4. Sorting

Goals

- Classify sorting algorithms by time complexities.
- Understand the limit of sorting algorithms, and learn linear-time sorting algorithms.
- Understand the space complexity of an algorithm

Sorting Algorithms

- Most algorithms: $O(n^2)$, $O(n\log n)$
- When input numbers satisfy special conditions: $O(n)$ sorting is possible
  - E.g., input numbers are integers between 1 and n

Basic Sorting Algorithms

- Sorting algorithms with worst-case and average-case $\Theta(n^2)$ time
  - Selection sort
  - Bubble sort
  - Insertion sort

Advanced Sorting Algorithms

- Sorting algorithms with average-case $\Theta(n \log n)$ time
  - Quicksort
  - Merge sort
  - Heap sort

Quicksort

```
quickSort(A[], p, r)    ▷ sort A[p ... r]
{
    if (p < r) then {
        q = partition(A, p, r);    ▷ partition
        quickSort(A, p, q-1);      ▷ sort left part
        quickSort(A, q+1, r);      ▷ sort right part
    }
}
```

```
partition(A[], p, r)
{
    pivot element: x = A[r]
    partition A[p ... r] into A[p ... q-1] ≤ A[q] = x < A[q+1 ... r]
    return q
}
```

Quicksort

Input numbers. Pivot element = 15

31	8	48	73	11	3	20	29	65	15

Partition

8	11	3	15	31	48	20	29	65	73	(a)

