YSC2229: Introductory Data Structures and Algorithms

Week 04: Advanced Sorting Techniques

Merge sort

- <u>Idea</u>: split the array to be sorted into two equal (± 1) parts, sort these arrays by *recursive* calls, and then *merge* them, preserving the ordering.

```
MergeSort(A[0 … n-1]) {
 if (n = 1) {
   return A;             // 1-element array, nothing to sort
 } else {
   m := n/2;
   L := A[0, …, m-1];    // split the array into Left and Right half (by copying)
   R := A[m, …, n-1];
   Merge(MergeSort(L), MergeSort(R), A)   // in-place merge results into A
   return A;
}
```

Merge sort by example

Recursive descent: *splitting* the array

0	1	2	3	4	5	6	7
1	2	12	8	18	3	4	6

0	1	2	3
1	2	12	8

0	1	2	3
18	3	4	6

0	1
1	2

0	1
12	8

0	1
18	3

0	1
4	6

0
1

0
2

0
12

0
8

0
18

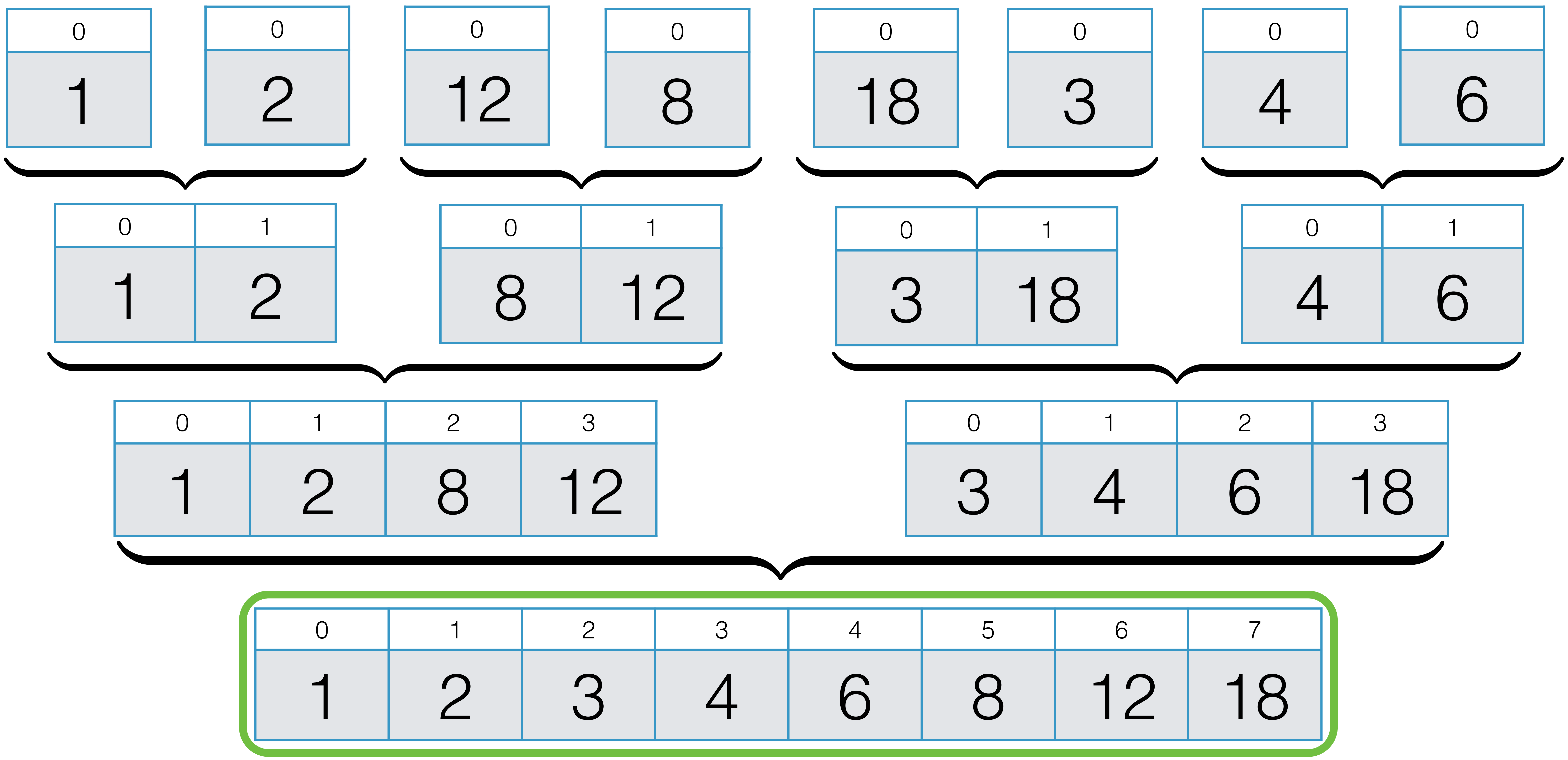
0
3

0
4

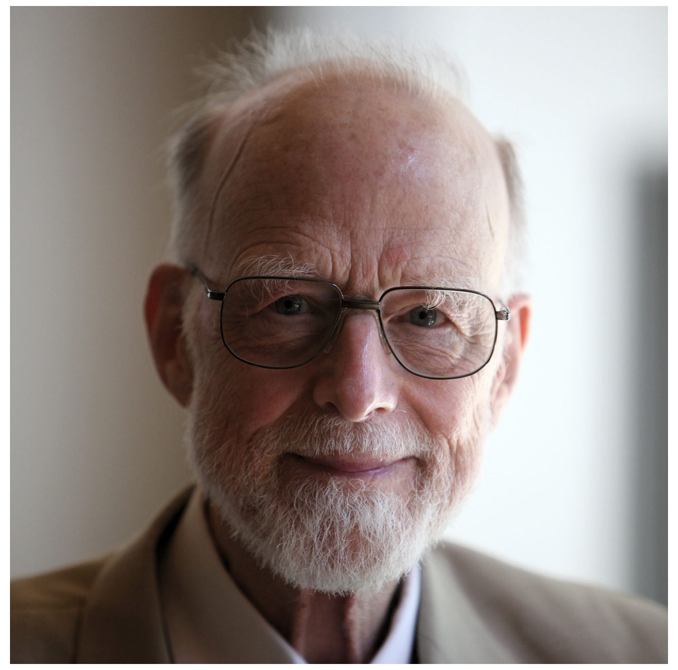
0
6

Merge sort by example

Merging the *sorted* sub-arrays



Quicksort

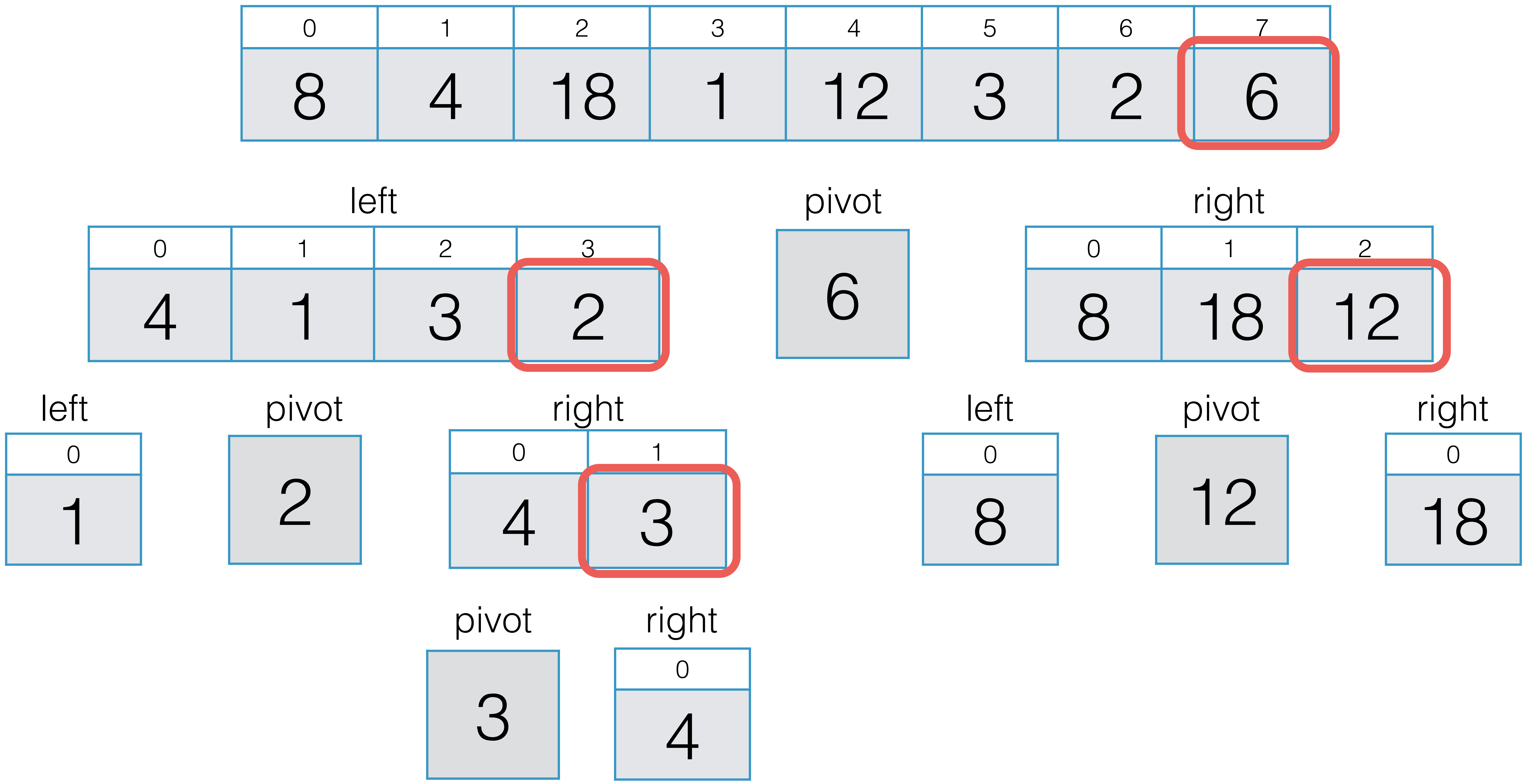


- Invented by Tony Hoare (the same as of Hoare triples) in 1961;
- <u>Idea</u>: divide-and-conquer with partially sorted sub-arrays;
- In practice, one of the *fastest* sorting algorithms as of today.

```
QuickSort (A[0 … n-1]) {
 if (n ≤ 1) { return A; }      // nothing to sort, return A
 else {
   l := 0; r := 0;
   pivot := A[n-1];            // take the last array element as a "pivot"
   for (i = 1 … n-1) {
    if (A[i] < pivot) then {
      L[l] := A[i];            // collect all elements of A smaller than pivot in
      l := l + 1;              // the "left" subarray L
    } else {
      R[r] := A[i];            // collect all elements of A greater or equal than pivot in
      r := r + 1;              // the "Right" subarray R
    }
   }
   Concat(QuickSort(L), pivot, QuickSort(R), A)  // run recursively on L, R, and then
   return A;                                      concatenate (L ++ [pivot] ++ R) into A
}
```

