BACS2063 Data Structures and Algorithms

OPTIONAL ALGORITHMS FOR SORTING

CHAPTER 8C (EXTRA READING)

RADIX SORT

Treats array elements as if they were strings of the same length or numbers with the same fixed number of digits

- Groups elements by a specified digit or character of the string
- Elements placed into "buckets" which match the digit (character)
- It sorts by processing the numbers digit by digit.
- The numbers are sorted according to the rightmost digit,then middle digit, and so on.

It is not appropriate for all data.

Radix sort is O(n) algorithm for certain data.

RADIX SORT

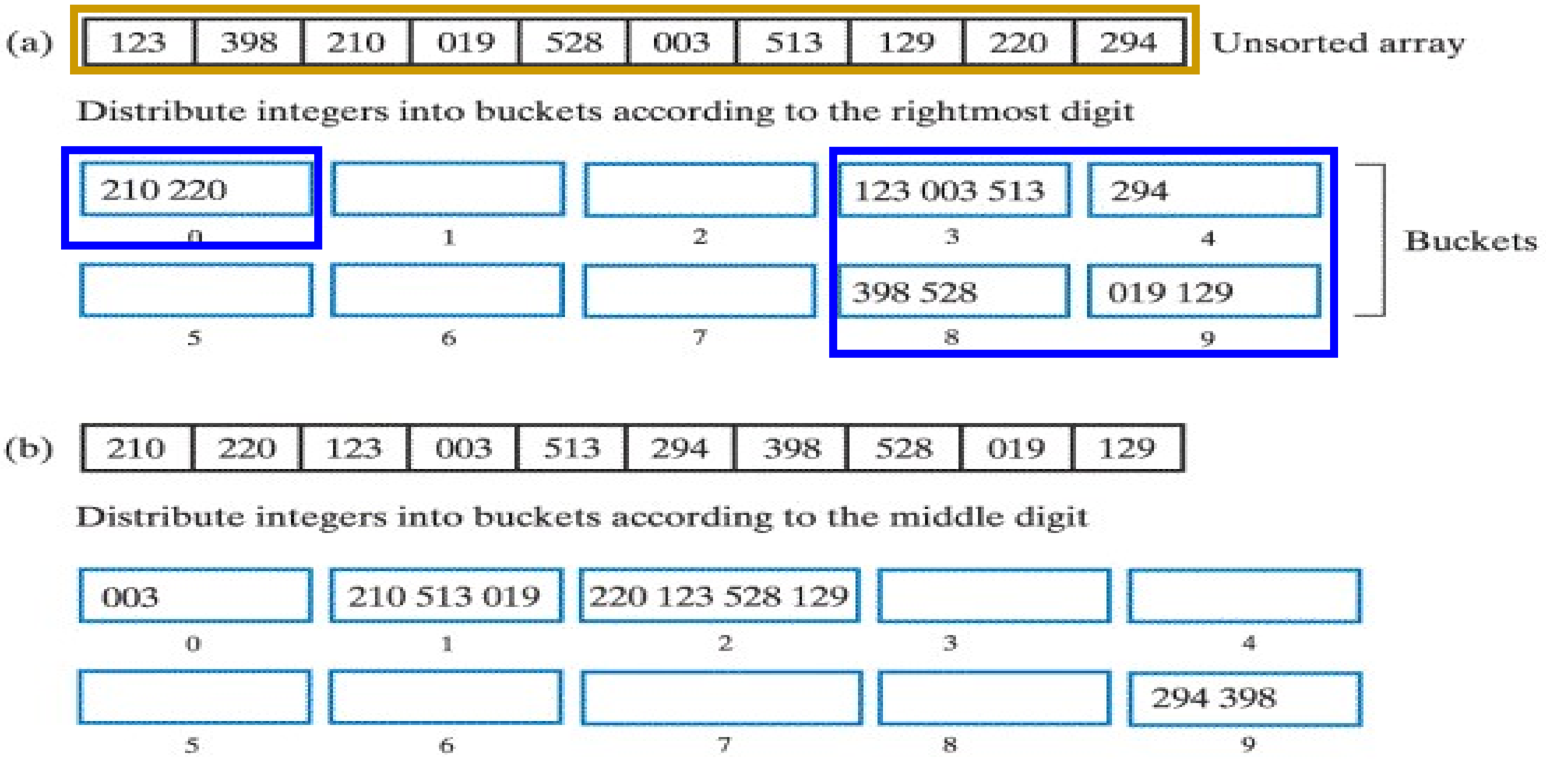


Fig. 12-9 (a) Original array and buckets after first distribution; (b) reordered array and buckets after second distribution … continued →

