}(n)$ with n the list length.

Equations:

$$W_{isort}(0) = c_0$$
$$W_{isort}(n) = c_1 + W_{isort}(n-1) + W_{ins}(n-1)$$

So: $W_{isort}(n) \leq c_1 + c_2 \cdot n + W_{isort}(n-1)$

Consequently: $W_{isort}(n)$ is $O(n^2)$.

Note: again, no opportunity for parallelism.

Work for insertion sort

```
fun isort ([ ] : int list) : int list = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work: $W_{isort}(n)$ with n the list length.

Equations:

$W_{isort}(0) = c_0$
$W_{isort}(n) = c_1 + W_{isort}(n-1) + W_{ins}(n-1)$

So: $W_{isort}(n) \leq c_1 + c_2 \cdot n + W_{isort}(n-1)$

Consequently: $W_{isort}(n)$ is $O(n^2)$.

Note: again, no opportunity for parallelism.

Can we do better?

Mergesort: divide and conquer

Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

```
[9, 7, 5, 3, 4]
```

Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

```
[9, 7, 5, 3, 4]
```

Divide the list into approximate halves:

Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

```
[9, 7, 5, 3, 4]
```

Divide the list into approximate halves:

```
[9, 7, 5, 3, 4]
```

Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

```
[9, 7, 5, 3, 4]
```

Divide the list into approximate halves:

```
[9, 7, 5, 3, 4]
```

Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

```
[9, 7, 5, 3, 4]
```

Divide the list into approximate halves:

```
[9, 7, 5, 3, 4]
```

Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

```
[9, 7, 5, 3, 4]
```

Divide the list into approximate halves:

```
[9, 7, 5, 3, 4]
```

```
[9
```

Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

```
[9, 7, 5, 3, 4]
```

Divide the list into approximate halves:

```
[9, 7, 5, 3, 4]
```

```
[9                                  [7
```

Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

```
[9, 7, 5, 3, 4]
```

Divide the list into approximate halves:

```
          [9, 7, 5, 3, 4]
         /               \
[9, 5                     [7
```

Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

```
[9, 7, 5, 3, 4]
```

Divide the list into approximate halves:

```
            [9, 7, 5, 3, 4]

[9, 5                            [7, 3
```

Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

```
[9, 7, 5, 3, 4]
```

Divide the list into approximate halves:

```
        [9, 7, 5, 3, 4]
[9, 5, 4]                [7, 3
```

Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

```
[9, 7, 5, 3, 4]
```

Divide the list into approximate halves:

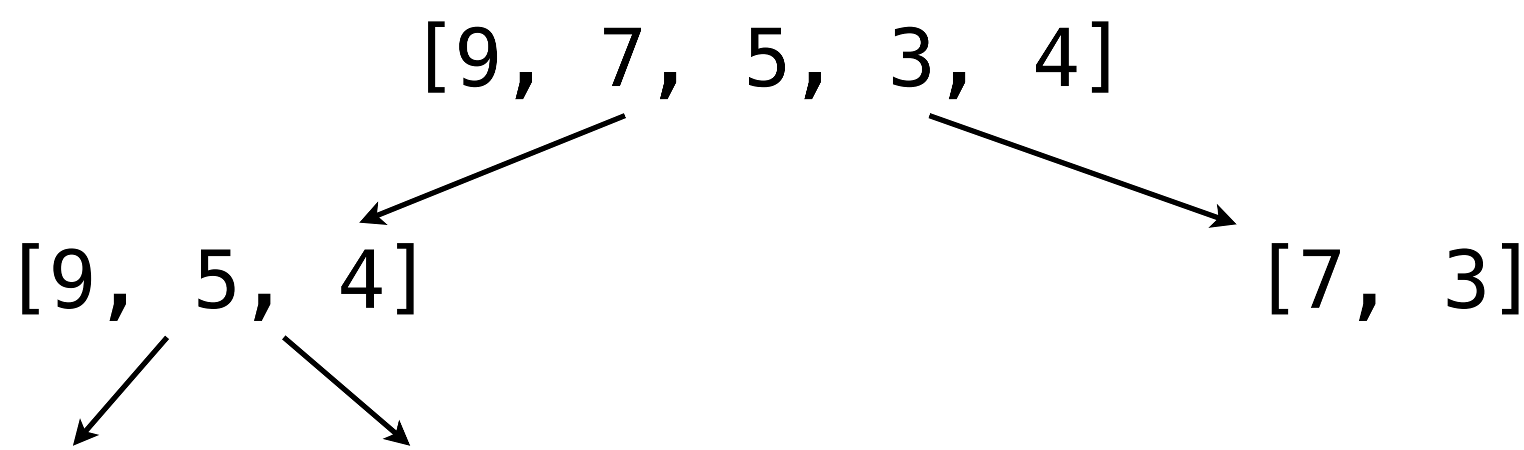
```
            [9, 7, 5, 3, 4]
           ↙               ↘
[9, 5, 4]                    [7, 3]
```

Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

```
[9, 7, 5, 3, 4]
```

Divide the list into approximate halves:

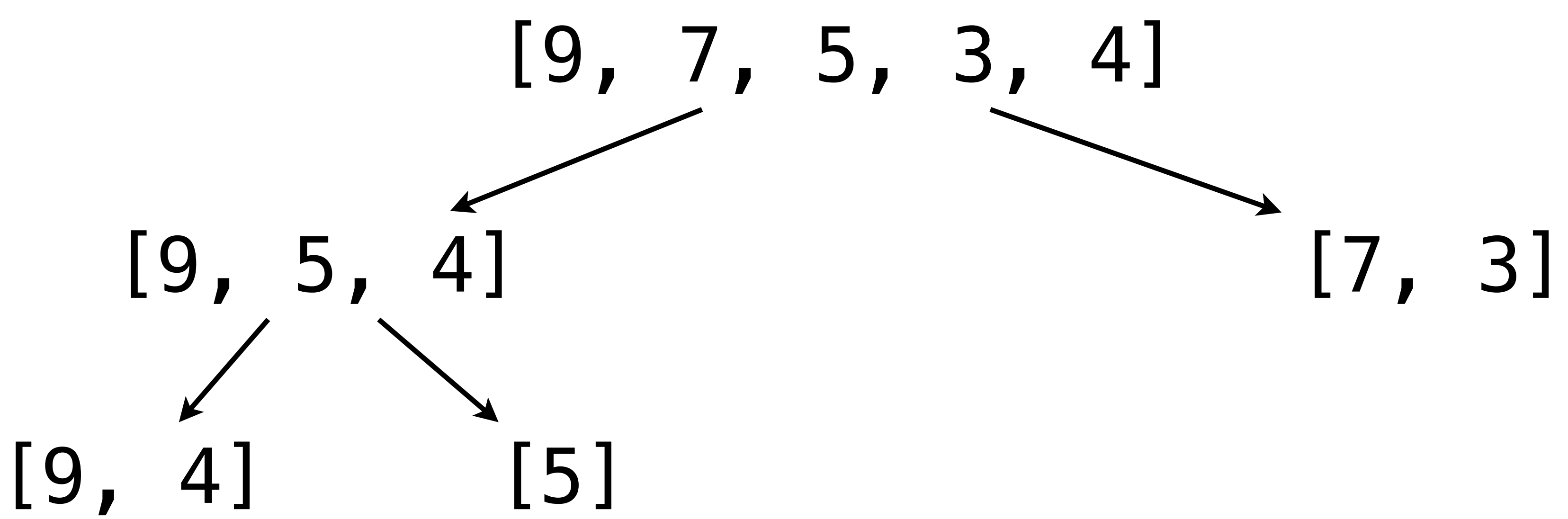


Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

`[9, 7, 5, 3, 4]`

Divide the list into approximate halves:

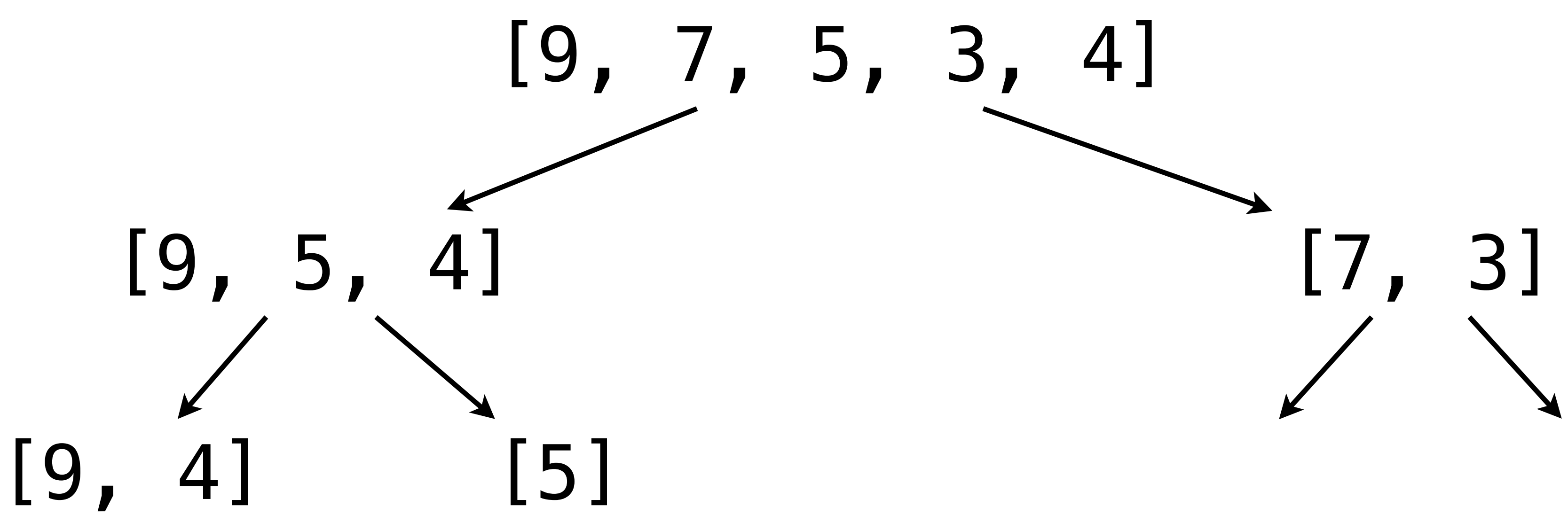


Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

