Chapter 3

Introduction to Searching and Sorting Algorithms

The Main Topics

- Learn about two basic searching algorithms

- Learn about the comparison-based sorting algorithms

  - Elementary sorting algorithms

  - Efficient sorting algorithms

- Learn about the non-comparison-based sorting algorithms

Searching Algorithms

Searching Algorithms

- A searching algorithm is a method of locating a specific item in a collection of data

- For simplicity, let's assume that our data is an array of n positive integers
  - It may or may not contain duplicate values

- Consider two algorithms for searching the contents of an array:
  - The linear search
  - The binary search

Searching Algorithms: Linear Search

- The linear search is a very simple algorithm. Sometimes it's called a *sequential search*

- Starting with the first element, the algorithm uses a loop to sequentially step through a given array

- It compares each element with the search key and stops when either

  – a match is encountered (successful search), or

  – the end of the array is encountered without finding a match (unsuccessful search)

Pseudo-code

```
i = 1;
while (i ≤ n && a[i] != k)
   i++;
return (i ≤ n);
```

Pseudo-code

```
a[n + 1] = k;

i = 1;

while (a[i] != k)

  i++;

return (i ≤ n);
```

Pseudo-code

sentinel

```
a[n + 1] = k;

i = 1;

while (a[i] != k)

  i++;

return (i ≤ n);
```

Searching Algorithms: Linear Search

- Advantages:

  – It's simple, easy to understand and implement

  – It doesn't require the data in the array to be stored in any particular order

- Disadvantage:

  – It's inefficient

Searching Algorithms: Binary Search

- Given a sorted array, *binary search* is always our choice

- The binary search is a clever algorithm that is much more efficient than the linear search

Searching Algorithms: Binary Search

- It starts with the element in the middle: $a[m]$
  - If $a[m] = k$: 👌
  - If $a[m] > k$: the desired value will be found somewhere in the *left half* of the array

    → *The right half of the array is eleminated from searching*

  - If $a[m] < k$: the desired value will be found somewhere in the *right half* of the array

    → *The left half of the array is eleminated from searching*

- This process continues until either the desired value is found or there are no more elements to test

Pseudo-code

```
left = 1, right = n;
while (left ≤ right) {
  middle = (left + right) / 2;
  if (a[middle] == k)
    return true;
  else
    if (k < a[middle])
      right = middle - 1;
    else
      left = middle + 1;
}
return false;
```

Searching Algorithms: Binary Search

- Advantages:
  - The binary search is much more efficient than the linear search
  - How to evaluate the efficiency of the binary search?

- Disadvantage:
  - The input array must be sorted

Searching Algorithms: Case Study

Finding both the smallest and largest values in an array of size n simultaneously

```
small = large = a[1];
for (i = 2; i ≤ n; i++) {
   if (small > a[i]) small = a[i];
   if (large < a[i]) large = a[i];
}


small = large = a[1];
for (i = 2; i ≤ n; i++)
   if (small > a[i]) small = a[i];
   else
      if (large < a[i]) large = a[i];
```

Sorting Algorithms

Why Do We Need Sorting Algorithms?

- Often the data in a list must be sorted in some order

  - A list of names is commonly sorted in alphabetical order

  - A list of student grades might be sorted from highest to lowest

- In addition, certain searching algorithms work only on sorted data

The comparison-based sorting algorithms

Elementary Sorting Approaches

- Bubble sort

- Selection sort

- Insertion sort

Bubble Sort

- In essence, this algorithm compares adjacent elements of the array and exchanges them if they are out of order

Bubble Sort: Pseudo-code

```
for (i = 2; i ≤ n; i++)
  for (j = n; j ≥ i; j--)
    if (a[j] < a[j - 1])
      a[j] ⇄ a[j - 1];
```

Analysis of Bubble Sort

```
for (i = 2; i ≤ n; i++)
    for (j = n; j ≥ i; j--)
        if (a[j] < a[j - 1])
            a[j] ⇄ a[j - 1];
```

- The data comparison is the basic operation

- The outer loop executes $n - 1$ times

  - During the first iteration of the outer loop, the number of iterations of the inner loop is $n - 1$

