CSE-2321

Data Structure

Part-6

Presented by

Asmaul Hosna Sadika

Adjunct Faculty

Dept of CSE, IIUC

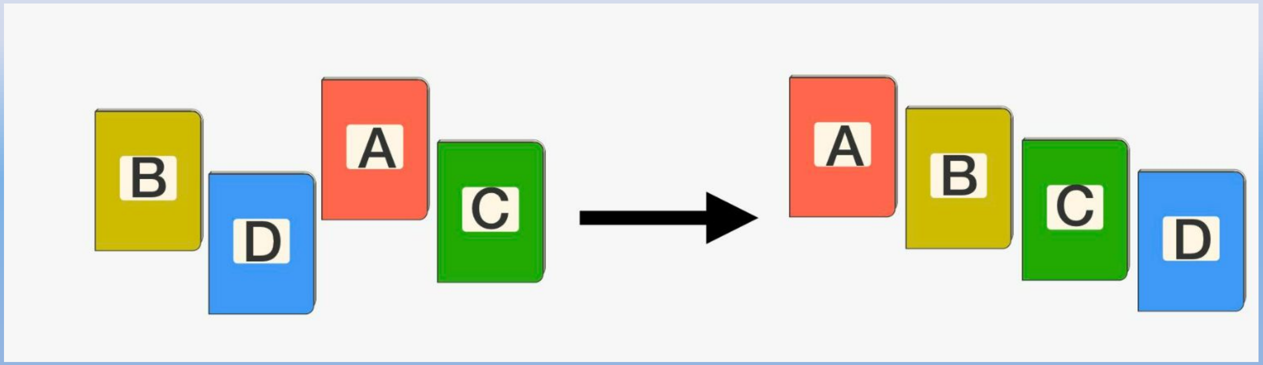
Slide credit: Professor Mohammed Shamsul Alam

Contents

What is sorting?

Sorting refers to the rearrangement of a given array or list of elements according to a comparison operator on the elements.

The comparison operator is used to decide the new order of elements in the respective data structure



Why Sorting Algorithms are Important

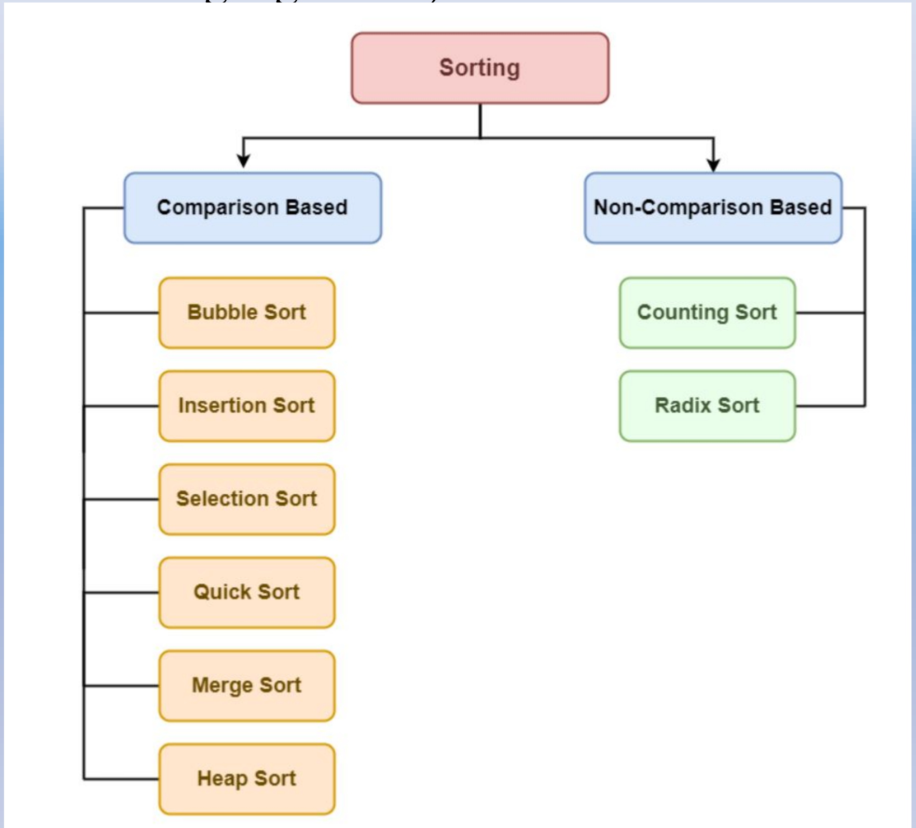
The sorting algorithm is important in Computer Science because it reduces the complexity of a problem. There is a wide range of applications for these algorithms, including searching algorithms, database algorithms, divide and conquer methods, and data structure algorithms.

In the following sections, we list some important scientific applications where sorting algorithms are used

- When you have hundreds of datasets you want to print, you might want to arrange them in some way.
- Once we get the data sorted, we can get the k-th smallest and k-th largest item in O(1) time.
- Searching any element in a huge data set becomes easy. We can use Binary search method for search if we have sorted data. So, Sorting become important here.
- They can be used in software and in conceptual problems to solve more advanced problems.

Types of Sorting Techniques

- **Comparison-based:** We compare the elements in a comparison-based sorting algorithm)
- **Non-comparison-based:** We do not compare the elements in a non-comparison-based sorting algorithm)



Sorting

- A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are **numerical order** and **lexicographical order**.

- **Iterative sorting algorithms (comparison based)**
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
- **Recursive sorting algorithms (comparison based)**
  - Merge Sort
  - Quick Sort
  - Heap Sort
- **Radix sort (non-comparison based)**

- **Given a set of *n* values, there can be *n!* permutations of these values.**
- **O (n log n) is the best possible for any comparison based sorting algorithm which sorts n items.**

Insertion Sort

Suppose an array A with N elements A[1], A[2], ... A[N] is in memory. The insertion sort algorithm scan A from A[1] to A[N], inserting each element A[K] into its proper position in the previously sorting sub array A[1], A[2], ...A[K-1]. That is:

Pass 1: A[1] by itself is trivially sorted

Pass 2: A[2] is inserted either before of after A[1] so that: A[1], A[2] is sorted.

Pass 3: A[3] is inserted into its proper place in A[1], A[2], that is, before A[1}, between A[1] and A[2], or after a[2], so that A[1], A[2], A[3] are sorted.

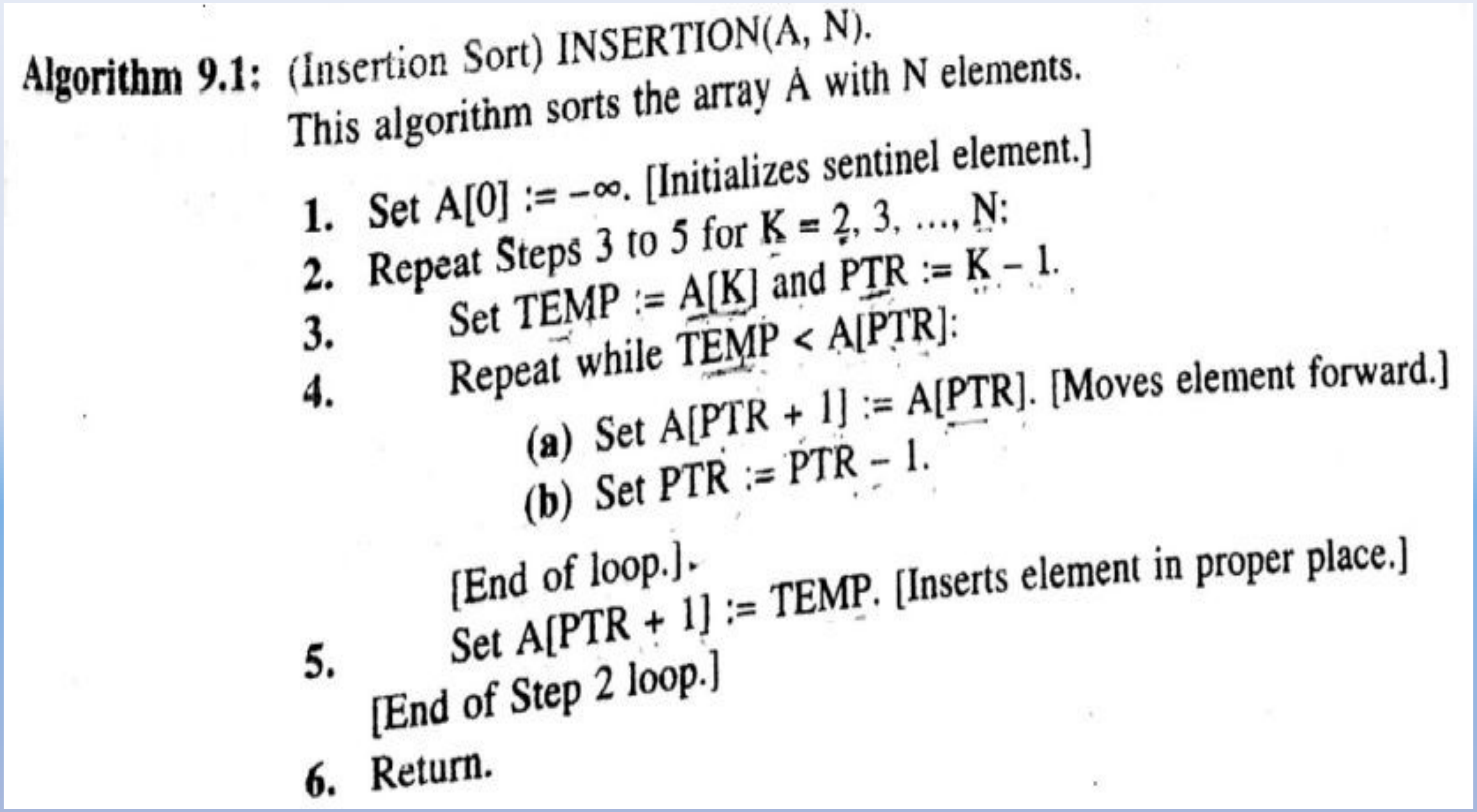
Pass 4: A[4] is inserted into its proper place in A[1], A[2], A[3] so that: A[1], A[2], A[3], A[4] are sorted.

. . . .

Pass N: A[N] is inserted into its proper place in A[1], A[2], ... A[N-1] so that: A[1], A[2], ... A[N] are sorted.

This sorting algorithm is frequently used when N is small. There remains only the problem of deciding how to insert A[K] in its proper place in the subarray A[1], A[2], ... A[K-1]. This can be accomplished by comparing A[K] with A[K-1], comparing A[K] with A[K-2], comparing A[K] with A[K-3], and so on, until first meeting an element A[J] such that A[J] ≤ A[K]. Then each of elements A[K-1], A[K-2], ... A[J+1] is moved forward one location, and A[K] is then inserted in the J+1 st position in the array.