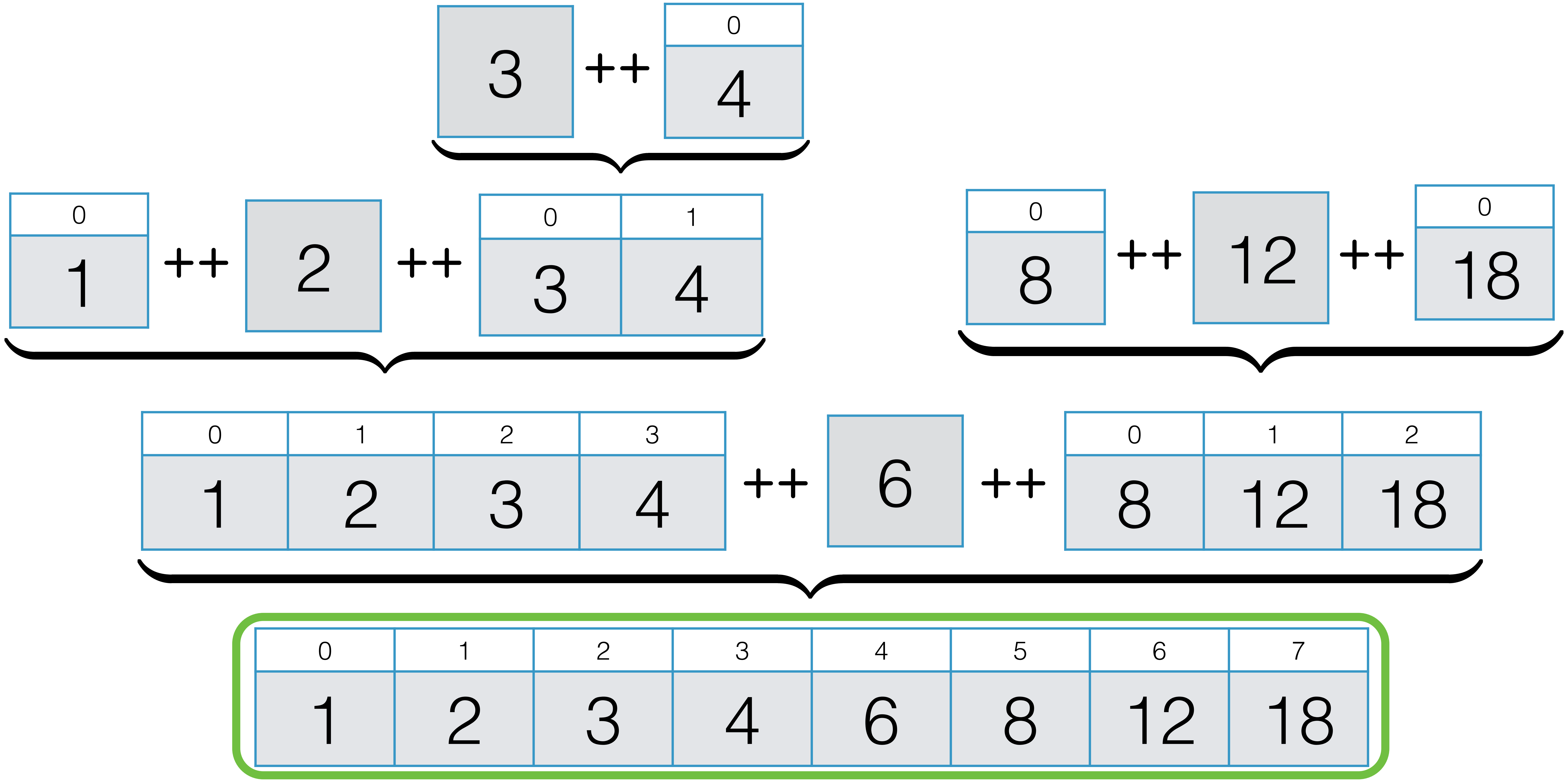
Quicksort by example

Recursive descent: *choosing pivots* and constructing sub-arrays



Quicksort by example

Combining *sorted* sub-arrays and pivots



Quicksort vs. Merge sort

- *Quicksort* can be seen as a complement to *Merge sort* in distributing the computational complexity;
- In *Merge sort*, creating sub-arrays is simply *copying*, whereas in *Quicksort* it requires *rearranging* elements wrt. the *pivot*;
- In *Merge sort*, combining partial results is *merging* (complicated, requires comparisons), whereas in *Quicksort* they are *concatenated* (simple, no comparisons).

