YSC2229: Introductory Data Structures and Algorithms

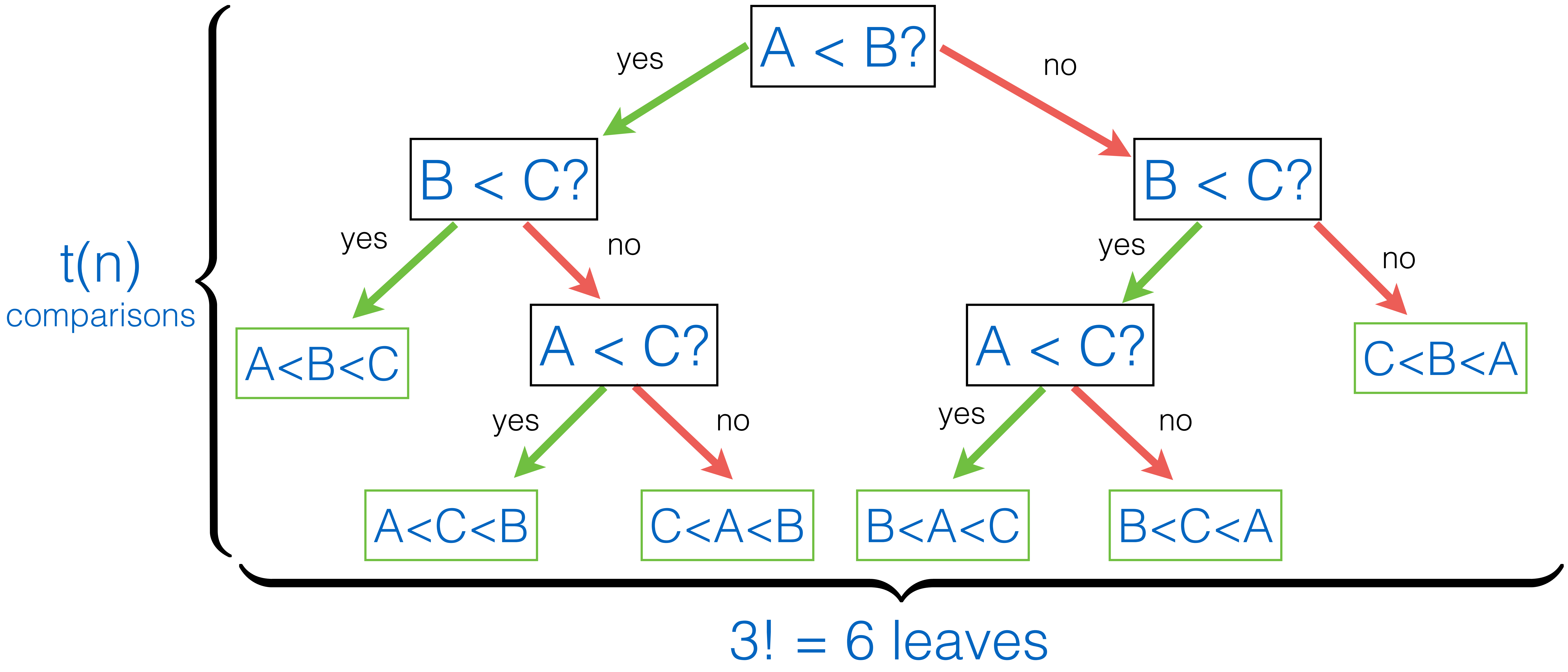
Week 05: Best-Worst Complexity of Sorting

Best worst time for comparison-based sorting

- *Quicksort*, *Merge sort* have complexity O(n log n)
- *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: $$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

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

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

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…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, …, 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…k;
- There *might be* repeated keys, some keys might be *absent*;
- **Idea**: allocate k "*buckets*" and put records into them, the "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

[]	[]	[]	[]	[]	[]