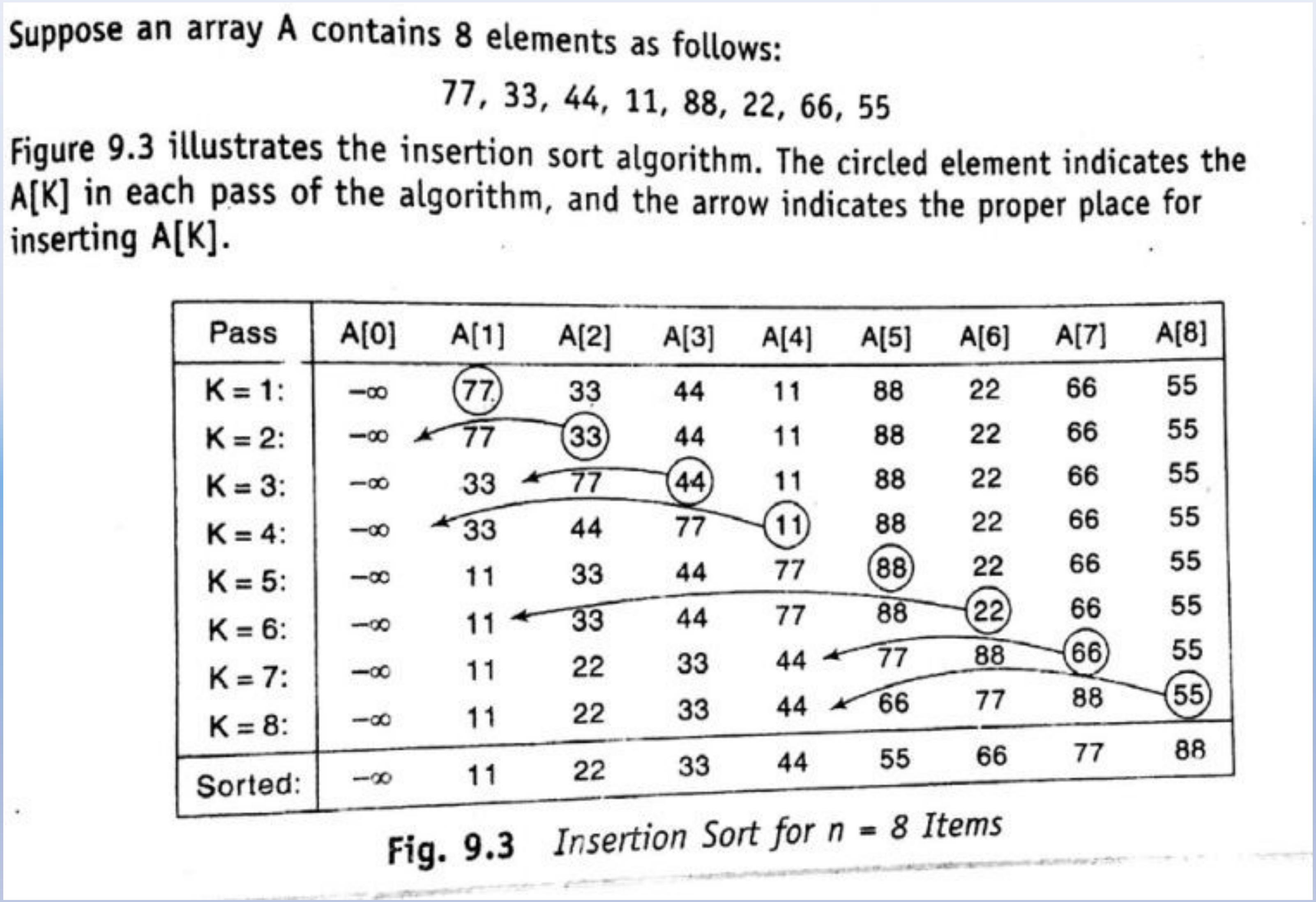
Insertion Sort



Algorithm 9.1: (Insertion Sort) INSERTION(A, N).
This algorithm sorts the array A with N elements.

```
1. Set A[0] := -∞. [Initializes sentinel element.]
2. Repeat Steps 3 to 5 for K = 2, 3, ..., N:
3.         Set TEMP := A[K] and PTR := K - 1.
4.         Repeat while TEMP < A[PTR]:
               (a) Set A[PTR + 1] := A[PTR]. [Moves element forward.]
               (b) Set PTR := PTR - 1.
           [End of loop.]
5.         Set A[PTR + 1] := TEMP. [Inserts element in proper place.]
   [End of Step 2 loop.]
6. Return.
```

Insertion Sort



Suppose an array A contains 8 elements as follows:

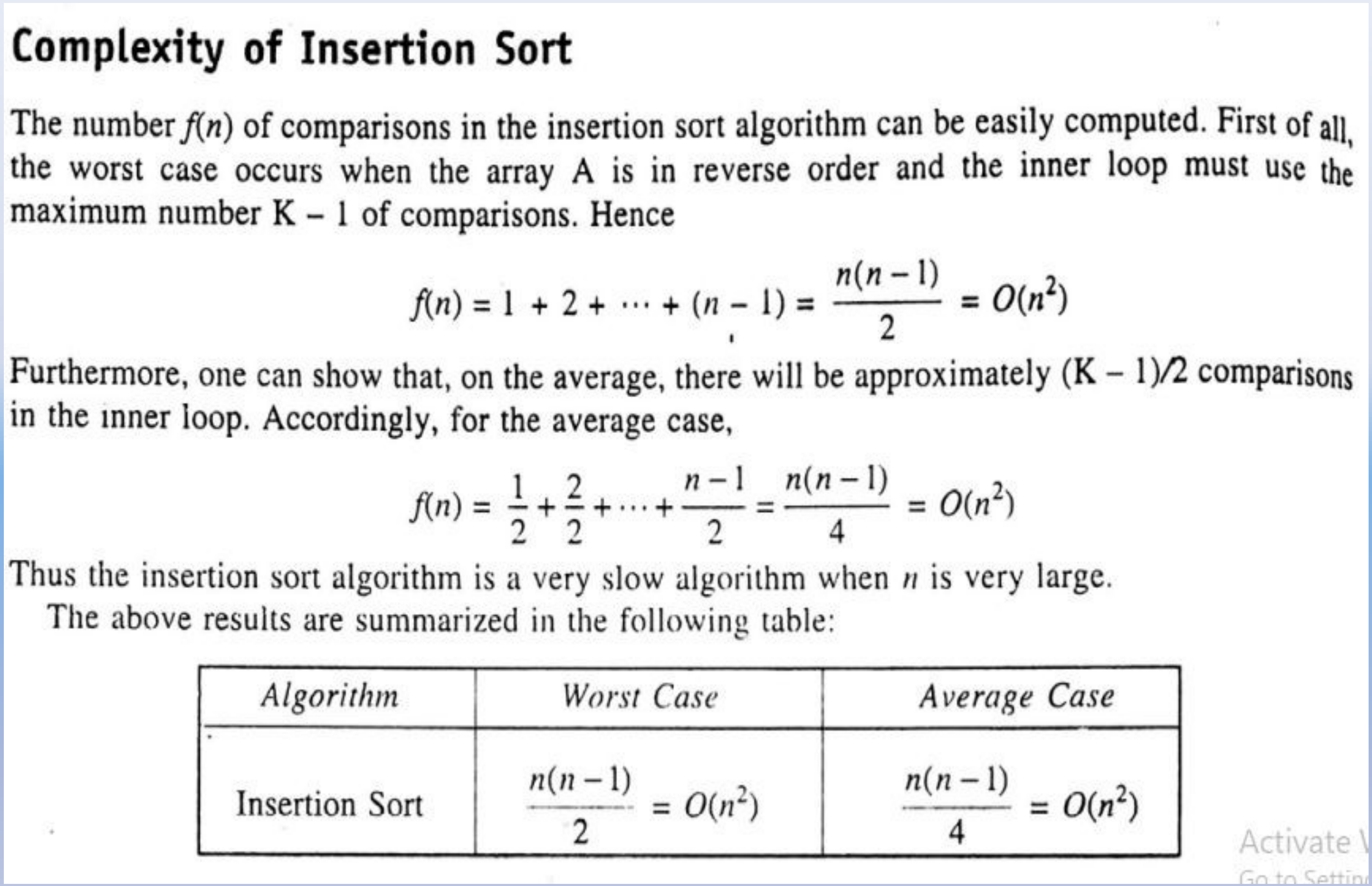
77, 33, 44, 11, 88, 22, 66, 55

Figure 9.3 illustrates the insertion sort algorithm. The circled element indicates the A[K] in each pass of the algorithm, and the arrow indicates the proper place for inserting A[K].

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K = 1:	−∞	77	33	44	11	88	22	66	55
K = 2:	−∞	77	33	44	11	88	22	66	55
K = 3:	−∞	33	77	44	11	88	22	66	55
K = 4:	−∞	33	44	77	11	88	22	66	55
K = 5:	−∞	11	33	44	77	88	22	66	55
K = 6:	−∞	11	33	44	77	88	22	66	55
K = 7:	−∞	11	22	33	44	77	88	66	55
K = 8:	−∞	11	22	33	44	66	77	88	55
Sorted:	−∞	11	22	33	44	55	66	77	88

Fig. 9.3 *Insertion Sort for n = 8 Items*

Insertion Sort



Complexity of Insertion Sort

The number $f(n)$ of comparisons in the insertion sort algorithm can be easily computed. First of all, the worst case occurs when the array A is in reverse order and the inner loop must use the maximum number K – 1 of comparisons. Hence

$$f(n) = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

Furthermore, one can show that, on the average, there will be approximately (K – 1)/2 comparisons in the inner loop. Accordingly, for the average case,

$$f(n) = \frac{1}{2} + \frac{2}{2} + \cdots + \frac{n-1}{2} = \frac{n(n-1)}{4} = O(n^2)$$

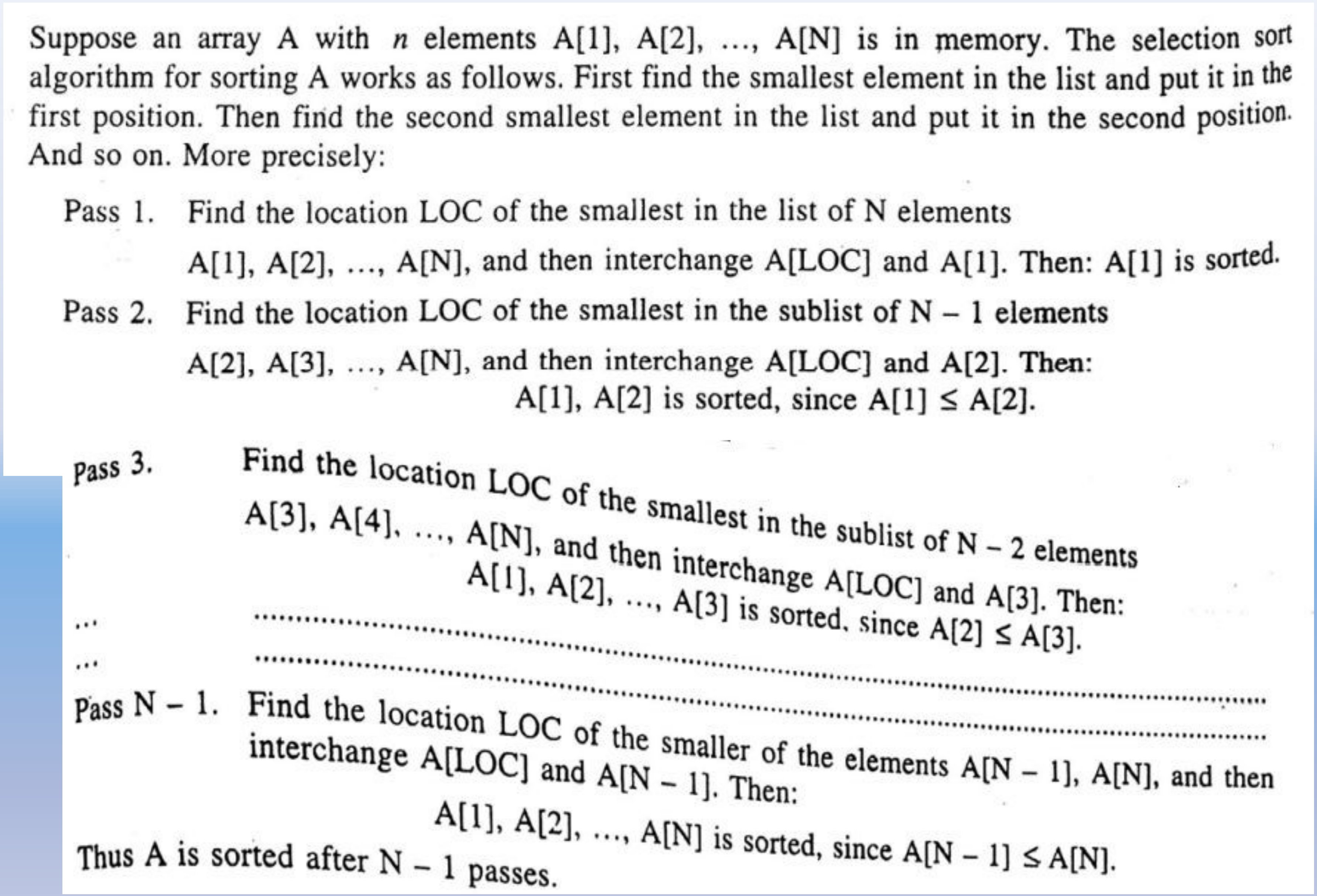
Thus the insertion sort algorithm is a very slow algorithm when n is very large.