RADIX SORT

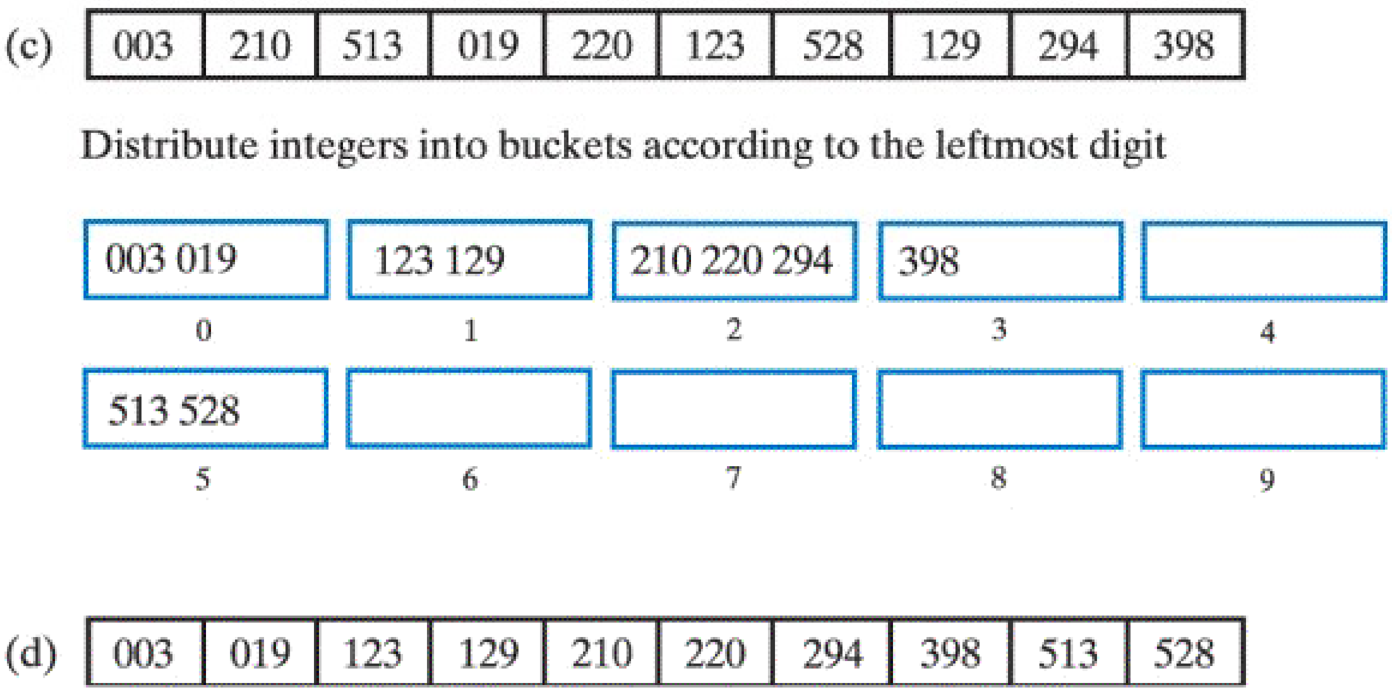


Fig. 12-9 (c) reordered array and buckets after third distribution; (d) sorted array

ALGORITHM FOR RADIX SORT

```
Algorithm radixSort (a, first, last, maxDigits) /
/ Sorts the array of positive decimal integers a[first..last] into ascending order;
// maxDigits is the number of digits in the longest integer.
            for (i = 0 to maxDigits - 1)  {
             Clear bucket [0], bucket [1], . . . , bucket [9]
             for (index = first to last)  {
                    digit = digit i of a [index]
                    Place a [index] at end of bucket [digit]
             }
             Place contents of bucket [0], bucket [1], . . . , bucket [9] into
    the array a
          }
```

SHELL SORT

A variation of the insertion sort

- But faster than $O(n^2)$

Done by sorting subarrays of equally spaced indices.

