

YSC2229: Introductory Data Structures and Algorithms



Week 04: Advanced Sorting Techniques

Merge sort

- Idea: 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

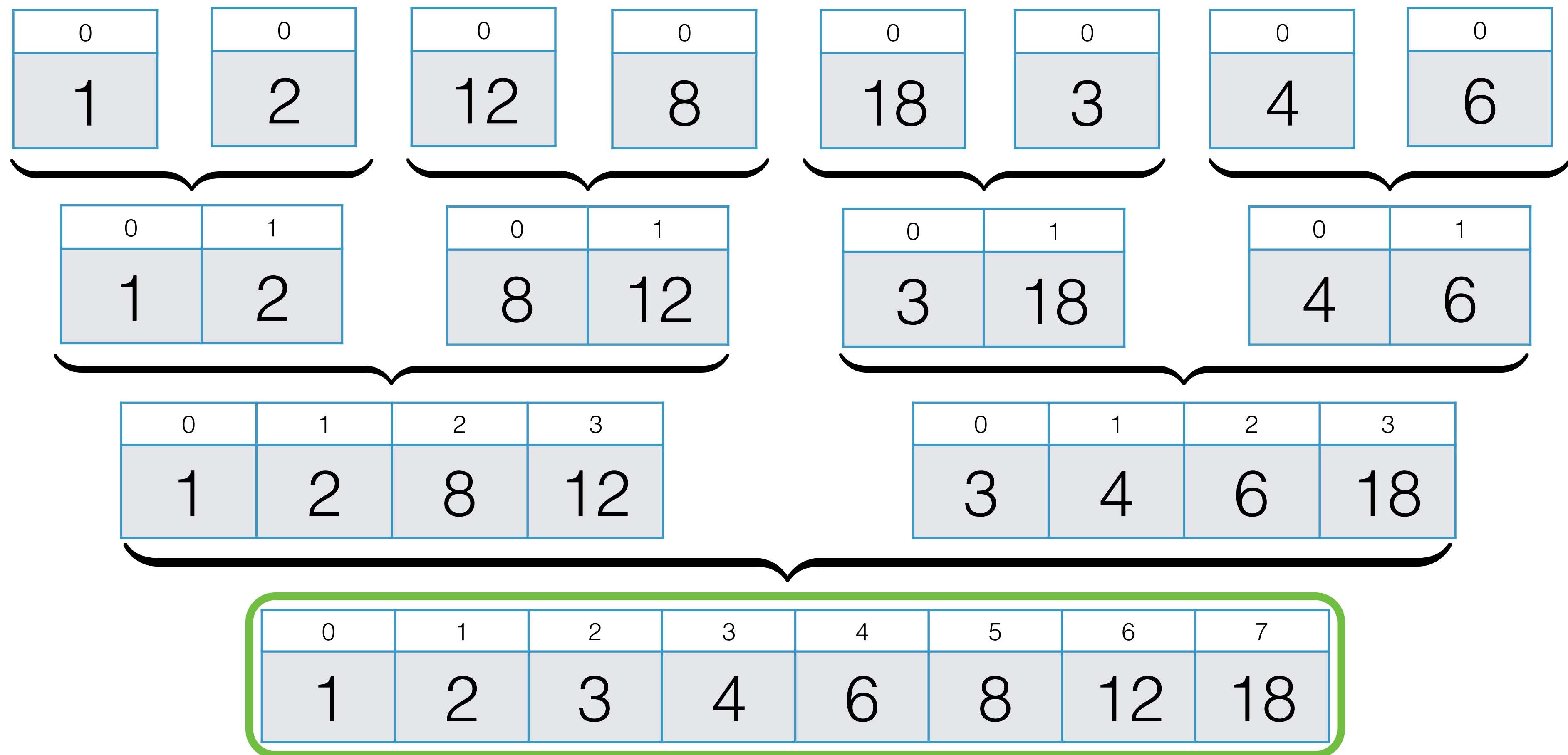
0
3

0
4

0
6

Merge sort by example

Merging the *sorted* sub-arrays



Merge sort complexity

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

M(1) = 0

copying: n/2
copying: n/2
merging: n comparisons
M(n/2)
M(n/2)