The above results are summarized in the following table:

Algorithm	*Worst Case*	*Average Case*
Insertion Sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{4} = O(n^2)$

Activate V
Go to Settin

Selection Sort



Suppose an array A with *n* elements A[1], A[2], …, A[N] is in memory. The selection sort algorithm for sorting A works as follows. First find the smallest element in the list and put it in the first position. Then find the second smallest element in the list and put it in the second position. And so on. More precisely:

Pass 1. Find the location LOC of the smallest in the list of N elements A[1], A[2], …, A[N], and then interchange A[LOC] and A[1]. Then: A[1] is sorted.

Pass 2. Find the location LOC of the smallest in the sublist of N – 1 elements A[2], A[3], …, A[N], and then interchange A[LOC] and A[2]. Then:

A[1], A[2] is sorted, since A[1] ≤ A[2].

Pass 3. Find the location LOC of the smallest in the sublist of N – 2 elements A[3], A[4], …, A[N], and then interchange A[LOC] and A[3]. Then:

A[1], A[2], …, A[3] is sorted, since A[2] ≤ A[3].

… ……………………………………………………………………………………

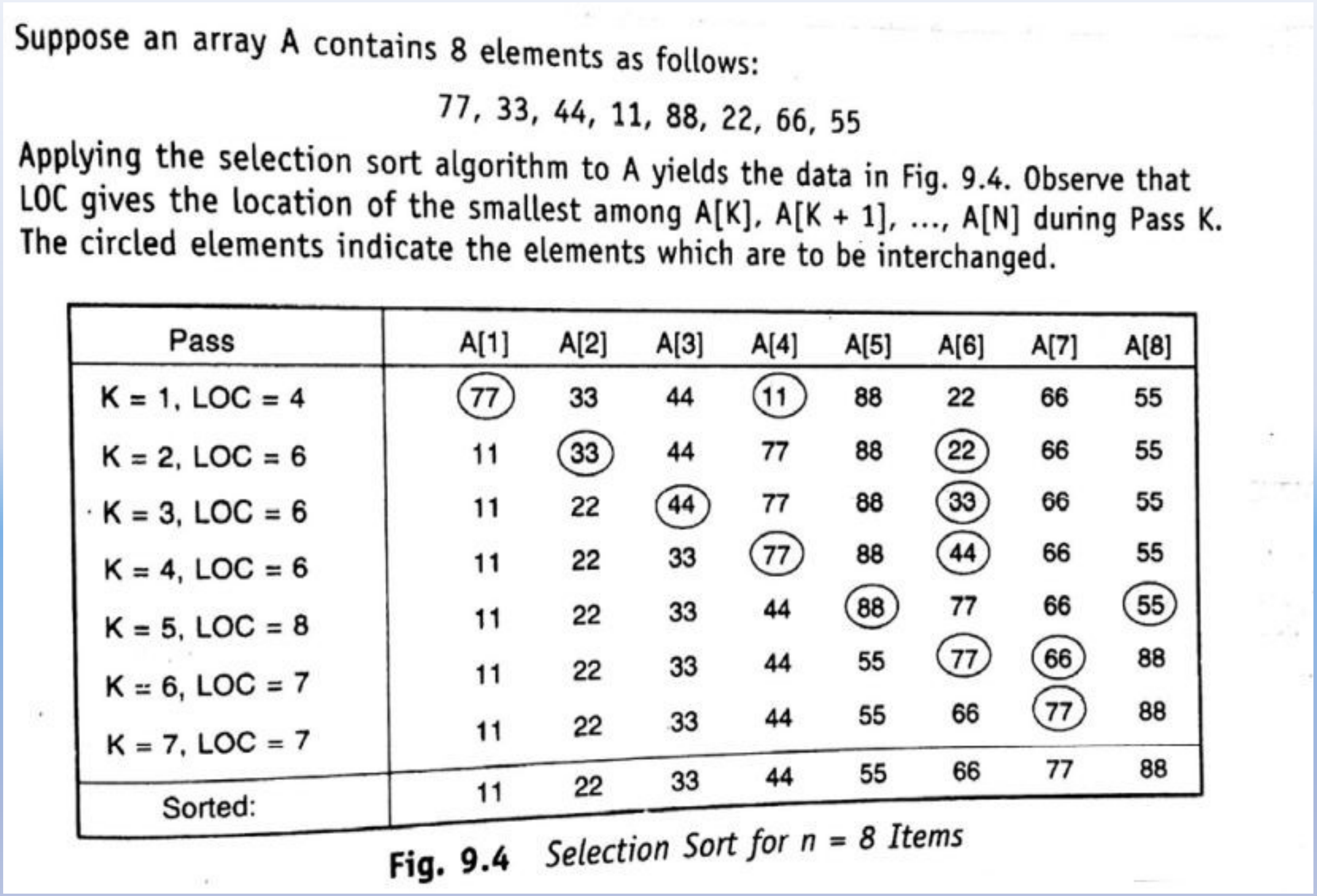
… ……………………………………………………………………………………

Pass N – 1. Find the location LOC of the smaller of the elements A[N – 1], A[N], and then interchange A[LOC] and A[N – 1]. Then:

A[1], A[2], …, A[N] is sorted, since A[N – 1] ≤ A[N].

Thus A is sorted after N – 1 passes.

Selection Sort



Suppose an array A contains 8 elements as follows:

77, 33, 44, 11, 88, 22, 66, 55

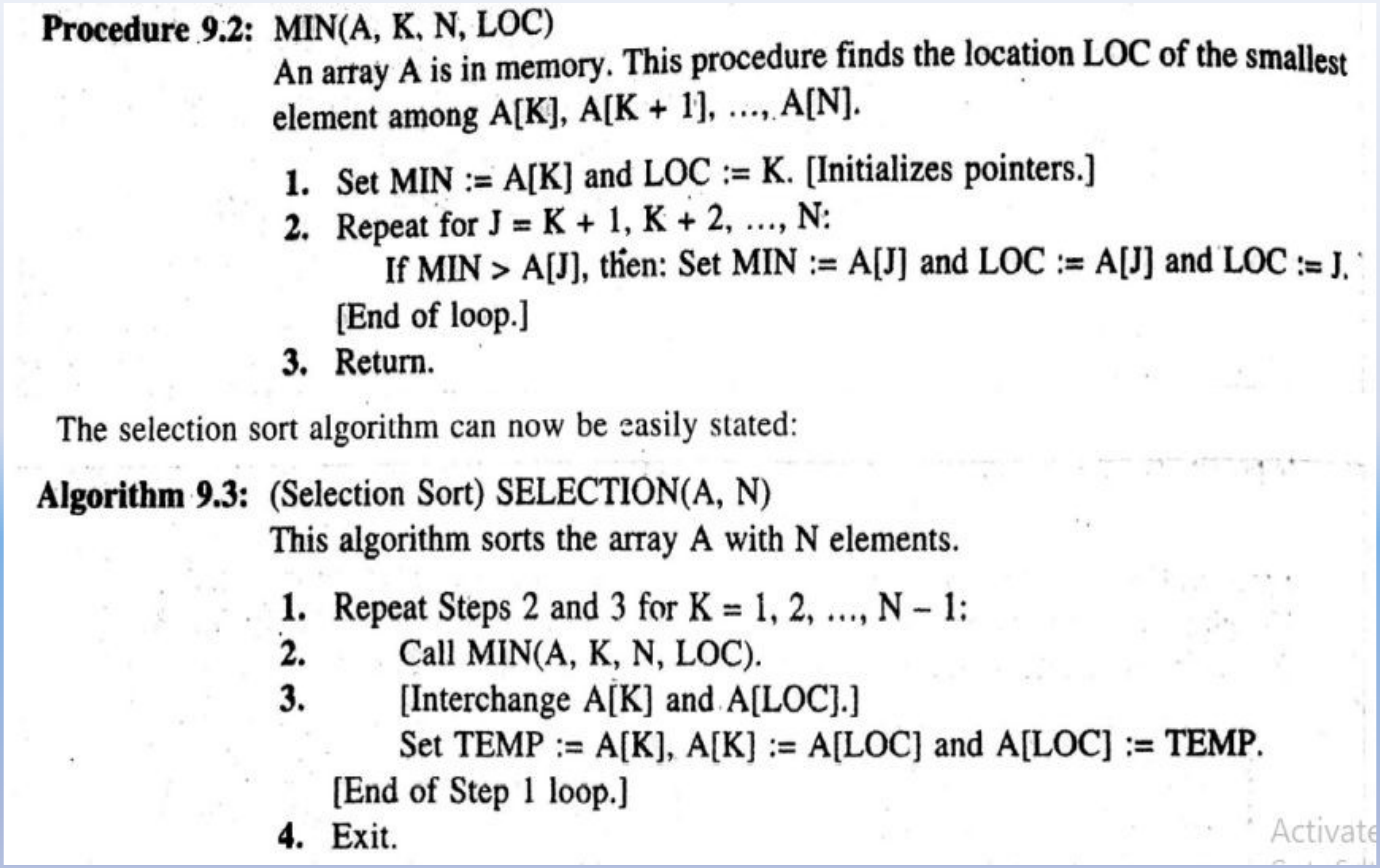
Applying the selection sort algorithm to A yields the data in Fig. 9.4. Observe that LOC gives the location of the smallest among A[K], A[K + 1], ..., A[N] during Pass K. The circled elements indicate the elements which are to be interchanged.

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K = 1, LOC = 4	$\boxed{77}$	33	44	$\boxed{11}$	88	22	66	55
K = 2, LOC = 6	11	$\boxed{33}$	44	77	88	$\boxed{22}$	66	55
K = 3, LOC = 6	11	22	$\boxed{44}$	77	88	$\boxed{33}$	66	55
K = 4, LOC = 6	11	22	33	$\boxed{77}$	88	$\boxed{44}$	66	55
K = 5, LOC = 8	11	22	33	44	$\boxed{88}$	77	66	$\boxed{55}$
K = 6, LOC = 7	11	22	33	44	55	$\boxed{77}$	$\boxed{66}$	88
K = 7, LOC = 7	11	22	33	44	55	66	$\boxed{77}$	88
Sorted:	11	22	33	44	55	66	77	88

Fig. 9.4 *Selection Sort for n = 8 Items*

Selection Sort

.