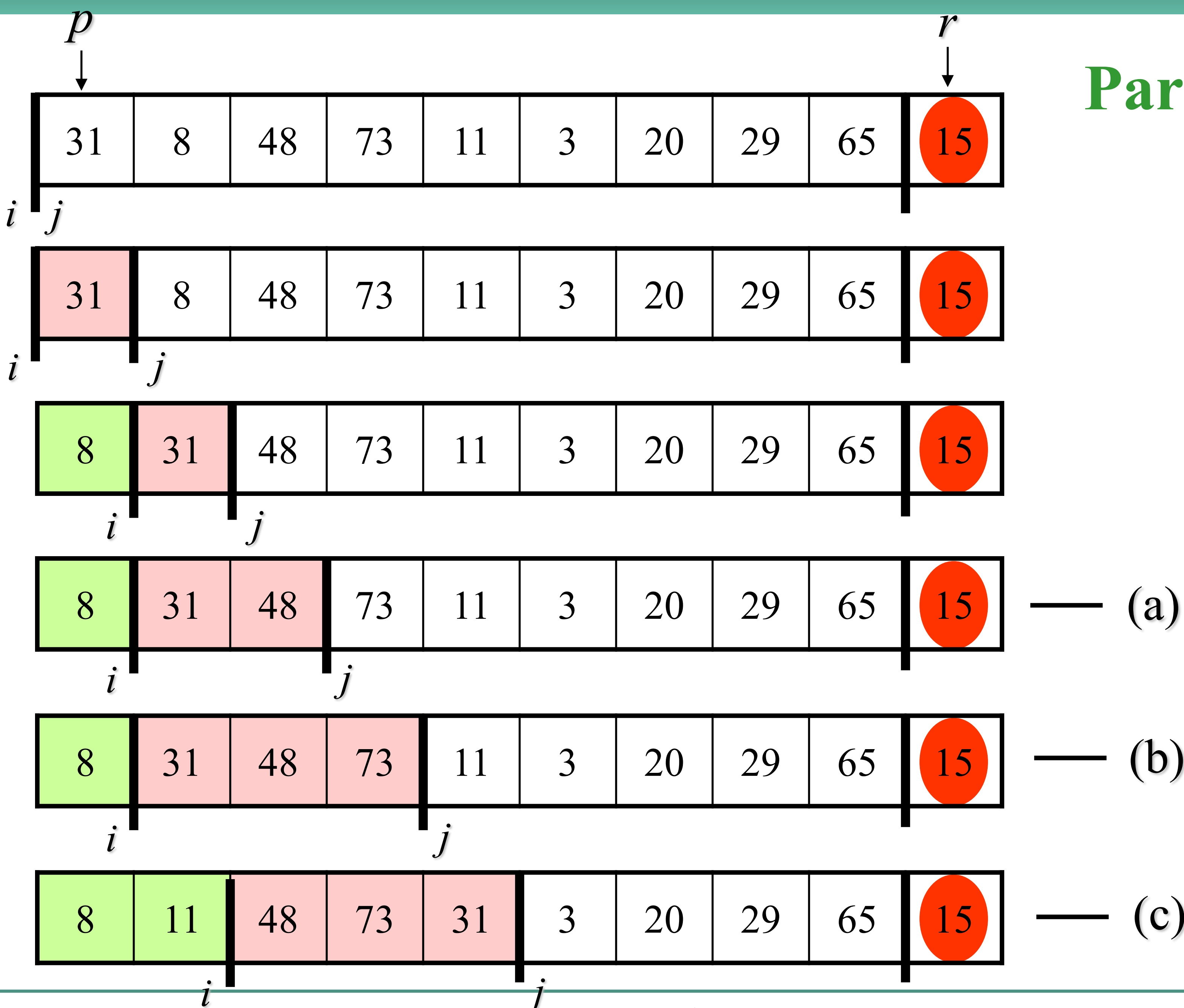
Recursive calls

3	8	11	15	20	29	31	48	65	73	(b)

Partition

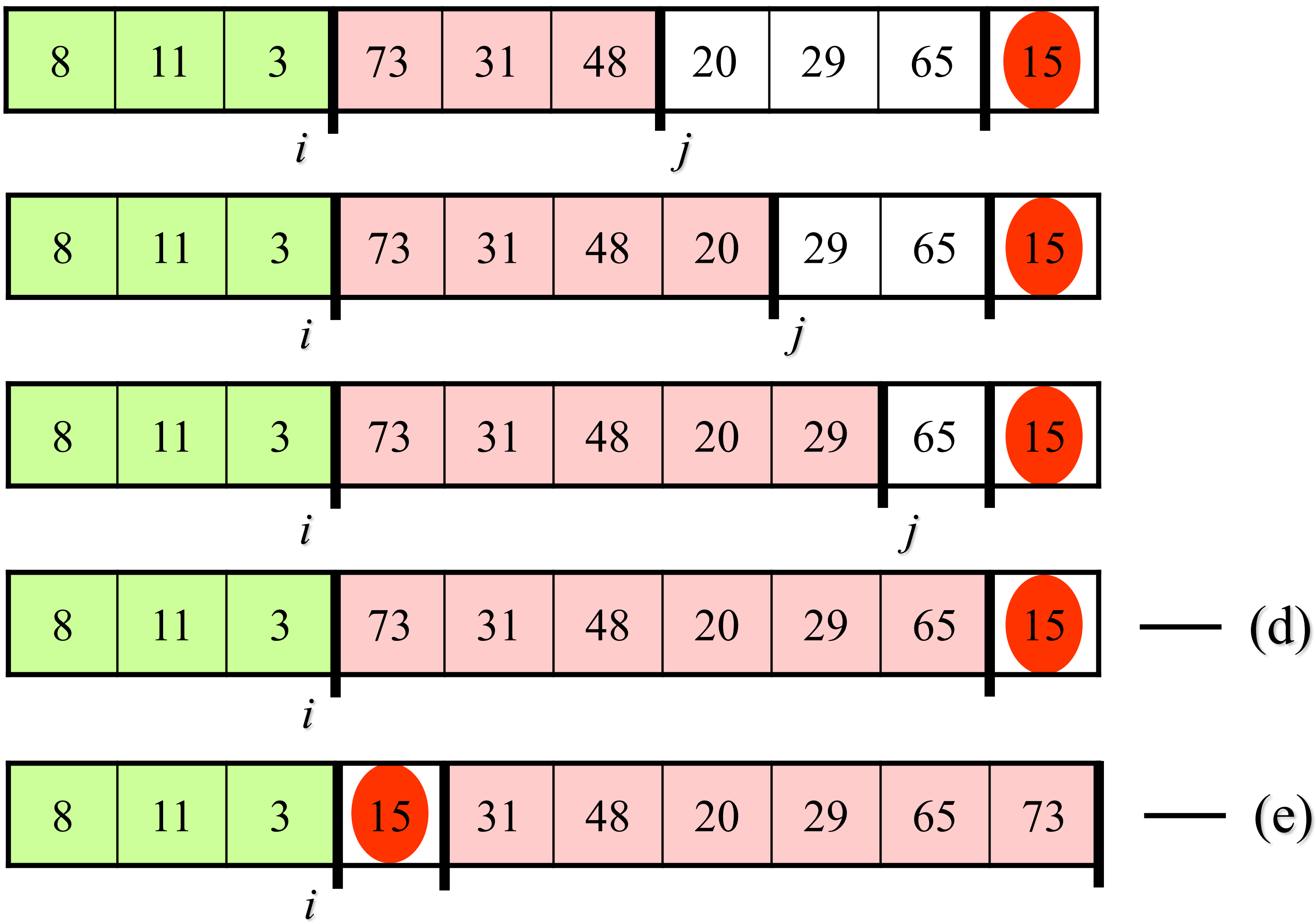
```
partition(A[], p, r)
    x = A[r]
    i = p – 1
    for j = p to r-1
        if A[j] ≤ x
            i = i + 1
            swap A[i] and A[j]
    swap A[i+1] and A[r]
    return i+1
```