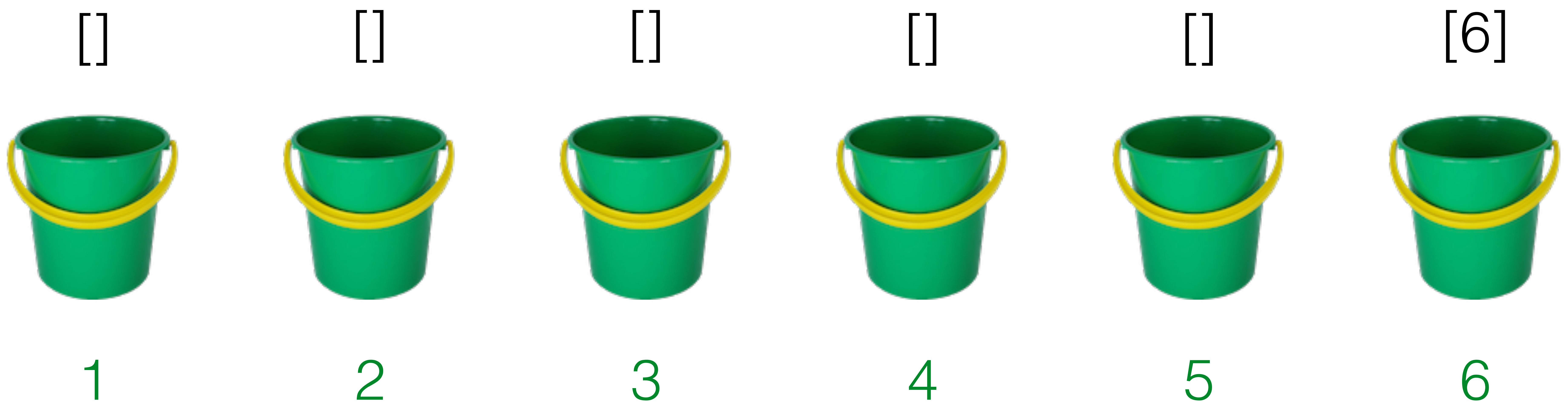
1	2	3	4	5	6

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



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]	[]	[]	[]	[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

[]	[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] [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] [2] [3] [] [5] [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] [2] [3, 3] [] [5] [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] [2] [3, 3] [] [5, 5] [6]