Faster than the bubble sort, selection sort and insertion sort

Instead of moving an element to an adjacent location, it moves several locations away

- Results in an almost sorted array
- This array sorted efficiently with ordinary insertion sort

SHELL SORT

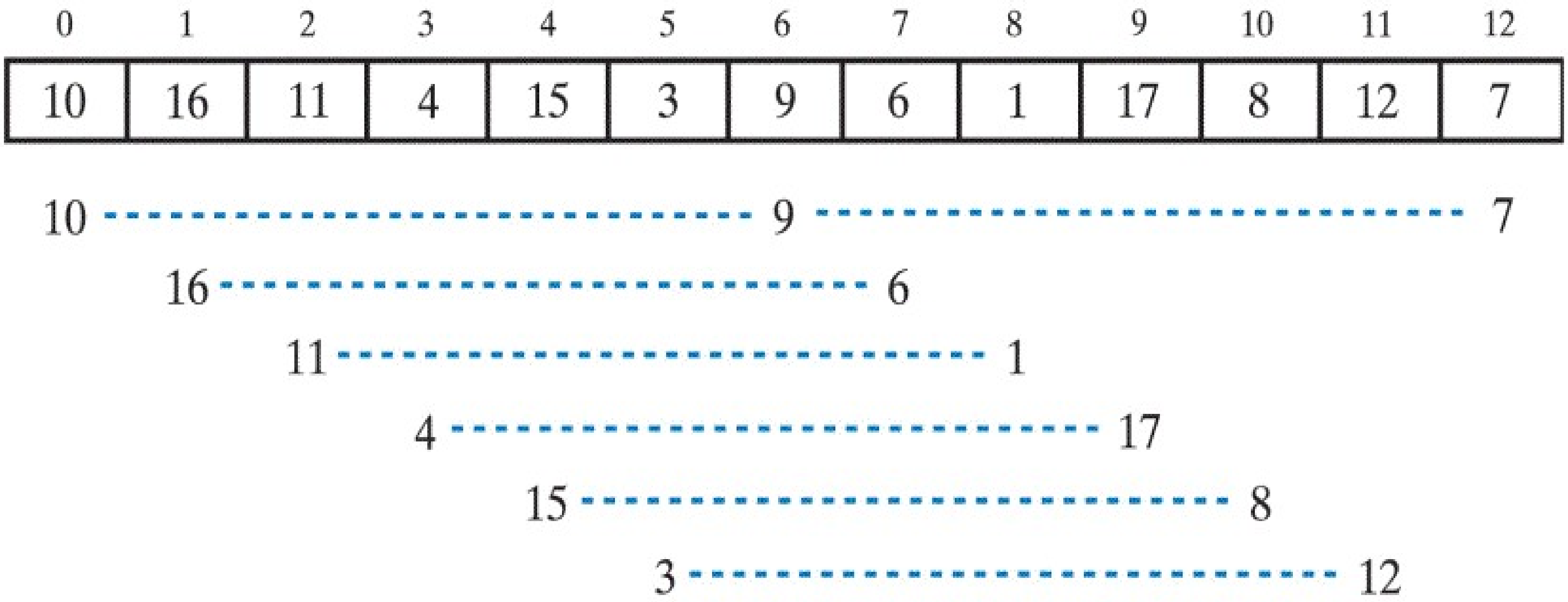
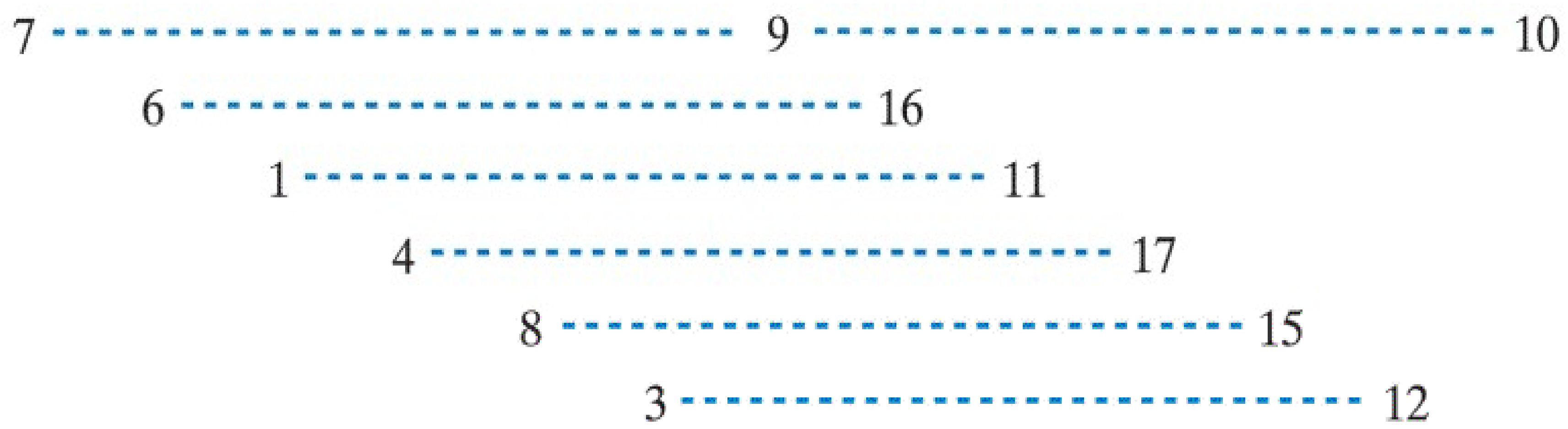