Procedure 9.2: MIN(A, K, N, LOC)
An array A is in memory. This procedure finds the location LOC of the smallest element among A[K], A[K + 1], …, A[N].

```
1. Set MIN := A[K] and LOC := K. [Initializes pointers.]
2. Repeat for J = K + 1, K + 2, …, N:
        If MIN > A[J], then: Set MIN := A[J] and LOC := A[J] and LOC := J.
   [End of loop.]
3. Return.
```

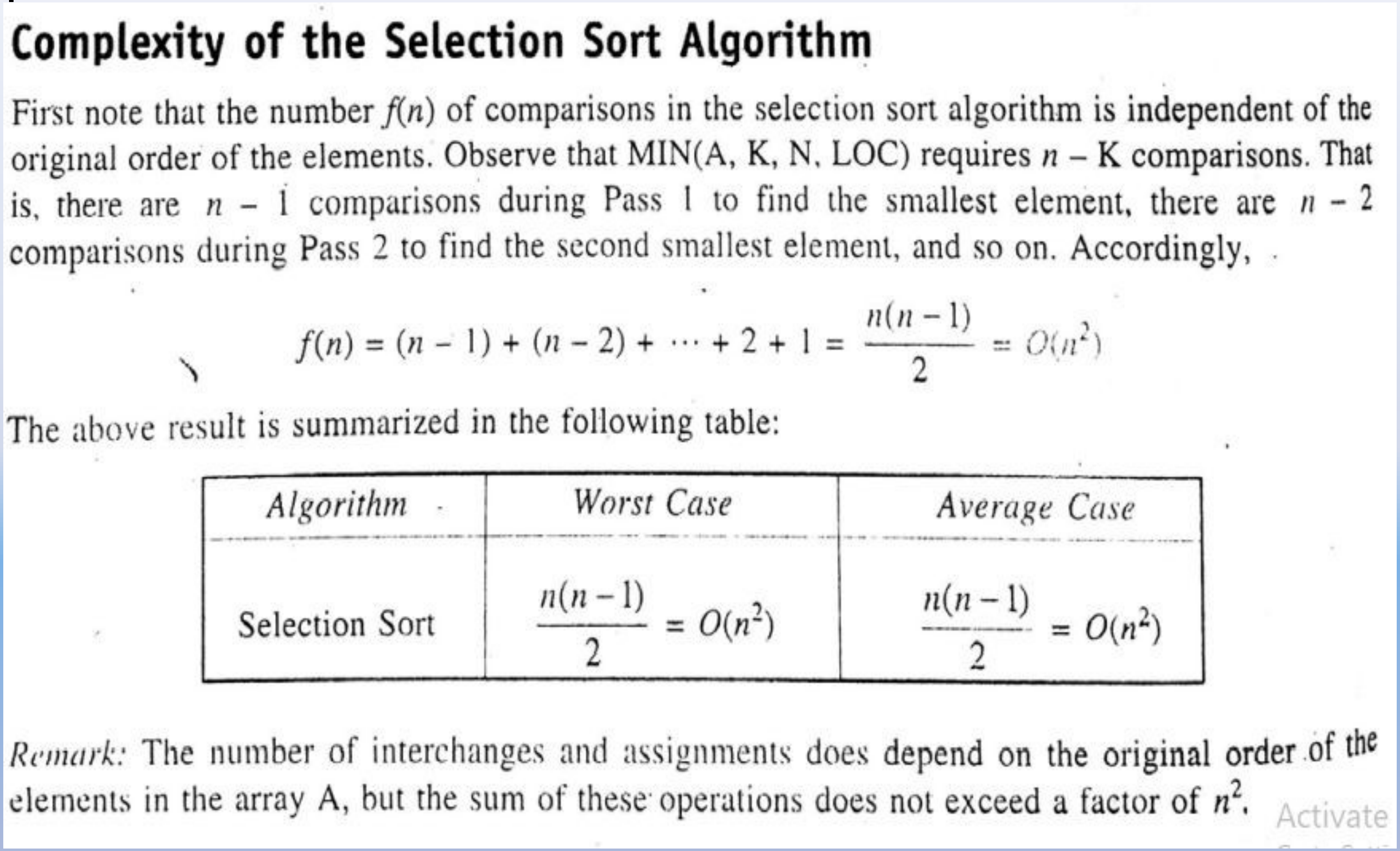
The selection sort algorithm can now be easily stated:

Algorithm 9.3: (Selection Sort) SELECTION(A, N)
This algorithm sorts the array A with N elements.

```
1. Repeat Steps 2 and 3 for K = 1, 2, …, N – 1:
2.      Call MIN(A, K, N, LOC).
3.      [Interchange A[K] and A[LOC].]
        Set TEMP := A[K], A[K] := A[LOC] and A[LOC] := TEMP.
   [End of Step 1 loop.]
4. Exit.
```

Activate

Selection Sort



Complexity of the Selection Sort Algorithm

First note that the number $f(n)$ of comparisons in the selection sort algorithm is independent of the original order of the elements. Observe that MIN(A, K, N, LOC) requires n – K comparisons. That is, there are $n - 1$ comparisons during Pass 1 to find the smallest element, there are $n - 2$ comparisons during Pass 2 to find the second smallest element, and so on. Accordingly,

$$f(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2} = O(n^2)$$

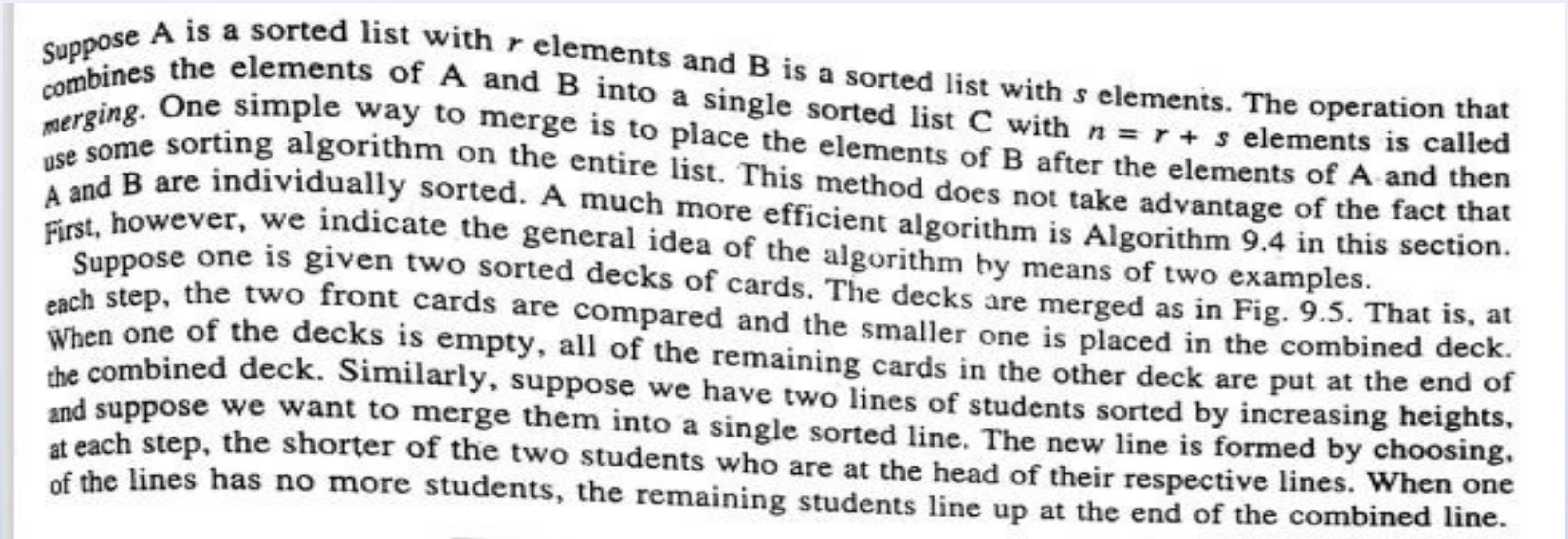
The above result is summarized in the following table:

Algorithm	*Worst Case*	*Average Case*
Selection Sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{2} = O(n^2)$

Remark: The number of interchanges and assignments does depend on the original order of the elements in the array A, but the sum of these operations does not exceed a factor of n^2.

Activate

Merging



Suppose A is a sorted list with r elements and B is a sorted list with s elements. The operation that combines the elements of A and B into a single sorted list C with $n = r + s$ elements is called *merging*. One simple way to merge is to place the elements of B after the elements of A and then use some sorting algorithm on the entire list. This method does not take advantage of the fact that A and B are individually sorted. A much more efficient algorithm is Algorithm 9.4 in this section. First, however, we indicate the general idea of the algorithm by means of two examples.

Suppose one is given two sorted decks of cards. The decks are merged as in Fig. 9.5. That is, at each step, the two front cards are compared and the smaller one is placed in the combined deck. When one of the decks is empty, all of the remaining cards in the other deck are put at the end of the combined deck. Similarly, suppose we have two lines of students sorted by increasing heights, and suppose we want to merge them into a single sorted line. The new line is formed by choosing, at each step, the shorter of the two students who are at the head of their respective lines. When one of the lines has no more students, the remaining students line up at the end of the combined line.

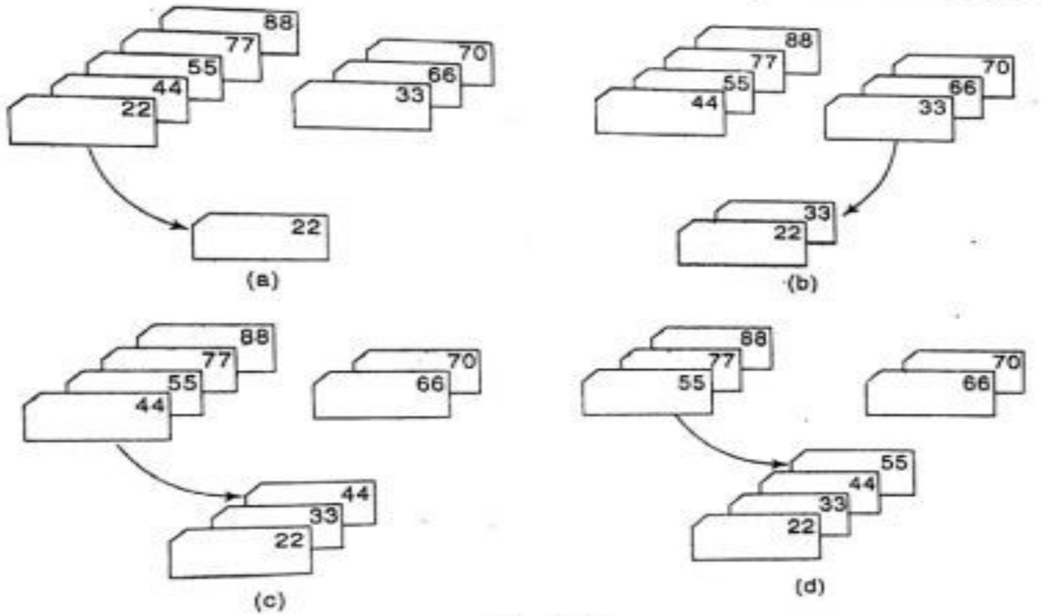
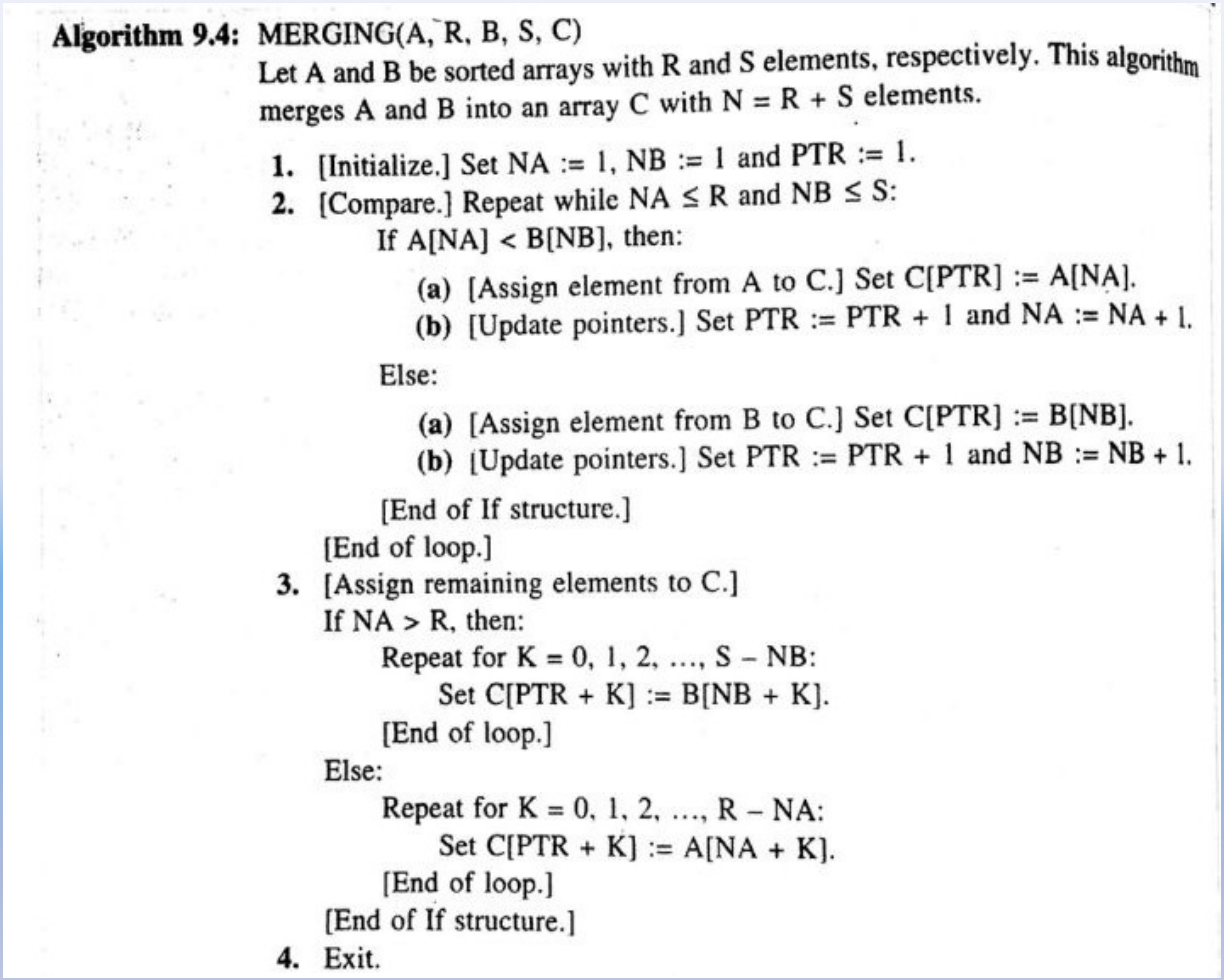