```
[9, 7, 5, 3, 4]
```

Divide the list into approximate halves:

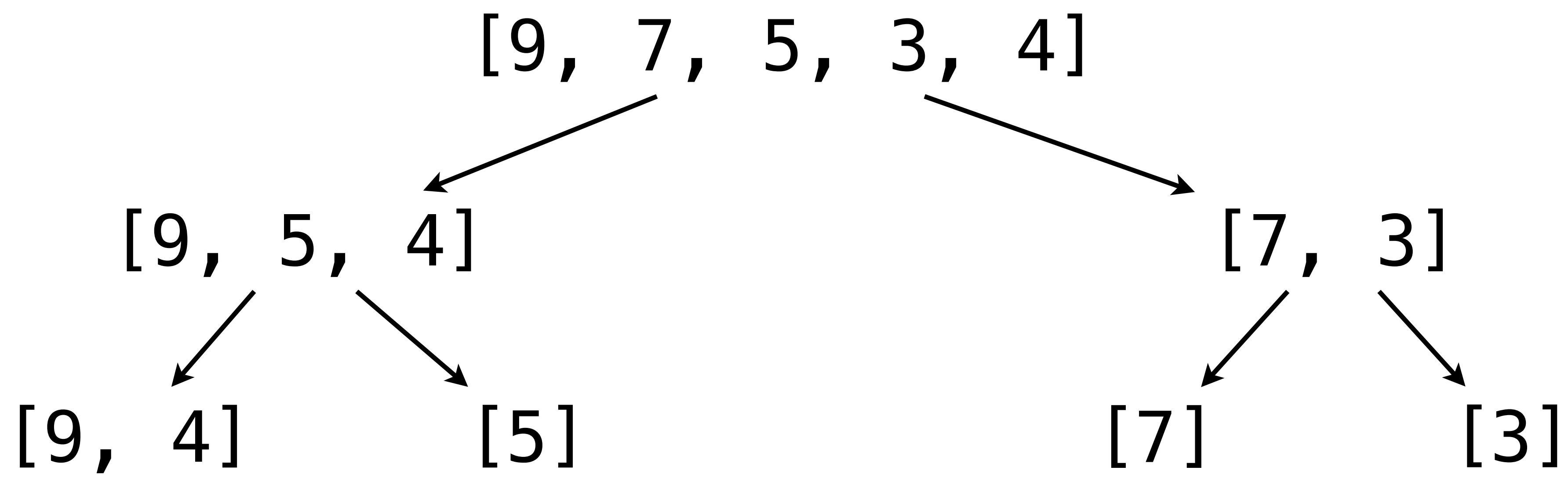


Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

`[9, 7, 5, 3, 4]`

Divide the list into approximate halves:

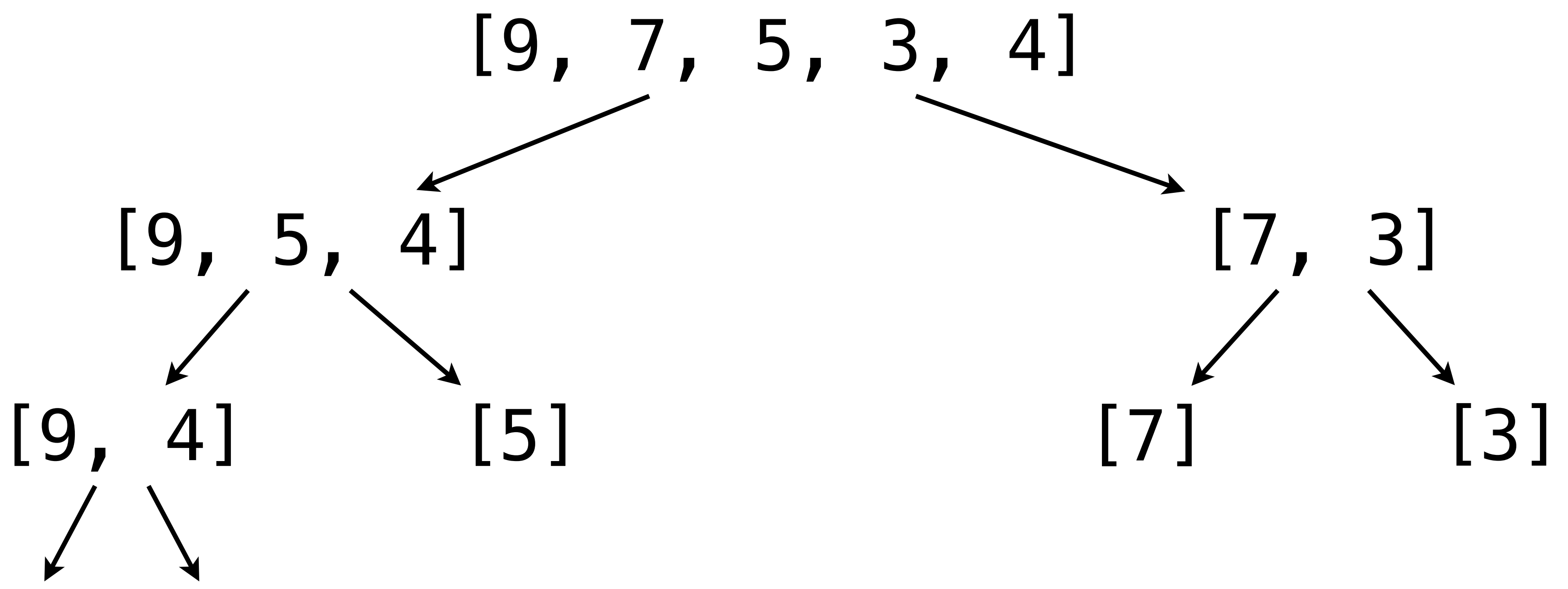


Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

`[9, 7, 5, 3, 4]`

Divide the list into approximate halves:

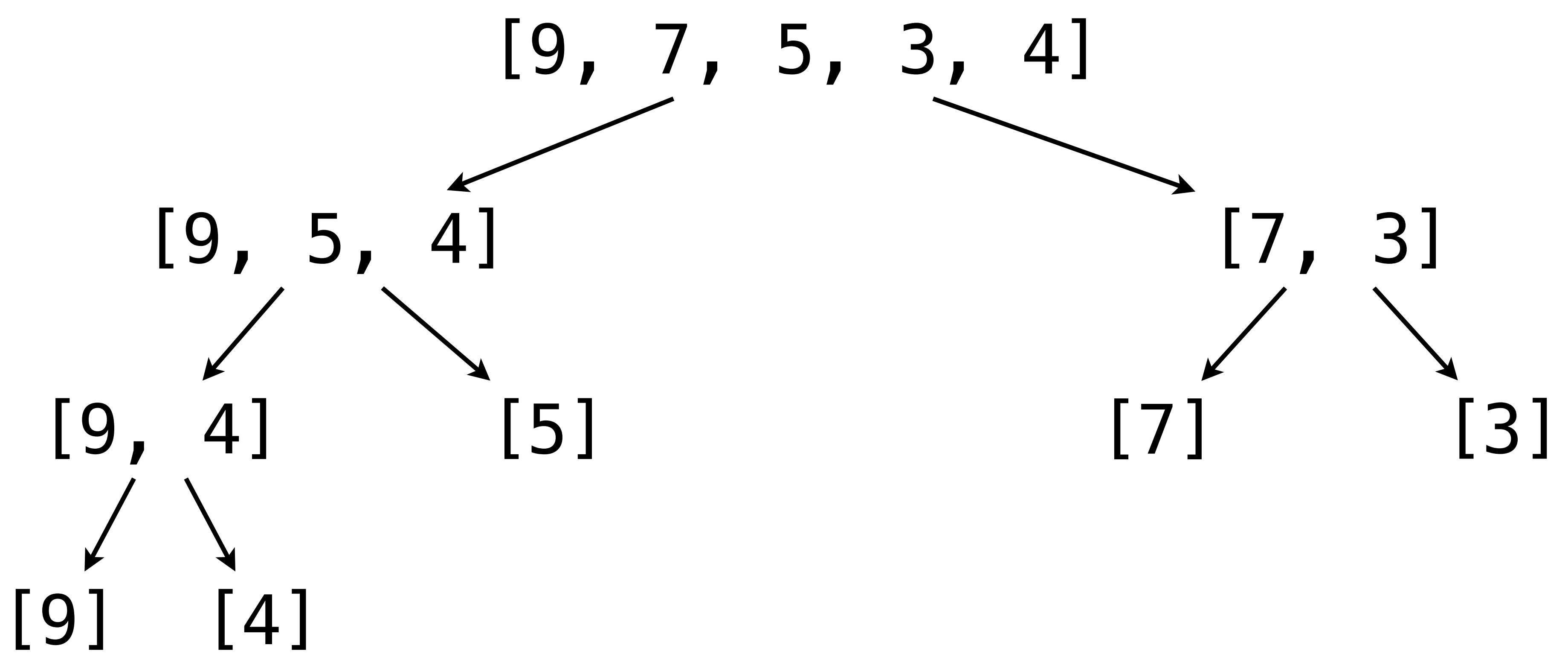


Mergesort: divide and conquer

Suppose, I have the list, that I want to **sort**

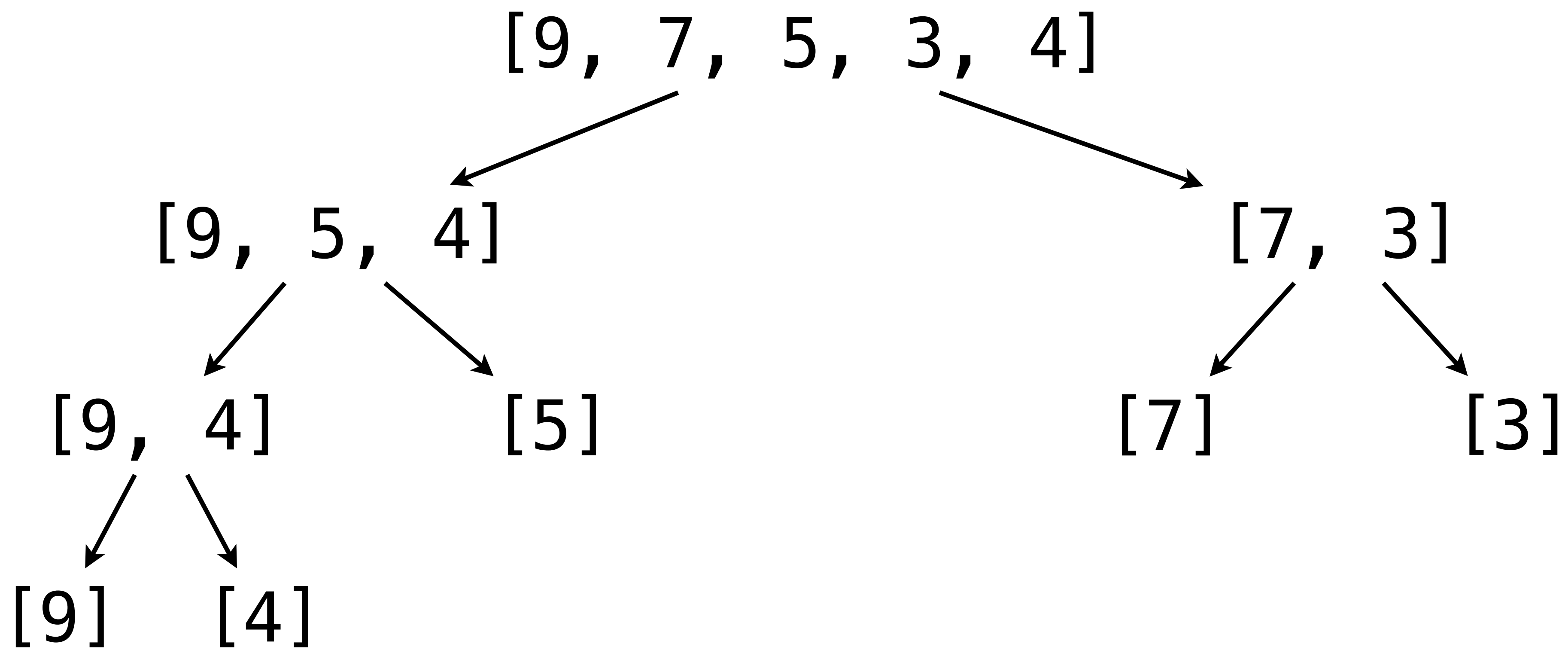
`[9, 7, 5, 3, 4]`

Divide the list into approximate halves:



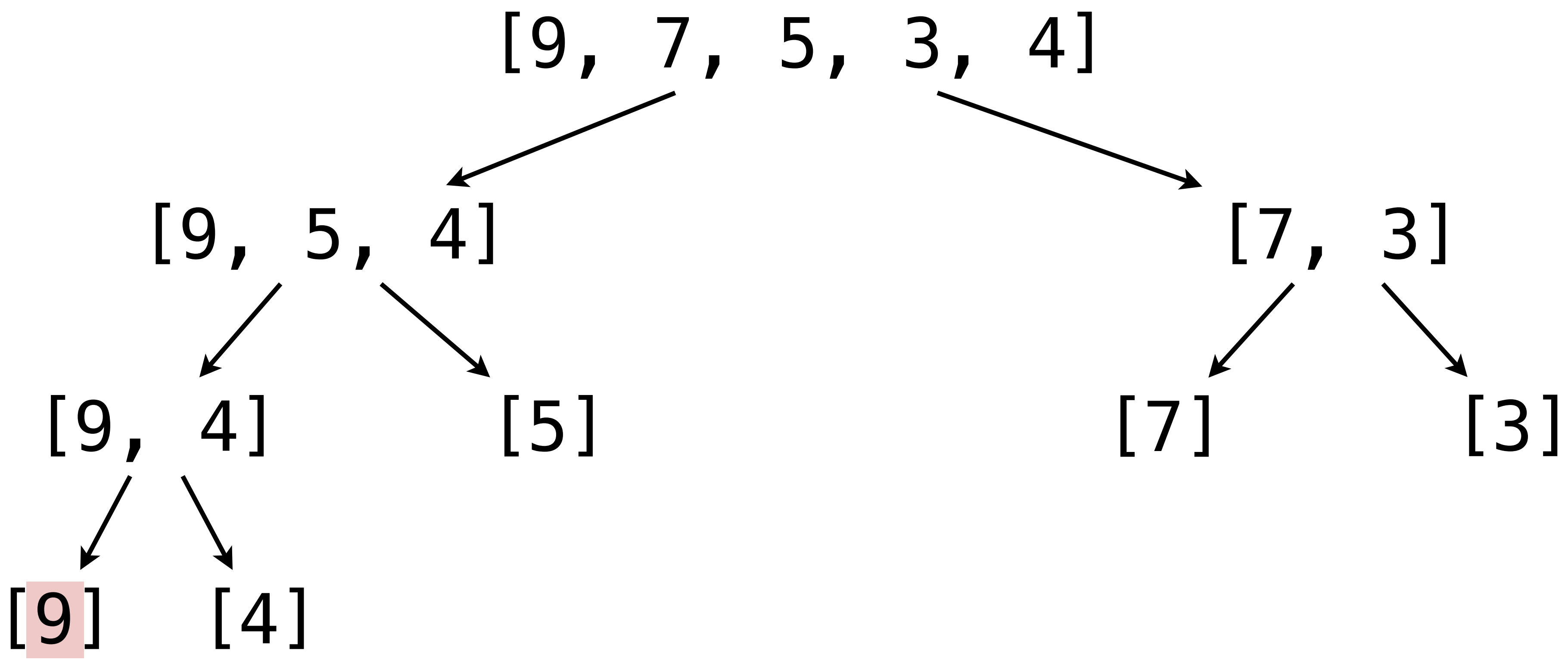
Mergesort: divide and conquer

Now, let's **merge**:



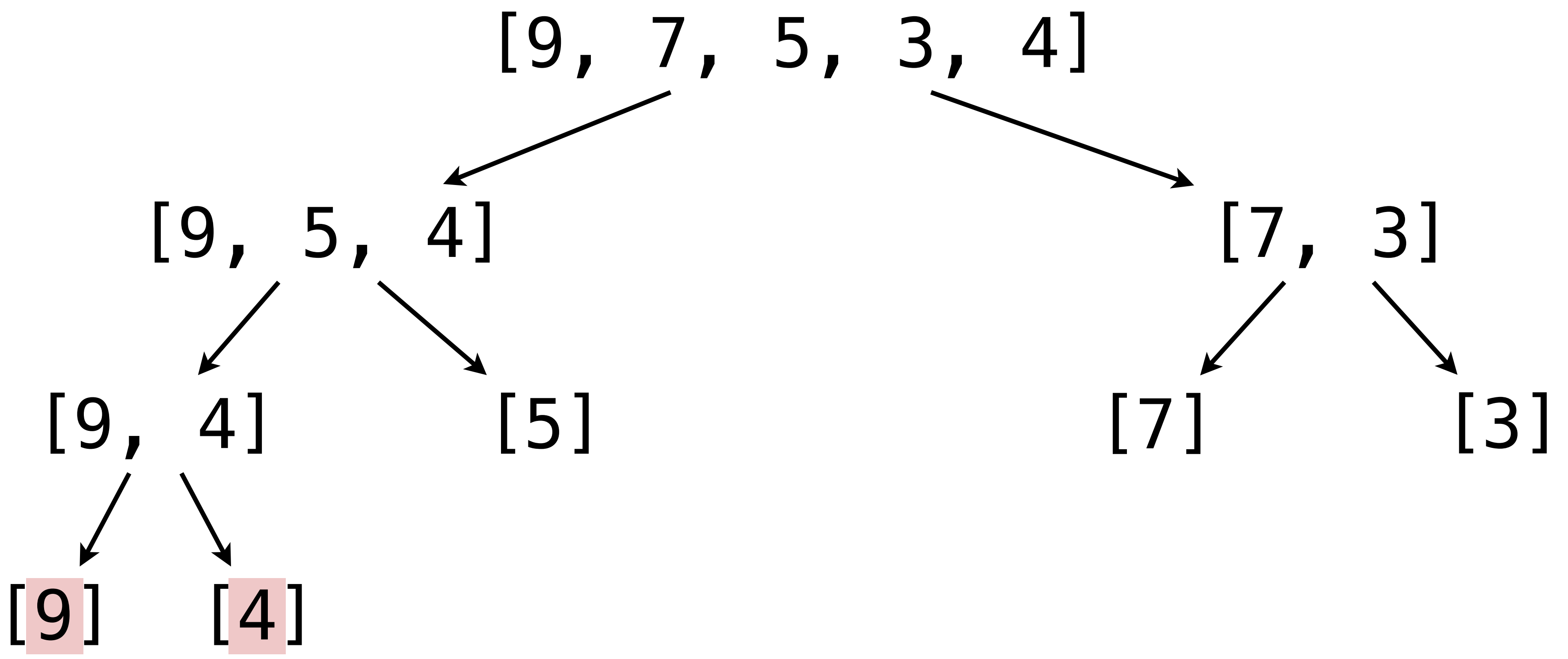
Mergesort: divide and conquer

Now, let's **merge**:



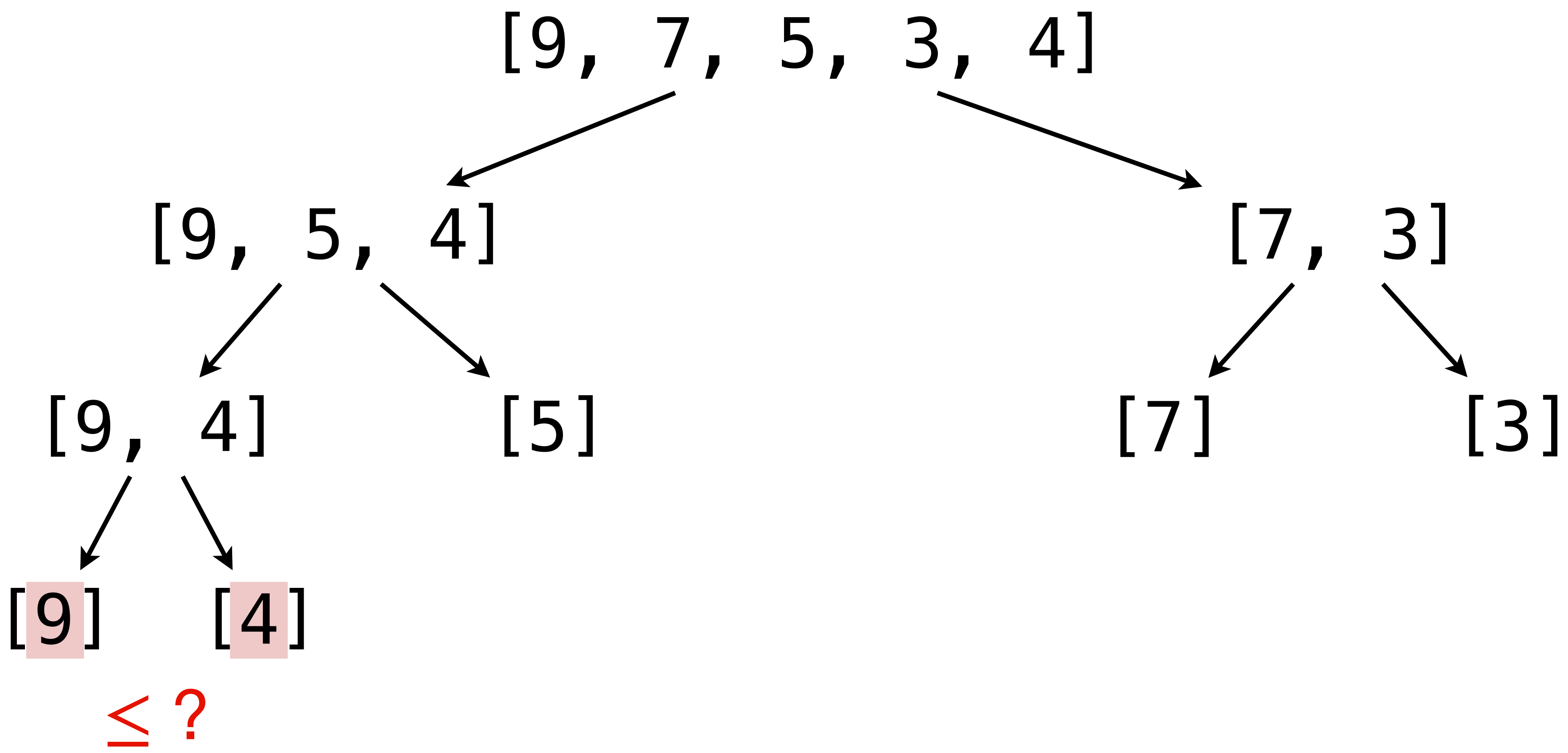
Mergesort: divide and conquer

Now, let's **merge**:



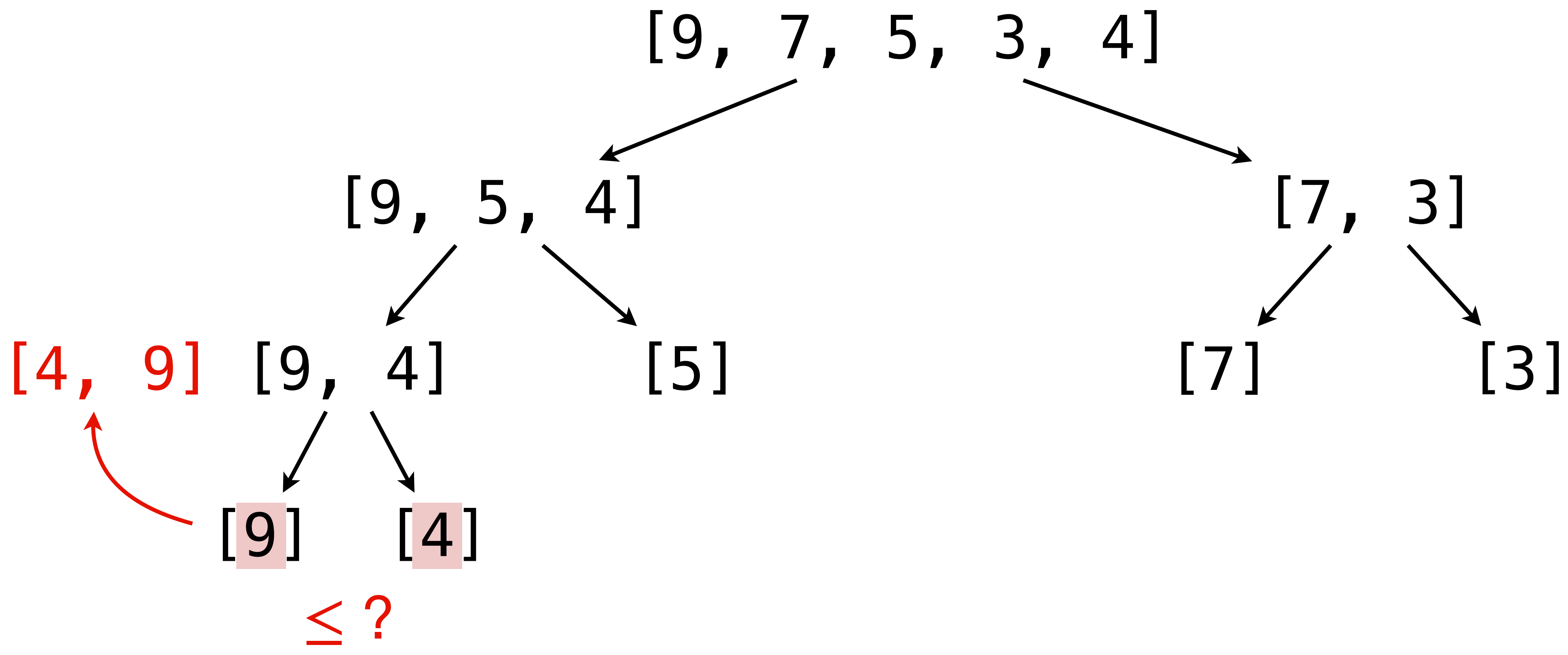
Mergesort: divide and conquer

Now, let’s **merge**:



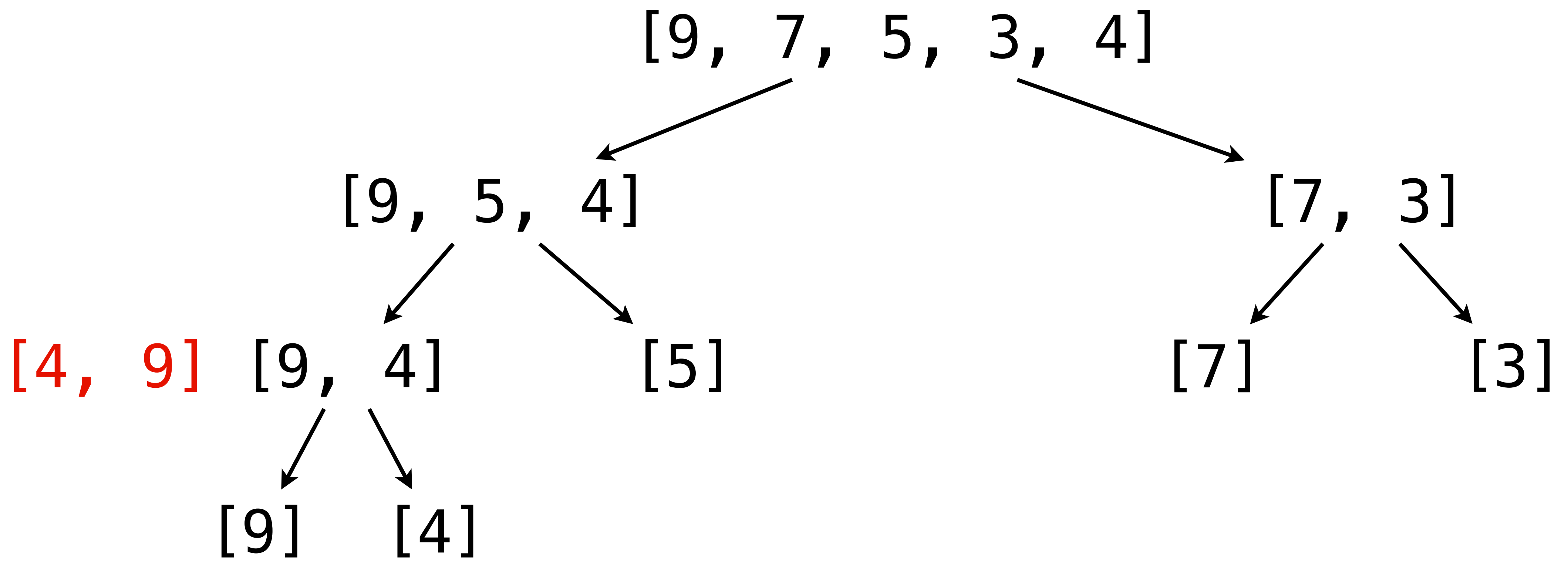
Mergesort: divide and conquer

Now, let's **merge**:



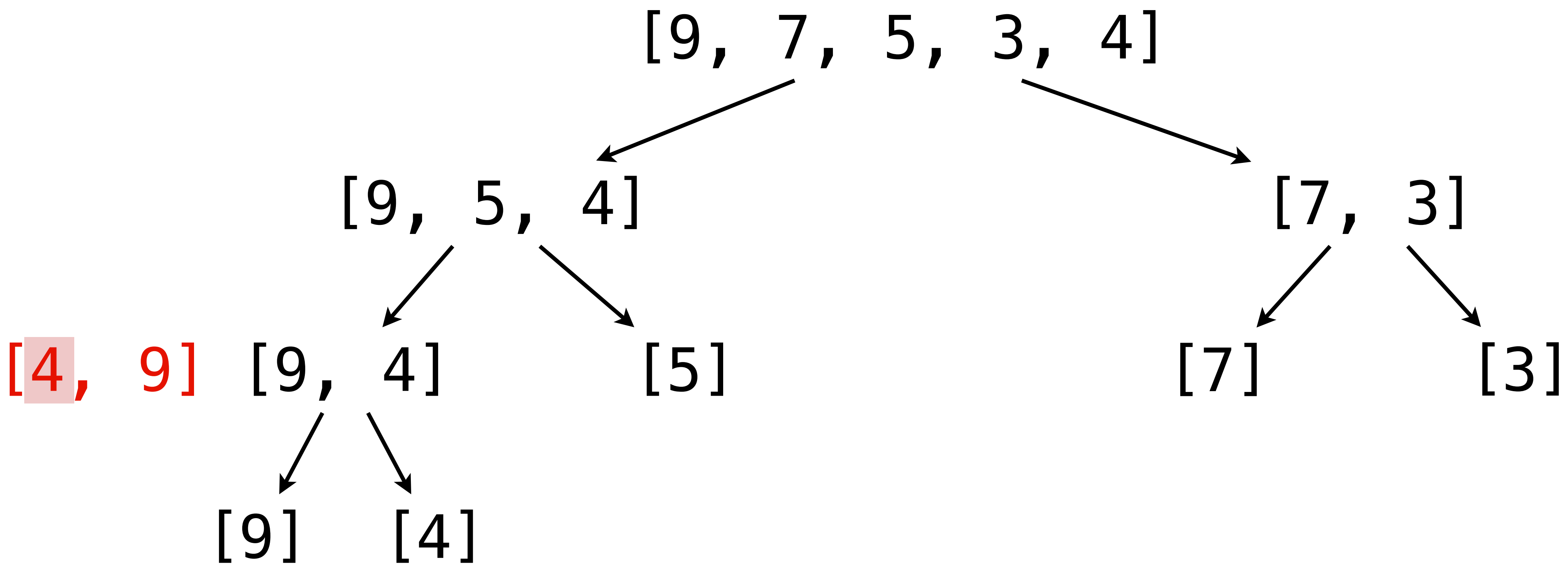
Mergesort: divide and conquer

Now, let's **merge**:



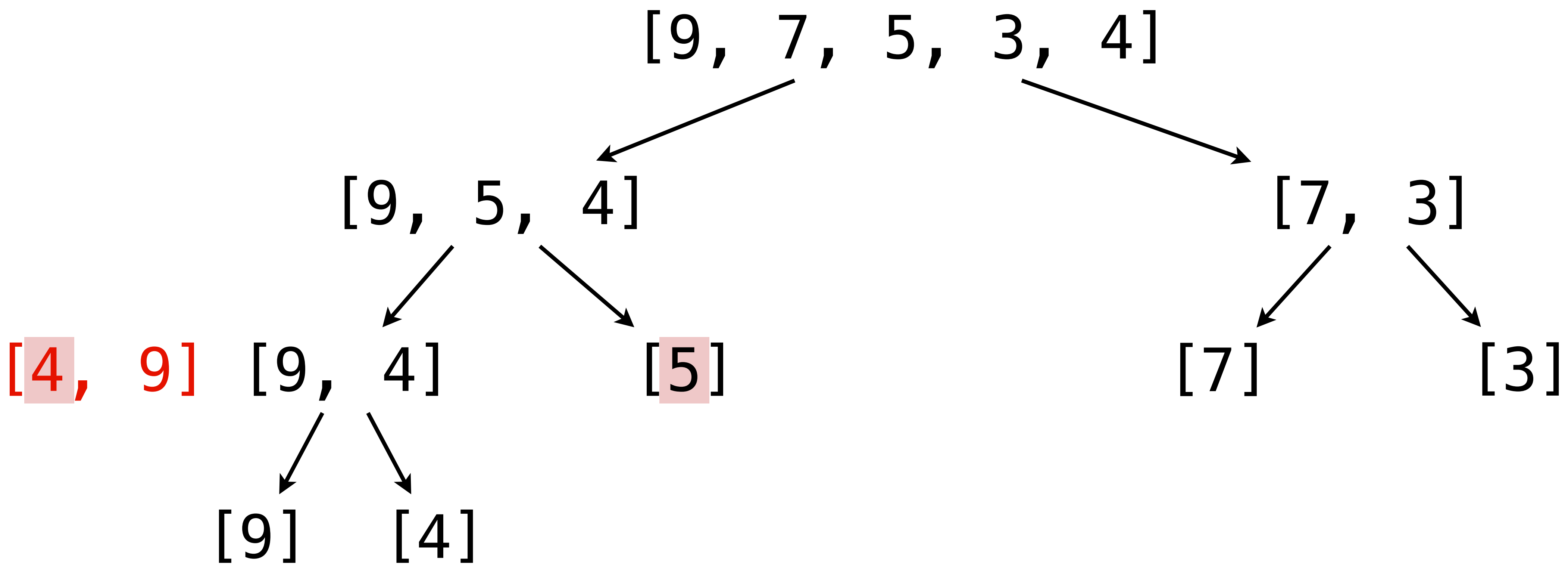
Mergesort: divide and conquer

Now, let’s **merge**:



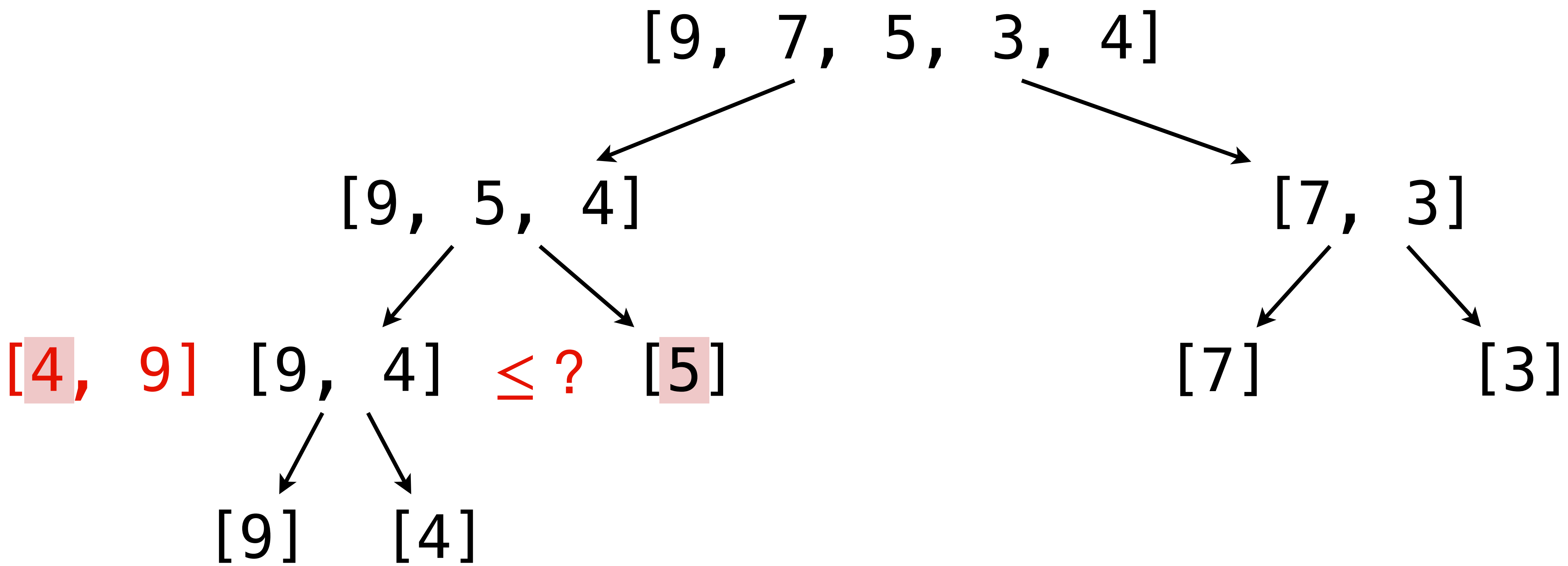
Mergesort: divide and conquer

Now, let's **merge**:



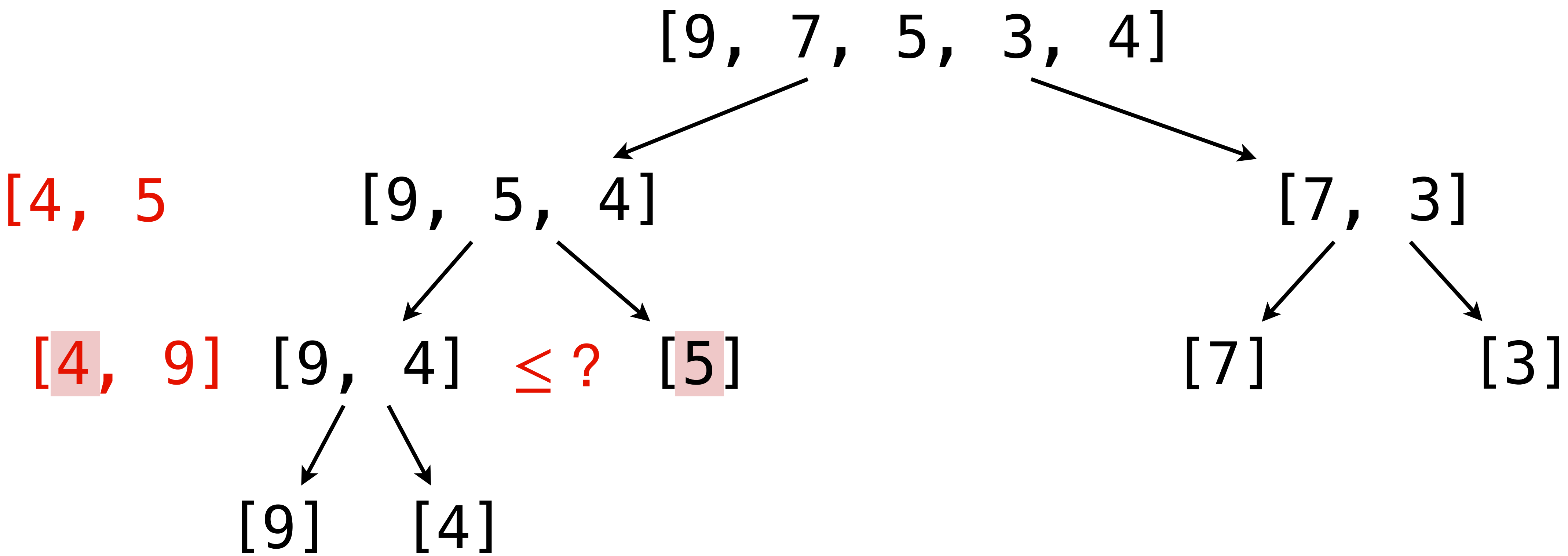
Mergesort: divide and conquer

Now, let's **merge**:



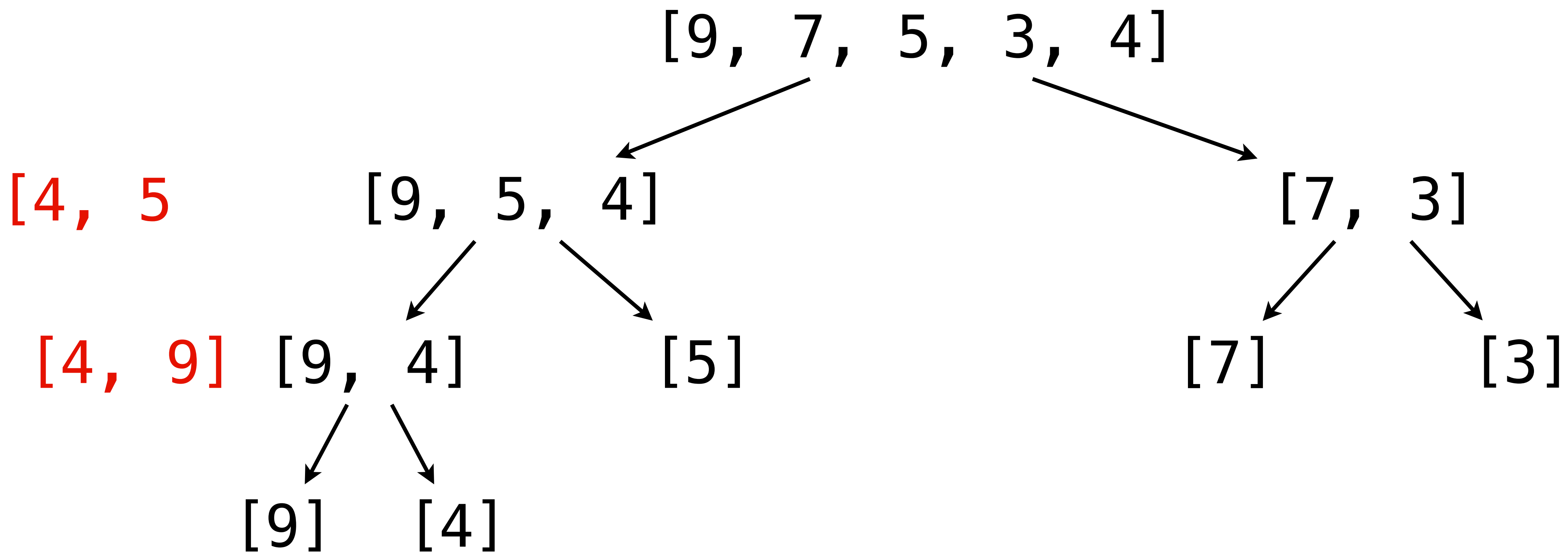
Mergesort: divide and conquer

Now, let's **merge**:



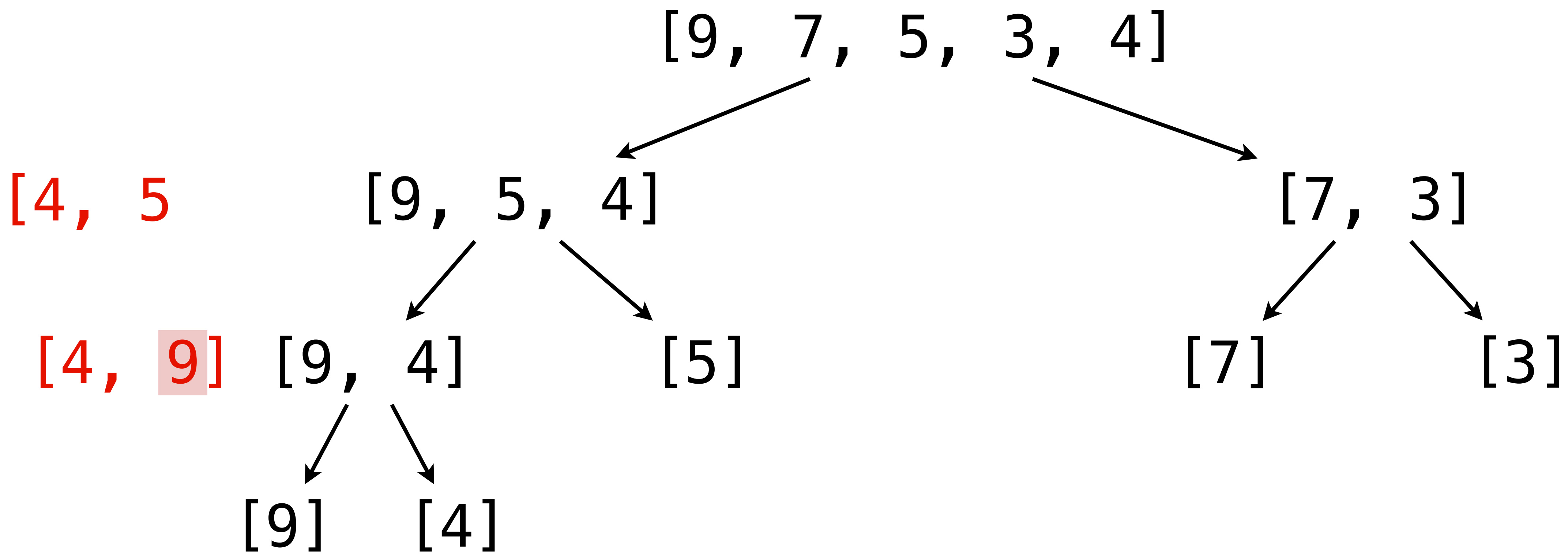
Mergesort: divide and conquer

Now, let’s **merge**:



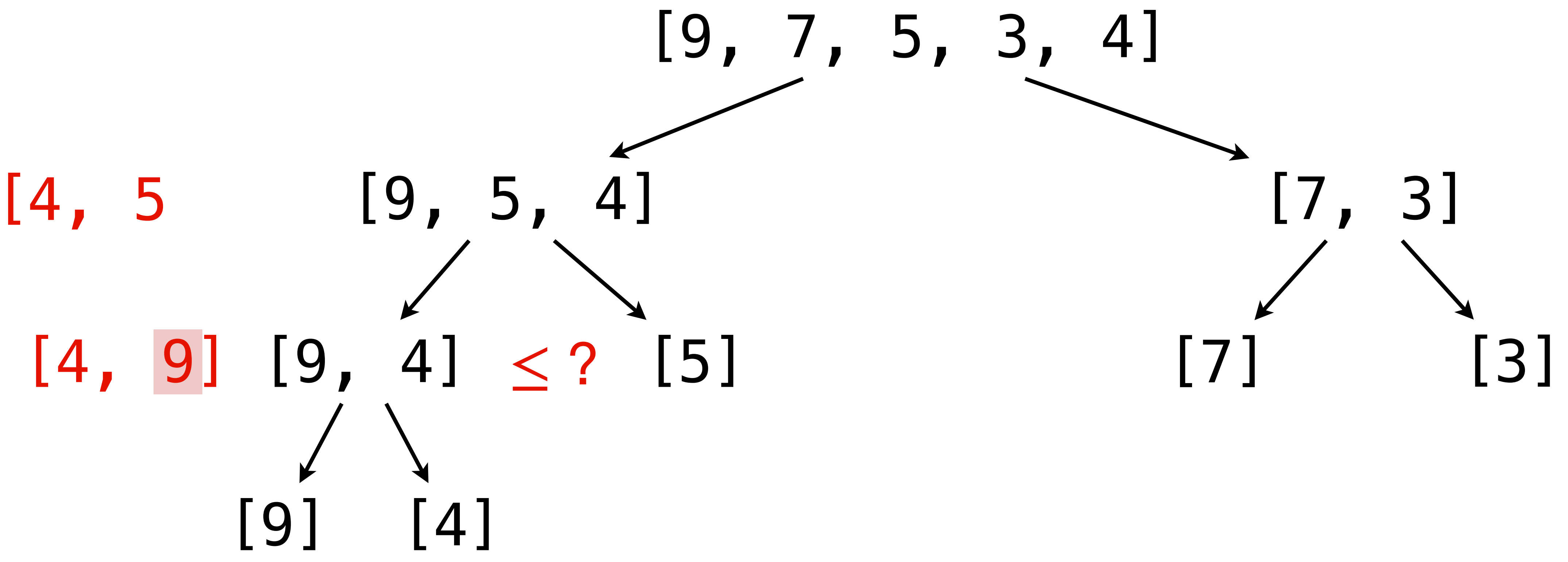
Mergesort: divide and conquer

Now, let's **merge**:



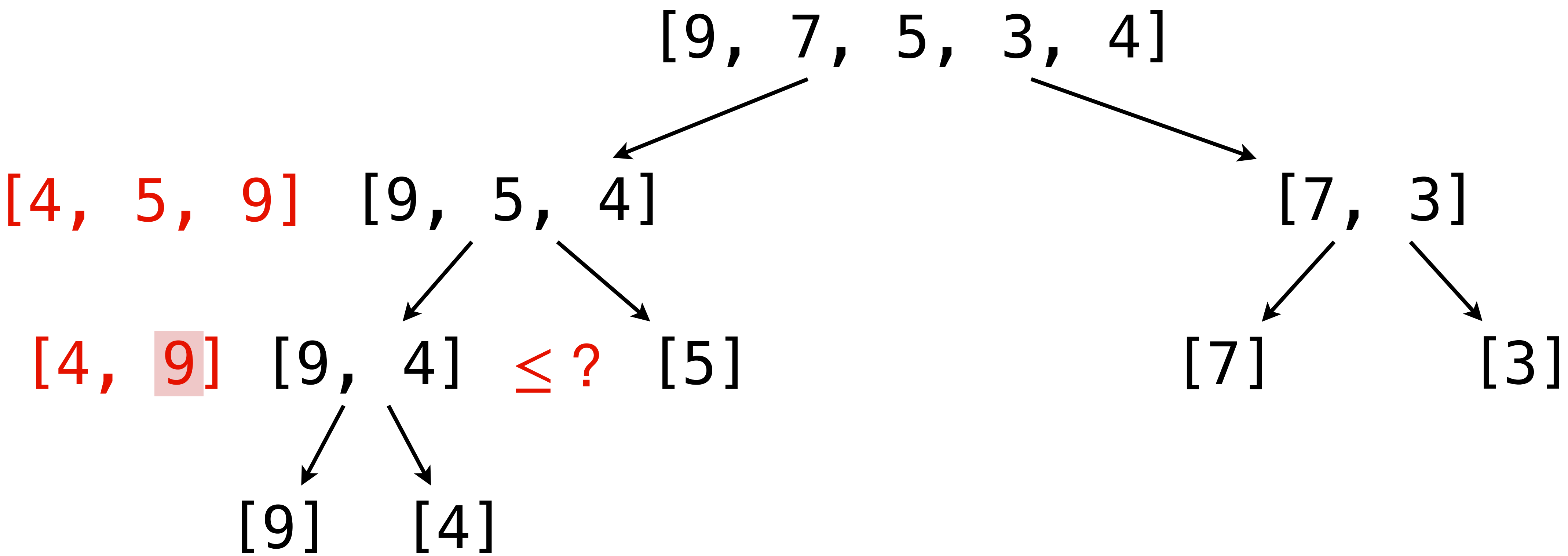
Mergesort: divide and conquer

Now, let's **merge**:



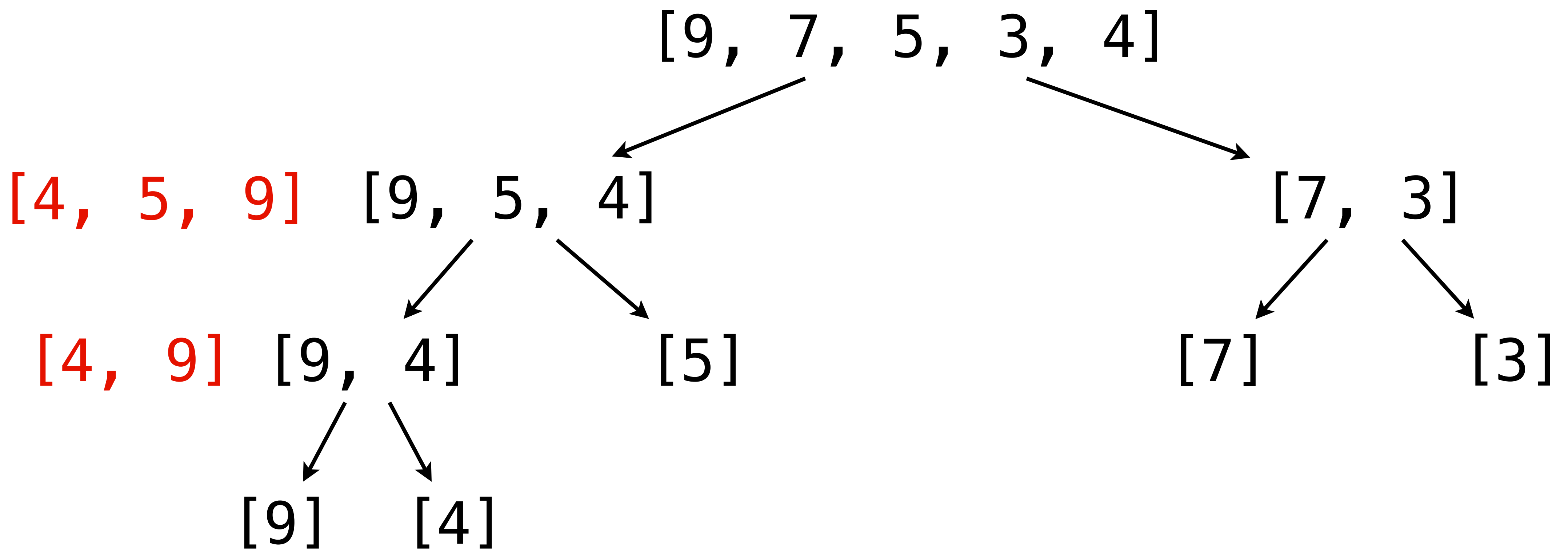
Mergesort: divide and conquer

Now, let's **merge**:



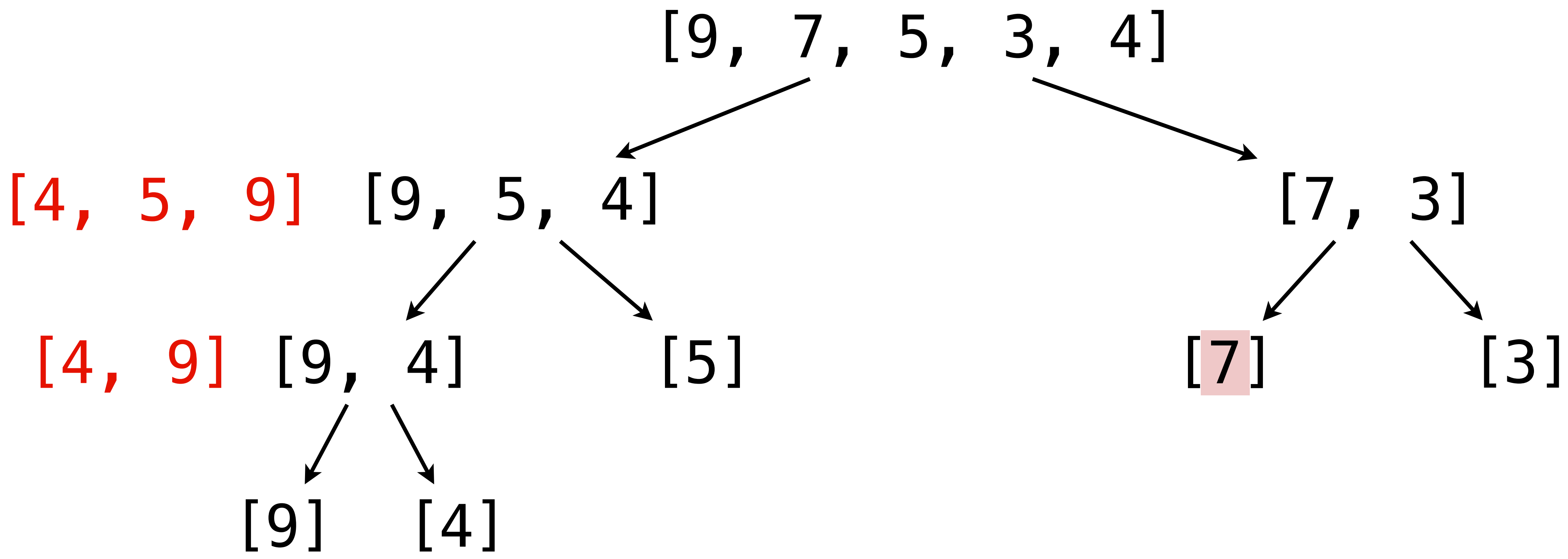