  - During the second iteration of the outer loop, the number of iterations of the inner loop is $n - 2$

  - …

- The total number of comparisons is

$$C(n) = \frac{n(n - 1)}{2} \in \Theta(n^2)$$

An Improvement of Bubble Sort

- Remembering whether or not any exchange had taken place during a pass

- Remembering the position of the last exchange

- Alternating the direction of consecutive passes

→ We call the resulting algorithm *Shakersort*

Shakersort: Pseudo-code

```
left = 2; right = n; k = n;
do {
  for (j = right; j ≥ left; j--)
    if (a[j - 1] > a[j]) {
      a[j - 1] ⇄ a[j];   k = j;
    }
  left = k + 1;
  for (j = left; j ≤ right; j++)
    if (a[j - 1] > a[j]) {
      a[j - 1] ⇄ a[j];   k = j;
    }
  right = k - 1;
} while (left ≤ right);
```

Selection Sort

- Selection sort moves elements immediately to their final position in the array

Selection Sort: Pseudo-code

```
for (i = 1; i < n; i++) {
  minIdx = i;
  minVal = a[i];
  for (j = i + 1; j ≤ n; j++)
    if (a[j] < minVal) {
      minIdx = j;
      minVal = a[j];
    }
  a[minIdx] = a[i];
  a[i] = minVal;
}
```

Analysis of Selection Sort

- The comparison `a[j] < minVal` is the basic operation

- The time complexity of this algorithm is $\Theta(n^2)$

Insertion Sort

- The insertion sort works in a slightly different way

Example

Insertion Sort: Pseudo-code

At first, the left part contains only one element

```
for (i = 2; i ≤ n; i++) {
  v = a[i];
  j = i - 1;
  while (j > 0 && a[j] > v) {
    a[j + 1] = a[j];
    j--;
  }
  a[j + 1] = v;
}
```

Analysis of Insertion Sort

- The comparison **a[j] > v** is the basic operation

```
for (i = 2; i ≤ n; i++) {
  a[0] = v = a[i];
  j = i - 1;
  while (a[j] > v) {
    a[j + 1] = a[j];
    j--;
  }
  a[j + 1] = v;
}
```

Analysis of Insertion Sort

- The algorithm depends on the distribution of the input
  - *The best case*:
$$B(n) = n - 1 \in \Theta(n)$$
  - *The worst case*:
$$W(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$$
  - *The average case*:
$$A(n) \approx \frac{n^2 + n}{4} + \frac{n}{2} - \ln n - 0.5772 \in \Theta(n^2)$$

An Improvement of Insertion Sort

- The act of scanning the left part is a sequential search

- All elements in the left part are in the sorted order

→ A replacement of a sequential search by a binary search is a good way to improve the algorithm

→ We call the resulting algorithm *Binary Insertion Sort*

Binary Insertion Sort: Pseudo-code

```
for (i = 2; i ≤ n; i++) {
  v = a[i];
  left = 1; right = i - 1;
  while (left ≤ right) {
    m = (left + right) / 2;
    if (v < a[m])    right = m - 1;
    else             left = m + 1;
  }
  for (j = i - 1; j ≥ left; j--)
    a[j + 1] = a[j];
  a[left] = v;
}
```

Conclusions

- Advantages
  - They are simple and easy to implement
  - They are in-place algorithms, so the space requirement is minimal
  - They are useful only when sorting an array of few elements
  - The *Bubble sort* is suitable for academic teaching but not for real-life applications
  - The *Selection sort* tends to minimize data movements
  - The *Insertion sort* can also be useful when the input array is almost sorted

Conclusions

- Disadvantage
  - They do not deal well with an array containing a huge number of elements

Efficient Sorting Approaches

- The elementary sorting algorithms are of the order $\Theta(n^2)$

- There are more efficient sorting algorithms that are of the order $O(n \log n)$

- Three typical algorithms are *heapsort*, *quicksort*, and *mergesort*

Heapsort (aka Tree Sort)

- Heapsort was invented by J. W. J. Williams in 1964

  - This was also the birth of the *heap*, presented already by Williams as a useful data structure

- Where does the new idea come from?