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

Merge sort complexity

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

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

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

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

$$g(k) = 2k$$

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

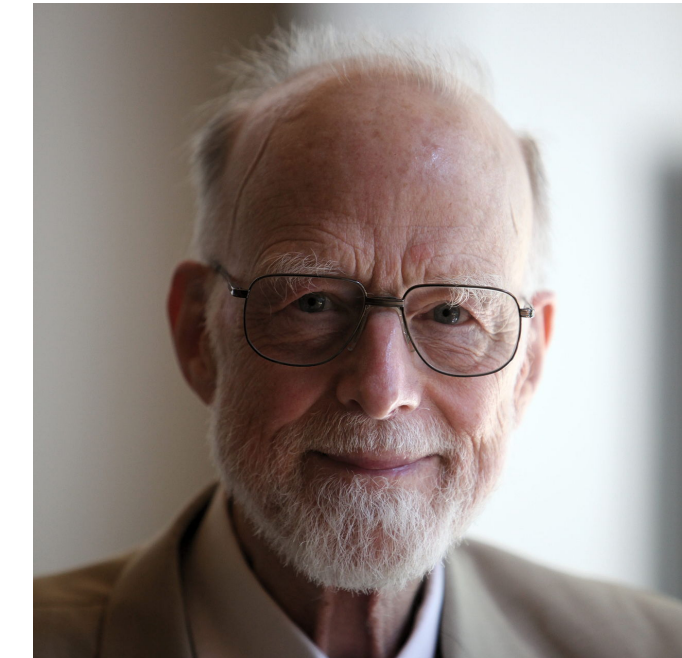
$$M(n) = 2n \log_2 n \in O(n \log_2 n \mid n \text{ is a power of } 2)$$

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

$$M(n) \in O(n \log n)$$

Quicksort

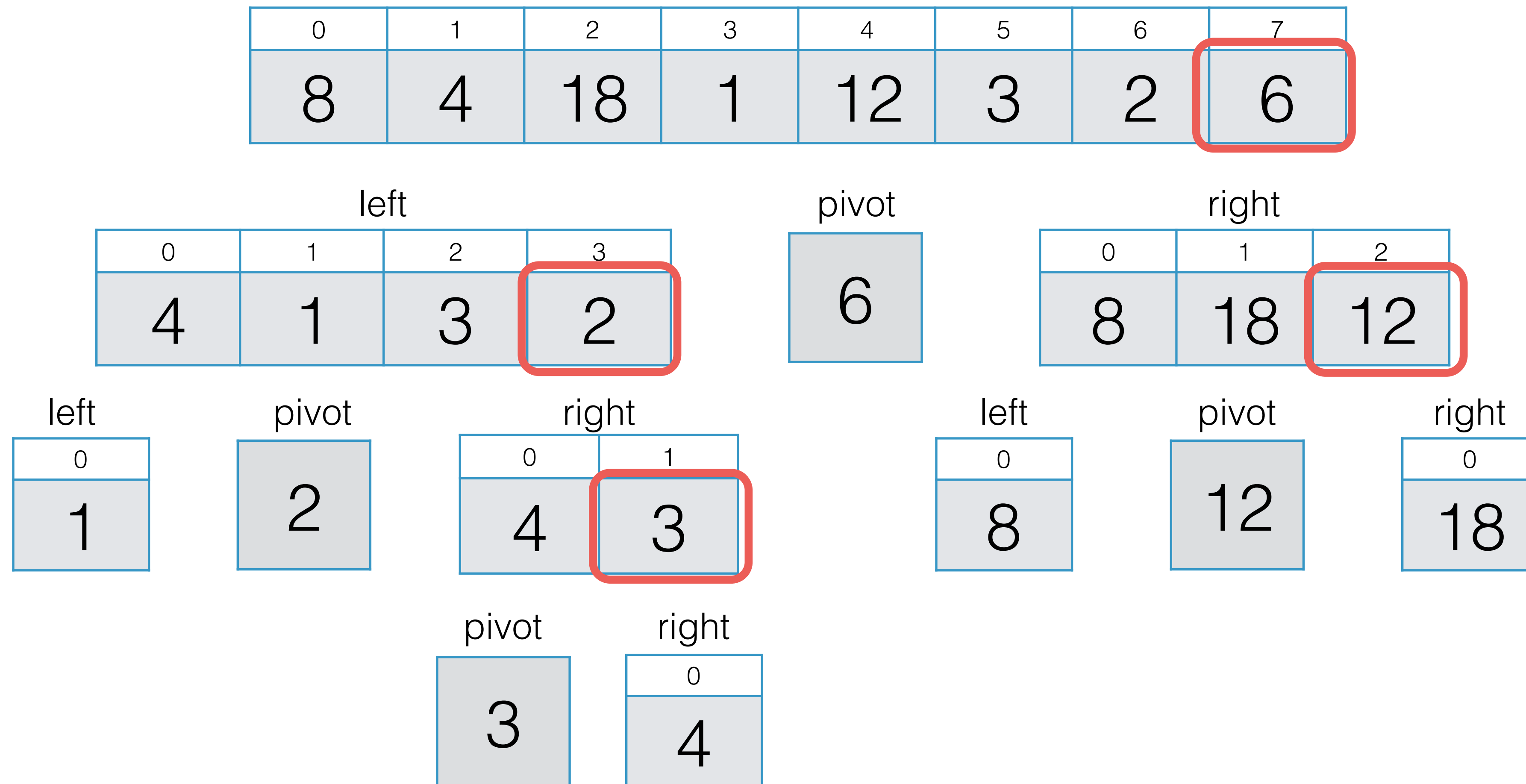
- Invented by Tony Hoare (the same as of [Hoare triples](#)) in 1961;
- Idea: divide-and-conquer with partially sorted sub-arrays;
- In practice, one of the *fastest* sorting algorithms as of today.



```
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  
  }  
}
```