Mergesort: divide and conquer

Now, let's **merge**:



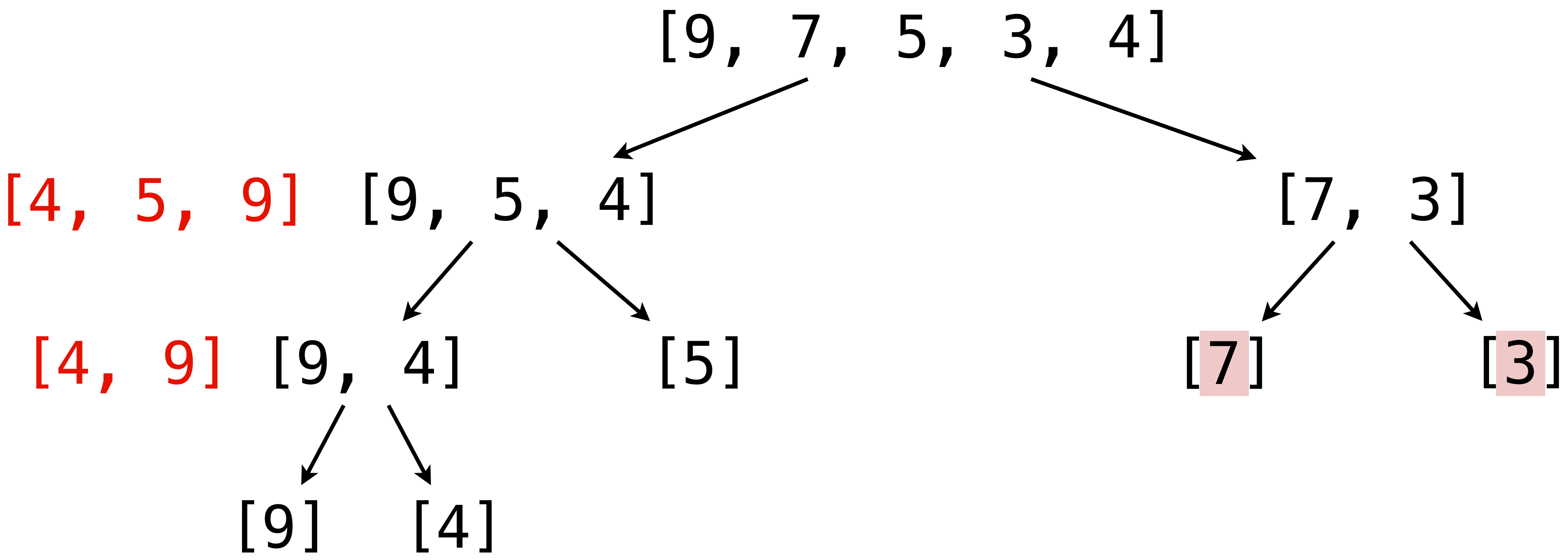
Mergesort: divide and conquer

Now, let's **merge**:



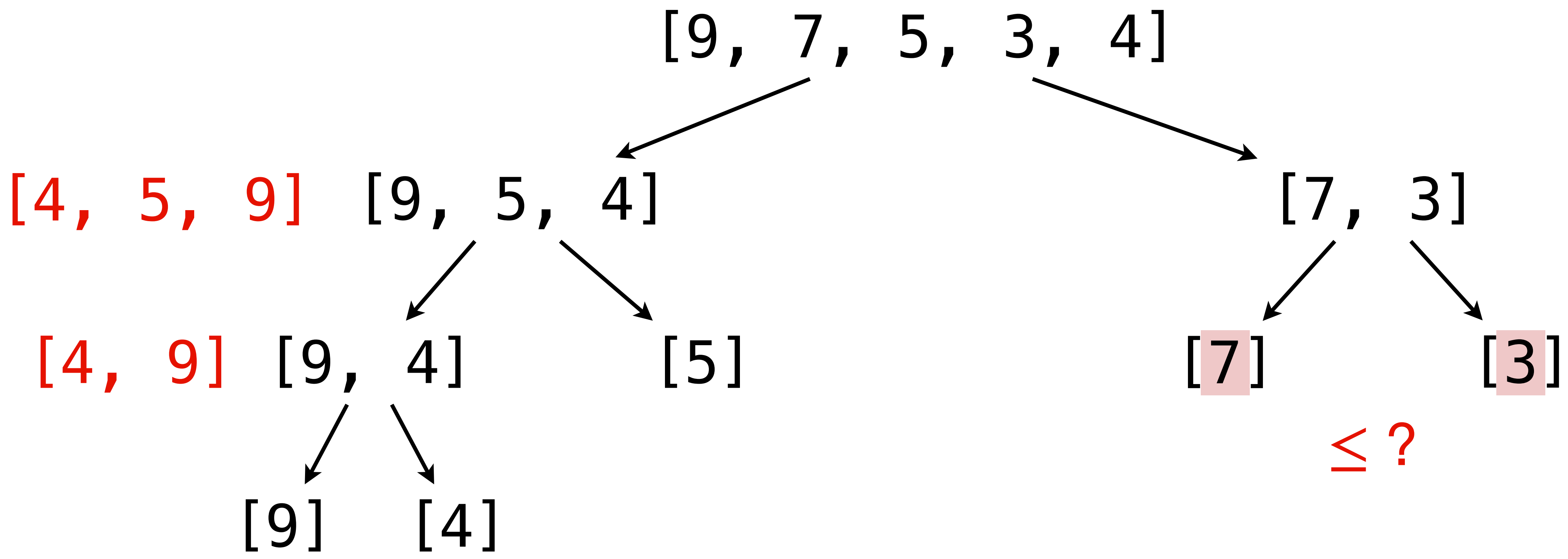
Mergesort: divide and conquer

Now, let's **merge**:



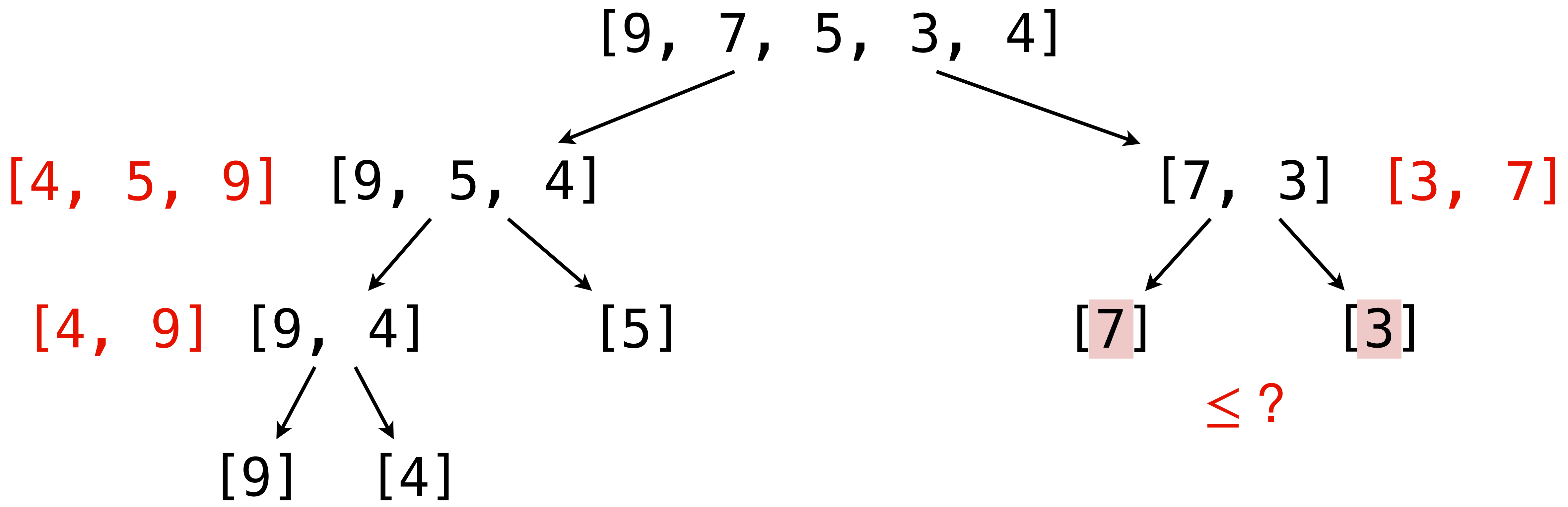
Mergesort: divide and conquer

Now, let's **merge**:



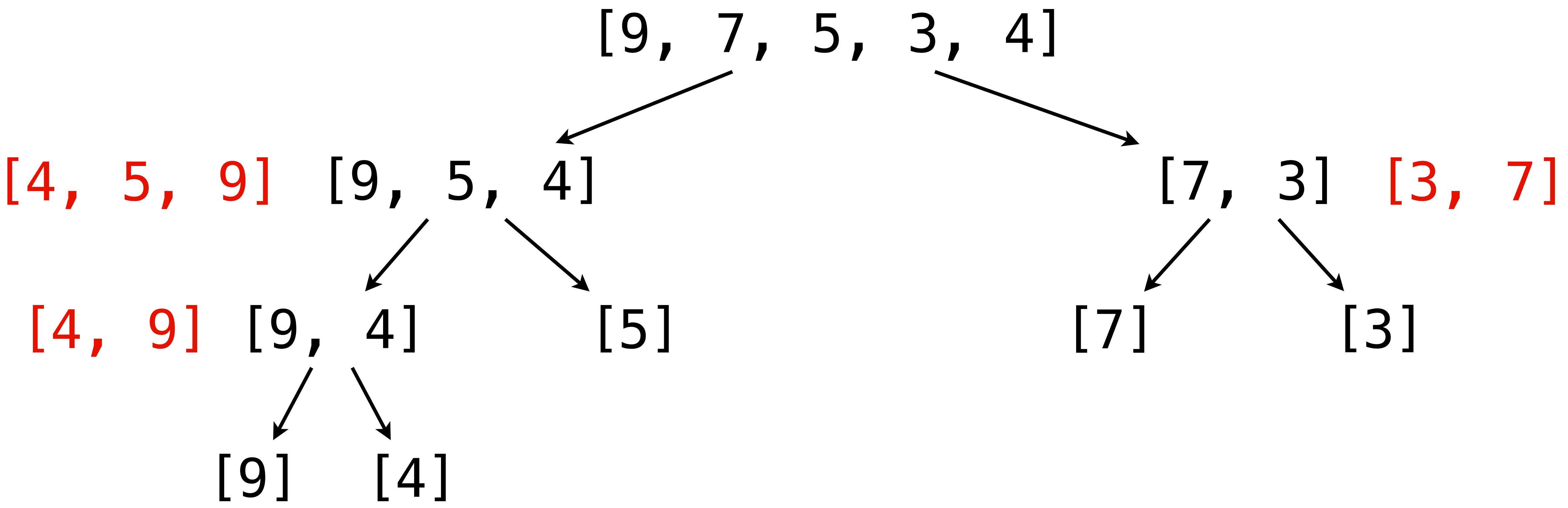
Mergesort: divide and conquer

Now, let's **merge**:



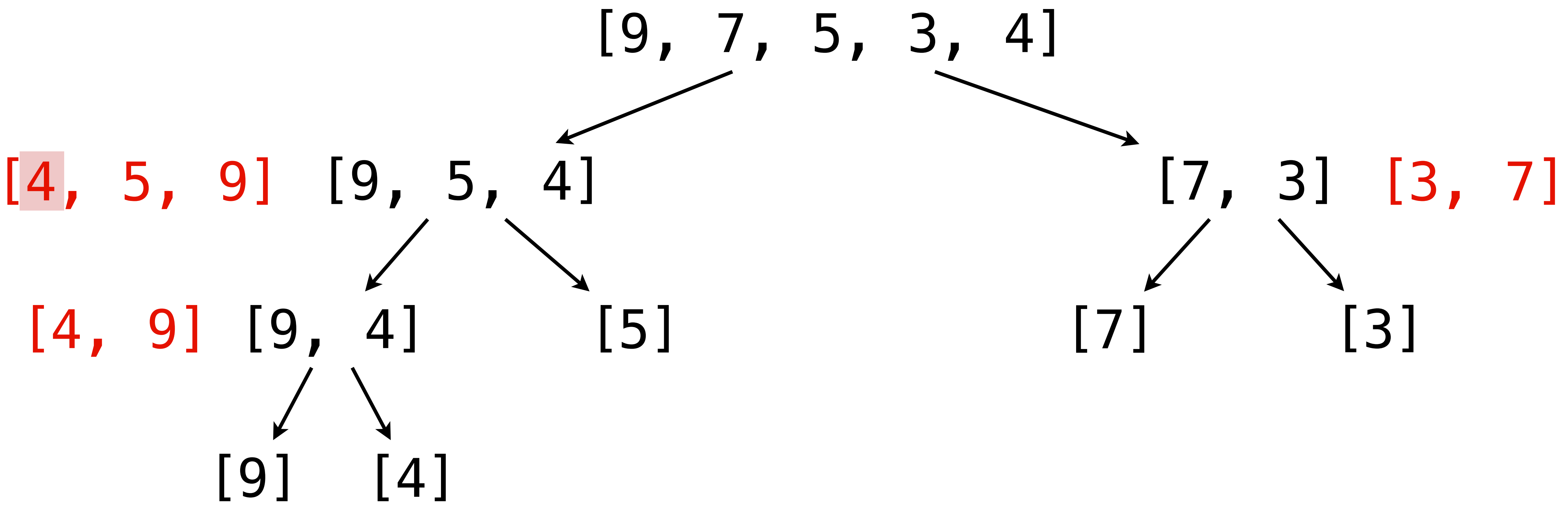
Mergesort: divide and conquer

Now, let's **merge**:



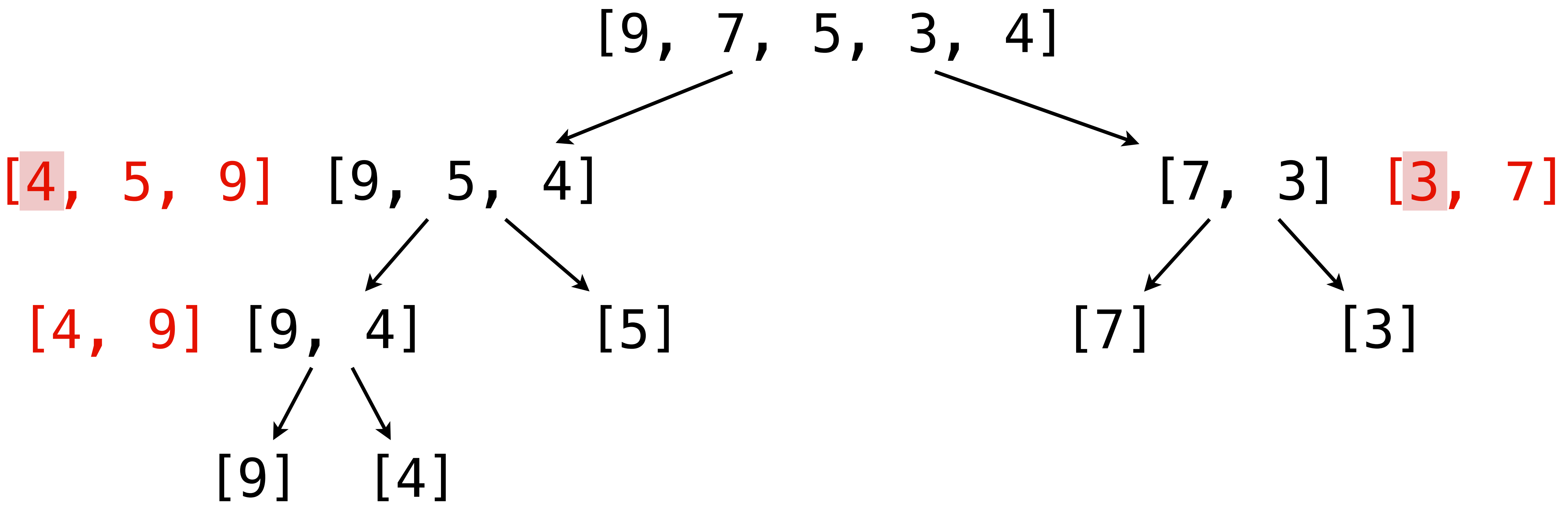
Mergesort: divide and conquer

Now, let's **merge**:



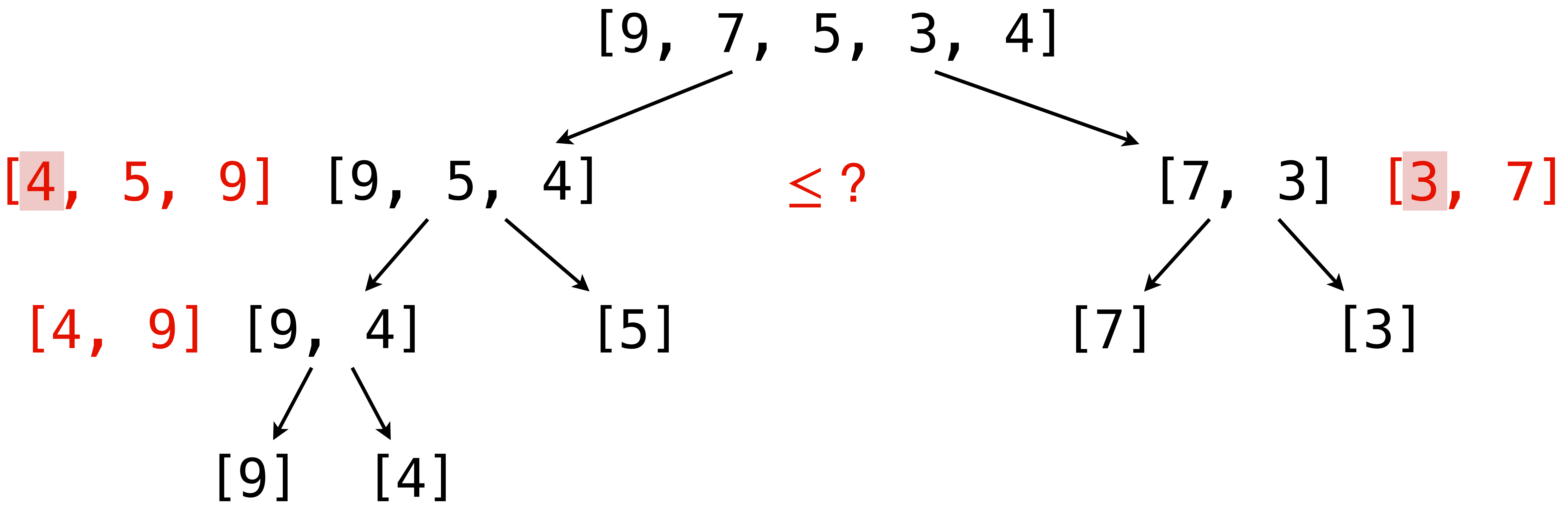
Mergesort: divide and conquer

Now, let's **merge**:



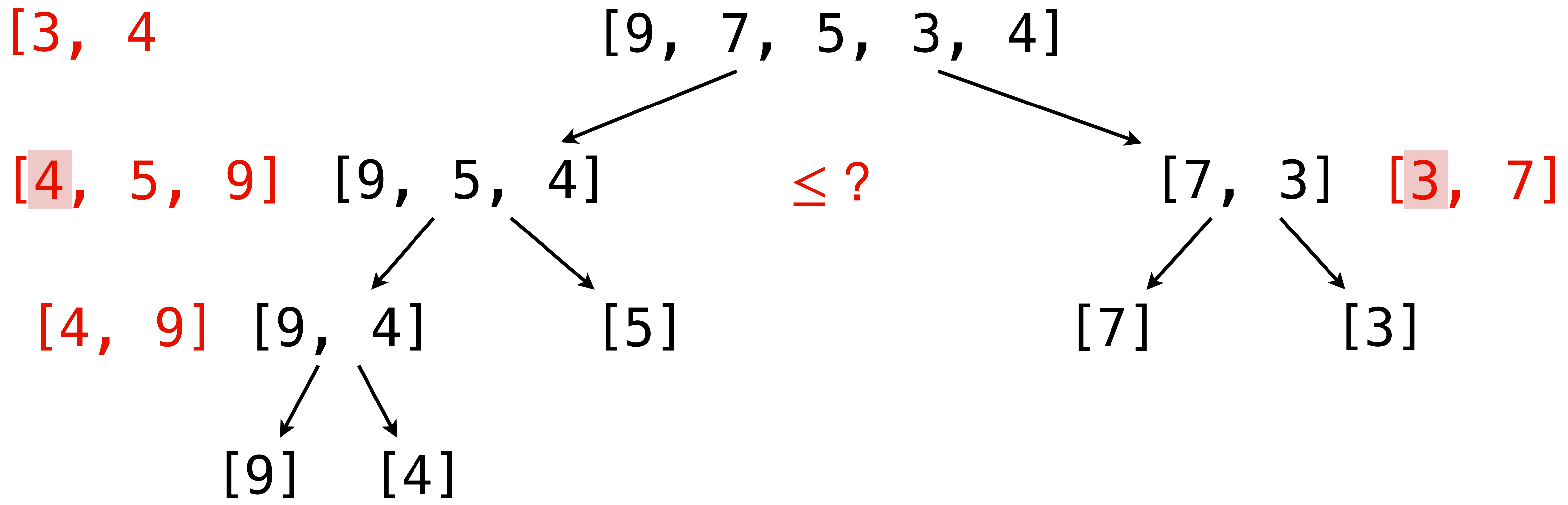
Mergesort: divide and conquer

Now, let's **merge**:



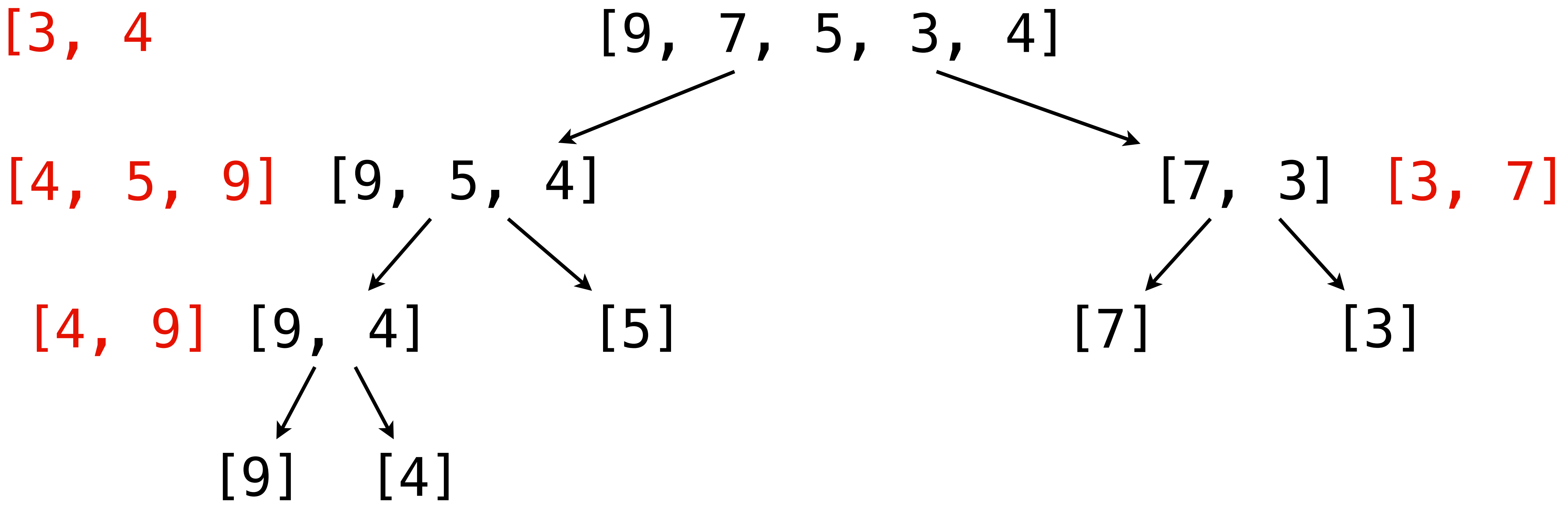
Mergesort: divide and conquer

Now, let’s **merge**:



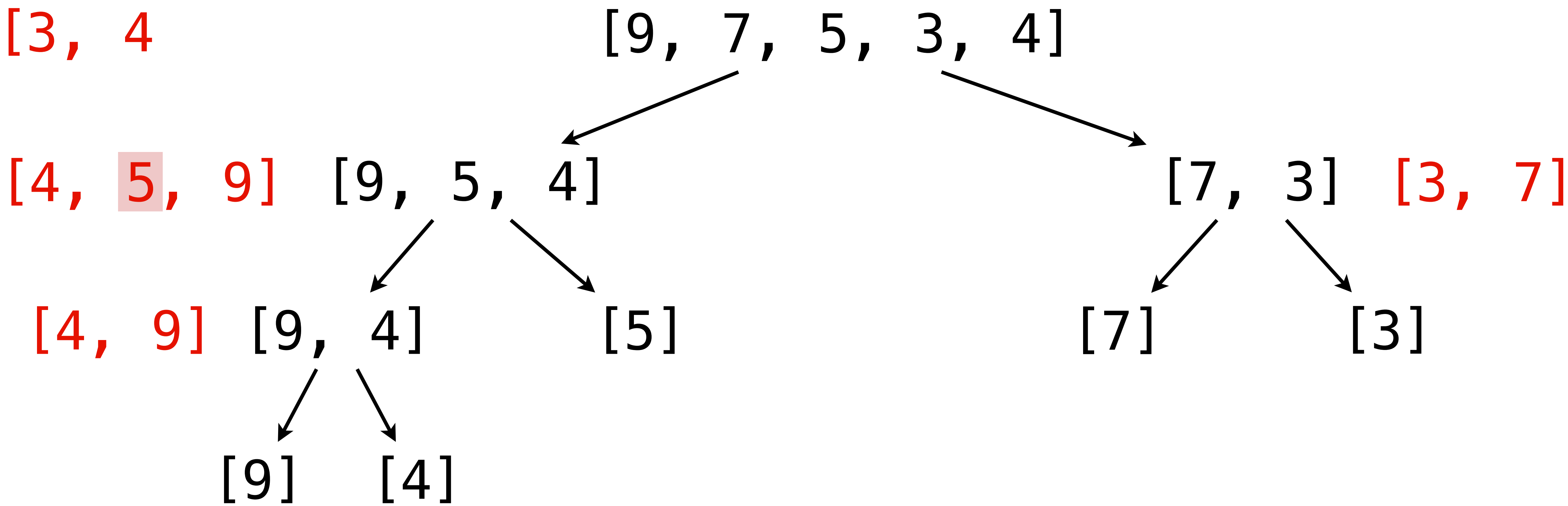
Mergesort: divide and conquer

Now, let's **merge**:



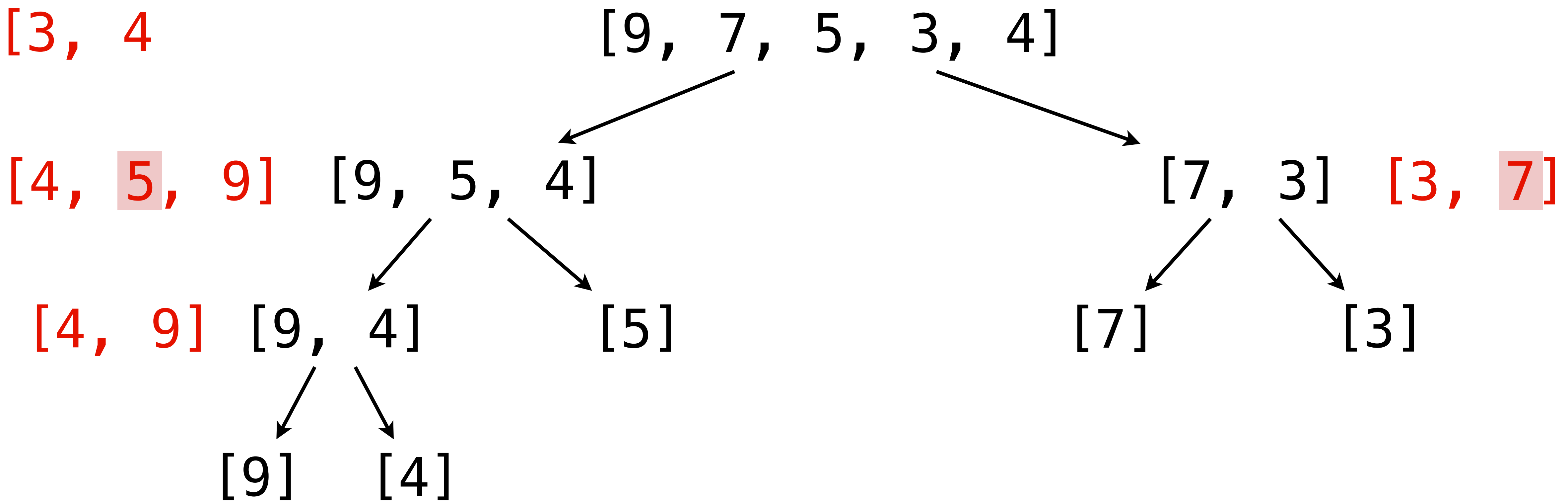
Mergesort: divide and conquer

Now, let's **merge**:



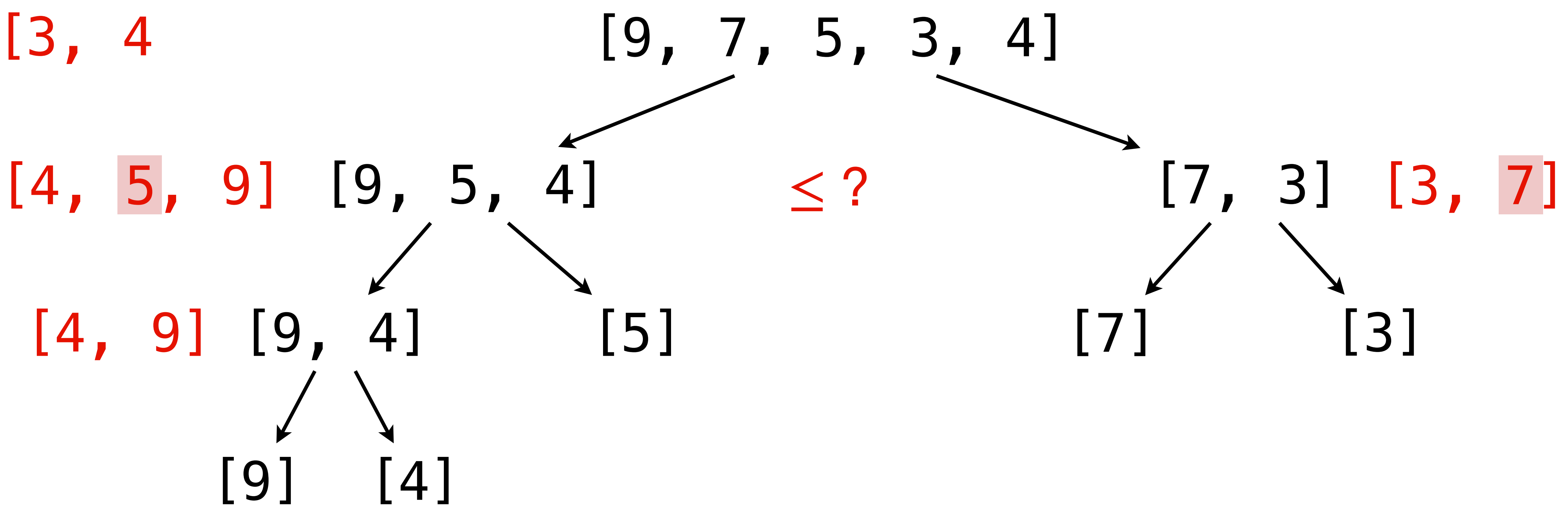
Mergesort: divide and conquer

Now, let's **merge**:



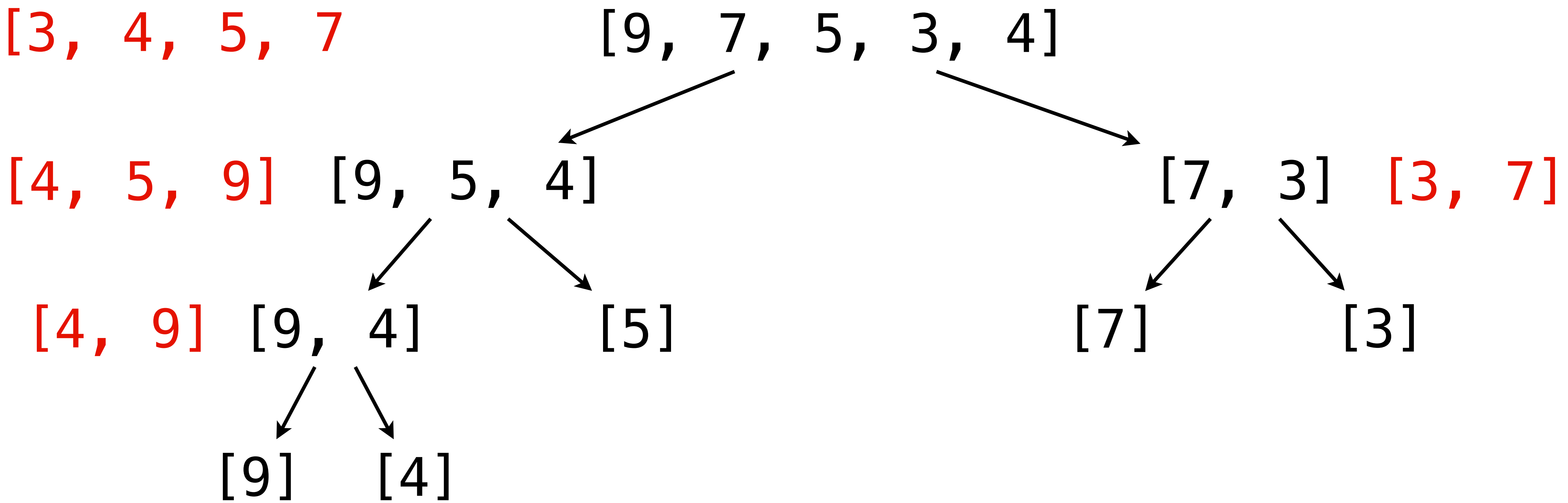
Mergesort: divide and conquer

Now, let's **merge**:



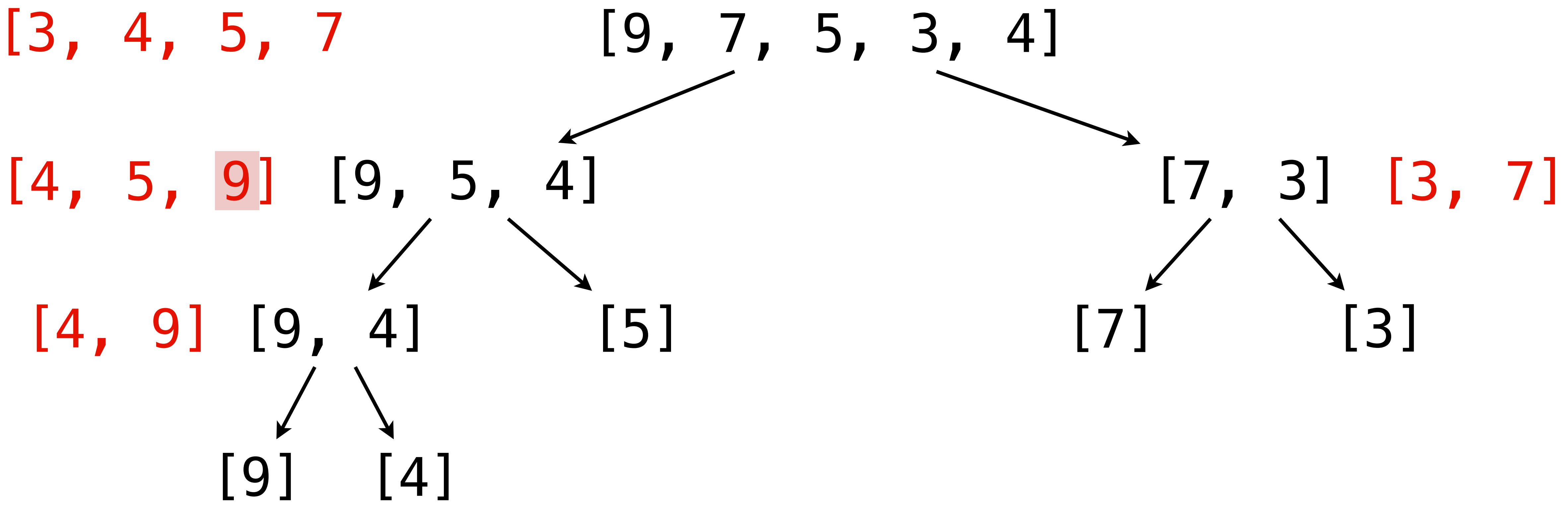
Mergesort: divide and conquer

Now, let’s **merge**:



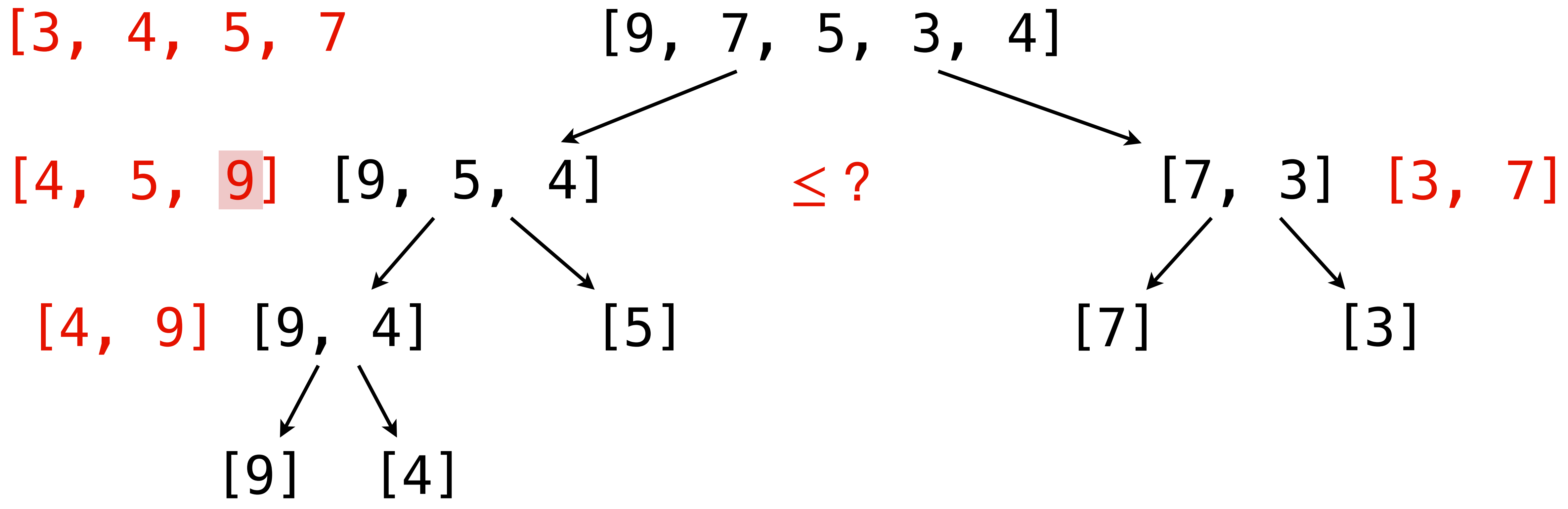
Mergesort: divide and conquer

Now, let's **merge**:



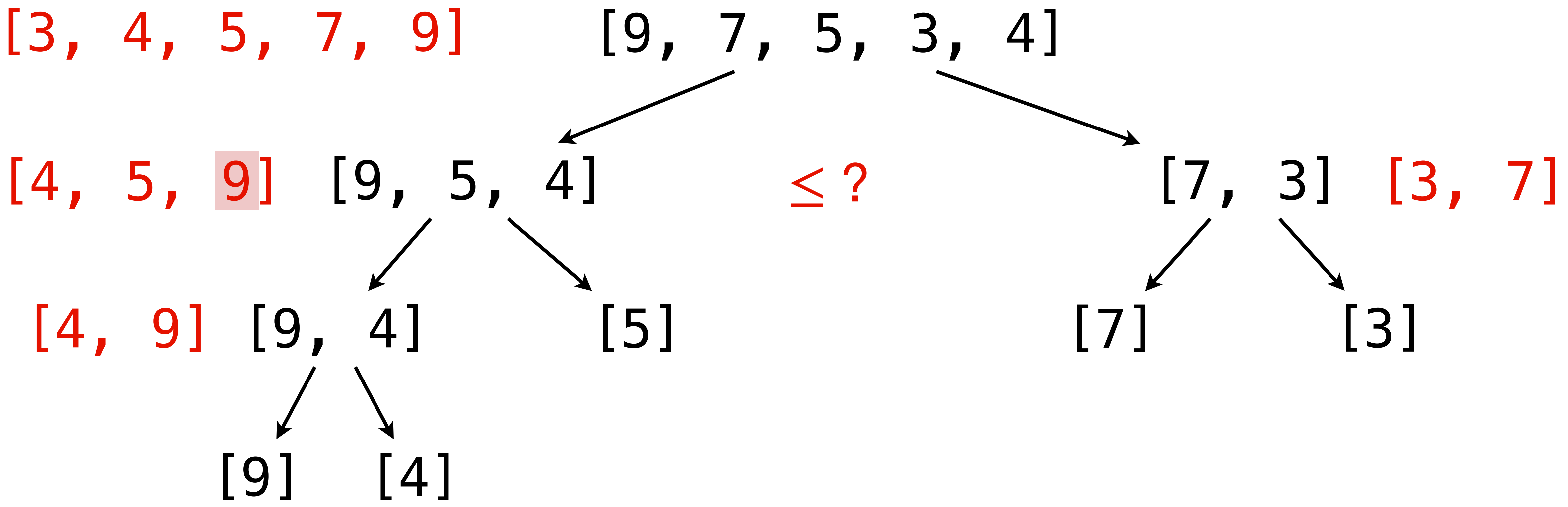
Mergesort: divide and conquer

Now, let’s **merge**:



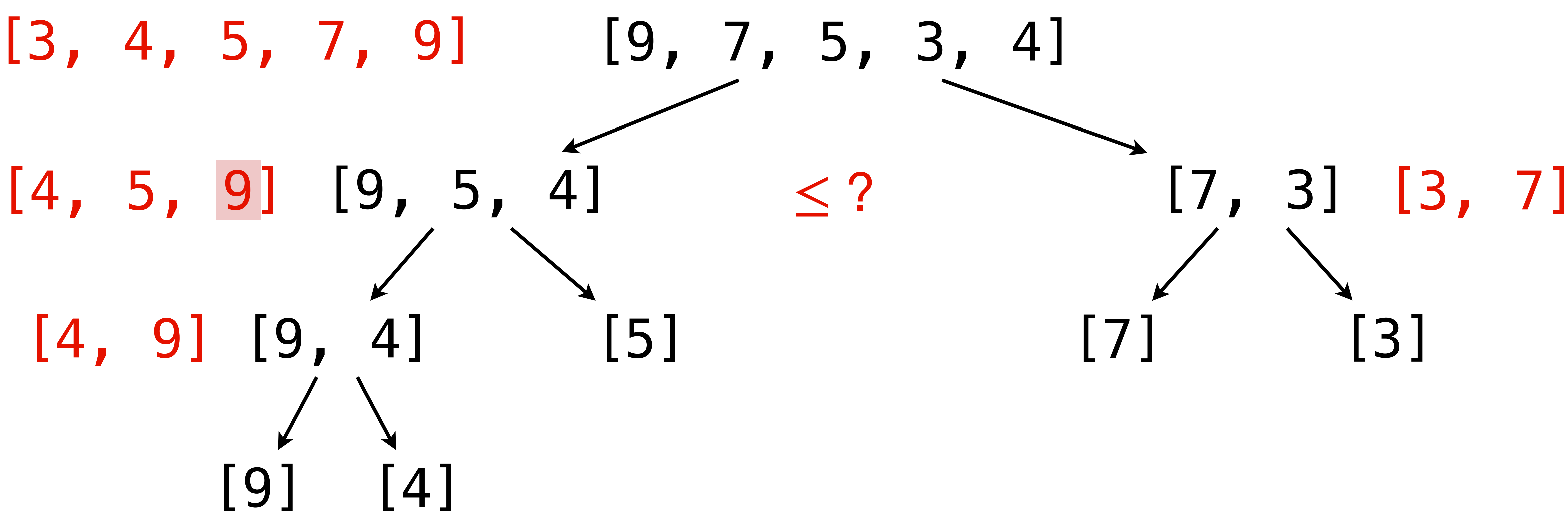
Mergesort: divide and conquer

Now, let's **merge**:



Mergesort: divide and conquer

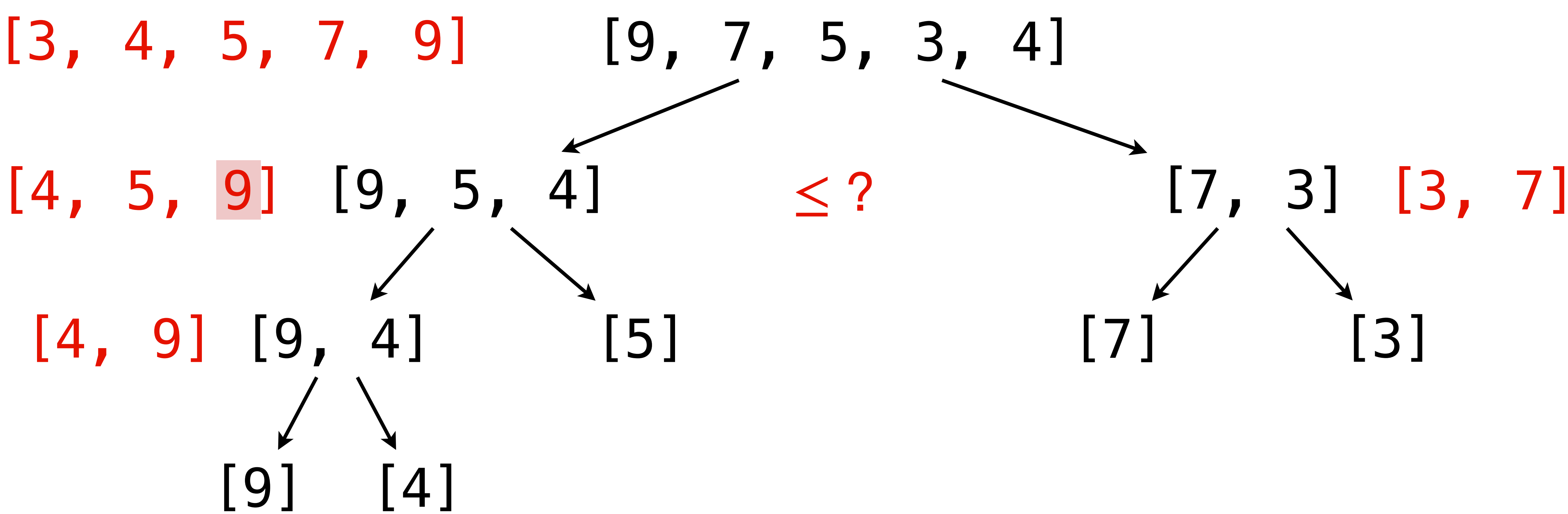
Now, let's **merge**:



Note, we use a list here.

Mergesort: divide and conquer

Now, let's **merge**:



- Note, we use a list here.
- But there is almost a tree emerging…

Let's write the mergesort function!

Let’s write the mergesort function!

