

Sorting

Sorting refers to the operation of arranging data in some order such as increasing or decreasing with **numerical data or alphabetically** with character data

Complexity of Sorting Algorithm

2

Each sorting algorithm **S** will be made up of the following operations, where A_1, A_2, \dots, A_n contain the items to be sorted and B is an auxiliary location:

- (a) **Comparisons**, which test whether $A_i < A_j$ or test $A_i < B$
- (b) **Interchange**, which switch the content of A_i and A_j or of A_i and B
- (c) **Assignments**, which set $B := A_i$ and then set $A_j := B$ or Set $A_j := A_i$

Sorting

Algorithms are divided into two categories:

Internal Sorts

and

External sorts.

Sorting

Internal Sort:

Any sort algorithm which uses main memory exclusively during the sort.

This assumes high-speed random access to all memory.

Sorting

External Sort:

Any sort algorithm which uses external memory, such as tape or disk, during the sort.

Internal Sorting

Bubble Sort

The oldest and simplest sort in use.

Unfortunately, also the slowest.

Works by comparing each item in the list with the item next to it, and swapping them if required.

This causes larger values to "bubble" to the end of the list

Bubble Sort

Suppose the list of number $A[1], [2], A[3], \dots, A[N]$ is in memory.
Algorithm works as follows.

[1] Compare $A[1]$ and $A[2]$, arrange them in the desired order so that $A[1] < A[2]$. Then Compare $A[2]$ and $A[3]$, arrange them in the desired order so that $A[2] < A[3]$. Continue until $A[N-1]$ is compared with $A[N]$, arrange them so that $A[N-1] < A[N]$.

Bubble Sort

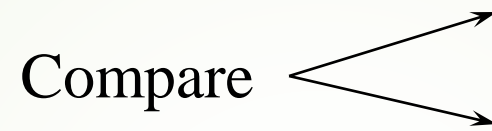
- 2 Repeat Step 1, Now stop after comparing and re-arranging $A[N-2]$ and $A[N-1]$.
- 3 Repeat Step 3, Now stop after comparing and re-arranging $A[N-3]$ and $A[N-2]$.
- .
- .
- [N-1] Compare $A[1]$ and $A[2]$ and arrange them in sorted order so that $A[1] < A[2]$.

After N-1 steps the list will be sorted in increasing order.

A Bubble Sort Example

9

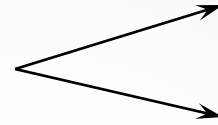
Compare



6
5
4
3
2
1

A Bubble Sort Example

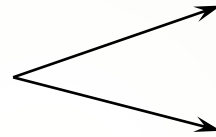
Swap



5
6
4
3
2
1

A Bubble Sort Example

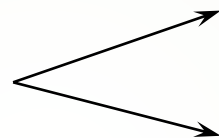
Compare



5
6
4
3
2
1

A Bubble Sort Example

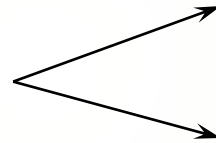
Swap



5
4
6
3
2
1

A Bubble Sort Example

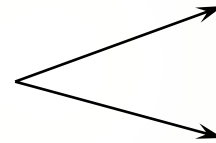
Compare



5
4
6
3
2
1

A Bubble Sort Example

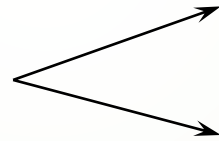
Swap



5
4
3
6
2
1

A Bubble Sort Example

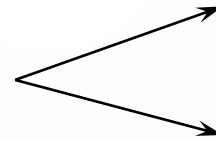
Compare



5
4
3
6
2
1

A Bubble Sort Example

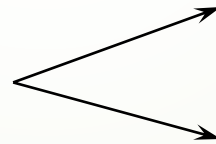
Swap



5
4
3
2
6
1

A Bubble Sort Example

Compare

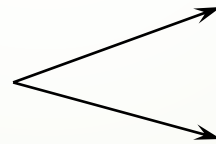


5
4
3
2
6
1

A Bubble Sort Example

As you can see, the largest number has “bubbled” down, or sunk to the bottom of the List after the first pass through the List.

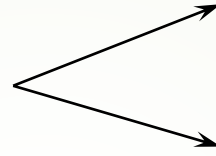
Swap



5
4
3
2
1
6

A Bubble Sort Example

Compare

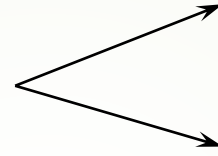


For our second pass through the List, we start by comparing these first two elements in the List.