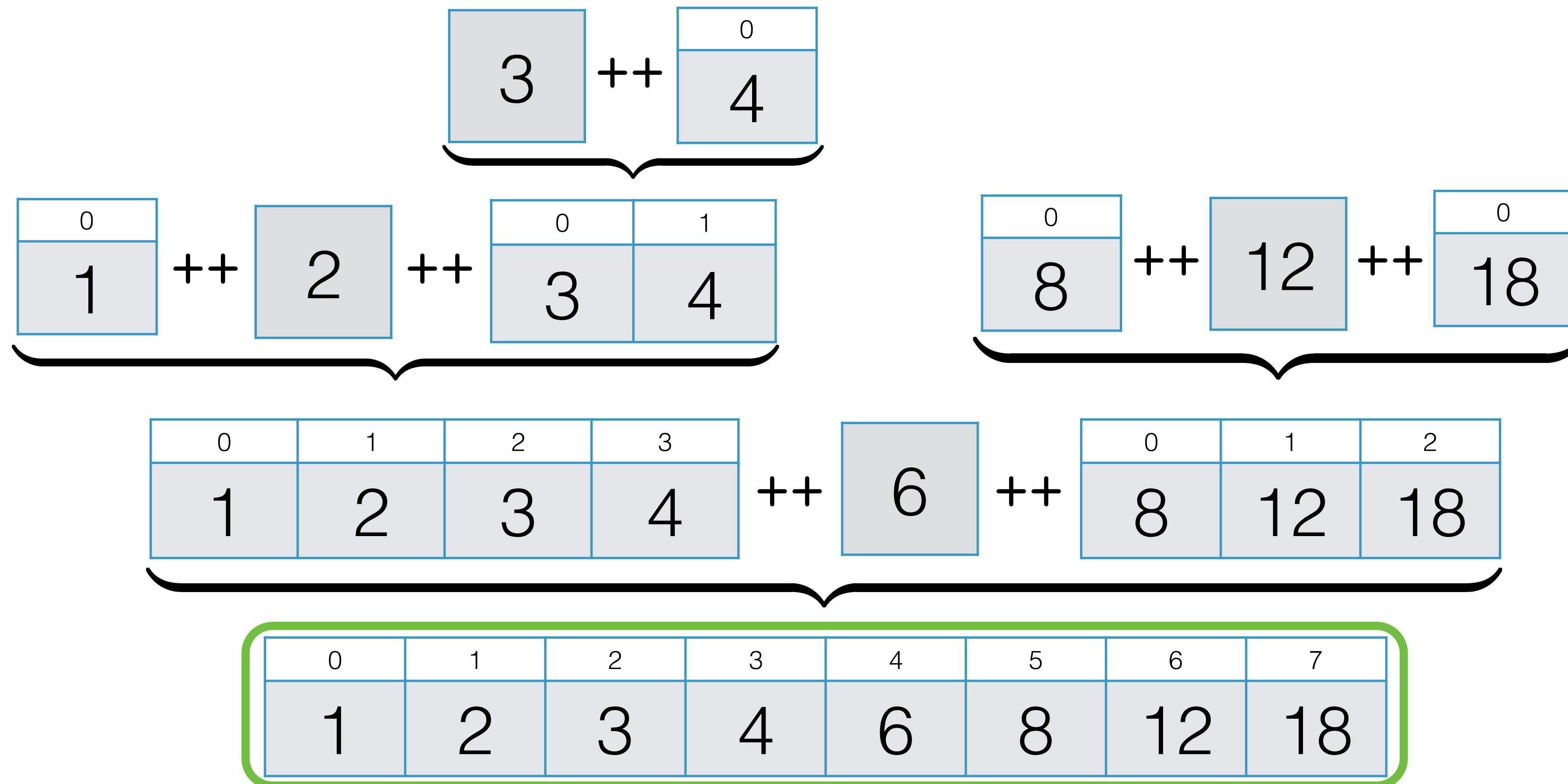

Quicksort by example

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).