Fig. 9.5

Merging

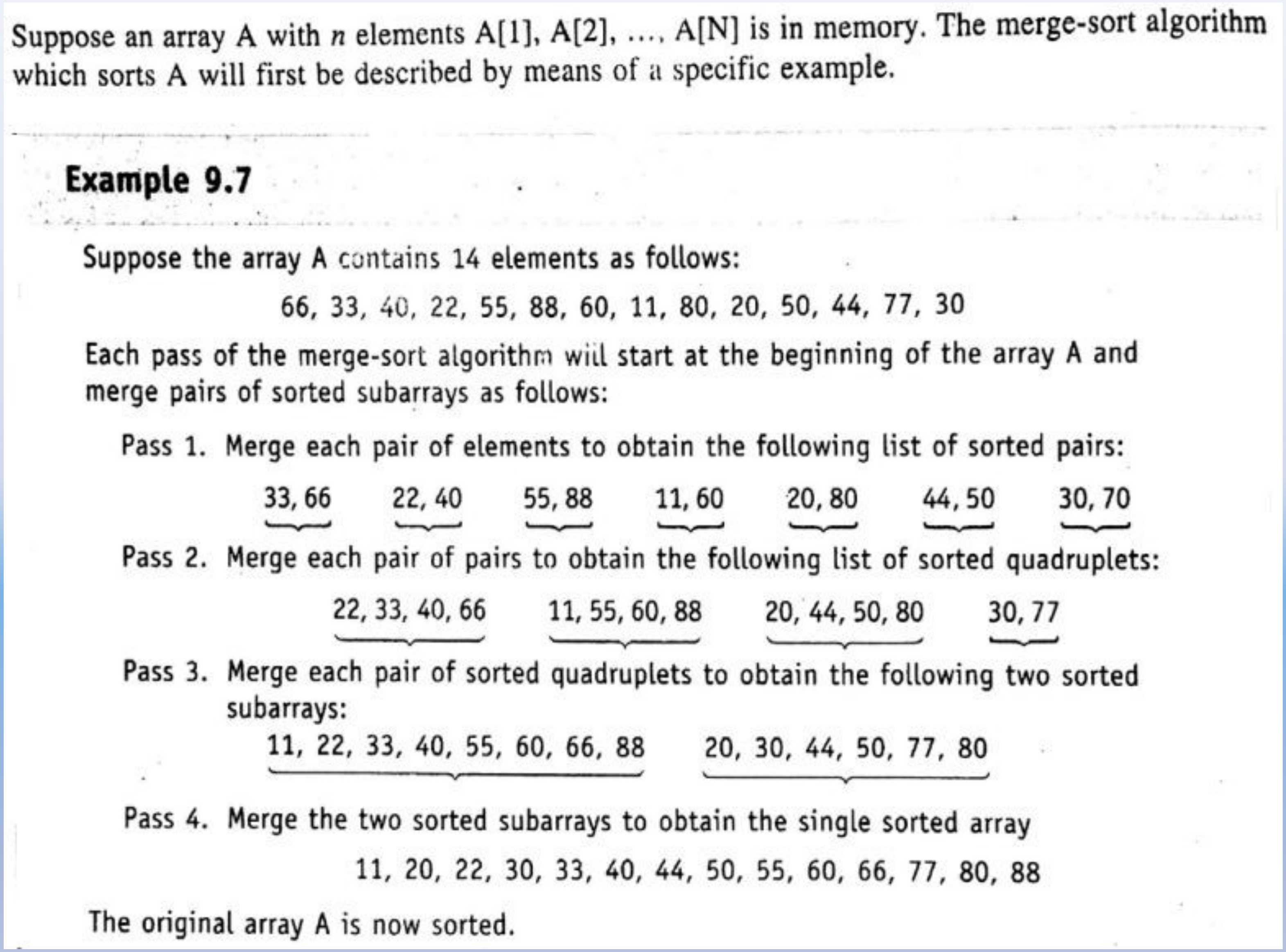
▪



Algorithm 9.4: MERGING(A, R, B, S, C)
Let A and B be sorted arrays with R and S elements, respectively. This algorithm merges A and B into an array C with N = R + S elements.

```
1. [Initialize.] Set NA := 1, NB := 1 and PTR := 1.
2. [Compare.] Repeat while NA ≤ R and NB ≤ S:
        If A[NA] < B[NB], then:
            (a) [Assign element from A to C.] Set C[PTR] := A[NA].
            (b) [Update pointers.] Set PTR := PTR + 1 and NA := NA + 1.
        Else:
            (a) [Assign element from B to C.] Set C[PTR] := B[NB].
            (b) [Update pointers.] Set PTR := PTR + 1 and NB := NB + 1.
        [End of If structure.]
   [End of loop.]
3. [Assign remaining elements to C.]
   If NA > R, then:
        Repeat for K = 0, 1, 2, ..., S – NB:
            Set C[PTR + K] := B[NB + K].
        [End of loop.]
   Else:
        Repeat for K = 0, 1, 2, ..., R – NA:
            Set C[PTR + K] := A[NA + K].
        [End of loop.]
   [End of If structure.]
4. Exit.
```

Merge Sort



Suppose an array A with *n* elements A[1], A[2], ..., A[N] is in memory. The merge-sort algorithm which sorts A will first be described by means of a specific example.

Example 9.7

Suppose the array A contains 14 elements as follows:

66, 33, 40, 22, 55, 88, 60, 11, 80, 20, 50, 44, 77, 30

Each pass of the merge-sort algorithm will start at the beginning of the array A and merge pairs of sorted subarrays as follows:

Pass 1. Merge each pair of elements to obtain the following list of sorted pairs:

$\underbrace{33, 66} \quad \underbrace{22, 40} \quad \underbrace{55, 88} \quad \underbrace{11, 60} \quad \underbrace{20, 80} \quad \underbrace{44, 50} \quad \underbrace{30, 70}$

Pass 2. Merge each pair of pairs to obtain the following list of sorted quadruplets:

$\underbrace{22, 33, 40, 66} \quad \underbrace{11, 55, 60, 88} \quad \underbrace{20, 44, 50, 80} \quad \underbrace{30, 77}$

Pass 3. Merge each pair of sorted quadruplets to obtain the following two sorted subarrays:

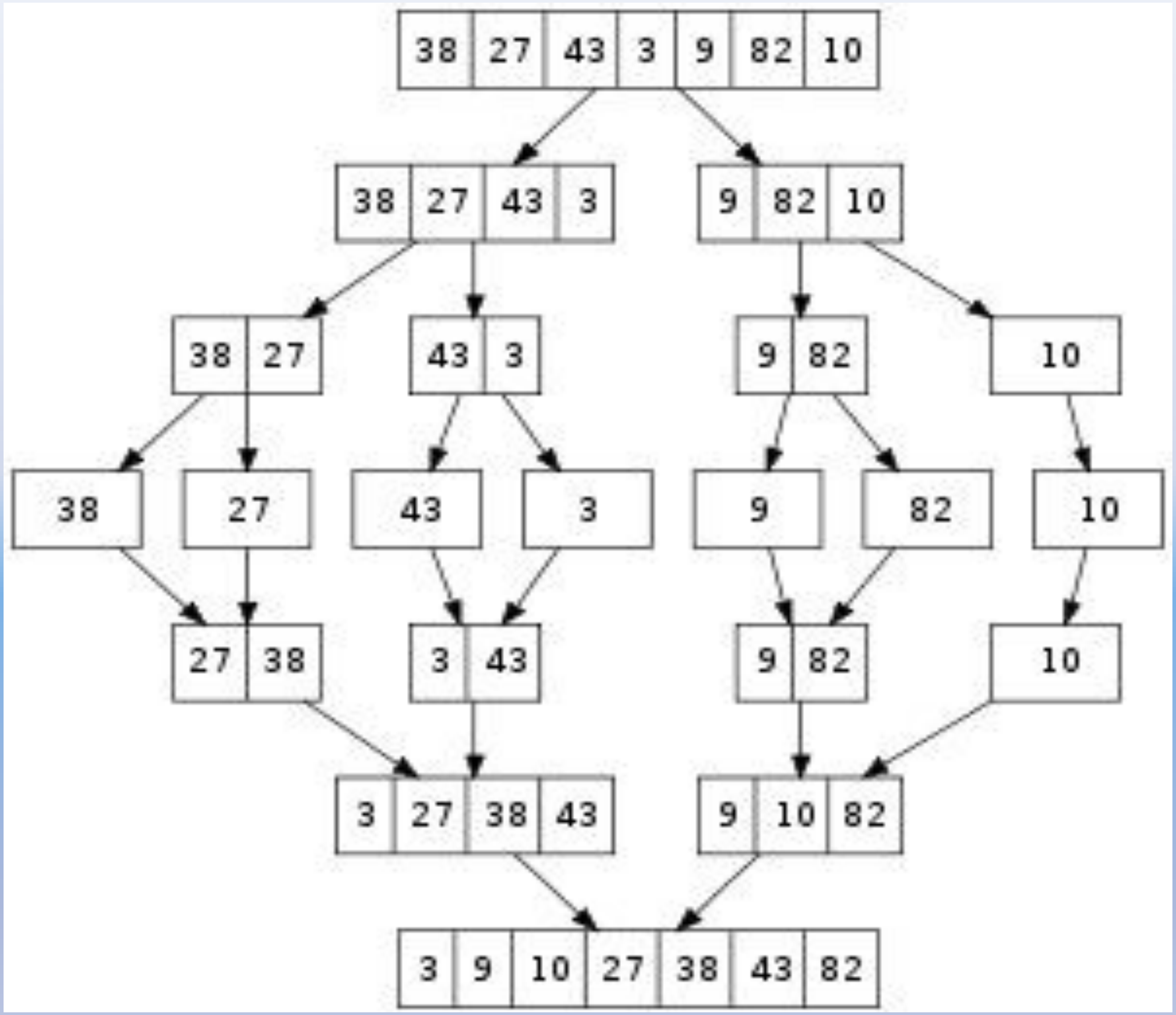
$\underbrace{11, 22, 33, 40, 55, 60, 66, 88} \quad \underbrace{20, 30, 44, 50, 77, 80}$