5
4
3
2
1
6

A Bubble Sort Example

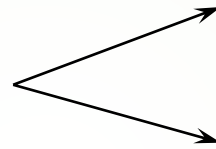
Swap



4
5
3
2
1
6

A Bubble Sort Example

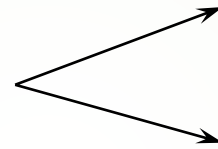
Compare



4
5
3
2
1
6

A Bubble Sort Example

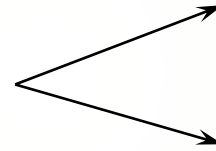
Swap



4
3
5
2
1
6

A Bubble Sort Example

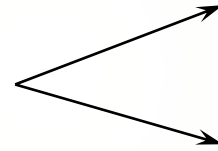
Compare



4
3
5
2
1
6

A Bubble Sort Example

Swap

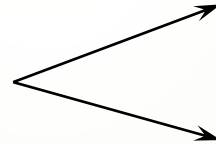


4
3
2
5
1
6

A Bubble Sort Example

25

Compare



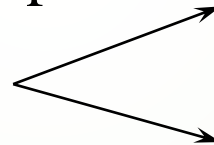
4
3
2
5
1
6

A Bubble Sort Example

26

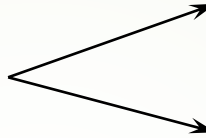
At the end of the second pass, we stop at element number $n - 1$, because the largest element in the List is already in the last position. This places the second largest element in the second to last spot.

Swap



4
3
2
1
5
6

A Bubble Sort Example

Compare 

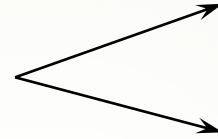
We start with the first two elements again at the beginning of the third pass.

4
3
2
1
5
6

A Bubble Sort Example

28

Swap

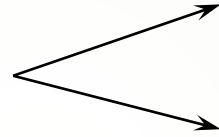


3
4
2
1
5
6

A Bubble Sort Example

29

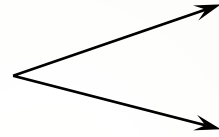
Compare



3
4
2
1
5
6

A Bubble Sort Example

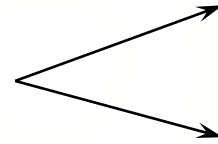
Swap



3
2
4
1
5
6

A Bubble Sort Example

Compare

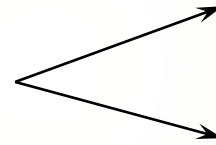


3
2
4
1
5
6

A Bubble Sort Example

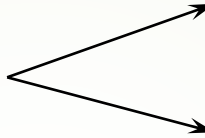
At the end of the third pass, we stop comparing and swapping at element number $n - 2$.

Swap



3
2
1
4
5
6

A Bubble Sort Example

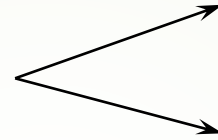
Compare 

The beginning of the fourth pass...

3
2
1
4
5
6

A Bubble Sort Example

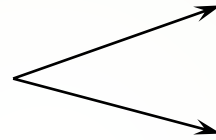
Swap



2
3
1
4
5
6

A Bubble Sort Example

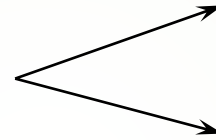
Compare



2
3
1
4
5
6

A Bubble Sort Example

Swap

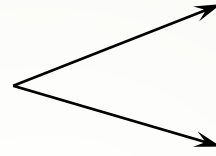


The end of the fourth pass
stops at element number $n - 3$.

2
1
3
4
5
6

A Bubble Sort Example

Compare

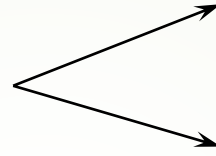


The beginning of the fifth pass...

2
1
3
4
5
6

A Bubble Sort Example

Swap

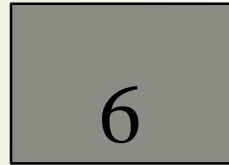


The last pass compares only the first two elements of the List. After this comparison and possible swap, the smallest element has “bubbled” to the top.