  - The method of sorting by the selection sort is based on the repeated selection of

    - the least key among n items

    - the least key among the remaining $n - 1$ items

    - …

  - How can the selection sort possibly be improved?

The General Idea

The General Idea: The First Stage

- The selection sort can only be improved by retaining from each scan more information than just the identification of the single least item

  - With $\frac{n}{2}$ comparisons, it is possible to determine the smaller key of each pair of items

  - With another $\frac{n}{4}$ comparisons, the smaller of each pair of such smaller keys can be selected

  - …

- With $n - 1$ comparisons, we can build a *selection tree* (or *heap*) and identify the root as the desired least key

Example

The General Idea: The Second Stage

- The following steps will be repeated until the *heap* is full of the key ∞

  - Extracting the least key at the root and replacing it by an empty hole

  - Descending down along the path marked by the least key and replacing it by an empty hole

  - Filling the empty hole at the bottom with ∞ and ascending up along the path marked by empty holes, filling them with ∞ or the key at the alternative branch

- Each of the above repetitions is called a *selection step*

Remarks

- Weaknesses:
  - The *heap* requires $2n - 1$ units of storage to store n items
  - The occurences of ∞ in the heap are the source of many unnecessary comparisons
  - It is easier said than done

- Strength:
  - After n selection steps, the tree is full of ∞, and the sorting process is terminated
  - Each of the n selection steps requires $\log_2 n$ comparisons
  - The second stage requires a linearithmic running time

Heap Definition

- A heap is a sequence of elements $h_1, h_2, \dots, h_n$ such that

$$h_i \leq h_{2i}$$
$$h_i \leq h_{2i+1}$$

for all $i = 1, 2, \dots, \left\lfloor \dfrac{n}{2} \right\rfloor$

- The sequence of elements $h_{\left\lfloor \frac{n}{2} \right\rfloor + 1}, \dots, h_n$ is a *natural heap*

- The element h_1 of a heap is its least value

Heap Construction: The Basic Idea

- How to reconstruct a heap if the element at the root is not the smallest key?

- The solution is as follows:

  A new heap is obtained by letting the element on top of the subheap "sift down" along the path of the smaller comparand, which at the same time move up

  The "sift down" process stops when the element on top of the subheap is smaller than or equal to both its comparands

Example

Heap Construction: The Sifting Algorithm

```
sift(a[], left, right) {
  i = left;    j = 2 * i;
  x = a[i];
  while (j ≤ right) {
    if (j < right)
      if (a[j] > a[j + 1]) j++;
    if (x ≤ a[j])   break;
    a[i] = a[j];
    i = j; j = 2 * i;
  }
  a[i] = x;
}
```

Heap Construction: Algorithm

- Given an array $a_1, a_2, \ldots, a_n$

  – The sequence of elements $a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n$ is a *natural heap*

- The heap is now extended to the left where in each step a new element is included and properly positioned by a sift

```
left = ⌊n/2⌋;
while (left > 0) {
  sift(a, left, n);
  left --;
}
```

Heap Construction: The Sifting Algorithm

```
sift(a[], left, right) {
  i = left;     j = 2 * i;
  x = a[i];
  while (j ≤ right) {
    if (j < right)
      if (a[j] > a[j + 1]) j++;
    if (x ≤ a[j])   break;
    a[i] = a[j];
    i = j; j = 2 * i;
  }
  a[i] = x;
}
```

```
left = ⌊n/2⌋;
while (left > 0) {
  sift(a, left, n);
  left --;
}
```

Example

The Sorting Process

- Given a heap of n elements. In order to obtain the elements sorted, n steps have to be executed

- Each step includes:
  - Picking off the element at the top of the heap (and store it somewhere)
  - Reconstructing the remaining elements to form a heap

- There are two questions:
  - Where to store the emerging top elements?
  - How to reconstruct the remaining elements to form a heap?