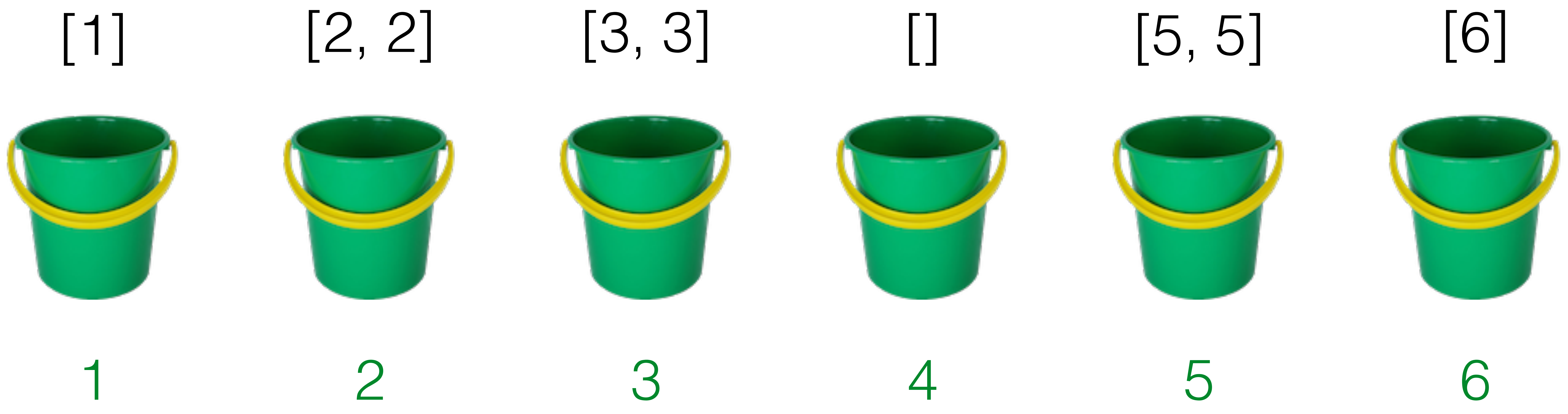
1 2 3 4 5 6

Bucket Sort by Example

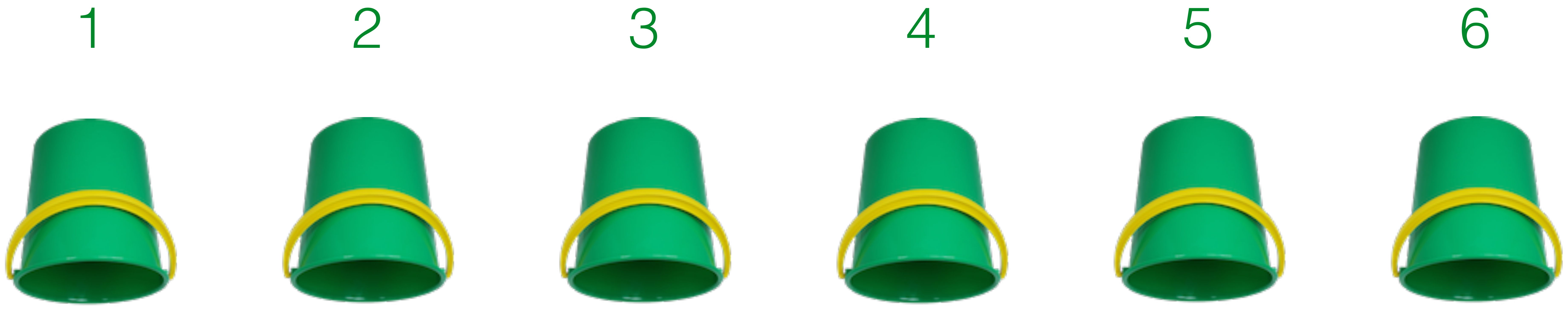
Keys are integer numbers, $k = 6$

A =

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



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) →

```
for (i = 0..n-1) {
  key := A[i].key;
  bucket := buckets[key];
  buckets[key] := bucket ++ [A[i]];
}
```

O(k) →

```
result = []
for (j = 0..k-1) {
  result := result ++ buckets[j];
}
return result;
```

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:
  - key = a + 10 b + 100 c + 1000 d + …
  - 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