Partition



p
r
31 8 48 73 11 3 20 29 65 15
i j
31 8 48 73 11 3 20 29 65 15
i j
8 31 48 73 11 3 20 29 65 15
i j
8 31 48 73 11 3 20 29 65 15
(a)
i j
8 31 48 73 11 3 20 29 65 15
(b)
i j
8 11 48 73 31 3 20 29 65 15
(c)
i j

Partition



8 11 3 73 31 48 20 29 65 15
i
j
8 11 3 73 31 48 20 29 65 15
i
j
8 11 3 73 31 48 20 29 65 15
i
j
8 11 3 73 31 48 20 29 65 15
(d)
i
8 11 3 15 31 48 20 29 65 73
(e)
i

(Worst-Case) Time Complexity of an Algorithm

- Insertion sort
  - Lower bound: $\Omega(n^2)$ for an input instance of decreasing numbers
  - Upper bound: $O(n^2)$ for all input instances
  - Time complexity of insertion sort: $\Theta(n^2)$

(Worst-Case) Time Complexity of a Problem

- Matrix multiplication
  - Standard method: $\Theta(n^3)$
  - Strassen's algorithm: $\Theta(n^{\log_2 7})$
  - Coppersmith and Winograd's algorithm: $O(n^{2.376})$
  - Upper bound: $O(n^{2.376})$
  - Lower bound: $\Omega(n^2)$
  - Time complexity of matrix mult: $n^2 \leq MM \leq n^{2.376}$

Time Complexity of Sorting

- Comparison sort: sorting algorithm that determines the sorting order based only on comparisons of input elements.
- Lower bound of comparison sorts is $\Omega(n \log n)$.
- Time complexity of comparison sorts: $\Theta(n \log n)$
- $\Theta(n)$ sorting
  - Counting Sort: input elements are integers in the range of $0 \sim O(n)$.
  - Radix Sort: input elements are d-digit numbers for constant d.

Lower Bound for Sorting

- Comparison sort: sorting algorithm that determines the sorting order based only on comparisons of input elements.
- Decision tree: binary tree that represents comparisons of a sorting algorithm (insertion sort, n=3)
  - internal node: comparison (i:j)
  - leaf: permutation that represents the sorting order
  - path from root to leaf: execution (comparisons) of sorting algorithm on an input
  - If sorting algorithm is correct, n! permutations should appear in the leaves.

Lower bound

- Longest path from root to leaf (tree height): worst case of sorting algorithm.
- Let h = height of decision tree, f = number of leaves.
  - $n! \leq f \leq 2^h$
  - $h \geq \log n!$ Hence $h = \Omega(n \log n)$.

Counting Sort

```
countingSort(A, B, n)
▷ A[1…n]: input array
▷ B[1…n]: output array
{
        for i = 1 to k
                C[i] ← 0;
        for j = 1 to n
                C[A[j]]++;
        ▷ C[i]: number of elements equal to i
        for i = 1 to k
                C[i] ← C[i] + C[i-1] ;
        ▷ C[i]: number of elements less than or equal to i
        for j ← n downto 1 {
                B[C[A[j]] ← A[j];
                C[A[j]]--;
        }
}
```

Counting Sort

- A: 2 5 3 0 2 3 0 3
- C: 2 0 2 3 0 1
- C: 2 2 4 7 7 8
- B: _ _ _ _ _ _ 3 _
- _ 0 _ _ _ _ 3 _

- Time: $\Theta(k + n)$
- If $k = O(n)$, time is $\Theta(n)$.
- Stable sort: elements with equal values appear in output in the same order as in input (satellite data)