The Sorting Process

- A solution is as follows:

  – In each step, the leftmost element and the rightmost element of the *current* heap are exchanged …

  – … and then, let the new leftmost element sift down into its proper position

- After $n - 1$ steps, the array will be sorted (in descending order)

The Sorting Process: Algorithm

```
right = n;
while (right > 1) {
  a[1] ⇄ a[right];
  right --;
  sift(a, 1, right);
}
```

Analysis of Heapsort

- The heapsort algorithm runs in $O(n \log n)$ because it consists of two stages:

  - The (heap) construction stage: The algorithm executes $\left\lfloor \frac{n}{2} \right\rfloor$ sift steps where each step runs in $O(\log n)$ time

    → This stage takes $O(n \log n)$ time

  - The sorting stage: The algorithm executes $n - 1$ steps where each step runs in $O(\log n)$ time

    → This stage takes $O(n \log n)$ time

- It is not recommended for small number of items

Quicksort

- The quicksort algorithm was developed in 1959 by Tony Hoare

- It was selected as one of the 10 algorithms "*with the greatest influence on the development and practice of science and engineering in the 20th century*"

  (The American Institute of Physics and the IEEE Computer Society, 2000)

- This sorting algorithm based on the idea of an array *partition*

Quicksort: A Partition

- It is an arrangement of the array's elements so that
$$\{a_1, \ldots, a_{s-1}\} \leq a_s \leq \{a_{s+1}, \ldots, a_n\}$$

- After a partition is achieved, a_s will be in its final position in the sorted array

- We can continue sorting the two subarrays to the left and to the right of a_s independently by the same method

Quicksort: Pseudo-code

```
Quicksort(a[1..n], left, right) {

  if (left < right) {

    s = Partition(a, left, right);

    Quicksort(a, left, s - 1);

    Quicksort(a, s + 1, right);

  }

}
```

Quicksort: `Partition()`

- `Partition()` starts by selecting a *pivot* p – an element of the (sub)array being partitioned

  – For simplicity: $p = a_{left}$

- Now the function scans the (sub)array from both ends, comparing the (sub)array's elements to the pivot p
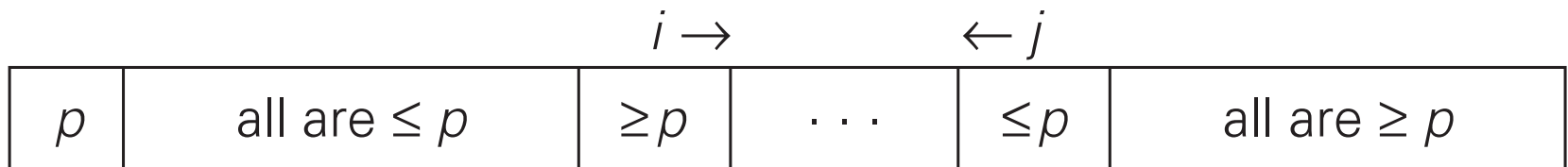
		$i \rightarrow$		$\leftarrow j$	
p	all are $\leq p$	$\geq p$	$\cdots$	$\leq p$	all are $\geq p$

Quicksort: `Partition()`

- After both scans stop, three situations may arise:

  - If $i < j$:

$$i \rightarrow \qquad \leftarrow j$$

p	all are $\leq p$	$\geq p$	$\cdots$	$\leq p$	all are $\geq p$
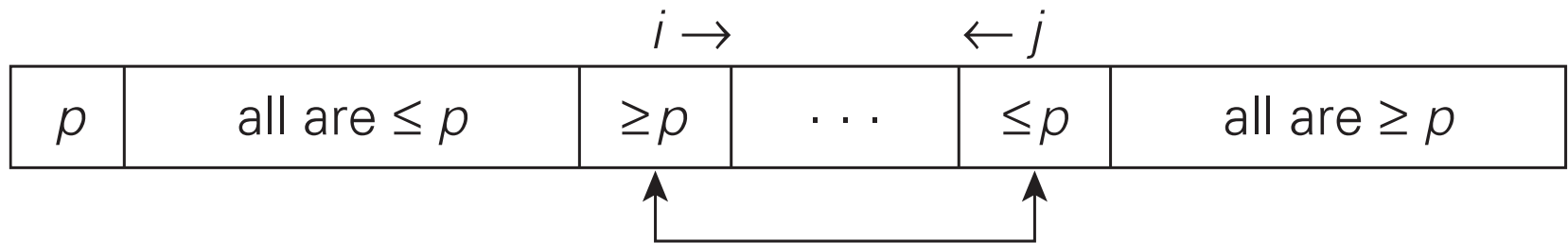
The function exchanges a_i and a_j and resume the scans by incrementing i and decrementing j by 1, respectively
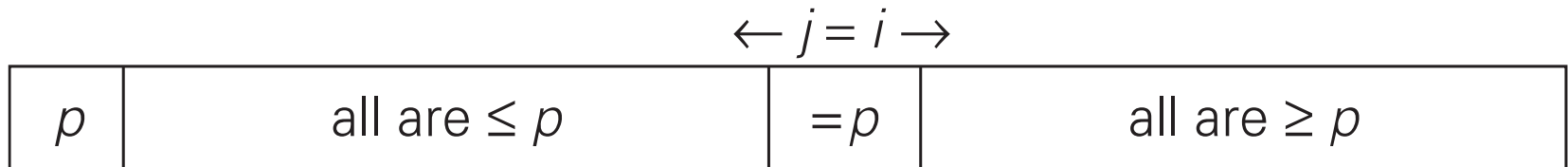
Quicksort: `Partition()`