Pass 4. Merge the two sorted subarrays to obtain the single sorted array

11, 20, 22, 30, 33, 40, 44, 50, 55, 60, 66, 77, 80, 88

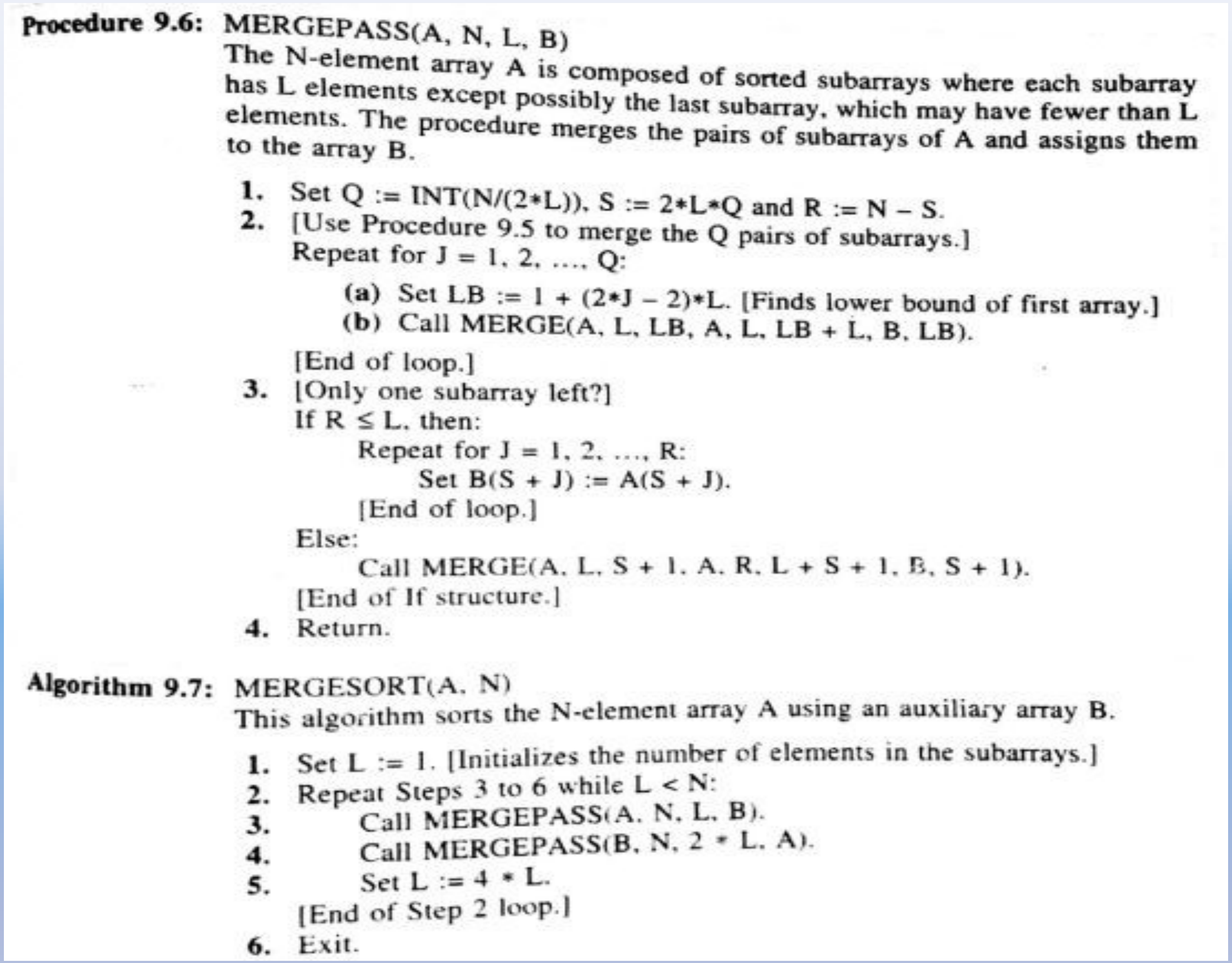
The original array A is now sorted.

Merge Sort



38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merge Sort



Procedure 9.6: MERGEPASS(A, N, L, B)

The N-element array A is composed of sorted subarrays where each subarray has L elements except possibly the last subarray, which may have fewer than L elements. The procedure merges the pairs of subarrays of A and assigns them to the array B.

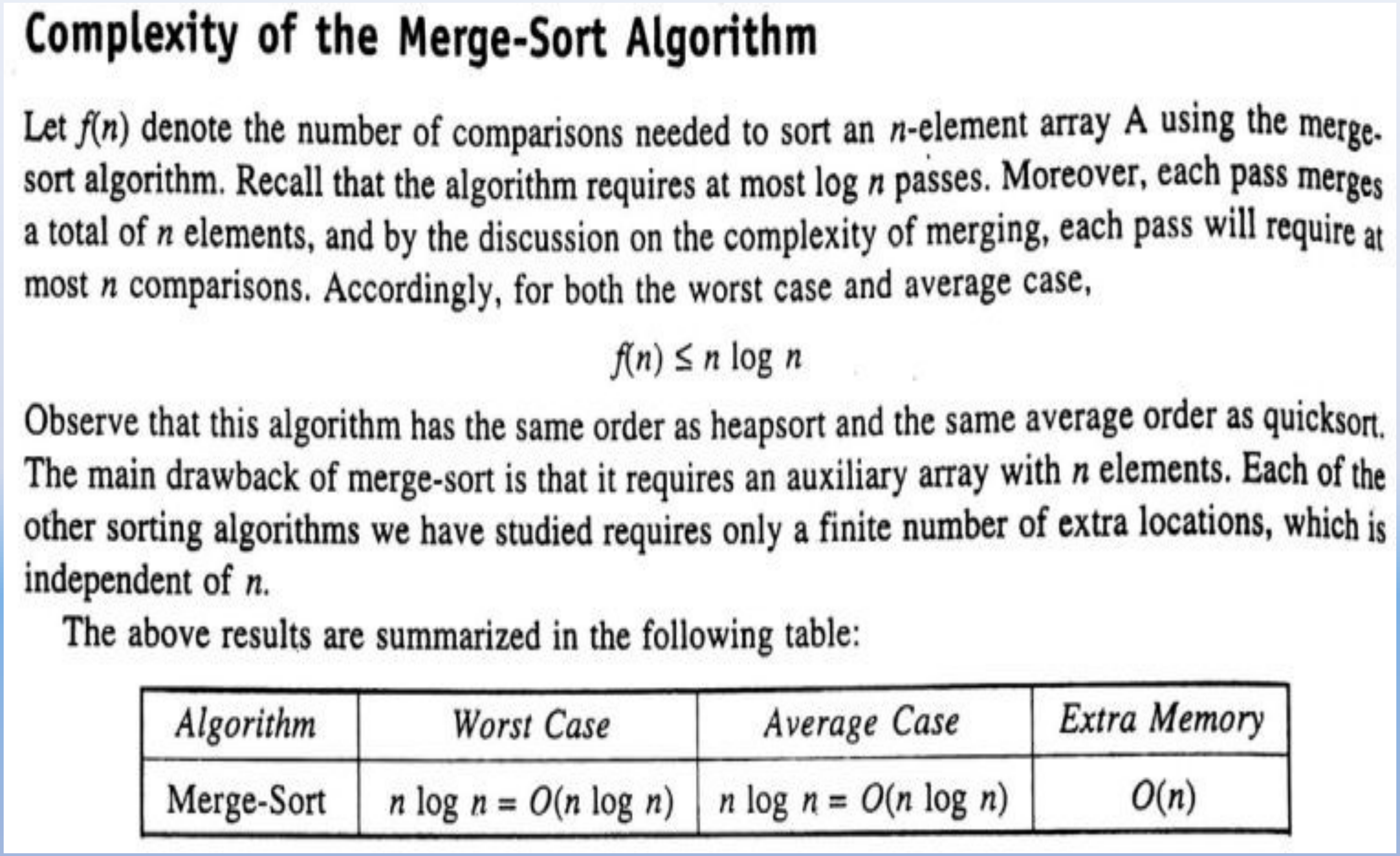
```
1.  Set Q := INT(N/(2*L)), S := 2*L*Q and R := N – S.
2.  [Use Procedure 9.5 to merge the Q pairs of subarrays.]
    Repeat for J = 1, 2, ..., Q:
        (a) Set LB := 1 + (2*J – 2)*L. [Finds lower bound of first array.]
        (b) Call MERGE(A, L, LB, A, L, LB + L, B, LB).
    [End of loop.]
3.  [Only one subarray left?]
    If R ≤ L, then:
        Repeat for J = 1, 2, ..., R:
            Set B(S + J) := A(S + J).
        [End of loop.]
    Else:
        Call MERGE(A, L, S + 1, A, R, L + S + 1, B, S + 1).
    [End of If structure.]
4.  Return.
```

Algorithm 9.7: MERGESORT(A, N)

This algorithm sorts the N-element array A using an auxiliary array B.

```
1.  Set L := 1. [Initializes the number of elements in the subarrays.]
2.  Repeat Steps 3 to 6 while L < N:
3.      Call MERGEPASS(A, N, L, B).
4.      Call MERGEPASS(B, N, 2 * L, A).
5.      Set L := 4 * L.
    [End of Step 2 loop.]
6.  Exit.
```

Merge Sort



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	$n \log n = O(n \log n)$	$n \log n = O(n \log n)$	$O(n)$

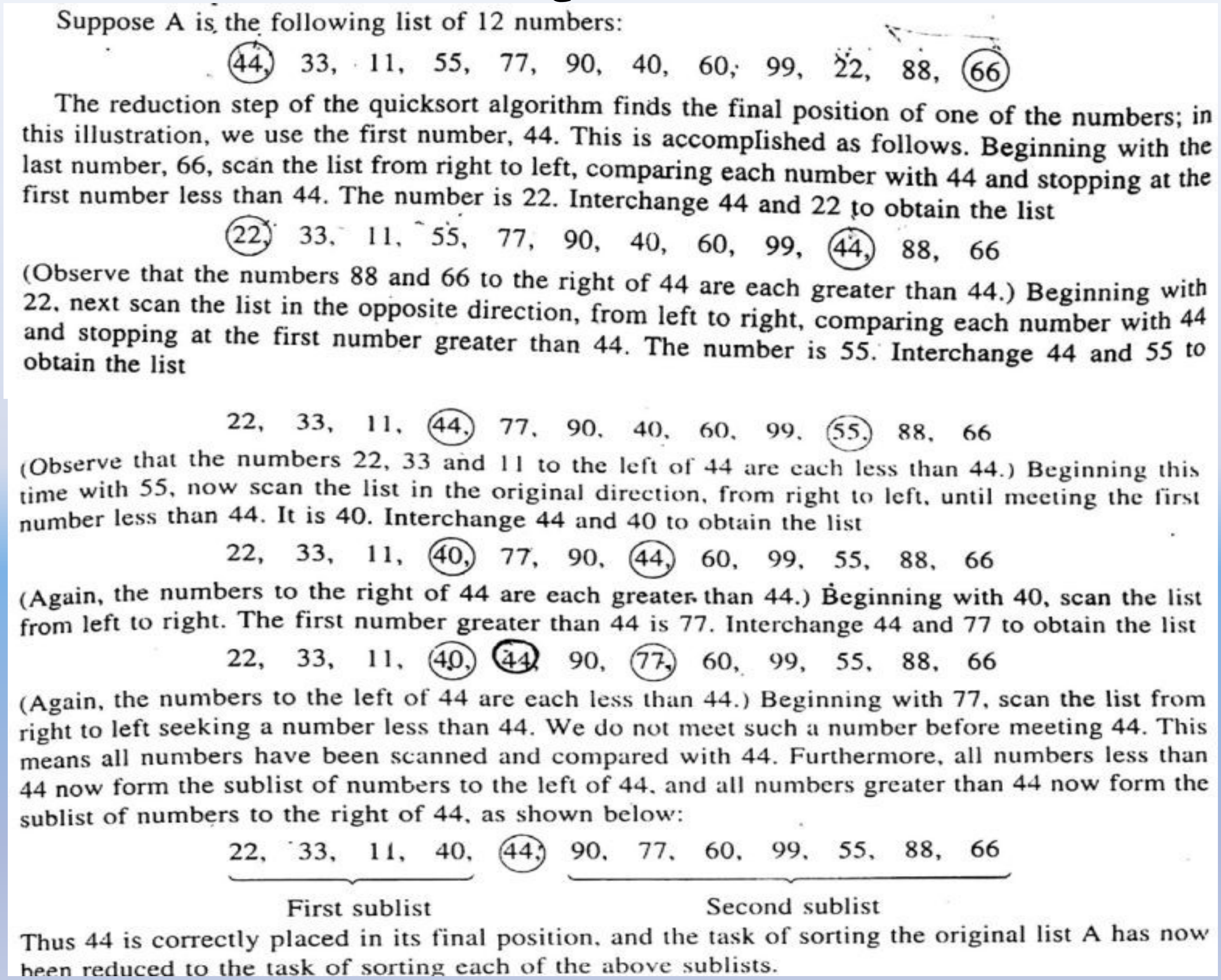
Quicksort

- Quick sort is a divide and conquer algorithm.
- Quick sort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quick sort can then recursively sort the sub-lists.
- The steps are:

1. Pick an element, called a pivot, from the list.
2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

The base case of the recursion are lists of size zero or one, which never need to be sorted.