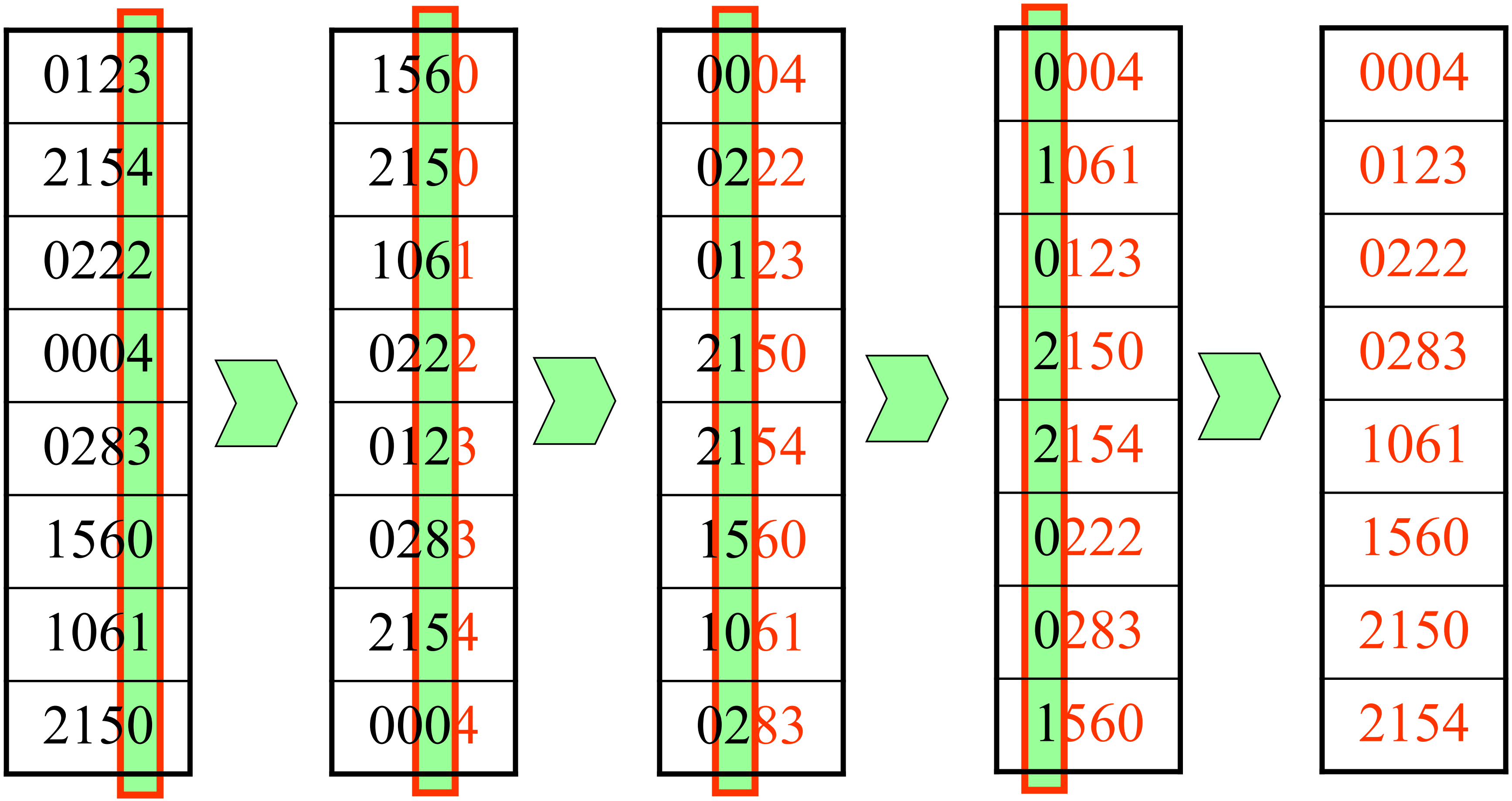
Radix Sort

radixSort(A[ ], n, d)

▷ input elements are d-digit numbers for constant d

▷ digit 1 is the least significant digit

```
{
    for i ← 1 to d
        use a stable sort to sort A[1…n] on digit i
}
```



✓ Running time: $\Theta(d(n+k))$

Time Complexities of Sorting Algorithms

	Worst Case	**Average Case**
Selection Sort	n^2	n^2
Bubble Sort	n^2	n^2
Insertion Sort	n^2	n^2
Quicksort	n^2	$n\log n$
Mergesort	$n\log n$	$n\log n$
Heapsort	$n\log n$	$n\log n$
Counting Sort	n	n
Radix Sort	n	n

(Worst-Case) Space Complexity of an Algorithm

- Counting sort
  - Input space: A, n → O(n), Output space: B → O(n)
  - Extra space: C, i, j, k → O(k)
- Insertion sort
  - Input, output space: A, n → O(n)
  - Extra space: i, j, key → O(1)
- Merge sort
  - Input, output space: O(n)
  - Extra space: O(n) (O(log n) for call stack)
- Quicksort
  - Extra space: O(n) for call stack

Thank you