- After both scans stop, three situations may arise:

  - If $i > j$:

$$\leftarrow j \qquad\qquad i \rightarrow$$

p	all are $\leq p$	$\leq p$	$\geq p$	all are $\geq p$
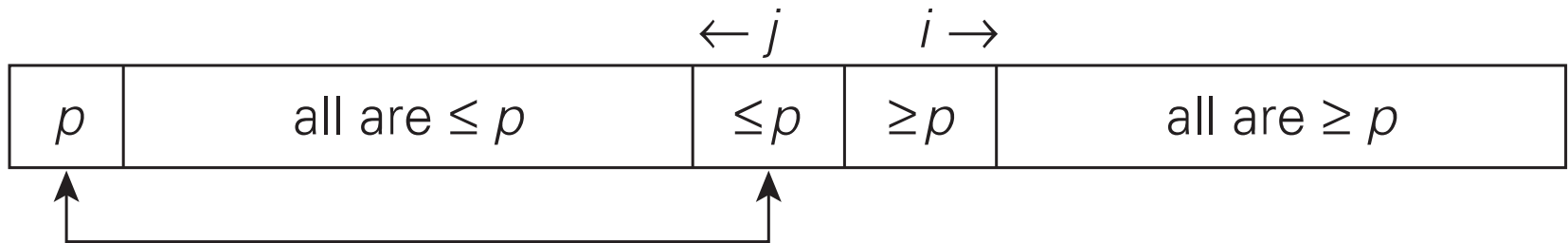
The function will have partitioned the (sub)array after exchanging the pivot p (a_{left}) with a_j. The split position is $s = j$

Quicksort: `Partition()`

- After both scans stop, three situations may arise:

  - If $i = j$:

$$\leftarrow j = i \rightarrow$$

p	all are $\leq p$	$= p$	all are $\geq p$

The value two scanning indices are pointing to must be equal to p. Thus, we have the (sub)array partitioned, with the split position $s = i = j$

- For the efficiency and legibility of the source code, we always exchange a_i with a_j

Quicksort: `Partition()`

```
Partition(a[], left, right) {
  p = a[left];
  i = left;
  j = right + 1;
  do {
    do i++; while (a[i] < p);
    do j--; while (a[j] > p);
    a[i] ⇄ a[j];
  } while (i < j);
  a[i] ⇄ a[j]; // undo last swap when i ≥ j
  a[left] ⇄ a[j];
  return  j;
}
```

Analysis of Quicksort

- The basic operation is the comparisons in do-while loop

- The efficiency of Quicksort depends on the value of p selected in each partition

  - *The best case*: The value of p is always the median of the (sub)array

  $$T(n) \in \Theta(n \log n)$$

  - *The worst case*: The value of p is always the maximum or minimum value of the (sub)array

  $$T(n) \in \Theta(n^2)$$

  - *The average case*: $T(n) \approx 1.39n \log_2 n \in \Theta(n \log n)$

Mergesort

- The Mergesort is a divide-and-conquer approach that was invented by John von Neumann in 1945