Quick Sort



Suppose A is the following list of 12 numbers:

44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

The reduction step of the quicksort algorithm finds the final position of one of the numbers; in this illustration, we use the first number, 44. This is accomplished as follows. Beginning with the last number, 66, scan the list from right to left, comparing each number with 44 and stopping at the first number less than 44. The number is 22. Interchange 44 and 22 to obtain the list

22, 33, 11, 55, 77, 90, 40, 60, 99, 44, 88, 66

(Observe that the numbers 88 and 66 to the right of 44 are each greater than 44.) Beginning with 22, next scan the list in the opposite direction, from left to right, comparing each number with 44 and stopping at the first number greater than 44. The number is 55. Interchange 44 and 55 to obtain the list

22, 33, 11, 44, 77, 90, 40, 60, 99, 55, 88, 66

(Observe that the numbers 22, 33 and 11 to the left of 44 are each less than 44.) Beginning this time with 55, now scan the list in the original direction, from right to left, until meeting the first number less than 44. It is 40. Interchange 44 and 40 to obtain the list

22, 33, 11, 40, 77, 90, 44, 60, 99, 55, 88, 66

(Again, the numbers to the right of 44 are each greater than 44.) Beginning with 40, scan the list from left to right. The first number greater than 44 is 77. Interchange 44 and 77 to obtain the list

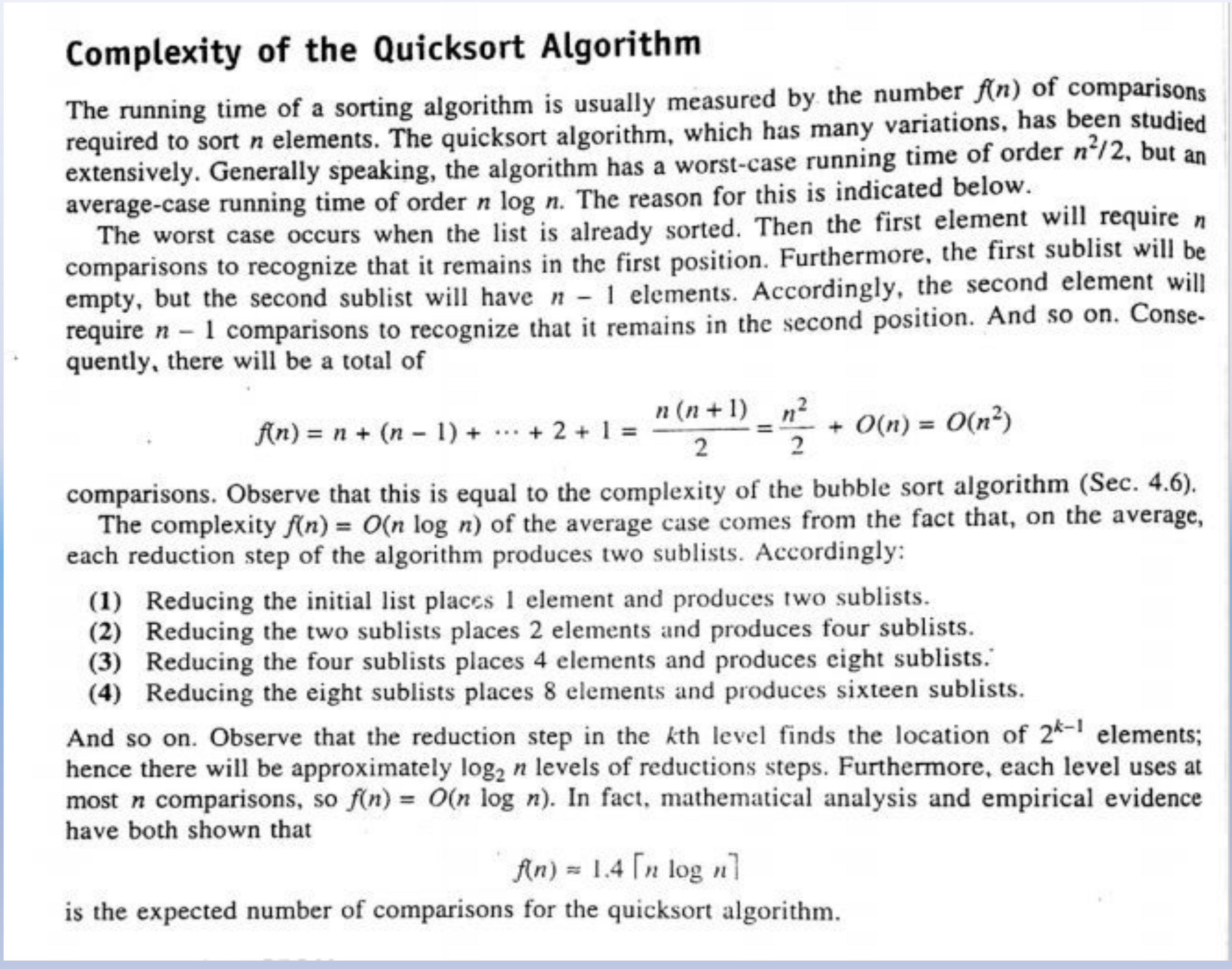
22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 88, 66

(Again, the numbers to the left of 44 are each less than 44.) Beginning with 77, scan the list from right to left seeking a number less than 44. We do not meet such a number before meeting 44. This means all numbers have been scanned and compared with 44. Furthermore, all numbers less than 44 now form the sublist of numbers to the left of 44, and all numbers greater than 44 now form the sublist of numbers to the right of 44, as shown below:

$$\underbrace{22,\ 33,\ 11,\ 40,}_{\text{First sublist}}\ 44,\ \underbrace{90,\ 77,\ 60,\ 99,\ 55,\ 88,\ 66}_{\text{Second sublist}}$$

Thus 44 is correctly placed in its final position, and the task of sorting the original list A has now been reduced to the task of sorting each of the above sublists.

Quick Sort



Complexity of the Quicksort Algorithm

The running time of a sorting algorithm is usually measured by the number $f(n)$ of comparisons required to sort n elements. The quicksort algorithm, which has many variations, has been studied extensively. Generally speaking, the algorithm has a worst-case running time of order $n^2/2$, but an average-case running time of order $n \log n$. The reason for this is indicated below.

The worst case occurs when the list is already sorted. Then the first element will require n comparisons to recognize that it remains in the first position. Furthermore, the first sublist will be empty, but the second sublist will have $n - 1$ elements. Accordingly, the second element will require $n - 1$ comparisons to recognize that it remains in the second position. And so on. Consequently, there will be a total of

$$f(n) = n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2} = \frac{n^2}{2} + O(n) = O(n^2)$$

comparisons. Observe that this is equal to the complexity of the bubble sort algorithm (Sec. 4.6).

The complexity $f(n) = O(n \log n)$ of the average case comes from the fact that, on the average, each reduction step of the algorithm produces two sublists. Accordingly:

(1) Reducing the initial list places 1 element and produces two sublists.
(2) Reducing the two sublists places 2 elements and produces four sublists.
(3) Reducing the four sublists places 4 elements and produces eight sublists.
(4) Reducing the eight sublists places 8 elements and produces sixteen sublists.

And so on. Observe that the reduction step in the kth level finds the location of 2^{k-1} elements; hence there will be approximately $\log_2 n$ levels of reductions steps. Furthermore, each level uses at most n comparisons, so $f(n) = O(n \log n)$. In fact, mathematical analysis and empirical evidence have both shown that

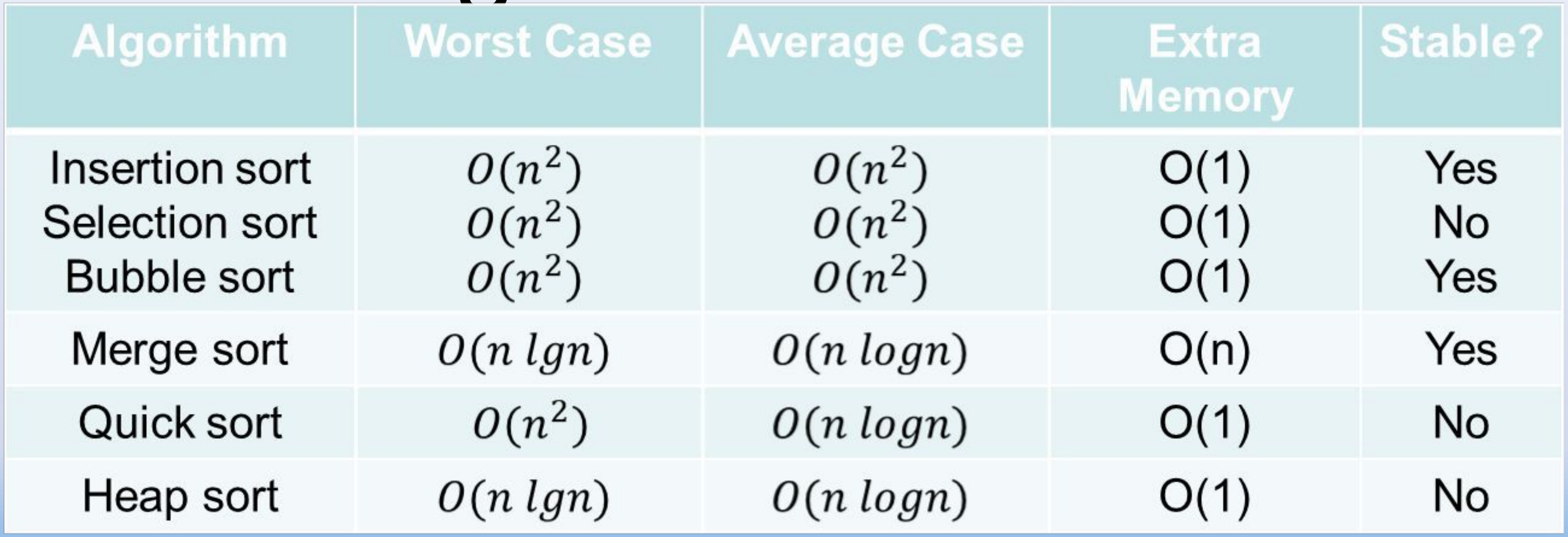
$$f(n) \approx 1.4 \lceil n \log n \rceil$$

is the expected number of comparisons for the quicksort algorithm.