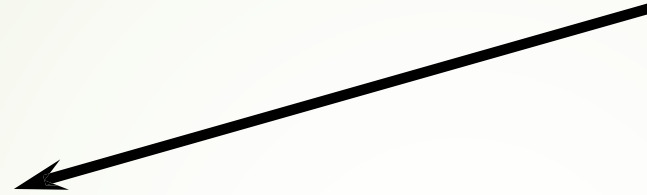
1
2
3
4
5
6

What “Swapping” Means

TEMP



Place the first element into the
Temporary Variable.



6
5
4
3
2
1

What “Swapping” Means

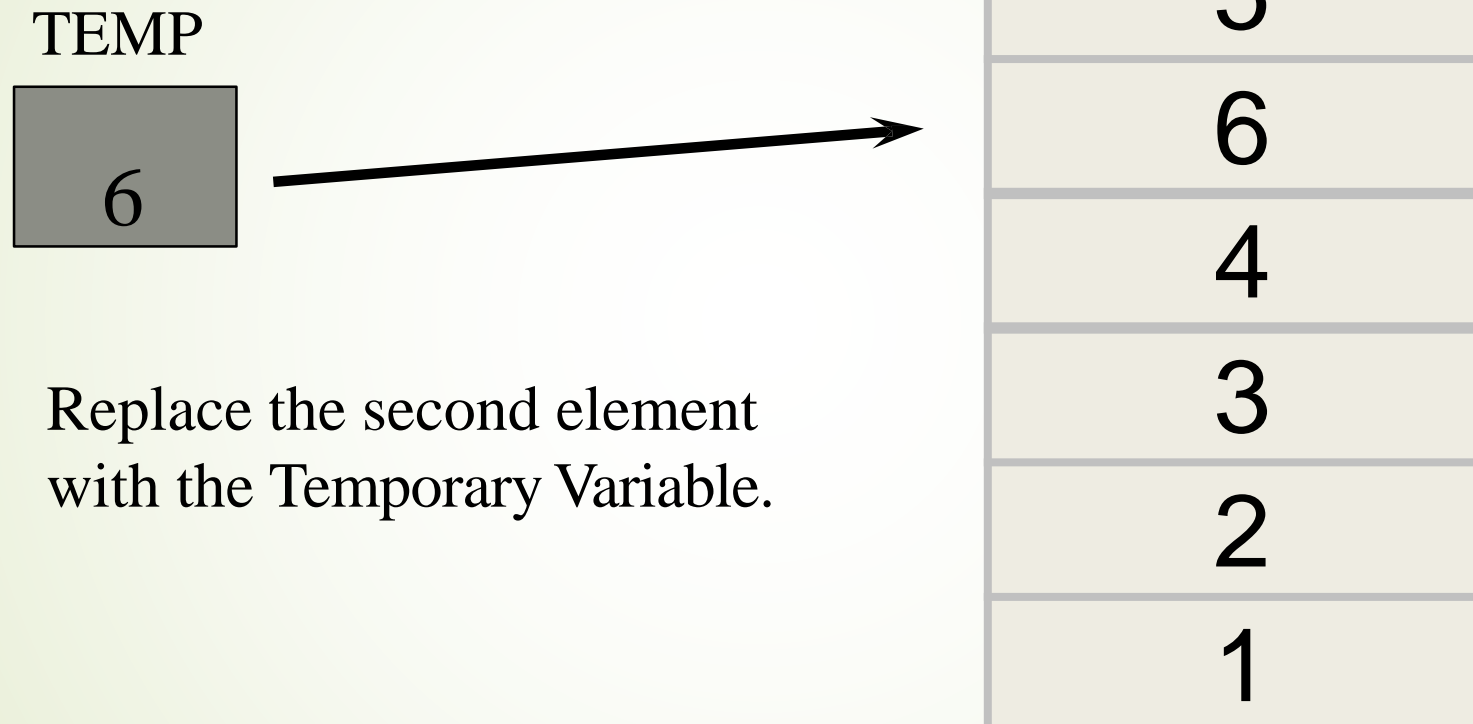
TEMP

6

Replace the first element with
the second element.

5
5
4
3
2
1

What “Swapping” Means



Bubble Sort

DATA is an array with N elements

- 1 Repeat Step 2 and 3 for $K = 1$ to $N-1$
- 2 Set $PTR := 1$
- 3 Repeat While $PTR \leq N - K$
 - (a) If $DATA[PTR] > DATA[PTR+1]$
Interchange $DATA[PTR]$ and $DATA[PTR + 1]$
 - (b) Set $PTR = PTR + 1$
- 1 Exit

Analysis of Bubble Sort

Best, Worst and Average cases are same:

$$\begin{aligned} f(n) &= (n-1) + (n-2) + \dots + 2+1 = n(n-1)/2 \\ &= O(n^2) \end{aligned}$$

Variation of Bubble Sort

Bubble sort with early termination

The algorithm repeats the process until it makes a pass all the way through the list without swapping any items.