Merge sort complexity

```
M(n)                        MergeSort(A[0 … n-1]) {
                             if (n = 1) {
M(1) = 0                       return A;
                             } else {
                               m := n/2;
copying: n/2                   L := A[0, …, m-1];
copying: n/2                   R := A[m, …, n-1];
merging: n comparisons         Merge(
M(n/2)                            MergeSort(L),
M(n/2)                            MergeSort(R), A)
                               return A;
                            }
```

- The complexity does *not* depend on the input *properties*, just its *size* ⇒ *worst-case = average case*.

Merge sort complexity

$$\boxed{\begin{array}{l} M(n) = 2\,M(n/2) + 2n, \text{ if } n > 1 \\ M(1) = 0 \end{array}}$$

Change variable $n \mapsto 2^k$: $M(n) = h(k) = 2\,h(k-1) + 2 \cdot 2^k$

Change of function: $h(k) = 2^k g(k)$

$h(0) = g(0) = M(1) = 0$

By substituting h: $2^k g(k) = 2 \cdot 2^{k-1} g(k-1) + 2 \cdot 2^k$

$g(k) = g(k-1) + 2$

By method of differences: $g(k) = 2k + M(1)$

Merge sort complexity

$$\boxed{\begin{array}{l} M(n) = 2\,M(n/2) + 2n, \text{ if } n > 1 \\ M(1) = 0 \end{array}}$$

$M(n) = h(k) = 2\,h(k-1) + 2 \cdot 2^k$ $\quad h(k) = 2^k g(k)$ $\quad g(k) = 2k + M(1)$

$g(k) = 2k$

$h(k) = 2 \cdot 2^k \cdot k$

$M(n) = 2n \log_2 n \in O(n \log_2 n \mid \textit{n is a power of 2})$

Since $n \cdot \log n$ is non-decreasing for $n > 1$, and it is also *smooth*,

$$\boxed{M(n) \in O(n \log n)}$$

Worst-case complexity of Quicksort

- Worst case is achieved when the arrays L and R are severely *imbalanced*;
- This happens, for instance, if the *pivot* is always the *smallest* element in the array.

Q(0) = 0
(no comparisons)

(n - 1) comparisons

Q(|L|) + Q(|R|)

```
QuickSort (A[0 … n-1]) {
 if (n ≤ 1) { return A; }
 else {
   l := 0; r := 0;
   pivot := A[n - 1];
   for (i = 1 … n-1) {
    if (A[i] < pivot) then {
       L[l] := A[i];
       l := l + 1;
     } else {
       R[r] := A[i];
       r := r + 1;
     }
   }
   Concat(QuickSort(L), pivot, QuickSort(R), A)
   return A;
}
```

Worst-case complexity of Quicksort

- In the worst case, $|L| = n - 1$, so we obtain the following recurrence relation:

$$Q(1) = 0$$
$$Q(n) = \underbrace{Q(n - 1)}_{Q(|L|)} + n - 1, \text{ if } n > 1$$

By method of differences:

$$Q(n) = \sum_{i=1}^{n} i - \sum_{i=1}^{n} 1 = \frac{n(n+1)}{2} - n = \frac{n(n-1)}{2} \in O(n^2)$$

