Since we want the sorted array to finally appear in the original array A; we must execute the procedure MERGEPASS an even number of times.

Complexity of the Merge-Sort Algorithm

Let $f(n)$ denote the number of comparisons needed to sort an n-element array A using the merge-sort algorithm. Recall that the algorithm requires at most $\log n$ passes. Moreover, each pass merges a total of n elements, and by the discussion on the complexity of merging, each pass will require at most n comparisons. Accordingly, for both the worst case and average case,

$$f(n) \le n \log n$$

Observe that this algorithm has the same order as heapsort and the same average order as quicksort. The main drawback of merge-sort is that it requires an auxiliary array with n elements. Each of the other sorting algorithms we have studied requires only a finite number of extra locations, which is independent of n.

The above results are summarized in the following table:

Algorithm	Worst Case	Average Case	Extra Memory
Merge-Sort	$n \log n = O(n \log n)$	$n \log n = O(n \log n)$	$O(n)$

?

9.7 RADIX SORT

2 ways of sorting.

Purpose:

Radix sort is the method that many people intuitively use or begin to use when alphabetizing a large list of names. (Here the radix is 26, the 26 letters of the alphabet.) Specifically, the list of names is first sorted according to the first letter of each name. That is, the names are arranged in 26 classes, where the first class consists of those names that begin with "A," the second class consists of those names that begin with "B," and so on. During the second pass, each class is alphabetized according to the second letter of the name. And so on. If no name contains, for example, more than 12 letters, the names are alphabetized with at most 12 passes. [illegible]

The radix sort is the method used by a card sorter. A card sorter contains 13 receiving pockets labeled as follows:

$$9,\ 8,\ 7,\ 6,\ 5,\ 4,\ 3,\ 2,\ 1,\ 0,\ 11,\ 12,\ \text{R (reject)}$$

Each pocket other than R corresonds to a row on a card in which a hole can be punched. Decimal numbers, where the radix is 10, are punched in the obvious way and hence use only the first 10 pockets of the sorter. The sorter uses a radix reverse-digit sort on numbers. That is, suppose a card sorter is given a collection of cards where each card contains a 3-digit number punched in columns 1 to 3. The cards are first sorted according to the units digit. On the second pass, the cards are sorted according to the tens digit. On the third and last pass, the cards are sorted according to the hundreds digit. We illustrate with an example.

EXAMPLE 9.8

Suppose 9 cards are punched as follows:

$$348,\ 143,\ 361,\ 423,\ 538,\ 128,\ 321,\ 543,\ 366$$

Given to a card sorter, the numbers would be sorted in three phases, as pictured in Fig. 9-6:

(*a*) In the first pass, the units digits are sorted into pockets. (The pockets are pictured upside down, so 348 is at the bottom of pocket 8.) The cards are collected pocket by pocket, from pocket 9 to pocket 0. (Note that 361 will now be at the bottom of the pile and 128 at the top of the pile.) The cards are now reinput to the sorter.

(*b*) In the second pass, the tens digits are sorted into pockets. Again the cards are collected pocket by pocket and reinput to the sorter.

...ite, no.s ...ically in ...umn in same order

Input	0	1	2	3	4	5	6	7	8	9
348									348	
143				143						
361		361								
423				423						
538									538	
128									128	
321		321								
543				543						
366							366			

331

→ last digit check and write no. in that column.
•(write with last digits sequence)
→ sort with last digit.

(*a*) First pass.

→ write values vertically in matri... by their order in first pass column.

Input	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366					543					
366							366			
348					348					
538				538						
128			128							

→ sort with 2nd digit.

(*b*) Second pass.

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128								
538						538				
143		143								
543						543				
348				348						
361				361						
366				366						

→ sort with first value digit.

(*c*) Third pass.

Fig. 9·6 finally, write output.

1-9

(c) In the third and final pass, the hundreds digits are sorted into pockets.

When the cards are collected after the third pass, the numbers are in the following order:

$$\boxed{128,\ 143,\ 321,\ 348,\ 361,\ 366,\ 423,\ 538,\ 543}$$

12

Thus the cards are now sorted.

The number C of comparisons needed to sort nine such 3-digit numbers is bounded as follows:

$$C \le 9 \cdot 3 \cdot 10$$

0-9

The 9 comes from the nine cards, the 3 comes from the three digits in each number, and the 10 comes from radix $d = 10$ digits.

Complexity of Radix Sort

Suppose a list A of n items $A_1, A_2, \ldots, A_n$ is given. Let d denote the radix (e.g., $d = 10$ for decimal digits, $d = 26$ for letters and $d = 2$ for bits), and suppose each item A_i is represented by means of s of the digits:

$$A_i = d_{i1} d_{i2} \cdots d_{is}$$

The radix sort algorithm will require s passes, the number of digits in each item. Pass K will compare each d_{iK} with each of the d digits. Hence the number $C(n)$ of comparisons for the algorithm is bounded as follows:

$$\boxed{C(n) \le d * s * n}$$

Although d is independent of n, the number s does depend on n. In the worst case, $s = n$, so $C(n) = O(n^2)$. In the best case, $s = \log_d n$, so $C(n) = O(n \log n)$. In other words, radix sort performs ~~well only~~ when the number s of digits in the representation of the A_i's is small.

Another drawback of radix sort is that one may need $d*n$ memory locations. This comes from the fact that all the items may be "sent to the same pocket" during a given pass. This drawback may be minimized by using linked lists rather than arrays to store the items during a given pass. However, one will still require $2*n$ memory locations.

9.3 SEARCHING AND DATA MODIFICATION

Suppose S is a collection of data maintained in memory by a table using some type of data structure. Searching is the operation which finds the location LOC in memory of some given ITEM of information or sends some message that ITEM does not belong to S. The search is said to be successful or unsuccessful according to whether ITEM does or does not belong to S. The searching algorithm that is used depends mainly on the type of data structure that is used to maintain S in memory.

Data modification refers to the operations of inserting, deleting and updating. Here data modification will mainly refer to inserting and deleting. These operations are closely related to searching, since usually one must search for the location of the ITEM to be deleted or one must search for the proper place to insert ITEM in the table. The insertion or deletion also requires a certain amount of execution time, which also depends mainly on the type of data structure that is used.

Generally speaking, there is a tradeoff between data structures with fast searching algorithms and data structures with fast modification algorithms. This situation is illustrated below, where we summarize the searching and data modification of three of the data structures previously studied in the text.

(1) *Sorted array.* Here one can use a binary search to find the location LOC of a given ITEM in time $O(\log n)$. On the other hand, inserting and deleting are very slow, since, on the average, $n/2 = O(n)$ elements must be moved for a given insertion or deletion. Thus a sorted array would likely be used when there is a great deal of searching but only very little data modification.