```
(* msort : int list -> int list
```

Let’s write the mergesort function!

```
(* msort : int list -> int list
   REQUIRES: true
```

Let’s write the mergesort function!

```
(* msort : int list -> int list
   REQUIRES: true
   ENSURES:  msort(L) evaluates to a sorted
             permutation of L.
*)
```

Let's write the mergesort function!

```
(* msort : int list -> int list
   REQUIRES: true
   ENSURES:  msort(L) evaluates to a sorted
             permutation of L.
*)
fun msort ([] : int list) : int list =
```

Let’s write the mergesort function!

```
(* msort : int list -> int list
   REQUIRES: true
   ENSURES:  msort(L) evaluates to a sorted
             permutation of L.
*)
fun msort ([] : int list) : int list = []
```

Let's write the mergesort function!

```
(* msort : int list -> int list
   REQUIRES: true
   ENSURES:  msort(L) evaluates to a sorted
             permutation of L.
*)
fun msort ([] : int list) : int list = []
  | msort [x] =
```

Let's write the mergesort function!

```
(* msort : int list -> int list
   REQUIRES: true
   ENSURES:  msort(L) evaluates to a sorted
             permutation of L.
*)
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
```

Let’s write the mergesort function!

```
(* msort : int list -> int list
   REQUIRES: true
   ENSURES:  msort(L) evaluates to a sorted
             permutation of L.
*)
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
```

Let's write the mergesort function!

```
(* msort : int list -> int list
   REQUIRES: true
   ENSURES:  msort(L) evaluates to a sorted
             permutation of L.
*)
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let
         val
      in

      end
```

Let's write the mergesort function!

```
(* msort : int list -> int list
   REQUIRES: true
   ENSURES:  msort(L) evaluates to a sorted
             permutation of L.
*)
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let
         val (A, B) = split L
      in

      end
```

Let's write the mergesort function!

```
(* msort : int list -> int list
   REQUIRES: true
   ENSURES:  msort(L) evaluates to a sorted
             permutation of L.
*)
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let
         val (A, B) = split L
      in
         merge(msort A, msort B)
      end
```

Now, let's write split!

Now, let's write split!

```
(* split : int list -> int list * int list
```

Now, let's write split!

```
(* split : int list -> int list * int list
   REQUIRES: true
```

Now, let's write split!

```
(* split : int list -> int list * int list
   REQUIRES: true
   ENSURES: split(L) evaluates to a pair of lists (A, B)
            such that length(A) and length(B) differ by
            at most 1, and A@B is a permutation of L.
*)
```

Now, let's write split!

```
(* split : int list -> int list * int list
   REQUIRES: true
   ENSURES: split(L) evaluates to a pair of lists (A, B)
            such that length(A) and length(B) differ by
            at most 1, and A@B is a permutation of L.
*)

fun split ([] : int list) : int list * int list =
```

Now, let's write split!

```
(* split : int list -> int list * int list
   REQUIRES: true
   ENSURES: split(L) evaluates to a pair of lists (A, B)
            such that length(A) and length(B) differ by
            at most 1, and A@B is a permutation of L.
*)

fun split ([] : int list) : int list * int list = ([], [])
```

Now, let's write split!

```
(* split : int list -> int list * int list
   REQUIRES: true
   ENSURES: split(L) evaluates to a pair of lists (A, B)
            such that length(A) and length(B) differ by
            at most 1, and A@B is a permutation of L.
*)

fun split ([] : int list) : int list * int list = ([], [])
  | split [x] =
```

Now, let's write split!

```
(* split : int list -> int list * int list
   REQUIRES: true
   ENSURES: split(L) evaluates to a pair of lists (A, B)
            such that length(A) and length(B) differ by
            at most 1, and A@B is a permutation of L.
*)

fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
```

Now, let's write split!

```
(* split : int list -> int list * int list
   REQUIRES: true
   ENSURES: split(L) evaluates to a pair of lists (A, B)
            such that length(A) and length(B) differ by
            at most 1, and A@B is a permutation of L.
*)

fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
```

Now, let's write split!

```
(* split : int list -> int list * int list
   REQUIRES: true
   ENSURES: split(L) evaluates to a pair of lists (A, B)
            such that length(A) and length(B) differ by
            at most 1, and A@B is a permutation of L.
*)

fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
       let
          val
       in

       end
```

Now, let's write split!

```
(* split : int list -> int list * int list
   REQUIRES: true
   ENSURES: split(L) evaluates to a pair of lists (A, B)
            such that length(A) and length(B) differ by
            at most 1, and A@B is a permutation of L.
*)

fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
      let
         val (A, B) = split L
      in

      end
```

Now, let's write split!

```
(* split : int list -> int list * int list
   REQUIRES: true
   ENSURES: split(L) evaluates to a pair of lists (A, B)
            such that length(A) and length(B) differ by
            at most 1, and A@B is a permutation of L.
*)

fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
      let
         val (A, B) = split L
      in
         (x::A, y::B)
      end
```

Now, let's write split!

```
(* split : int list -> int list * int list
   REQUIRES: true
   ENSURES: split(L) evaluates to a pair of lists (A, B)
            such that length(A) and length(B) differ by
            at most 1, and A@B is a permutation of L.
*)

fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
      let
         val (A, B) = split L
      in
         (x::A, y::B)
      end
```

Now, let's write split!

```
(* split : int list -> int list * int list
   REQUIRES: true
   ENSURES: split(L) evaluates to a pair of lists (A, B)
            such that length(A) and length(B) differ by
            at most 1, and A@B is a permutation of L.
*)

fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
      let
         val (A, B) = split L
      in
         (x::A, y::B)
      end
```

Have we established post-condition?

Now, let's write split!

```
(* split : int list -> int list * int list
   REQUIRES: true
   ENSURES: split(L) evaluates to a pair of lists (A, B)
            such that length(A) and length(B) differ by
            at most 1, and A@B is a permutation of L.
*)

fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
      let
         val (A, B) = split L
      in
         (x::A, y::B)
      end
```

Have we established post-condition?

Prove in your head as you write code!

Work for split

Work for split

```
fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
      let
         val (A, B) = split L
      in
         (x::A, y::B)
      end
```

Work for split

```
fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
       let
          val (A, B) = split L
       in
          (x::A, y::B)
       end
```

Work: $W_{split}(n)$ with n the list length.

Work for split

```
fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
      let
         val (A, B) = split L
      in
         (x::A, y::B)
      end
```

Work: $W_{split}(n)$ with n the list length.

Equations:

Work for split

```
fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
       let
          val (A, B) = split L
       in
          (x::A, y::B)
       end
```

Work: $W_{split}(n)$ with n the list length.

Equations:

$W_{split}(0) =$

Work for split

```
fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
      let
         val (A, B) = split L
      in
         (x::A, y::B)
      end
```

Work: $W_{\texttt{split}}(\texttt{n})$ with $\texttt{n}$ the list length.

Equations:

$W_{\texttt{split}}(0) = c_0$

Work for split

```
fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
      let
         val (A, B) = split L
      in
         (x::A, y::B)
      end
```

Work: $W_{split}(n)$ with n the list length.

Equations:

$W_{split}(0) = c_0$
$W_{split}(1) =$

Work for split

```
fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
      let
         val (A, B) = split L
      in
         (x::A, y::B)
      end
```

Work: $W_{split}(n)$ with n the list length.

Equations:

$W_{split}(0) = c_0$
$W_{split}(1) = c_1$

Work for split

```
fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
       let
          val (A, B) = split L
       in
          (x::A, y::B)
       end
```

Work: $W_{split}(n)$ with n the list length.

Equations:

$W_{split}(0) = c_0$
$W_{split}(1) = c_1$
$W_{split}(n) =$

Work for split

```
fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
       let
          val (A, B) = split L
       in
          (x::A, y::B)
       end
```

Work: $W_{split}(n)$ with n the list length.

Equations:

$W_{split}(0) = c_0$

$W_{split}(1) = c_1$

$W_{split}(n) = c_2 + W_{split}(n-2)$, for $n \geq 2$

Work for split

```
fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
      let
         val (A, B) = split L
      in
         (x::A, y::B)
      end
```

Work: $W_{split}(n)$ with n the list length.

Equations:

$W_{split}(0) = c_0$
$W_{split}(1) = c_1$
$W_{split}(n) = c_2 + W_{split}(n-2)$, for $n \geq 2$

Consequently: $W_{split}(n)$ is $O(n)$.

Work for split

```
fun split ([] : int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
      let
         val (A, B) = split L
      in
         (x::A, y::B)
      end
```

no opportunity for parallelism

Work: $W_{split}(n)$ with n the list length.

Equations:

$W_{split}(0) = c_0$
$W_{split}(1) = c_1$
$W_{split}(n) = c_2 + W_{split}(n-2)$, for $n \geq 2$

Consequently: $W_{split}(n)$ is $O(n)$.

Now, let’s write merge!

Now, let's write merge!

```
(* merge : int list * int list -> int list
```

Now, let’s write merge!

```
(* merge : int list * int list -> int list
   REQUIRES: A and B are sorted lists.
```

Now, let's write merge!

```
(* merge : int list * int list -> int list
   REQUIRES: A and B are sorted lists.
   ENSURES:  merge(A,B) evaluates to a sorted
             permutation of A@B.
*)
```

Now, let's write merge!

```
(* merge : int list * int list -> int list
   REQUIRES: A and B are sorted lists.
   ENSURES:  merge(A,B) evaluates to a sorted
             permutation of A@B.
*)

fun merge ([] : int list, B : int list) : int list =
```

Now, let's write merge!

```
(* merge : int list * int list -> int list
   REQUIRES: A and B are sorted lists.
   ENSURES:  merge(A,B) evaluates to a sorted
             permutation of A@B.
*)

fun merge ([] : int list, B : int list) : int list = B
```

Now, let's write merge!

```
(* merge : int list * int list -> int list
   REQUIRES: A and B are sorted lists.
   ENSURES:  merge(A,B) evaluates to a sorted
             permutation of A@B.
*)

fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) =
```

Now, let's write merge!

```
(* merge : int list * int list -> int list
   REQUIRES: A and B are sorted lists.
   ENSURES:  merge(A,B) evaluates to a sorted
             permutation of A@B.
*)

fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
```

Now, let's write merge!

```
(* merge : int list * int list -> int list
   REQUIRES: A and B are sorted lists.
   ENSURES:  merge(A,B) evaluates to a sorted
             permutation of A@B.
*)

fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) =
```

Now, let's write merge!

```
(* merge : int list * int list -> int list
   REQUIRES: A and B are sorted lists.
   ENSURES:  merge(A,B) evaluates to a sorted
             permutation of A@B.
*)

fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
```

Now, let's write merge!

```
(* merge : int list * int list -> int list
   REQUIRES: A and B are sorted lists.
   ENSURES:  merge(A,B) evaluates to a sorted
             permutation of A@B.
*)

fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                               LESS =>
```

Now, let's write merge!

```
(* merge : int list * int list -> int list
   REQUIRES: A and B are sorted lists.
   ENSURES:  merge(A,B) evaluates to a sorted
             permutation of A@B.
*)

fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                              LESS => x :: merge(A, y::B)
```

Now, let's write merge!

```
(* merge : int list * int list -> int list
   REQUIRES: A and B are sorted lists.
   ENSURES:  merge(A,B) evaluates to a sorted
             permutation of A@B.
*)

fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                               LESS => x :: merge(A, y::B)
                             | EQUAL =>
```

Now, let's write merge!

```
(* merge : int list * int list -> int list
   REQUIRES: A and B are sorted lists.
   ENSURES:  merge(A,B) evaluates to a sorted
             permutation of A@B.
*)

fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                              LESS => x :: merge(A, y::B)
                            | EQUAL => x::y::merge(A, B)
```

Now, let's write merge!

```
(* merge : int list * int list -> int list
   REQUIRES: A and B are sorted lists.
   ENSURES:  merge(A,B) evaluates to a sorted
             permutation of A@B.
*)

fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                              LESS => x :: merge(A, y::B)
                            | EQUAL => x::y::merge(A, B)
                            | GREATER =>
```

Now, let's write merge!

```
(* merge : int list * int list -> int list
   REQUIRES: A and B are sorted lists.
   ENSURES:  merge(A,B) evaluates to a sorted
             permutation of A@B.
*)

fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                                 LESS => x :: merge(A, y::B)
                               | EQUAL => x::y::merge(A, B)
                               | GREATER => y :: merge(x::A, B))
```

Work for merge

Work for merge

```
fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                               LESS => x :: merge(A, y::B)
                             | EQUAL => x::y::merge(A, B)
                             | GREATER => y :: merge(x::A, B))
```

Work for merge

```
fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                              LESS => x :: merge(A, y::B)
                            | EQUAL => x::y::merge(A, B)
                            | GREATER => y :: merge(x::A, B))
```

Work: $W_{merge}(n,m)$ for `merge(A,B)` with `n`, `m` the length of `A`, `B`, resp.

Work for merge

```
fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                              LESS => x :: merge(A, y::B)
                            | EQUAL => x::y::merge(A, B)
                            | GREATER => y :: merge(x::A, B))
```

Work: $W_{merge}(n,m)$ for `merge(A,B)` with `n`, `m` the length of `A`, `B`, resp.

Equations:

Work for merge

```
fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                              LESS => x :: merge(A, y::B)
                            | EQUAL => x::y::merge(A, B)
                            | GREATER => y :: merge(x::A, B))
```

Work: $W_{merge}(n,m)$ for `merge(A,B)` with `n`, `m` the length of `A`, `B`, resp.

Equations:

$W_{merge}(0,m) =$

Work for merge

```
fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                              LESS => x :: merge(A, y::B)
                            | EQUAL => x::y::merge(A, B)
                            | GREATER => y :: merge(x::A, B))
```

Work: $W_{merge}(n,m)$ for `merge(A,B)` with `n`, `m` the length of `A`, `B`, resp.

Equations:

$W_{merge}(0,m) = c_0$, for all $m \geq 0$

Work for merge

```
fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                              LESS => x :: merge(A, y::B)
                            | EQUAL => x::y::merge(A, B)
                            | GREATER => y :: merge(x::A, B))
```

Work: $W_{merge}(n,m)$ for `merge(A,B)` with `n`, `m` the length of `A`, `B`, resp.

Equations:

$W_{merge}(0,m) = c_0$, for all $m \geq 0$

$W_{merge}(n,0) =$

Work for merge

```
fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                              LESS => x :: merge(A, y::B)
                            | EQUAL => x::y::merge(A, B)
                            | GREATER => y :: merge(x::A, B))
```

Work: $W_{merge}(n,m)$ for `merge(A,B)` with `n`, `m` the length of `A`, `B`, resp.

Equations:

$W_{merge}(0,m) = c_0$, for all $m \geq 0$

$W_{merge}(n,0) = c_1$, for all $n \geq 0$

Work for merge

```
fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                              LESS => x :: merge(A, y::B)
                            | EQUAL => x::y::merge(A, B)
                            | GREATER => y :: merge(x::A, B))
```

Work: $W_{merge}(n,m)$ for `merge(A,B)` with `n`, `m` the length of `A`, `B`, resp.

Equations:

$W_{merge}(0,m) = c_0$, for all $m \geq 0$

$W_{merge}(n,0) = c_1$, for all $n \geq 0$

$W_{merge}(n,m) =$

Work for merge

```
fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                             LESS => x :: merge(A, y::B)
                           | EQUAL => x::y::merge(A, B)
                           | GREATER => y :: merge(x::A, B))
```

Work: $W_{merge}(n,m)$ for `merge(A,B)` with `n`, `m` the length of `A`, `B`, resp.

Equations:

$W_{merge}(0,m) = c_0$, for all $m \geq 0$

$W_{merge}(n,0) = c_1$, for all $n \geq 0$

$W_{merge}(n,m) = k_1 + W_{merge}(n-1,m)$, for $n, m > 0$ and case LESS

Work for merge

```
fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                              LESS => x :: merge(A, y::B)
                            | EQUAL => x::y::merge(A, B)
                            | GREATER => y :: merge(x::A, B))
```

Work: $W_{merge}(n,m)$ for `merge(A,B)` with `n`, `m` the length of `A`, `B`, resp.

Equations:

$W_{merge}(0,m) = c_0$, for all $m \geq 0$

$W_{merge}(n,0) = c_1$, for all $n \geq 0$

$W_{merge}(n,m) = k_1 + W_{merge}(n-1,m)$, for $n, m > 0$ and case LESS

$W_{merge}(n,m) =$

Work for merge

```
fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                              LESS => x :: merge(A, y::B)
                            | EQUAL => x::y::merge(A, B)
                            | GREATER => y :: merge(x::A, B))
```

Work: $W_{merge}(n,m)$ for `merge(A,B)` with `n`, `m` the length of `A`, `B`, resp.

Equations:

$W_{merge}(0,m) = c_0$, for all $m \geq 0$

$W_{merge}(n,0) = c_1$, for all $n \geq 0$

$W_{merge}(n,m) = k_1 + W_{merge}(n-1,m)$, for n, m > 0 and case LESS

$W_{merge}(n,m) = k_2 + W_{merge}(n-1,m-1)$, for n, m > 0 and case EQUAL

Work for merge

```
fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                              LESS => x :: merge(A, y::B)
                            | EQUAL => x::y::merge(A, B)
                            | GREATER => y :: merge(x::A, B))
```

Work: $W_{merge}(n,m)$ for `merge(A,B)` with `n`, `m` the length of `A`, `B`, resp.

Equations:

$W_{merge}(0,m) = c_0$, for all $m \geq 0$
$W_{merge}(n,0) = c_1$, for all $n \geq 0$
$W_{merge}(n,m) = k_1 + W_{merge}(n-1,m)$, for $n, m > 0$ and case LESS
$W_{merge}(n,m) = k_2 + W_{merge}(n-1,m-1)$, for $n, m > 0$ and case EQUAL
$W_{merge}(n,m) =$

Work for merge

```
fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x::A, y::B) = (case compare(x,y) of
                              LESS => x :: merge(A, y::B)
                            | EQUAL => x::y::merge(A, B)
                            | GREATER => y :: merge(x::A, B))
```

Work: $W_{merge}(n,m)$ for `merge(A,B)` with `n`, `m` the length of `A`, `B`, resp.

Equations:

$W_{merge}(0,m) = c_0$, for all $m \geq 0$
$W_{merge}(n,0) = c_1$, for all $n \geq 0$
$W_{merge}(n,m) = k_1 + W_{merge}(n-1,m)$, for n, m > 0 and case LESS
$W_{merge}(n,m) = k_2 + W_{merge}(n-1,m-1)$, for n, m > 0 and case EQUAL
$W_{merge}(n,m) = k_3 + W_{merge}(n,m-1)$, for n, m > 0 and case GREATER

Work for merge

Work: $W_{merge}(n,m)$ for `merge(A,B)` with `n`, `m` the length of `A`, `B`, resp.

Equations:

$W_{merge}(0,m) = c_0$, for all $m \geq 0$
$W_{merge}(n,0) = c_1$, for all $n \geq 0$
$W_{merge}(n,m) = k_1 + W_{merge}(n-1,m)$, for n, m > 0 and case LESS
$W_{merge}(n,m) = k_2 + W_{merge}(n-1,m-1)$, for n, m > 0 and case EQUAL
$W_{merge}(n,m) = k_3 + W_{merge}(n,m-1)$, for n, m > 0 and case GREATER

Work for merge

Work: W_{merge}(n,m) for merge(A,B) with n, m the length of A, B, resp.

Equations:

$W_{merge}(0,m) = c_0$, for all $m \geq 0$
$W_{merge}(n,0) = c_1$, for all $n \geq 0$
$W_{merge}(n,m) = k_1 + W_{merge}(n-1,m)$, for n, m > 0 and case LESS
$W_{merge}(n,m) = k_2 + W_{merge}(n-1,m-1)$, for n, m > 0 and case EQUAL
$W_{merge}(n,m) = k_3 + W_{merge}(n,m-1)$, for n, m > 0 and case GREATER

Consequently: $W_{merge}(n,m)$ is $O(n+m)$.

Work for merge

Work: W_{merge}`(n,m)` for `merge(A,B)` with `n`, `m` the length of `A`, `B`, resp.

Equations:

$W_{merge}(0,m) = c_0$, for all $m \geq 0$
$W_{merge}(n,0) = c_1$, for all $n \geq 0$
$W_{merge}(n,m) = k_1 + W_{merge}(n-1,m)$, for n, m > 0 and case LESS
$W_{merge}(n,m) = k_2 + W_{merge}(n-1,m-1)$, for n, m > 0 and case EQUAL
$W_{merge}(n,m) = k_3 + W_{merge}(n,m-1)$, for n, m > 0 and case GREATER

Consequently: $W_{merge}(n,m)$ is $O(n+m)$.

Note: again, no opportunity for parallelism.

Finally, work for mergesort!

Finally, work for mergesort!