Alternate passes in opposite direction

A pass moves the largest element in the sub-array to the end of sub-array. This is same as a pass in standard bubble sort.

Next pass moves the smallest element in sub-array to the beginning of sub-array.

Largest and smallest values are settled in alternate passes.

This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list.

Insertion Sort

Insertion sort is frequently used by bridge players when they first sort their cards.

Insertion sort is frequently used when n is small.

General Idea:

An array A with N elements $A[1], A[2] \dots A[N]$ is in memory

Insertion Sort scan A from $A[1]$ to $A[N]$ inserting each elements $A[k]$ into its proper position in the previously sorted subarray $A[1], A[2], \dots A[k-1]$

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

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

Pass 3: $A[3]$ is inserted in 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 $A[1]$, $A[2]$, $A[3]$ is sorted.

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

▪

▪

Pass N : $A[N]$ is inserted in its proper place in $A[1], A[2], A[3], \dots A[N-1]$ so that $A[1], A[2], A[3], A[4], \dots A[N]$ is sorted.

Insertion Sort Example

Sort an array A with 8 elements

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

Insertion Sort

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K=1	$-\infty$	77	33	44	11	88	22	66	55
K=2	$-\infty$	77	33	44	11	88	22	66	55
K=3	$-\infty$	33	77	44	11	88	22	66	55
K=4	$-\infty$	33	44	77	11	88	22	66	55
K=5	$-\infty$	11	33	44	77	88	22	66	55
K=6	$-\infty$	11	33	44	77	88	22	66	55
K=7	$-\infty$	11	22	33	44	77	88	66	55
K=8	$-\infty$	11	22	33	44	66	77	88	55
Sorted	$-\infty$	11	22	33	44	55	66	77	88

Insertion Algorithm

This algorithm sort an array with N elements

- 1 Set $A[0] = -\infty$ [Initialize a delimiter]
- 2 Repeat Steps 3 to 5 for $K = 2, 3, \dots, N$
- 3 Set $TEMP = A[K]$ and $PTR = K-1$
- 4 Repeat while $TEMP < A[PTR]$
 - (a) Set $A[PTR+1] = A[PTR]$
 - (b) Set $PTR = PTR - 1$
- 5 Set $A[PTR+1] = TEMP$
- 6 Exit

Complexity of Insertion Sort

Best case: When the array is completely sorted- $4n-4 = O(n)$

Worst Case: When the array is in reverse order- $n(n-1)/2 = O(n^2)$

Average Case: $n(n-1)/4 = O(n^2)$

Improvement in Performance:

Use binary search in inner loop to find location where $A[k]$ is to be inserted: $\log_2 k$ comparisons
but $(k-1)/2$ movements still required so order of complexity remain same

Selection Sort

52

Suppose an array A with N elements is in memory. Selection sort 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.

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

Pass 2: Find the location LOC of the smallest element in the sublist $A[2]$, $A[3]$, ... $A[N]$. Then interchange $A[LOC]$ and $A[2]$. Then: $A[1], A[2]$ is sorted since $A[1] \leq A[2]$.

Pass 3: Find the location LOC of the smallest element in the sublist $A[3]$, $A[4]$, ... $A[N]$. Then interchange $A[LOC]$ and $A[3]$. Then: $A[1], A[2], A[3]$ is sorted, since $A[2] \leq A[3]$.

Pass N-1: Find the location LOC of the smallest element in the sublist $A[N-1], A[N]$. Then interchange $A[LOC]$ and $A[N-1]$.
Then: $A[1], A[2], \dots, A[N]$ is sorted, since $A[N-1] \leq A[N]$.

A is sorted after N-1 pass.

Selection Sort

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

(Selection Sort) SELECTION(A, N)

This algorithm sorts the array A with N elements.

1. Repeat Steps 2 and 3 for $K = 1, 2, \dots, N - 1$:
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.

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 $J = K + 1, K + 2, \dots, N$:
 If $MIN > A[J]$, then: Set $MIN := A[J]$ and $LOC := J$.
 [End of loop.]
3. Return.