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.

```
QuickSort (A[0 ... n-1]) {  
     $Q(0) = 0$   $\longrightarrow$  if (n  $\leq$  1) { return A; }  
    (no comparisons)  
    else {  
        l := 0; r := 0;  
        pivot := A[n - 1];  
        {  
            (n - 1) comparisons  
            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;  
                }  
            }  
        }  
         $Q(|L|) + Q(|R|)$   $\longrightarrow$  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:

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

By method of differences:

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

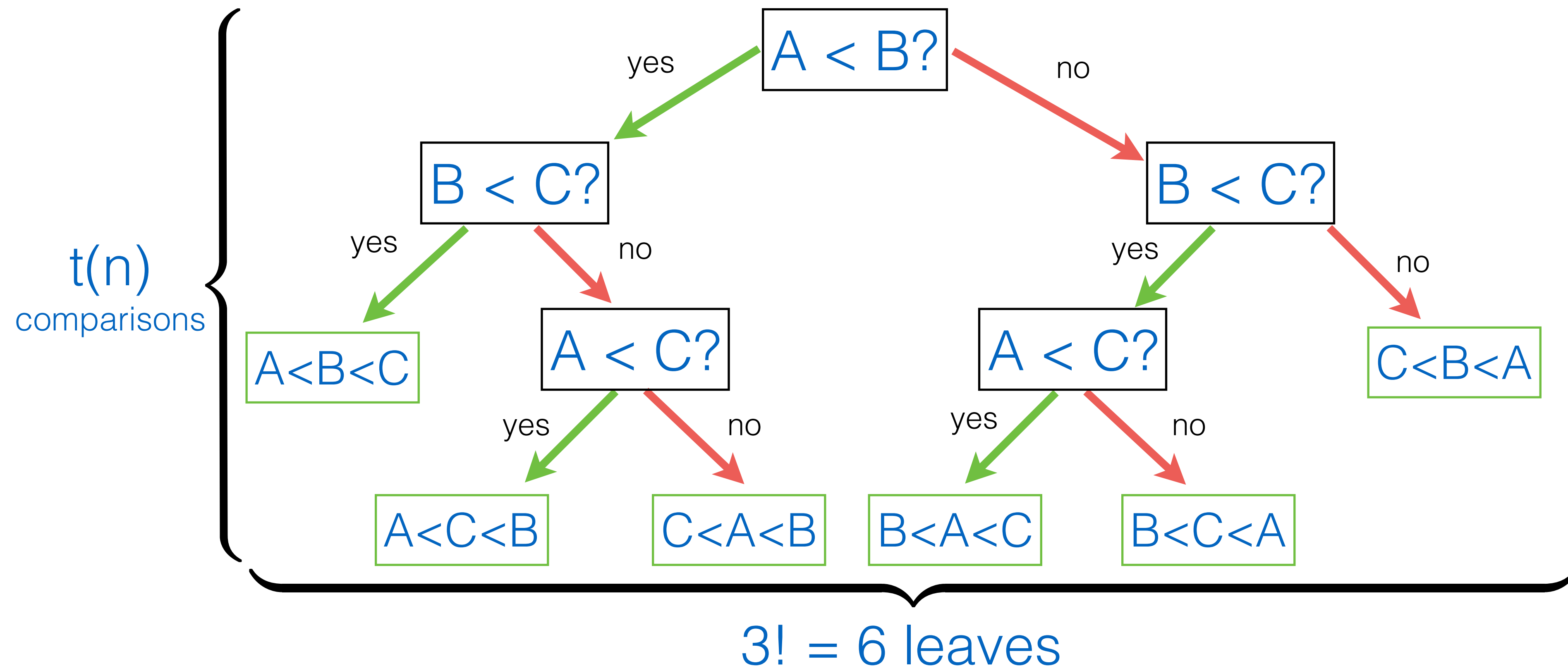
But for *Quicksort*, this worst case is *highly* improbable.

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 : $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$

$$\begin{aligned} t(n) &\approx n \log_e n \\ &= (\log_e 2) n \log_2 n \end{aligned}$$

$$t(n) \in O(n \log n)$$

Can we do sorting better than
in $O(n \log n)$?

Yes, if we don't base it on *comparisons*.

Quiz

- We want to sort n integer numbers, all in the range $1 \dots n$;
- *No repetitions*, all numbers are present *exactly once*;
- What is the worst-case complexity?

Quiz

- We want to sort n integer numbers, all in the range $1 \dots n$;
- *No repetitions*, all numbers are present *exactly once*;
- What is the worst-case complexity?

Answer: $O(n)$

- We know that it has to be $1, 2, \dots, n-1, n$, so just generate this sequence.

Bucket sort

- We want to sort an array A of n records, whose *keys* are integer numbers;
- All keys in A are in the range $1 \dots k$;
- There *might be* repeated keys, some keys might be *absent*;
- **Idea:** allocate k “*buckets*” and put records into them, then “flush” the buckets in their order.

Bucket sort

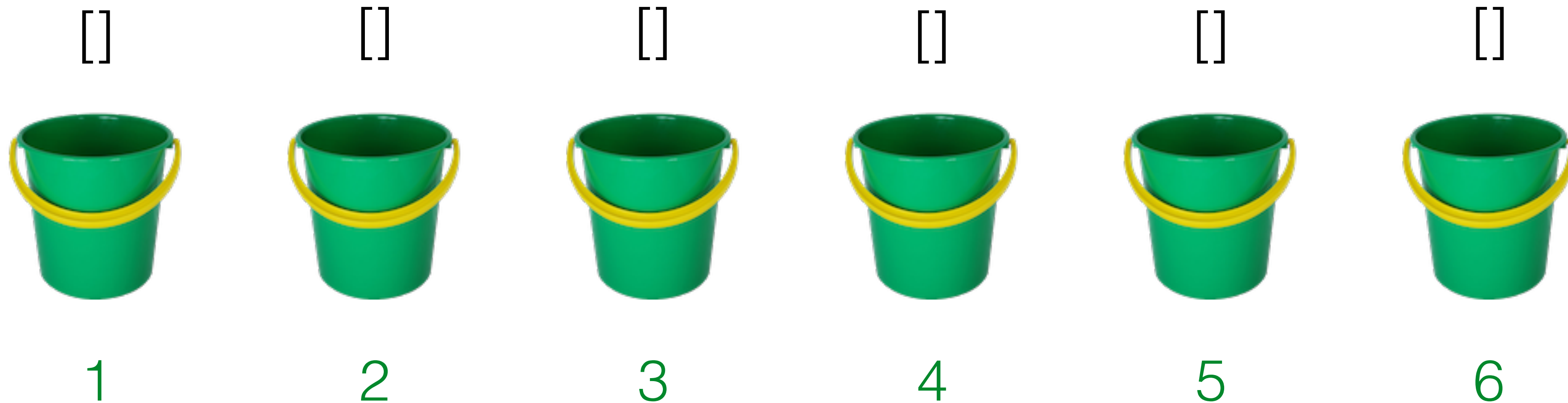
```
BucketSort (A[0 .. n-1], k) {  
    buckets := array of k empty lists; // create k empty buckets  
  