```
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let
         val (A, B) = split L
      in
         merge(msort A, msort B)
      end
```

Finally, work for mergesort!

```
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let
         val (A, B) = split L
      in
         merge(msort A, msort B)
      end
```

Work: $W_{msort}(n)$ with n the list length.

Equations:

Finally, work for mergesort!

```
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let
         val (A, B) = split L
      in
         merge(msort A, msort B)
      end
```

Work: $W_{msort}(n)$ with n the list length.

Equations:

$W_{msort}(0) =$

Finally, work for mergesort!

```
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let
         val (A, B) = split L
      in
         merge(msort A, msort B)
      end
```

Work: $W_{msort}(n)$ with n the list length.

Equations:

$W_{msort}(0) = c_0$

Finally, work for mergesort!

```
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let
         val (A, B) = split L
      in
         merge(msort A, msort B)
      end
```

Work: $W_{msort}(n)$ with n the list length.

Equations:

$W_{msort}(0) = c_0$

$W_{msort}(1) =$

Finally, work for mergesort!

```
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let
         val (A, B) = split L
      in
         merge(msort A, msort B)
      end
```

Work: $W_{msort}(n)$ with n the list length.

Equations:

$W_{msort}(0) = c_0$
$W_{msort}(1) = c_1$

Finally, work for mergesort!

```
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let
         val (A, B) = split L
      in
         merge(msort A, msort B)
      end
```

Work: $W_{msort}(n)$ with n the list length.

Equations:

$W_{msort}(0) = c_0$
$W_{msort}(1) = c_1$
$W_{msort}(n) =$

Finally, work for mergesort!

```
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let
         val (A, B) = split L
      in
         merge(msort A, msort B)
      end
```

Work: $W_{msort}(n)$ with n the list length.

Equations:

$W_{msort}(0) = c_0$
$W_{msort}(1) = c_1$
$W_{msort}(n) = c_2 + W_{split}(n) +$

$n \geq 2$

Finally, work for mergesort!

```
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let
         val (A, B) = split L
      in
         merge(msort A, msort B)
      end
```

Work: $W_{msort}(n)$ with n the list length.

Equations:

$$W_{msort}(0) = c_0$$
$$W_{msort}(1) = c_1$$
$$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$$

$n \geq 2$

Finally, work for mergesort!