- Conceptually, the algorithm works as follows:
  - Divide the unsorted array into n subarrays, each containing one element
  - Repeatedly merge sorted subarrays to produce new sorted subarrays until there is only one subarray remaining
    - → This will be the sorted array

- Studying the details of the algorithm is left as an exercise for you

Split

Merge

3 2 7 4 9 1 6 8

3 2 7 4

9 1 6 8

3 2

7 4

9 1

6 8

3

2

7

4

9

1

6

8

2 3

4 7

1 9

6 8

2 3 4 7

1 6 8 9

1 2 3 4 6 7 8 9
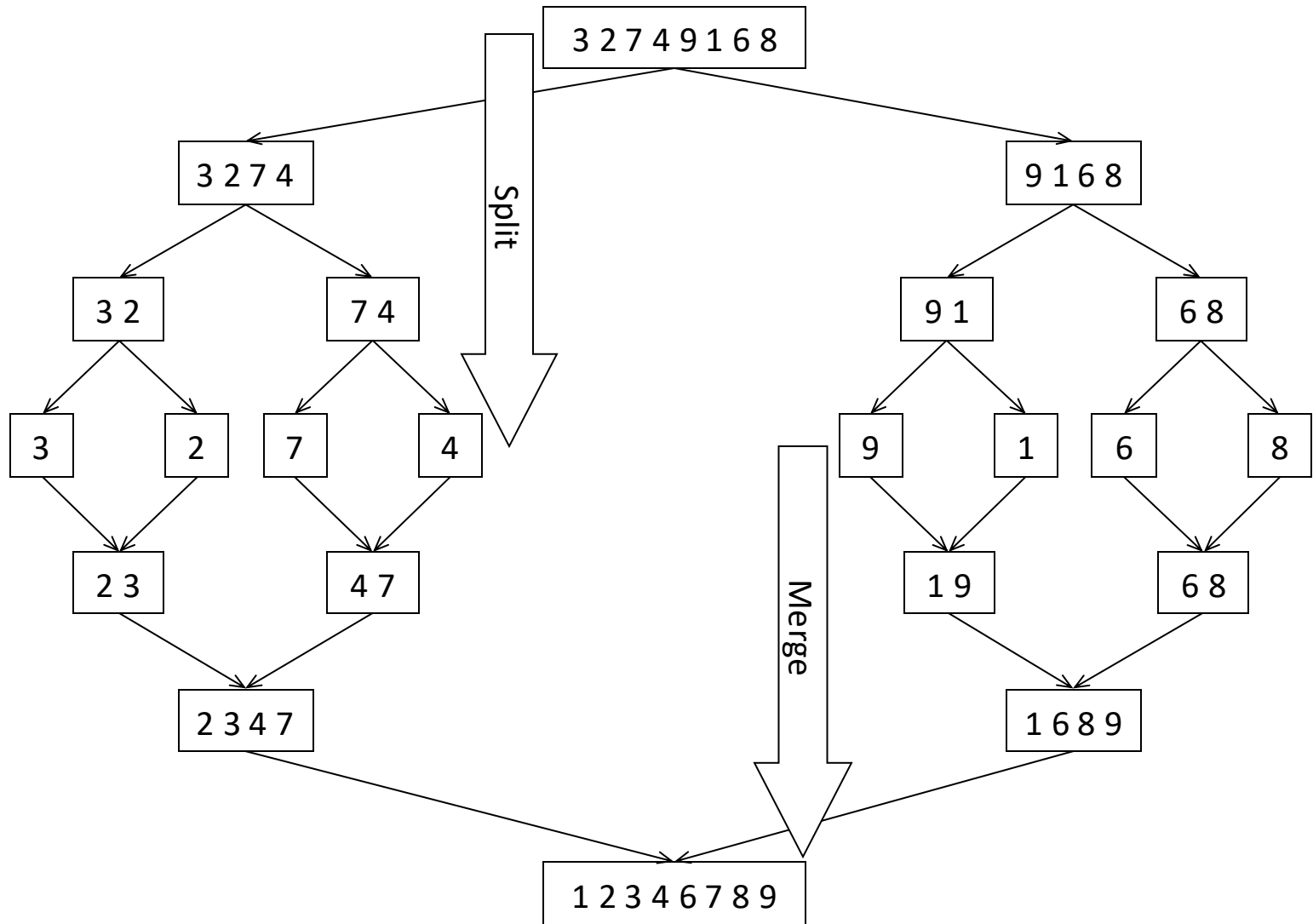
The non-comparison-based sorting algorithms

The Stability Of a Sorting Algorithm

- What is the stability of a sorting algorithm?