    for (i = 0..n-1) {  
        key := A[i].key; // get the next key  
        bucket := buckets[key]; // find the bucket for the key  
        buckets[key] := bucket ++ [A[i]]; // add the record into bucket  
    }  
  
    result = []  
    for (j = 0..k-1) { // concatenate all buckets  
        result := result ++ buckets[j];  
    }  
    return result;  
}
```

Bucket Sort by Example

Keys are integer numbers, $k = 6$

A =

0	1	2	3	4	5	6	7
6	2	3	1	5	3	5	2



Bucket Sort by Example

Keys are integer numbers, $k = 6$

A =

0	1	2	3	4	5	6	7
6	2	3	1	5	3	5	2



[]



1

[]



2

[]



3

[]



4

[]



5

[6]



6

Bucket Sort by Example

Keys are integer numbers, $k = 6$

A =

0	1	2	3	4	5	6	7
6	2	3	1	5	3	5	2



[]



1

[2]



2

[]



3

[]



4

[]



5

[6]



6

Bucket Sort by Example

Keys are integer numbers, $k = 6$

A =

0	1	2	3	4	5	6	7
6	2	3	1	5	3	5	2



[]

[2]

[3]

[]

[]

[6]



1

2

3

4

5

6

Bucket Sort by Example

Keys are integer numbers, $k = 6$

A =

0	1	2	3	4	5	6	7
6	2	3	1	5	3	5	2



[1]



1

[2]



2

[3]



3

[]



4

[]



5

[6]



6

Bucket Sort by Example

Keys are integer numbers, $k = 6$

A =

0	1	2	3	4	5	6	7
6	2	3	1	5	3	5	2



[1]



1

[2]



2

[3]



3

[]



4

[5]



5

[6]



6

Bucket Sort by Example

Keys are integer numbers, $k = 6$

A =

0	1	2	3	4	5	6	7
6	2	3	1	5	3	5	2



[1]



1

[2]



2

[3, 3]



3

[]



4

[5]



5

[6]



6

Bucket Sort by Example

Keys are integer numbers, $k = 6$

A =

0	1	2	3	4	5	6	7
6	2	3	1	5	3	5	2



[1]



1

[2]



2

[3, 3]



3

[]



4

[5, 5]



5

[6]



6

Bucket Sort by Example

Keys are integer numbers, $k = 6$

A =

0	1	2	3	4	5	6	7
6	2	3	1	5	3	5	2



[1]

[2, 2]

[3, 3]

[]

[5, 5]

[6]



1

2

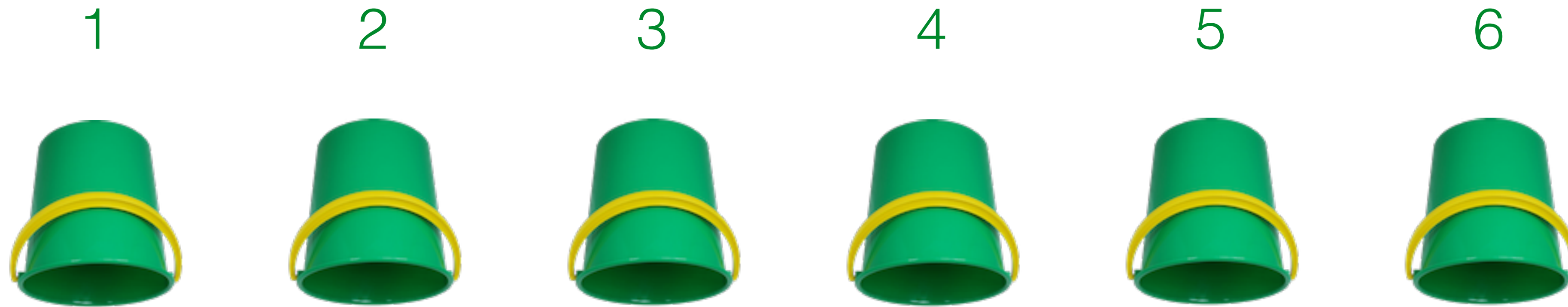
3

4

5

6


Bucket Sort by Example





[1] ++ [2, 2] ++ [3, 3] ++ [] ++ [5, 5] ++ [6]

result = [1, 2, 2, 3, 3, 5, 5, 6]

Bucket Sort Worst-case Complexity

$O(k)$  `buckets := array of k empty lists;`

$O(n)$  $\left\{ \begin{array}{l} \text{for } (i = 0..n-1) \{ \\ \text{key} := A[i].\text{key}; \\ \text{bucket} := \text{buckets}[\text{key}]; \\ \text{buckets}[\text{key}] := \text{bucket} ++ [A[i]]; \\ \} \end{array} \right.$

$O(k)$  $\left\{ \begin{array}{l} \text{result} = [] \\ \text{for } (j = 0..k-1) \{ \\ \text{result} := \text{result} ++ \text{buckets}[j]; \\ \} \\ \text{return result;} \end{array} \right.$

Overall complexity: $O(n + k)$

Remarks on Bucket Sort

- Bucket sort works for any sets of keys, known *in advance*;
- For instance, it can work with a *pre-defined set of strings*;
- But what if the size k of the set of keys is *much larger* than n ?
 - The complexity $O(n + k)$ is not so good in this case.

Stability of Sorting Algorithms

A sorting algorithm is **stable** if, when two records in the original array have the same key, they stay in *their original order* in the sorted result.

- Is *Insertion sort* stable?
 - **Yes**
- What about *Bucket sort*?
 - **Yes**
- *Merge sort*?
 - **Maybe**. It depends on how we divide the list into two and how we merge them, resolving situations for elements with the same key.
- *Quicksort*?
 - **Maybe**. Depends on the implementation of the partition step.

Radix sort

- An enhancement of the *Bucket sort*'s idea, for the case when the size of key set k in the array A is *very large*;
- **Idea:** partition each *key* using its decimal representation:
 - $\text{key} = a + 10b + 100c + 1000d + \dots$
 - then, sort keys by each register of the decimal representation, *right-to-left*, using *Bucket sort*
 - For each internal bucket sort $k = 10$ (the base of decimal representation);
- Essentially:

```
RadixSort(A) {  
    BucketSort A by a with k = 10;  
    BucketSort A by b with k = 10;  
    BucketSort A by c with k = 10;  
    ...  
}
```

Radix sort

(in very crude pseudocode)

```
RadixSort(A) {  
    L := zip(A.keys, A);  
    while (some key in L.fst is non zero) {  
        L := BucketSort(L[keys mod 10], 10); // sort by last register  
        L.fst := L.fst / 10; // shift L keys' representation to the next register  
    }  
    return L.snd; // return sorted second component  
}
```

Radix sort by Example

A =

0	1	2	3	4	5	6	7
234	124	765	238	976	157	235	953

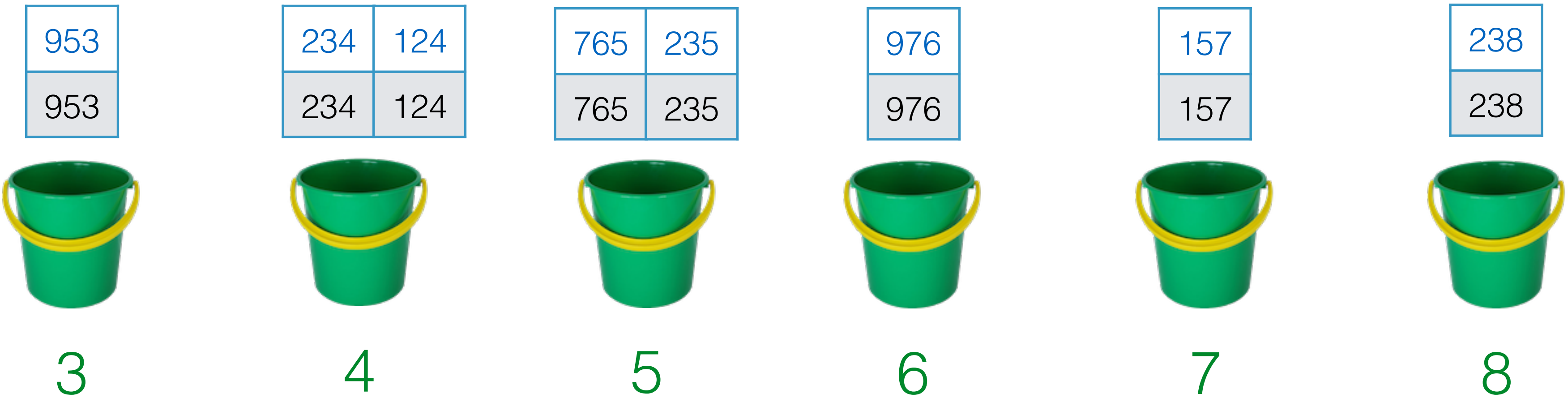
L =

234	124	765	238	976	157	235	953
234	124	765	238	976	157	235	953

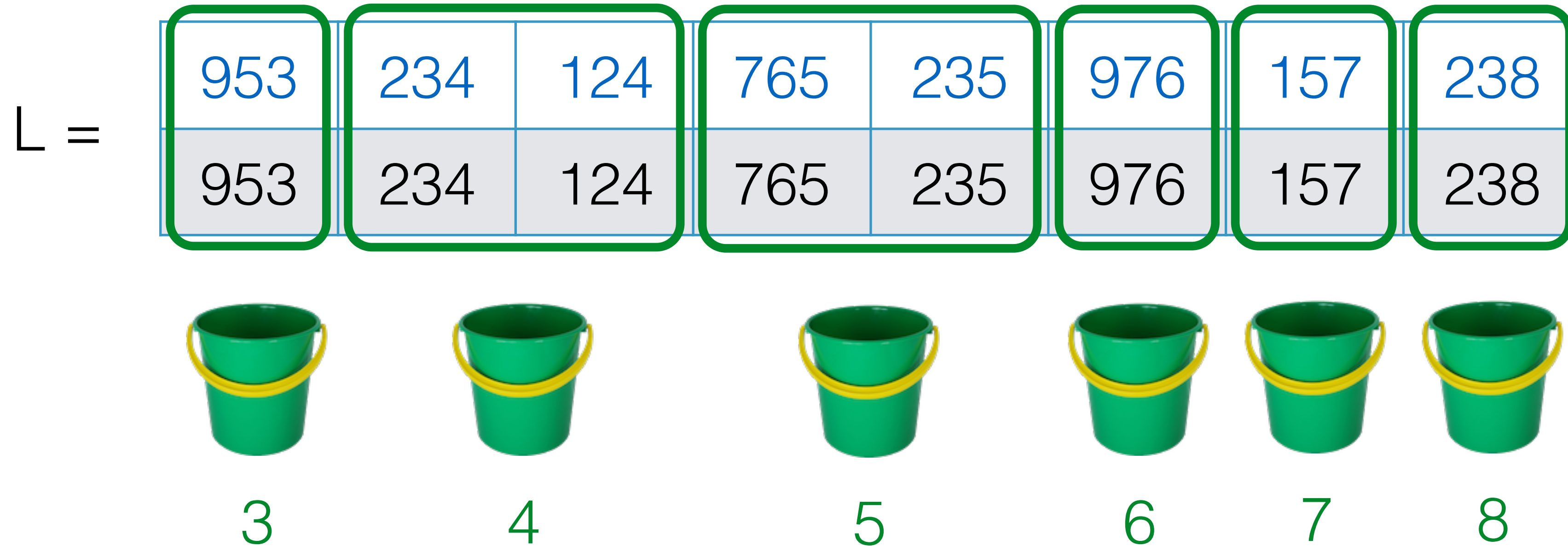
Radix sort by Example

L =

234	124	765	238	976	157	235	953
234	124	765	238	976	157	235	953



Radix sort by Example



Radix sort by Example

L =

95	23	12	76	23	97	15	23
953	234	124	765	235	976	157	238

12
124



2

23	23	23
234	235	238



3

95	15
953	157



5

76
765



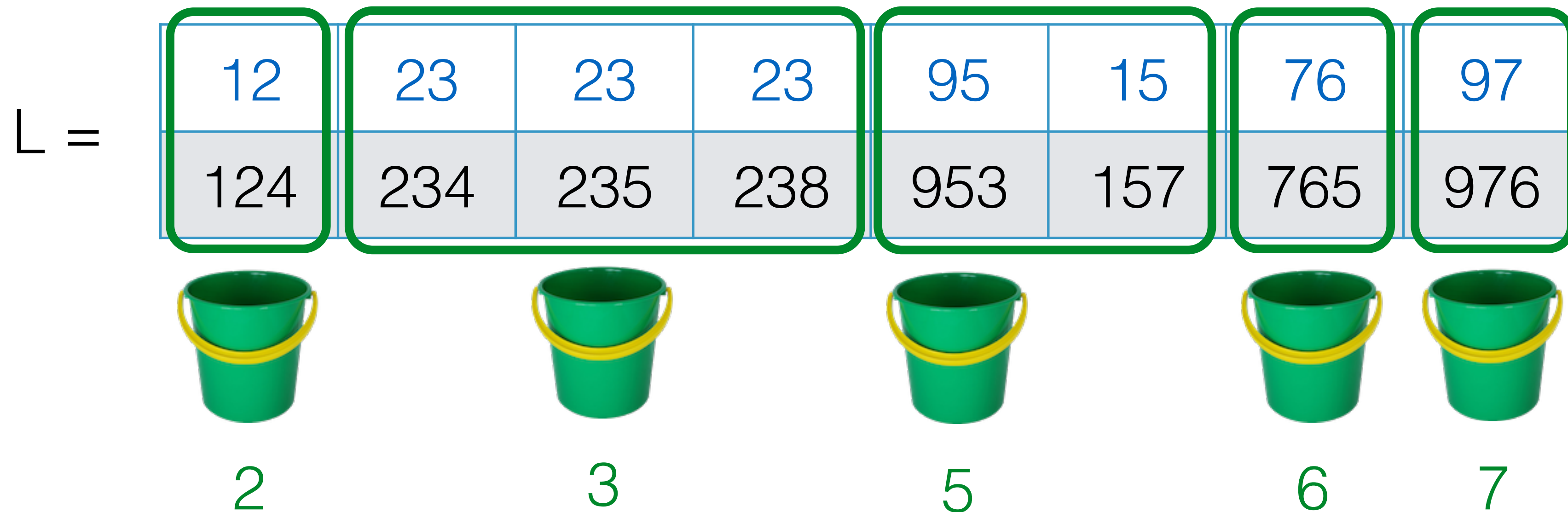
6

97
976



7

Radix sort by Example



- Thanks to *stability* of Bucket sort, values within buckets remain *sorted* with respect to *lower* registers (e.g., for bucket 3).