Complexity

57

Best, worst and average case complexities are same.

$$\begin{aligned} f(n) &= (n-1) + (n-2) + \dots + 2 + 1 = n(n-1)/2 \\ &= O(n^2) \end{aligned}$$



Merge Sort

- **Merge sort:**

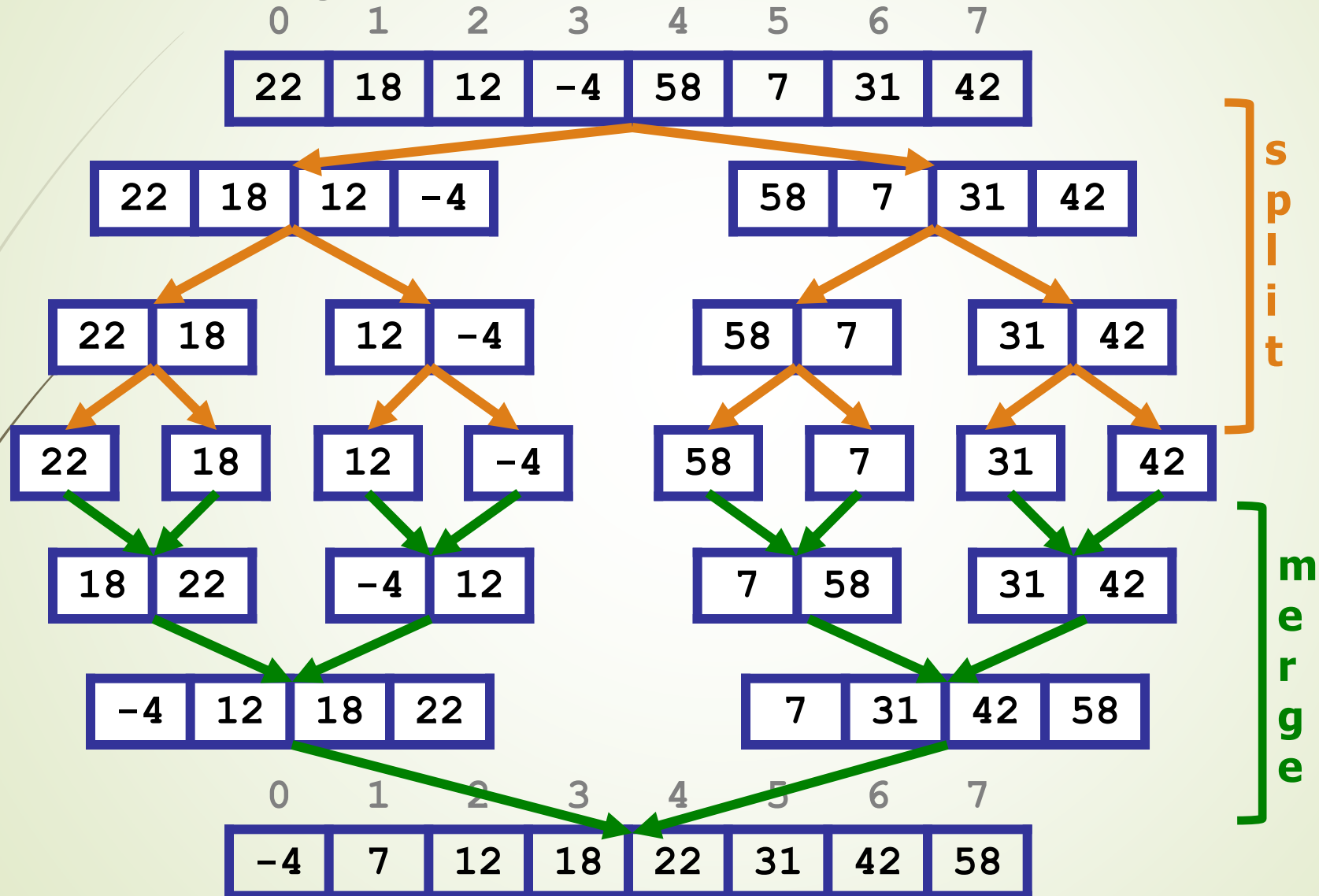
- divide a list into two halves
- sort the halves
- recombine the sorted halves into a sorted whole