Fig. 11-12 An array and the subarrays formed by grouping elements whose indices are 6 apart.

SHELL SORT



7	6	1	4	8	3	9	16	11	17	15	12	10
0	1	2	3	4	5	6	7	8	9	10	11	12

Fig. 11-13 The subarrays of Fig. 11-12 after they are sorted, and the array that contains them.

SHELL SORT

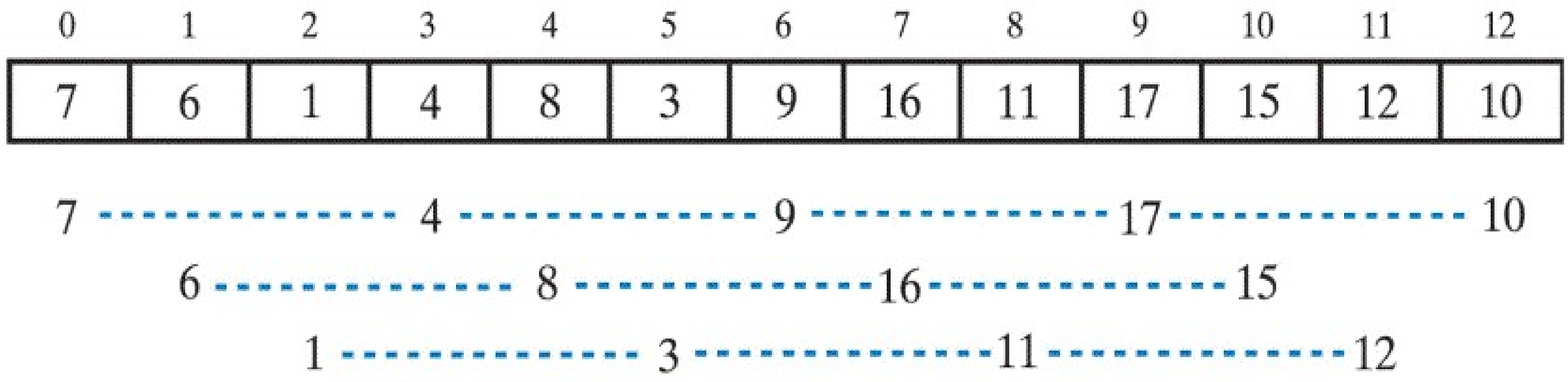


Fig. 11-14 The subarrays of the array in Fig. 11-13 formed by grouping elements whose indices are 3 apart