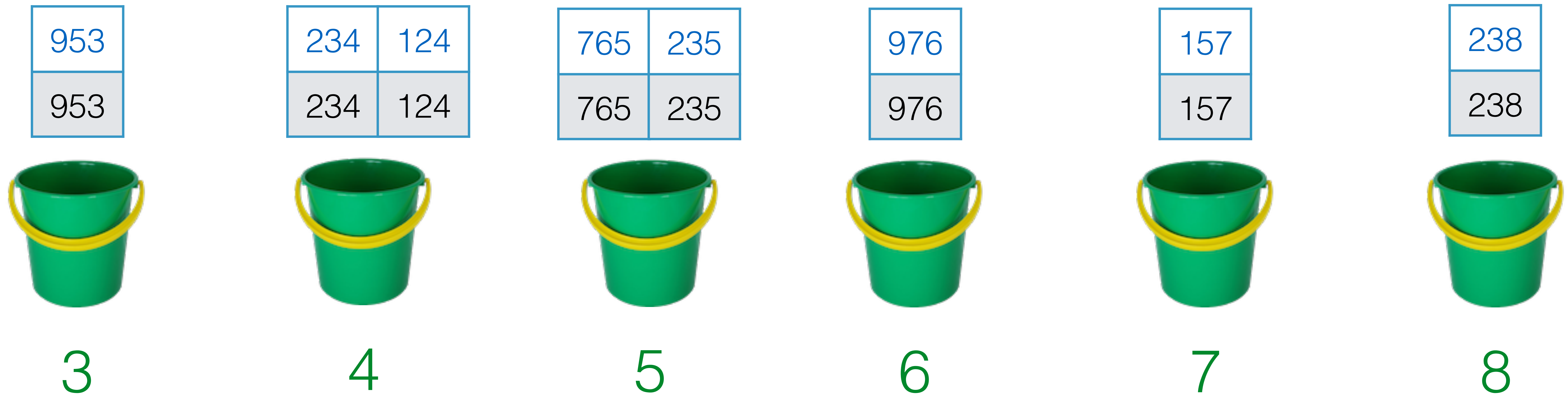
L =

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

L =

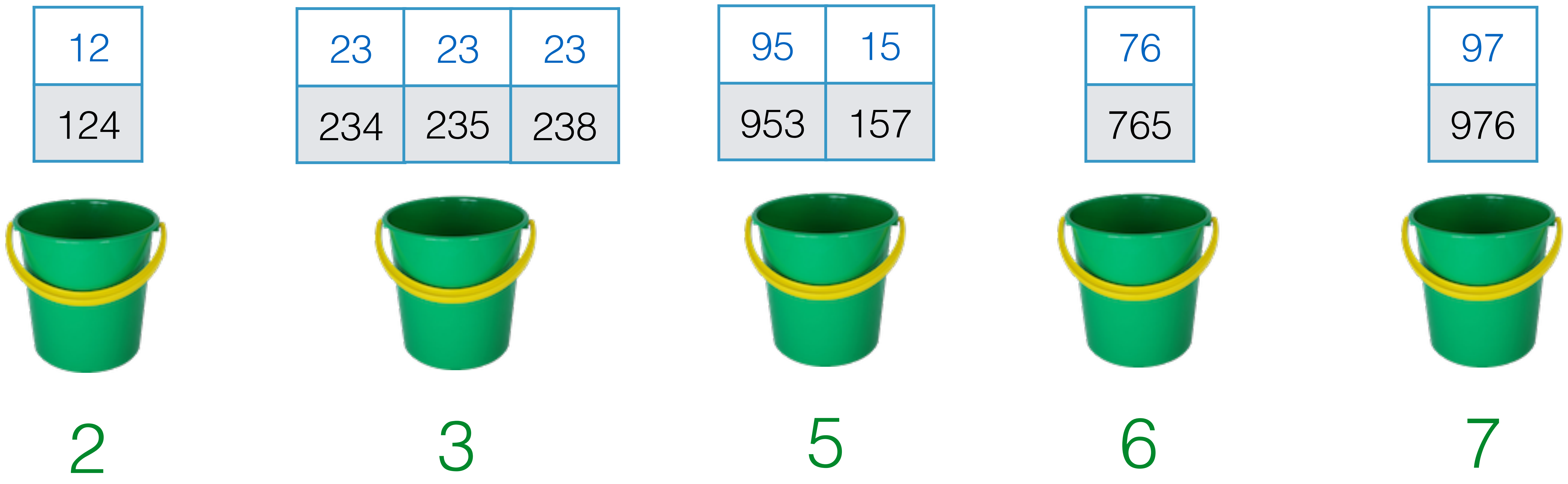
953	234	124	765	235	976	157	238
953	234	124	765	235	976	157	238

3 4 5 6 7 8

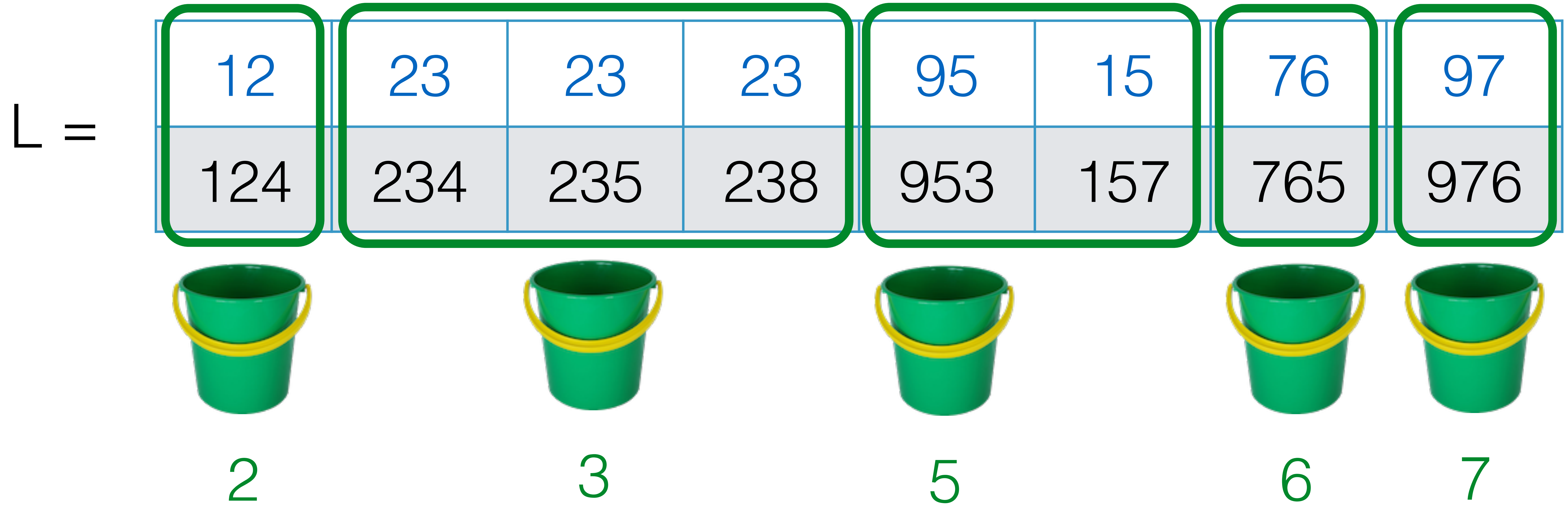
Radix sort by Example

L =

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



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

L =

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

1

2

7

9

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

Complexity of Radix sort

```
RadixSort(A) {
  L := zip(A.keys, A);
  while (some key in L.fst is non zero) {
    L := BucketSort(L[keys mod 10], 10);
    L.fst := L.fst / 10;
  }
  return L.snd;
}
```

O(n) → `L := zip(A.keys, A);`

$O(\log_{10} k)$ iterations, O(n) each → while loop

Overall complexity: O(n log k)