```
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let
         val (A, B) = split L
      in
         merge(msort A, msort B)
      end
```

Work: $W_{\texttt{msort}}(n)$ with n the list length.

Equations:

$$
\begin{aligned}
W_{\texttt{msort}}(0) &= c_0 \\
W_{\texttt{msort}}(1) &= c_1 \\
W_{\texttt{msort}}(n) &= c_2 + W_{\texttt{split}}(n) + W_{\texttt{msort}}(n_a) + W_{\texttt{msort}}(n_b) \\
&\quad + W_{\texttt{merge}}(n_a, n_b), \text{ for } n = n_a + n_b \text{ and } n \geq 2
\end{aligned}
$$

Finally, work for mergesort!

Work: $W_{msort}(n)$ with n the list length.

Equations:

$$W_{msort}(0) = c_0$$
$$W_{msort}(1) = c_1$$
$$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$$
$$+ W_{merge}(n_a, n_b), \text{ for } n = n_a + n_b \text{ and } n \geq 2$$

Finally, work for mergesort!

Work: $\texttt{W}_{\texttt{msort}}\texttt{(n)}$ with $\texttt{n}$ the list length.

Equations:

$$\texttt{W}_{\texttt{msort}}\texttt{(0)} = c_0$$
$$\texttt{W}_{\texttt{msort}}\texttt{(1)} = c_1$$
$$\texttt{W}_{\texttt{msort}}\texttt{(n)} = c_2 + \texttt{W}_{\texttt{split}}\texttt{(n)} + \texttt{W}_{\texttt{msort}}\texttt{(n}_{\texttt{a}}\texttt{)} + \texttt{W}_{\texttt{msort}}\texttt{(n}_{\texttt{b}}\texttt{)}$$
$$+ \texttt{W}_{\texttt{merge}}\texttt{(n}_{\texttt{a}}\texttt{,n}_{\texttt{b}}\texttt{)}, \text{ for } \texttt{n} = \texttt{n}_{\texttt{a}} + \texttt{n}_{\texttt{b}} \text{ and } \texttt{n} \geq 2$$

Finally, work for mergesort!

Work: $W_{\texttt{msort}}(\texttt{n})$ with $\texttt{n}$ the list length.

Equations:

$$= \lfloor n/2 \rfloor$$

$$W_{\texttt{msort}}(\texttt{0}) = c_0$$

$$W_{\texttt{msort}}(\texttt{1}) = c_1$$

$$W_{\texttt{msort}}(\texttt{n}) = c_2 + W_{\texttt{split}}(\texttt{n}) + W_{\texttt{msort}}(\texttt{n}_\texttt{a}) + W_{\texttt{msort}}(\texttt{n}_\texttt{b})$$

$$+ W_{\texttt{merge}}(\texttt{n}_\texttt{a},\texttt{n}_\texttt{b}), \text{ for } \texttt{n} = \texttt{n}_\texttt{a} + \texttt{n}_\texttt{b} \text{ and } \texttt{n} \geq 2$$

Finally, work for mergesort!

Work: $W_{msort}(n)$ with n the list length.

Equations:

$= \lfloor n/2 \rfloor$

$$W_{msort}(0) = c_0$$
$$W_{msort}(1) = c_1$$
$$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$$
$$+ W_{merge}(n_a, n_b), \text{ for } n = n_a + n_b \text{ and } n \geq 2$$

Finally, work for mergesort!

Work: $W_{msort}(n)$ with n the list length.

Equations:

$= \lfloor n/2 \rfloor$ $= \lceil n/2 \rceil$

$W_{msort}(0) = c_0$

$W_{msort}(1) = c_1$

$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$
$+ W_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

Finally, work for mergesort!

Work: $W_{msort}(n)$ with n the list length.

Equations:

$$W_{msort}(0) = c_0$$
$$W_{msort}(1) = c_1$$
$$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$$
$$+ W_{merge}(n_a, n_b), \text{ for } n = n_a + n_b \text{ and } n \geq 2$$

Finally, work for mergesort!

Work: $W_{msort}(n)$ with n the list length.

Equations:

$W_{msort}(0) = c_0$

$W_{msort}(1) = c_1$

$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$
$+ W_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

Finally, work for mergesort!

Work: $\texttt{W}_{\texttt{msort}}\texttt{(n)}$ with $\texttt{n}$ the list length.

Equations:

$$\texttt{W}_{\texttt{msort}}\texttt{(0)} = c_0$$

$$\texttt{W}_{\texttt{msort}}\texttt{(1)} = c_1$$

$$\texttt{W}_{\texttt{msort}}\texttt{(n)} = c_2 + \texttt{W}_{\texttt{split}}\texttt{(n)} + \texttt{W}_{\texttt{msort}}\texttt{(n}_{\texttt{a}}\texttt{)} + \texttt{W}_{\texttt{msort}}\texttt{(n}_{\texttt{b}}\texttt{)}$$

$$+ \texttt{W}_{\texttt{merge}}\texttt{(n}_{\texttt{a}}\texttt{,n}_{\texttt{b}}\texttt{)}, \text{ for } \texttt{n} = \texttt{n}_{\texttt{a}} + \texttt{n}_{\texttt{b}} \text{ and } \texttt{n} \geq 2$$

$c\ \texttt{n}$

Finally, work for mergesort!

Work: $W_{msort}(n)$ with n the list length.

Equations:

$$W_{msort}(0) = c_0$$
$$W_{msort}(1) = c_1$$
$$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b) + W_{merge}(n_a, n_b), \text{ for } n = n_a + n_b \text{ and } n \geq 2$$

$c\,n$

Finally, work for mergesort!

Work: $W_{msort}(n)$ with n the list length.

Equations:

$W_{msort}(0) = c_0$

$W_{msort}(1) = c_1$

$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$
$+ W_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

$c\,n + c'\,n$

Finally, work for mergesort!

Work: $W_{msort}(n)$ with n the list length.

Equations:

$W_{msort}(0) = c_0$

$W_{msort}(1) = c_1$

$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$

$+ W_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

$c\,n + c'\,n = (c + c')\,n$

Finally, work for mergesort!

Work: $W_{msort}(n)$ with n the list length.

Equations:

$W_{msort}(0) = c_0$

$W_{msort}(1) = c_1$

$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$
$\quad + W_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

$c\ n + c'\ n = (c + c')\ n = c_3\ n$

Finally, work for mergesort!

Work: $W_{msort}(n)$ with n the list length.

Equations:

$= \lfloor n/2 \rfloor$ $= \lceil n/2 \rceil$

$W_{msort}(0) = c_0$
$W_{msort}(1) = c_1$
$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$
$+ W_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

$c\, n + c'\, n = (c + c')\, n = c_3\, n$

Finally, work for mergesort!

Work: $W_{msort}(n)$ with n the list length.

Equations:

$= \lfloor n/2 \rfloor$ $= \lceil n/2 \rceil$

$W_{msort}(0) = c_0$
$W_{msort}(1) = c_1$
$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$
$+ W_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

$c\ n + c'\ n = (c + c')\ n = c_3\ n$

$W_{msort}(n) \leq c_2 + c_3\ n + 2\ W_{msort}(n/2)$

Finally, work for mergesort!

Work: $W_{msort}(n)$ with n the list length.

Equations:

$= \lfloor n/2 \rfloor$ $= \lceil n/2 \rceil$

$W_{msort}(0) = c_0$
$W_{msort}(1) = c_1$
$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$
$+ W_{merge}(n_a,n_b)$, for $n = n_a + n_b$ and $n \geq 2$

$c\ n + c'\ n = (c + c')\ n = c_3\ n$

$W_{msort}(n) \leq c_2 + c_3\ n + 2\ W_{msort}(n/2)$
$W_{msort}(n) \leq c_4\ n + 2\ W_{msort}(n/2)$

Finally, work for mergesort!

Work: $W_{msort}(n)$ with n the list length.

Equations:

$= \lfloor n/2 \rfloor$ $= \lceil n/2 \rceil$

$W_{msort}(0) = c_0$
$W_{msort}(1) = c_1$
$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$
$+ W_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

$c\, n + c'\, n = (c + c')\, n = c_3\, n$

$W_{msort}(n) \leq c_2 + c_3\, n + 2\, W_{msort}(n/2)$
$W_{msort}(n) \leq c_4\, n + 2\, W_{msort}(n/2)$

➡ Let's look at the tree method to find a closed form.

Finally, work for mergesort!

$$W_{\mathtt{msort}}(n) \leq c_4\, n + 2\, W_{\mathtt{msort}}(n/2)$$

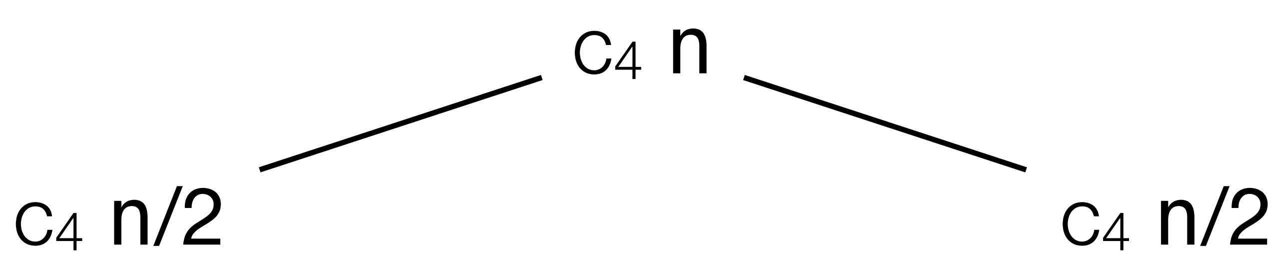
Finally, work for mergesort!

$$W_{msort}(n) \leq c_4\, n + 2\, W_{msort}(n/2)$$

$$c_4\, n$$

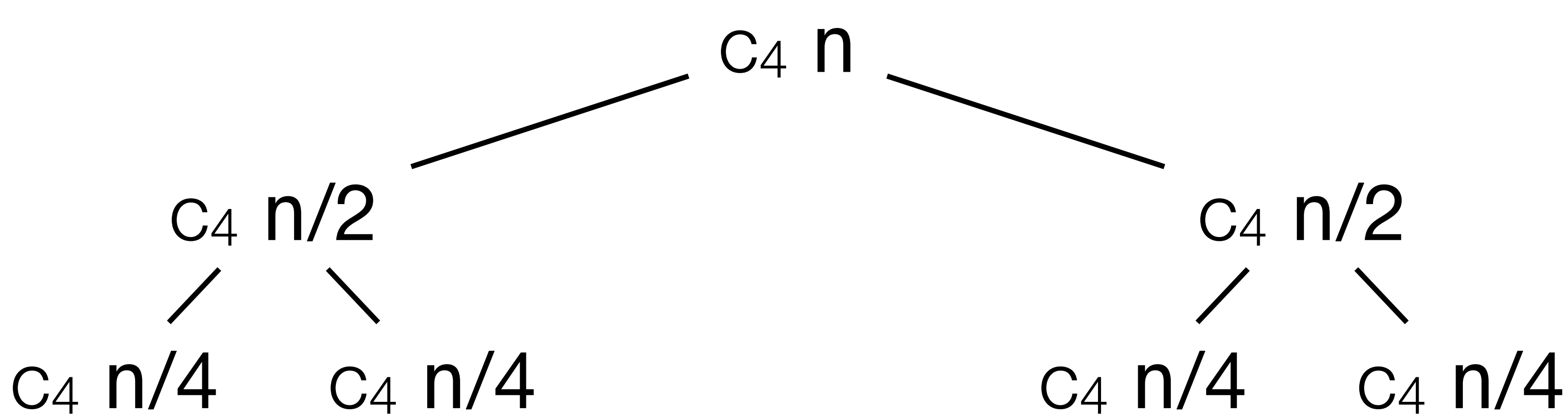
Finally, work for mergesort!

$$W_{\mathtt{msort}}(n) \leq c_4\, n + 2\, W_{\mathtt{msort}}(n/2)$$



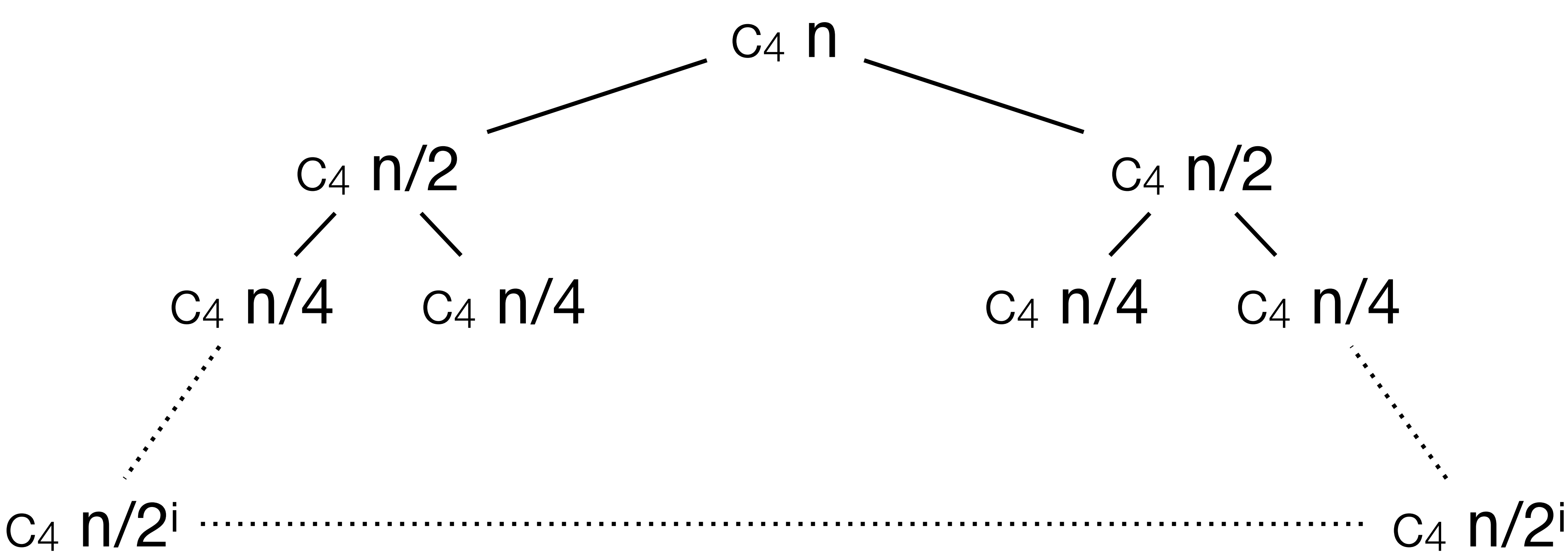
Finally, work for mergesort!

$$W_{msort}(n) \leq c_4\, n + 2\, W_{msort}(n/2)$$



Finally, work for mergesort!

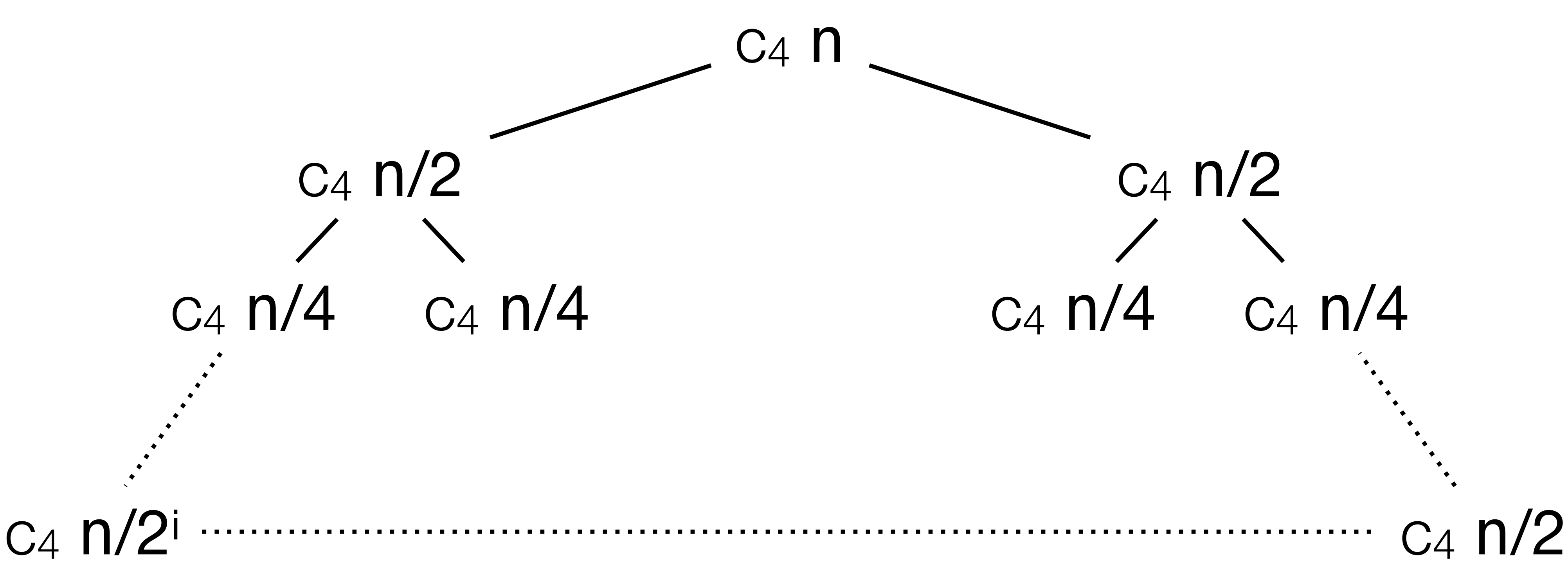
$$W_{\texttt{msort}}(n) \leq c_4\, n + 2\, W_{\texttt{msort}}(n/2)$$



Finally, work for mergesort!

$W_{\texttt{msort}}(n) \leq c_4\, n + 2\, W_{\texttt{msort}}(n/2)$

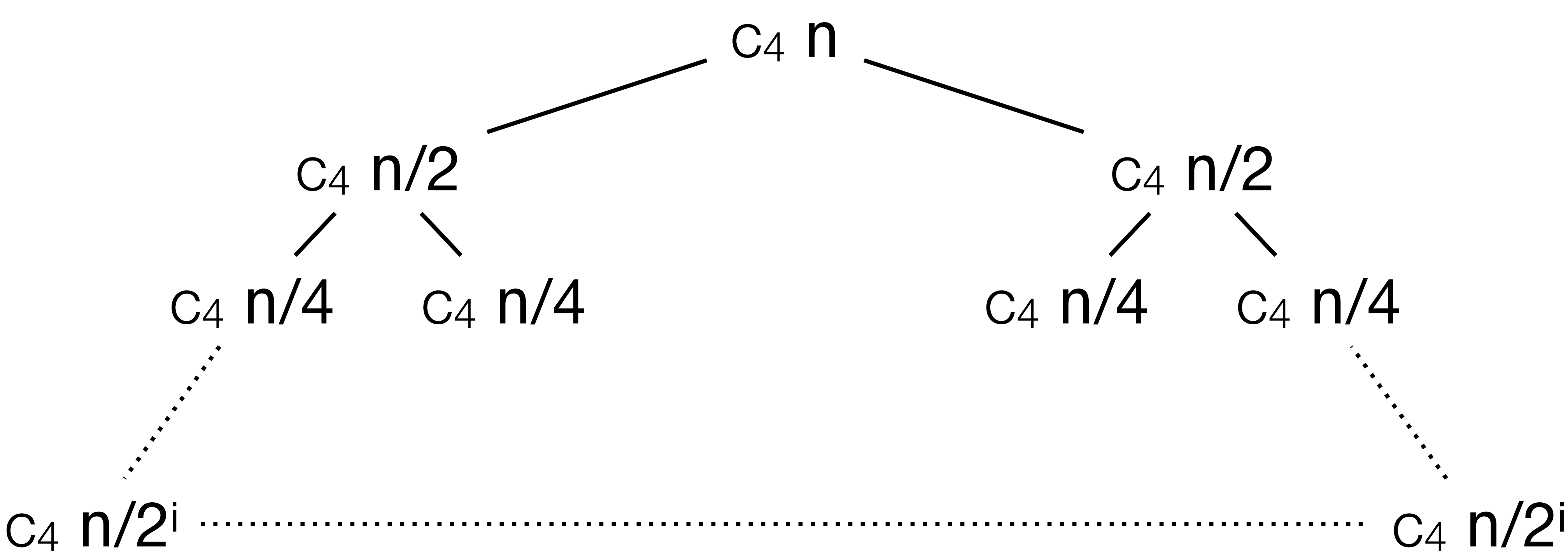
work per level:



Finally, work for mergesort!

$$W_{\texttt{msort}}(\texttt{n}) \leq c_4\, n + 2\, W_{\texttt{msort}}(n/2)$$

work per level:

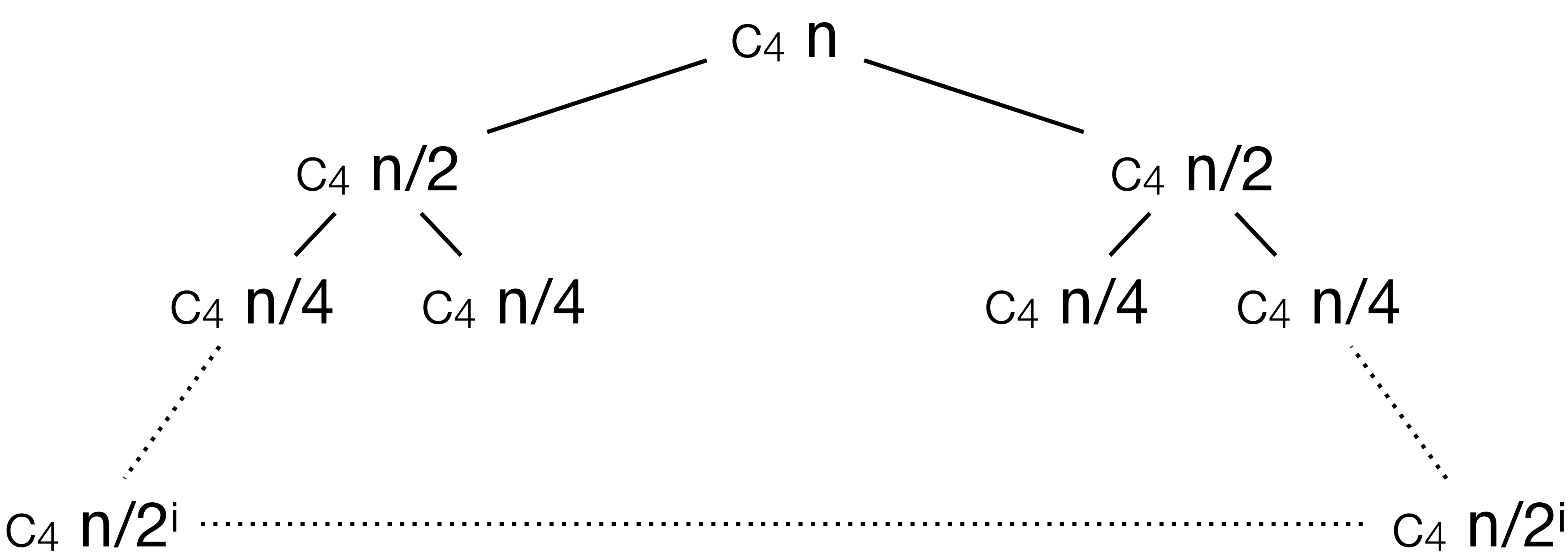


$c_4\, n$

Finally, work for mergesort!

$$W_{\texttt{msort}}(n) \leq c_4\, n + 2\, W_{\texttt{msort}}(n/2)$$

work per level:

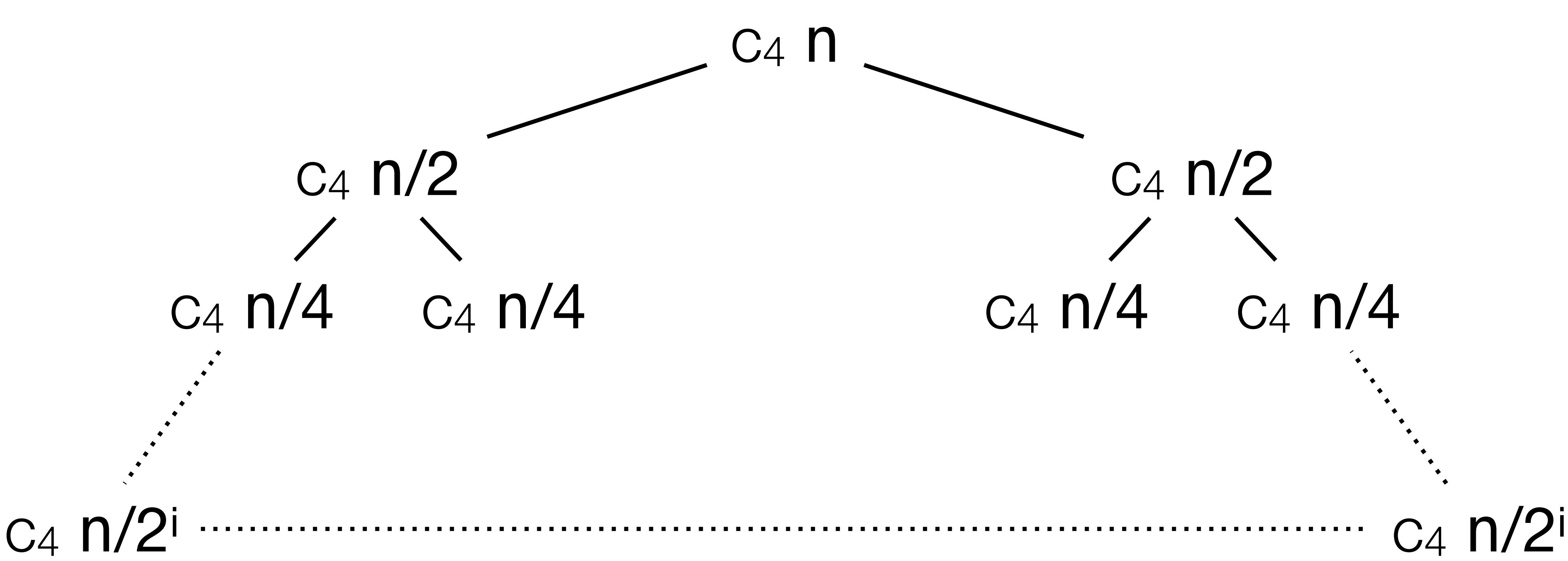


$c_4\, n$

$c_4\, n$

Finally, work for mergesort!

$$W_{\texttt{msort}}(n) \leq c_4\, n + 2\, W_{\texttt{msort}}(n/2)$$



work per level:

$c_4\, n$

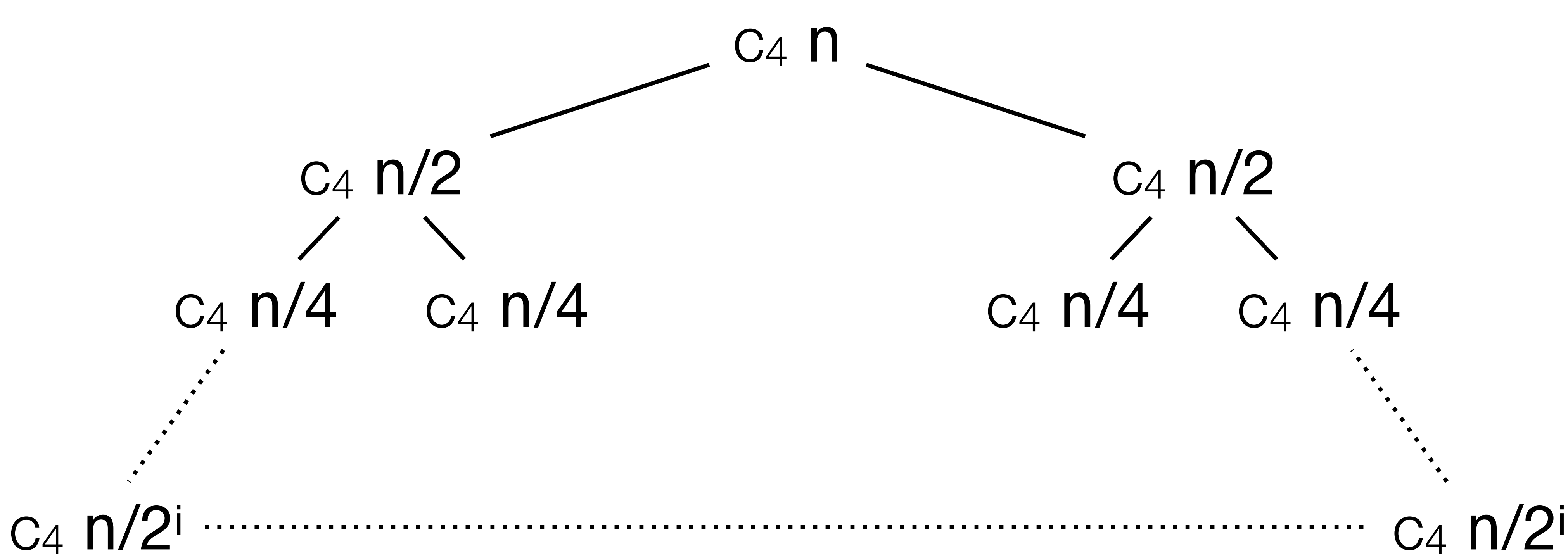
$c_4\, n$

$c_4\, n$

Finally, work for mergesort!

$$W_{\texttt{msort}}(n) \leq c_4\, n + 2\, W_{\texttt{msort}}(n/2)$$

work per level:



c_4 n

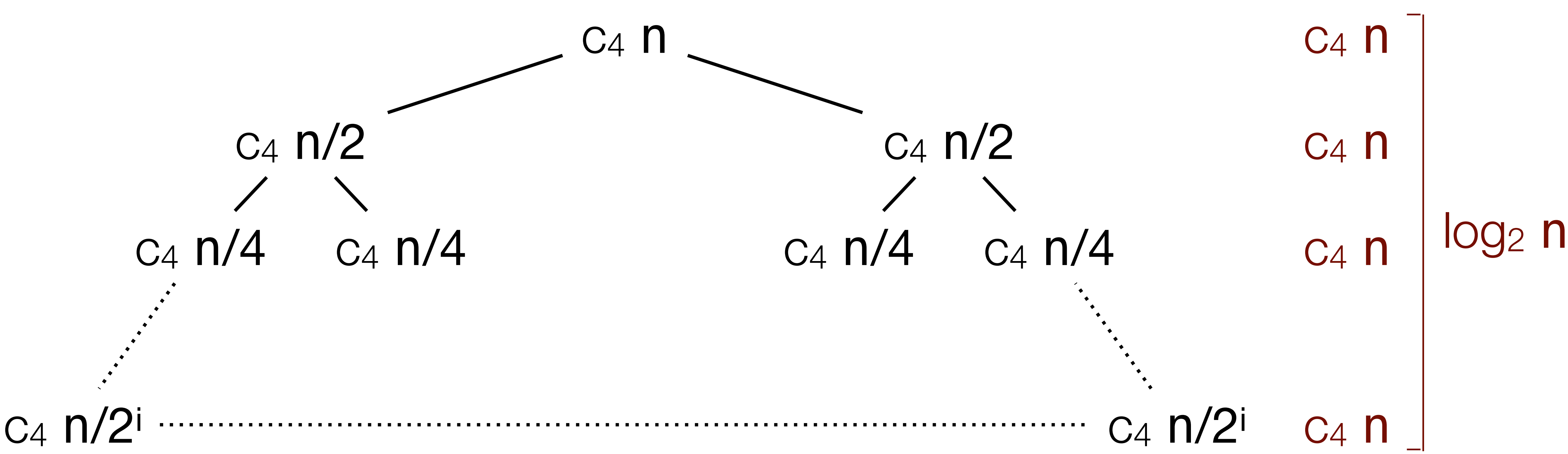
c_4 n

c_4 n

c_4 n

Finally, work for mergesort!

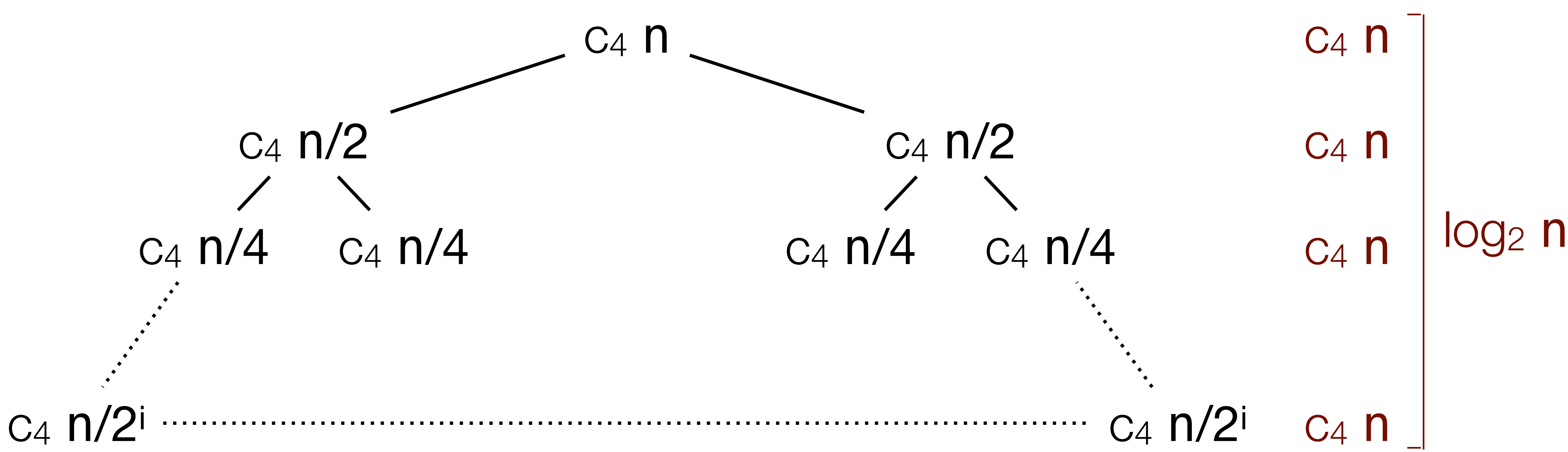
$$W_{\mathtt{msort}}(n) \leq c_4\, n + 2\, W_{\mathtt{msort}}(n/2)$$



Finally, work for mergesort!

$$W_{\texttt{msort}}(n) \leq c_4\, n + 2\, W_{\texttt{msort}}(n/2)$$

work per level:

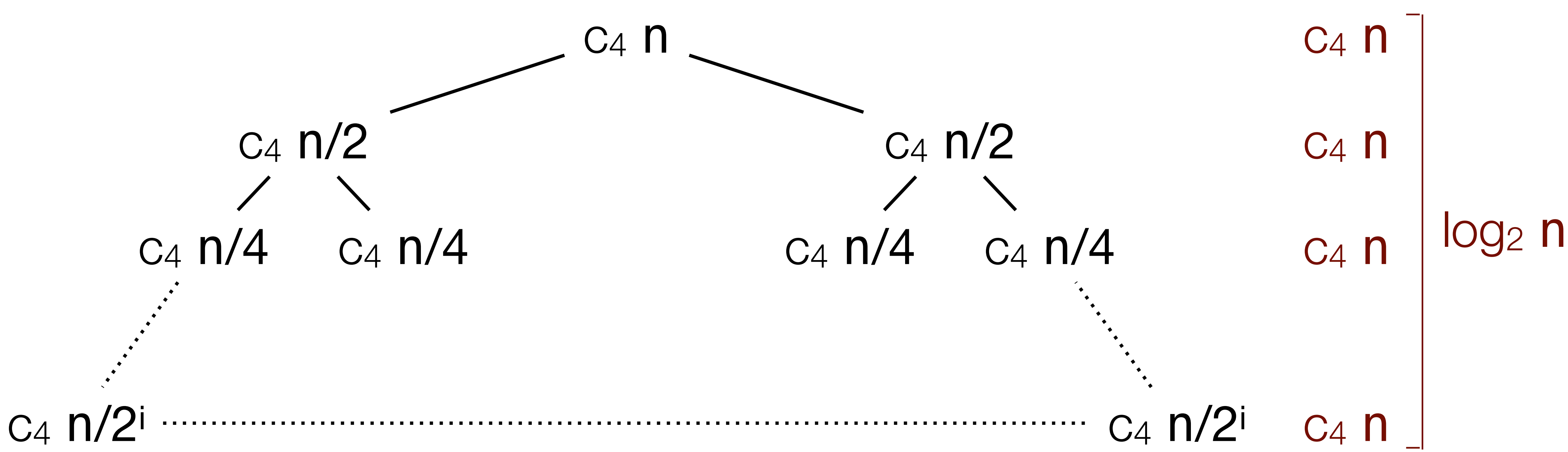


Consequently:

Finally, work for mergesort!

$W_{msort}(n) \leq c_4\, n + 2\, W_{msort}(n/2)$

work per level:

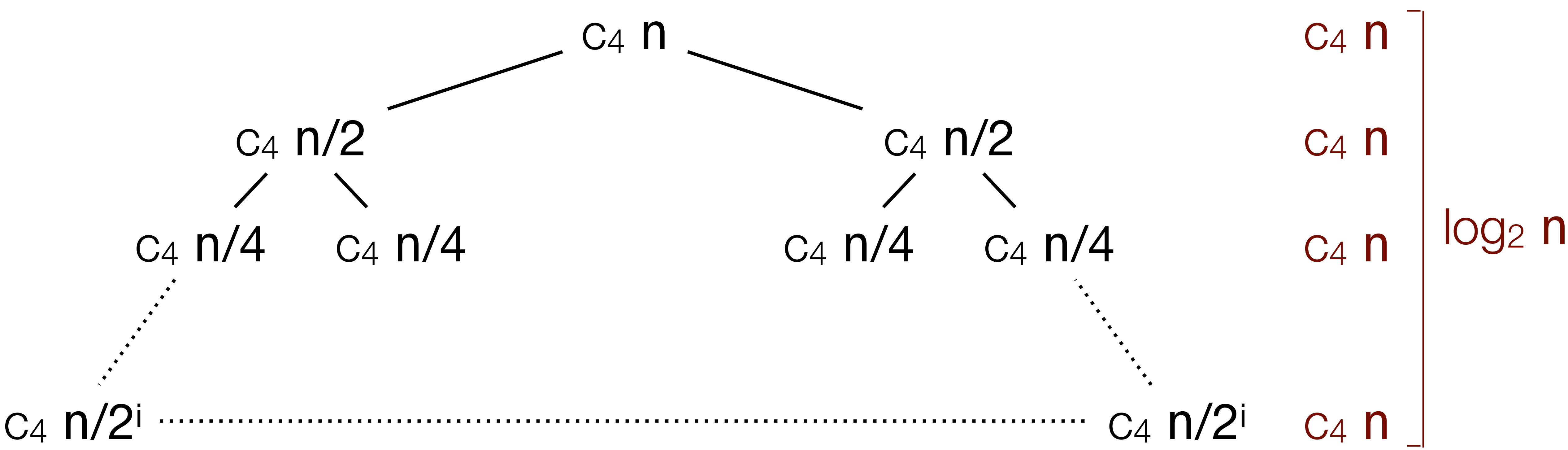


Consequently: $W_{msort}(n)$ is $O(n \log n)$.

Finally, work for mergesort!

$$W_{\texttt{msort}}(n) \leq c_4\, n + 2\, W_{\texttt{msort}}(n/2)$$

work per level:



Consequently: $W_{\texttt{msort}}(n)$ is $O(n \log n)$.

Is there an opportunity for parallelism?