    It is the property that *the elements that compare to be equal preserve their original order after sorting*

- *A sorting algorithm is stable* if elements that are of the same value do not change their order after the sorting

Introduction to Counting Sort

- For each element of an array to be sorted, the total number of elements smaller than this element will indicate the position of the element in the sorted array

  - For simplicity, assume that the array contains no duplicate elements

- The sorted order this approach determines is based on *counting*

Counting Sort: A Silly Approach

```
count[1..n] = 0;
for (i = 1; i ≤ n - 1; i++)
  for (j = i + 1; j ≤ n; j++)
    if (a[i] < a[j])
      count[j]++;
    else
      count[i]++;
for (i = 1; i ≤ n; i++)
  b[count[i] + 1] = a[i];
a[1..n] = b[1..n];
```

Counting Sort: Step by Step

- The counting idea works in a situation in which elements to be sorted belong to a *known small set of values*

  - $\forall i \in [1, n]: a_i \in \mathbb{N} \wedge a_i \in [l, u]$

    → For simplicity, $l = 0$

- At first, we compute the frequency of each of those values and store them in array $f[0..u]$:

```
f[0..u] = 0;
for (i = 1; i ≤ n; i++)
  f[a[i]] ++;
```

Counting Sort: Step by Step

- Now, we run the following code segment:

```
for (i = 1; i ≤ u; i++)
  f[i] = f[i - 1] + f[i];
```

  - Since such accumulated sums of frequencies are called a distribution in statistics, the method itself is known as *distribution counting*

- The distribution values indicate the proper positions for the *last* occurrences of their elements in the sorted array

Counting Sort: Step by Step

- The last step is as follows:

```
for (i = n; i ≥ 1; i--) {
  b[f[a[i]]] = a[i];
  f[a[i]] --;
}
a[1..n] = b[1..n];
```

Analysis of Counting Sort

- This is a linear algorithm because it makes just two consecutive passes through its input array

  - This is a better time-efficiency class than that of the efficient sorting algorithms: mergesort, quicksort, and heapsort
  - However, the space complexity is high

- In practice, this algorithm should be used when $u \leq n$

  - It would be a disaster if $u \gg n$

Introduction to Radix Sort

- In principle, the radix sort algorithm works with whole numbers in all bases

  - Assume that the numbers to be sorted are in decimal base

- Let d be the number of digits of the largest element of an input array

  - The algorithm runs d passes to complete the sorting process

- As an auxiliary data structure, the algorithm also uses ten bins to store numbers

  - These bins are always empty before each of d passes

Radix Sort: Algorithm

- It sorts the array on the *least significant digit* (LSD) first
  - A number with the LSD i is stored in the i bin
  - All numbers are combined into an array, with the numbers in the 0 bin preceding the numbers in the 1 bin preceding the numbers in the 2 bin, and so on
- Then it sorts the array on the *second-least significant digit* and recombines the array in a like manner
- The process continues until the numbers have been sorted on all d digits

The Radix Sort: Example

- Let's consider an array of 6 integers:

$$28, 143, 550, 7, 911, 220$$

- After the first pass:

$$550, 220, 911, 143, 7, 28$$

- After the second pass:

$$7, 911, 220, 28, 143, 550$$

- After the third pass:

$$7, 28, 143, 220, 550, 911$$

The Radix Sort: Example

- Let's consider an array of 6 integers:

$$28, 143, 550, 7, 911, 220$$

- After the first pass:

$$550, 220, 911, 143, 7, 28$$

- After the second pass:

$$7, 911, 220, 28, 143, 550$$

- After the third pass:

$$7, 28, 143, 220, 550, 911$$
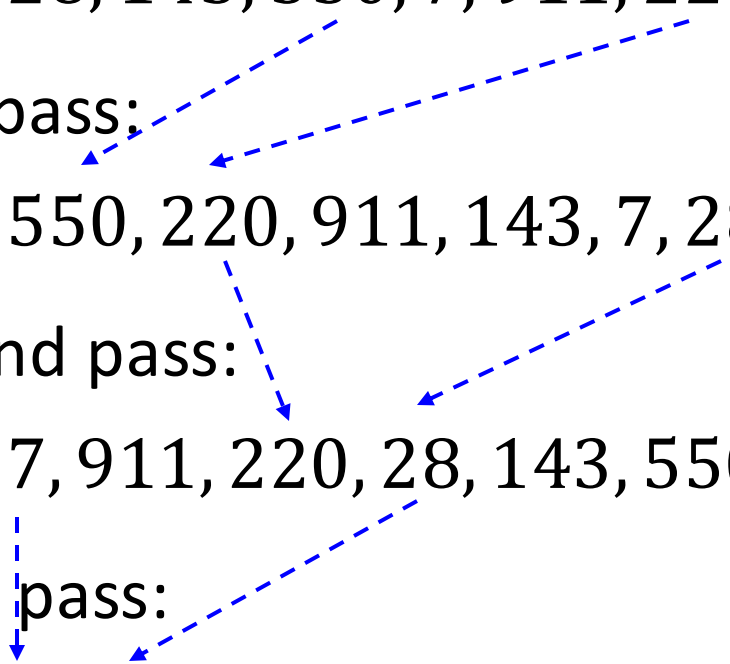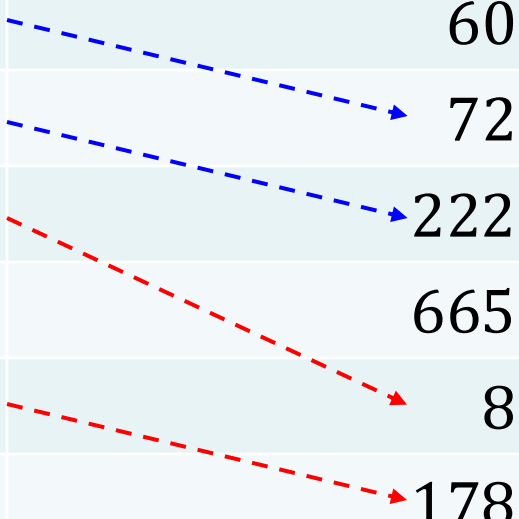
Counting Radix Sort

- It is an alternative implementation of the radix sort that avoids using bins

- It once again needs to run d passes to complete the sorting process

- In general, in the ith pass, it uses an *efficient sorting algorithm* to sort the numbers on the ith least significant digit

Counting Radix Sort: Example

	After the 1ˢᵗ pass	After the 2ⁿᵈ pass	After the 3ʳᵈ pass
72	60		
222	72		
8	222		
665	665		
178	8		
60	178		

Counting Radix Sort: Example

	After the 1st pass	After the 2nd pass	After the 3rd pass
72	60	8	
222	72	222	
8	222	60	
665	665	665	
178	8	72	
60	178	178	

Counting Radix Sort: Example

	After the 1st pass	After the 2nd pass	After the 3rd pass
72	60	8	8
222	72	222	60
8	222	60	72
665	665	665	178
178	8	72	222
60	178	178	665

Counting Radix Sort: Example

	After the 1st pass	After the 2nd pass	After the 3rd pass
72	60	8	8
222	72	222	60
8	222	60	72
665	665	665	178
178	8	72	222
60	178	178	665

An efficient sorting algorithm!!!

Counting Radix Sort: Pseudo-code

```
sort(a[1..n], k) {
  f[0..9] = {0};
  for (i = 1; i ≤ n; i++) f[digit(a[i], k)]++;
  for (i = 1; i ≤ 9; i++) f[i] += f[i - 1];
  for (i = n; i ≥ 1; i--) {
    j = digit(a[i], k);
    b[f[j]] = a[i];
    f[j] --;
  }
  a[1..n] = b[1..n];
}
LSDRadixSort(a[1..n], d) {
  for (k = 1; k ≤ d; k++)     sort(a, k);
}
```