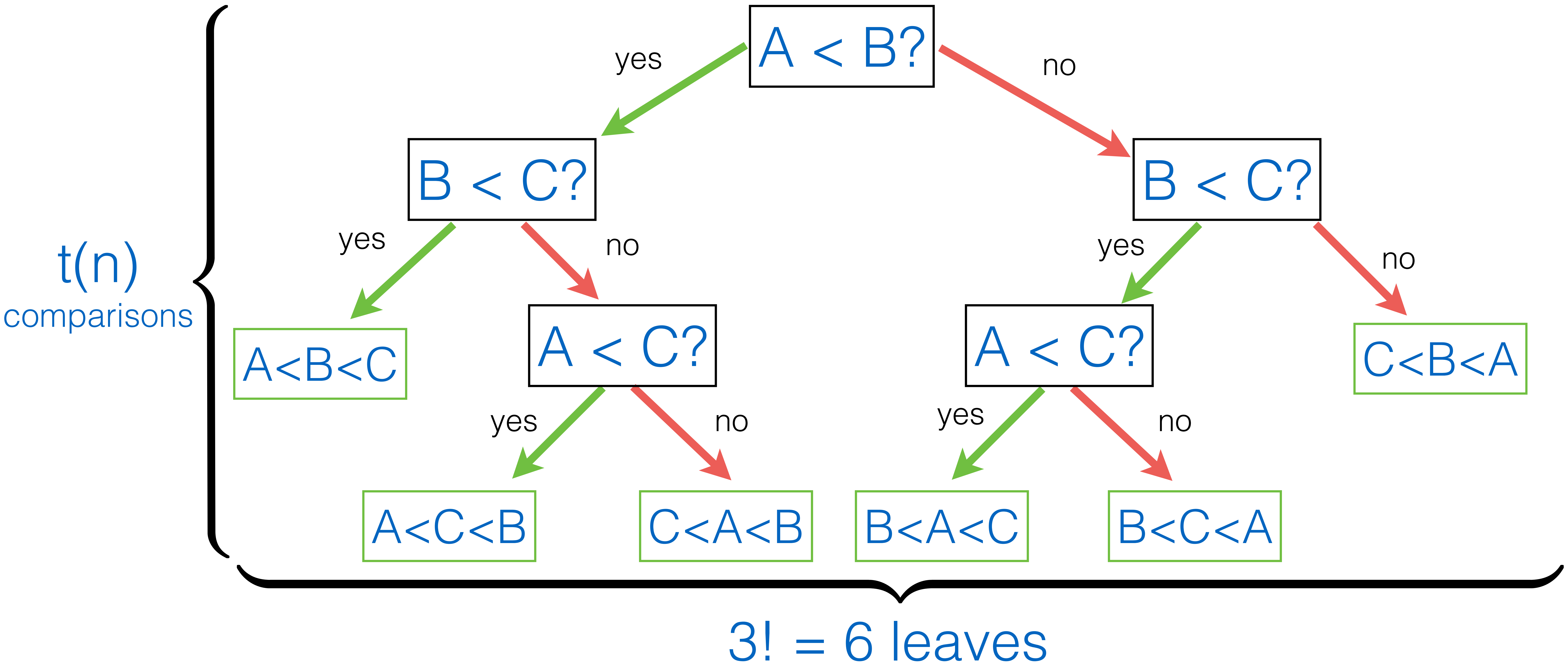
But for *Quicksort*, this worst case is *highly* improbable.

Best worst time for comparison-based sorting

- *Quicksort*, *Insertion sort*, *Merge sort* are all *comparison-based* sorting algorithms: they compare elements *pairwise*;
- An "ideal" algorithm will *always* perform no more than $t(n)$ comparisons, where n is the size of the array being sorted;
  - What is then $t(n)$?
- A number of *possible orderings* of n elements is $n!$, and such an algorithm should find "the right one" by following a path in a *binary tree*, where each node corresponds to comparing just *two* elements.

Decision tree of a comparison-based sorting

- **Example**: array [A, B, C] of three elements;
- All possible orderings between A, B, and C are possible.



Best-worst case complexity analysis

- By making $t(n)$ steps in a *decision tree*, the algorithm should be able to say, which ordering it is;
- The number of reachable leaves in $t(n)$ steps is $2^{t(n)}$;
- The number of possible orderings is $n!$ is, therefore

$$2^{t(n)} \geq n!$$

Best-worst case complexity analysis

$$2^{t(n)} \geq n!$$

$$t(n) \geq \log_2(n!)$$

Stirling's formula for large n: $$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

$$t(n) \approx n \log_e n$$
$$= (\log_e 2)\, n \log_2 n$$

$$\boxed{t(n) \in O(n \log n)}$$