SHELL SORT

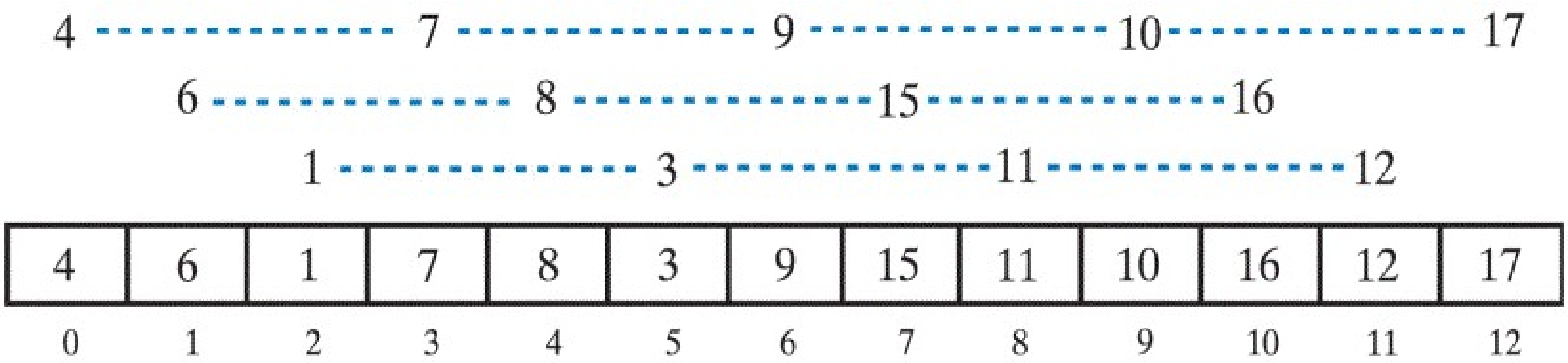


Fig. 11-15 The subarrays of Fig. 11-14 after they are sorted, and the array that contains them.

SOURCE CODE FOR **INCREMENTALINSERTIONSORT**

```
private static < T extends Comparable < ? super T >>
  void incrementalInsertionSort (T[] a, int first, int last, int space) {
     int unsorted, index;
     for (unsorted = first + space ; unsorted <= last ; unsorted = unsorted + space) {
        T firstUnsorted = a [unsorted];
        for (index = unsorted - space ; (index >= first) &&
              (firstUnsorted.compareTo (a [index]) < 0) ;  index = index - space)  {
                 a[index + space] = a [index];
        } // end for
        a [index + space] = firstUnsorted;
     } // end for
} // end incrementalInsertionSort
```

SOURCE CODE FOR **SHELLSORT**

```
public static < T extends Comparable < ? super T >>
   void shellSort (T [] a, int first, int last)  {
        int n = last - first + 1; // number of array elements
        for (int space = n / 2 ; space > 0 ; space = space / 2)  {
            for (int begin = first ; begin < first + space ; begin++)
                 incrementalInsertionSort (a, begin, last, space);
            } // end for
} // end shellSort
```

EFFICIENCY OF SHELL SORT

Efficiency is $O(n^2)$ for worst case

If *n* is a power of 2

- Average-case behavior is $O(n^{1.5})$

When the variable space index is even, add 1 to space. This results in consecutive increments that have no factor in common.

- Improve the worst case behavior to $O(n^{1.5})$

COMPARING THE SORTING ALGORITHMS

	Best Case	Average Case	Worst Case
Selection sort	O(n^2)	O(n^2)	O(n^2)
Insertion sort	O(n)	O(n^2)	O(n^2)
Shell sort	O(n)	O($n^{1.5}$)	O($n^{1.5}$) or O(n^2)
Bubble sort	O(n)	O(n^2)	O(n^2)
Radix sort	O(n)	O(n)	O(n)
Quick sort	O(n log n)	O(n log n)	O(n^2)
Merge sort	O(n log n)	O(n log n)	O(n log n)

Fig. 11-16 The time efficiencies of sorting algorithms, expressed in Big Oh notation.