Quick Sort

```
// C qsort and C++ sort() algorithm
#include <bits/stdc++.h>
using namespace std;
#define N 100
// A comparator function used by qsort
int compare(const void * a, const void * b)
{
        return ( *(int*)a - *(int*)b );
}
int main()
{
        int arr[N+1], Arr2[N+1];
        srand(time(NULL));
        // generate random input
        for (int i = 0; i < N; i++)
                Arr2[i] = arr[i] = rand()%100000;
        qsort(arr, N, sizeof(int), compare);
        for (int i = 0; i < N; i++)
                cout<< arr[i] << " ";
        sort(Arr2, Arr2 + N);
        cout<<endl;
        for (int i = 0; i < N; i++)
                cout<< Arr2[i] << " ";
        return 0;
}
```

Sorting Algorithms



Algorithm	Worst Case	Average Case	Extra Memory	Stable?
Insertion sort	$O(n^2)$	$O(n^2)$	O(1)	Yes
Selection sort	$O(n^2)$	$O(n^2)$	O(1)	No
Bubble sort	$O(n^2)$	$O(n^2)$	O(1)	Yes
Merge sort	$O(n\ lgn)$	$O(n\ logn)$	O(n)	Yes
Quick sort	$O(n^2)$	$O(n\ logn)$	O(1)	No
Heap sort	$O(n\ lgn)$	$O(n\ logn)$	O(1)	No

- **A sort algorithm is said to be an in-place sort, If it requires only a constant amount (i.e. O(1)) of extra space during the sorting process.**

- **A sorting algorithm is stable if the relative order of elements with the same key value is preserved by the algorithm.**

Searching

- Sequential search and binary search are two algorithms for searching for an element in the array.
- Sequential search works by comparing each element in the array to the target value, starting at the beginning of the array.
- Binary search works by dividing the array in half and then searching the half that is more likely to contain the target value.
- Sequential search is simpler to implement than binary search, but it is less efficient.
- Binary search is more efficient than sequential search, but it is more complex to implement.

Sequential Search Analysis

Sequential search algorithm performance
- Examine worst case and average case
-Count number of key comparisons

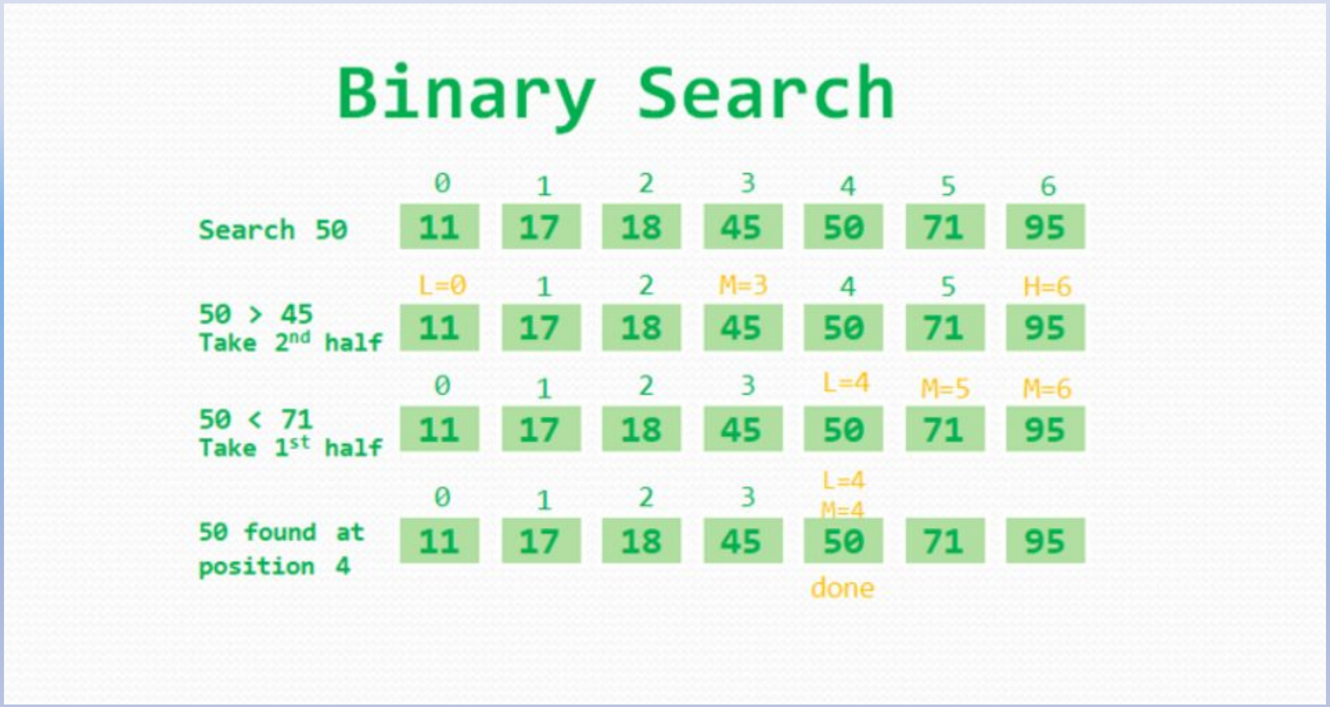
Unsuccessful search
-Search item not in list
-Make n comparisons

Conducting algorithm performance analysis
-Best case: make one key comparison
-Worst case: algorithm makes n comparisons

Determining the average number of comparisons
-Consider all possible cases
-Find number of comparisons for each case
-Add number of comparisons, divide by number of cases

Binary Search

- Binary Search Performed only on ordered lists
- Uses divide-and-conquer technique



What is Hashing?

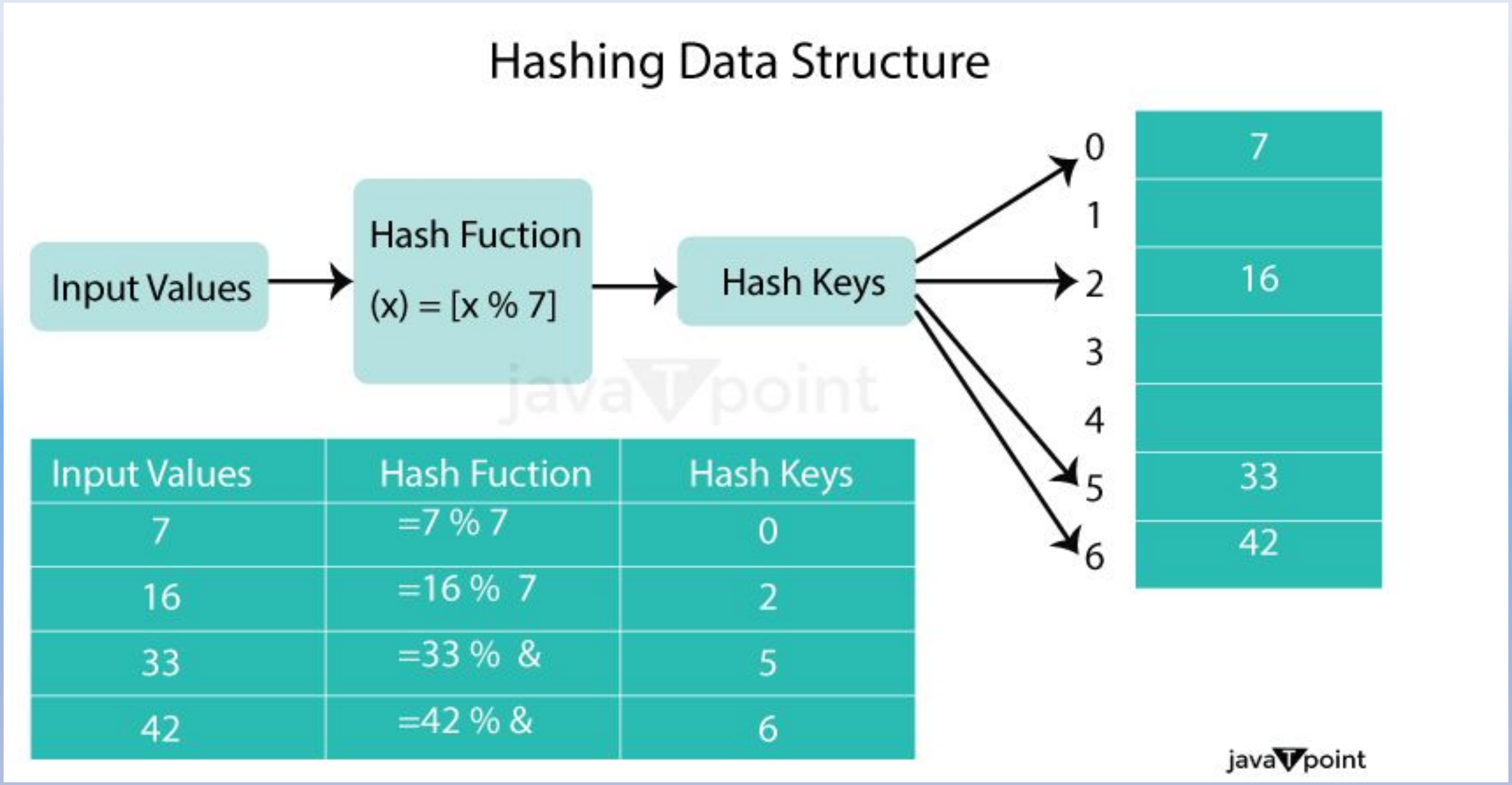
- A hashing algorithm is used to convert an input (such as a string or integer) into a fixed-size output (referred to as a hash code or hash value). The data is then stored and retrieved using this hash value as an index in an array or hash table.
- The hash function must be deterministic, which guarantees that it will always yield the same result for a given input.
- Hashing is commonly used to create a unique identifier for a piece of data, which can be used to quickly look up that data in a large dataset.
- For example, a web browser may use hashing to store website passwords securely. When a user enters their password, the browser converts it into a hash value and compares it to the stored hash value to authenticate the user.

How Hashing Works?

The process of hashing can be broken down into three steps:

- **Input:** The data to be hashed is input into the hashing algorithm.
- **Hash Function:** The hashing algorithm takes the input data and applies a mathematical function to generate a fixed-size hash value. The hash function should be designed so that different input values produce different hash values, and small changes in the input produce large changes in the output.
- **Output:** The hash value is returned, which is used as an index to store or retrieve data in a data structure.

Hashing process



Input Values	Hash Fuction	Hash Keys
7	=7 % 7	0
16	=16 % 7	2
33	=33 % &	5
42	=42 % &	6

Hash Function

Hash Function: A hash function is a type of mathematical operation that takes an input (or key) and outputs a fixed-size result known as a hash code or hash value. The hash function must always yield the same hash code for the same input in order to be deterministic. Additionally, the hash function should produce a unique hash code for each input, which is known as the hash property.

There are different types of hash functions, including:

1. Division method:

This method involves dividing the key by the table size and taking the remainder as the hash value. For example, if the table size is 10 and the key is 23, the hash value would be 3 (23 % 10 = 3).

2. Multiplication method:

This method involves multiplying the key by a constant and taking the fractional part of the product as the hash value. For example, if the key is 10 and the constant is 0.618, the hash value would be 2 (floor(10*(0.61823 - floor(0.61823))) = floor(2.236) = 2).

3. Universal hashing:

This method involves using a random hash function from a family of hash functions. This ensures that the hash function is not biased towards any particular input and is resistant to attacks.

Collision Resolution

One of the main challenges in hashing is handling collisions, which occur when two or more input values produce the same hash value. There are various techniques used to resolve collisions, including:

- **Chaining:** In this technique, each hash table slot contains a linked list of all the values that have the same hash value. This technique is simple and easy to implement, but it can lead to poor performance when the linked lists become too long.
- **Open addressing:** In this technique, when a collision occurs, the algorithm searches for an empty slot in the hash table by probing successive slots until an empty slot is found. This technique can be more efficient than chaining when the load factor is low, but it can lead to clustering and poor performance when the load factor is high.
- **Double hashing:** This is a variation of open addressing that uses a second hash function to determine the next slot to probe when a collision occurs. This technique can help to reduce clustering and improve performance.

Thank you !

ANY QUESTION?