Radix sort by Example

L =

1	2	2	2	9	1	7	9
124	234	235	238	953	157	765	976

1	1
124	157



1

2	2	2
234	235	238



2

7
765



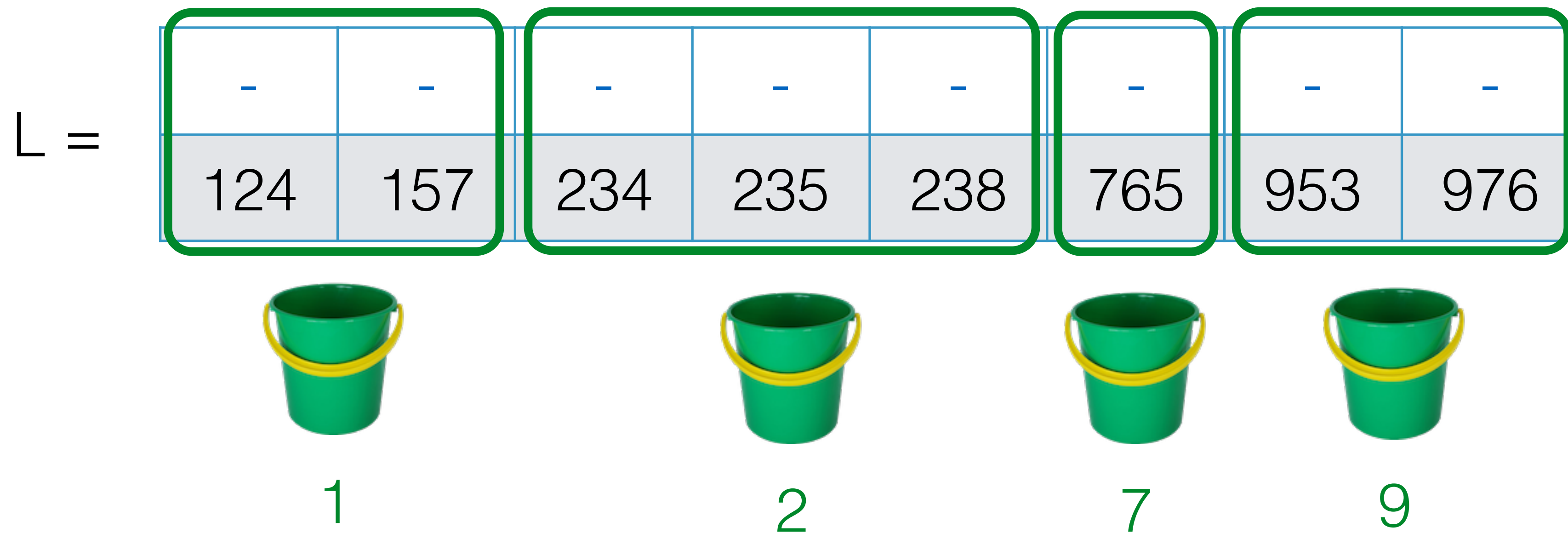
7

9	9
953	976





9

Radix sort by Example



0	1	2	3	4	5	6	7
124	157	234	235	238	765	953	976

Complexity of Radix sort

$O(n)$  RadixSort(A) {
 $O(\log_{10}k)$ iterations,
 $O(n)$ each  { **while** (some key in L.fst is *non zero*) {
 L := BucketSort(L[**keys mod 10**], **10**);
 L.fst := L.fst / 10;
 }
 return L.snd;
}

Overall complexity: $O(n \log k)$