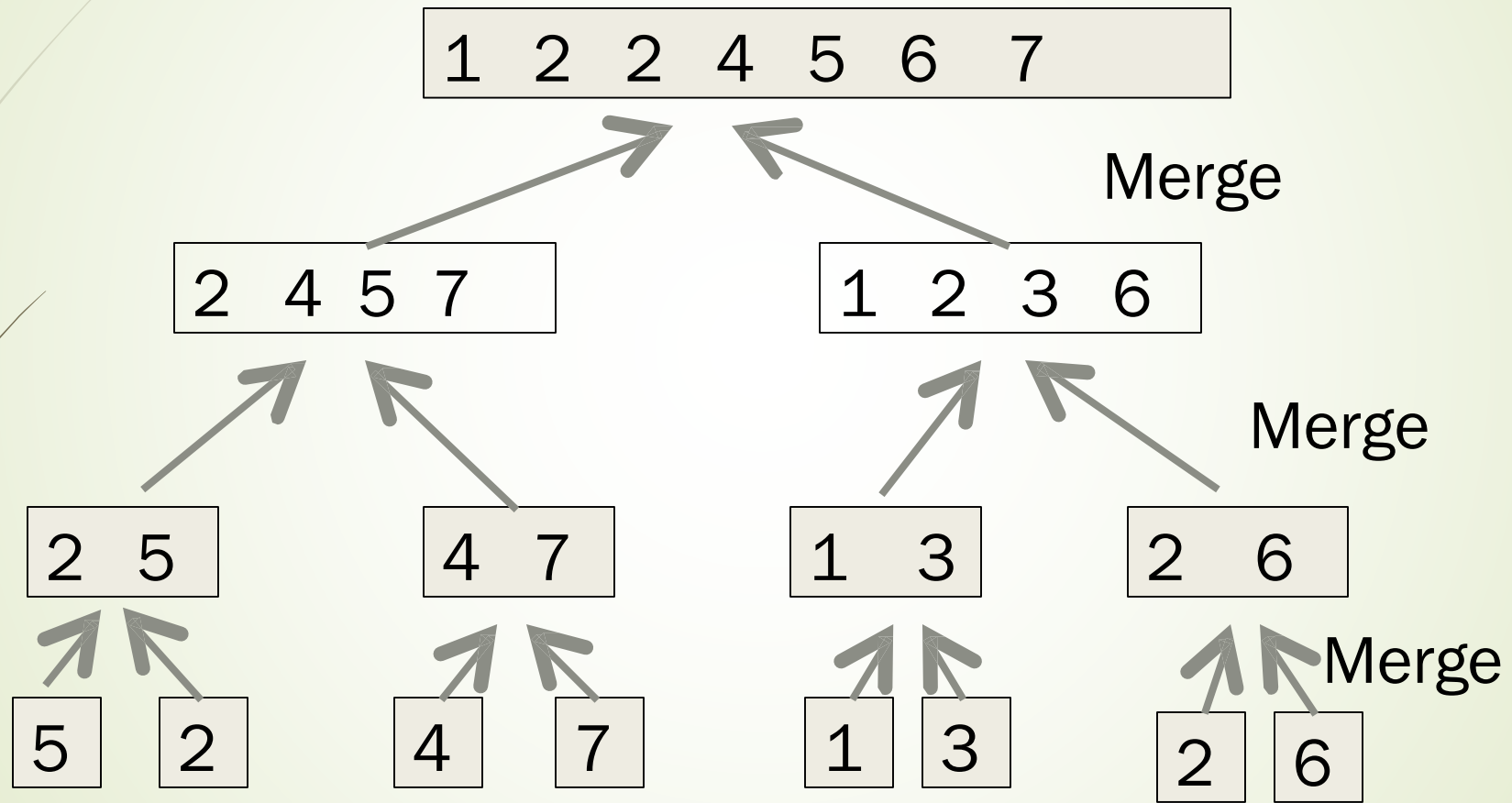
- Merge sort is an example of a “divide and conquer” algorithm

- **divide and conquer algorithm:** an algorithm that repeatedly divides the given problem into smaller pieces that can be solved more easily

- it's easier to sort the two small lists than the one big list

Merge Sort Picture





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 $NA \leq R$ and $NB \leq S$:
 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 $K = 0, 1, 2, \dots, S - NB$:
 Set $C[PTR + K] := B[NB + K]$.
 [End of loop.]
 Else:
 Repeat for $K = 0, 1, 2, \dots, R - NA$:
 Set $C[PTR + K] := A[NA + K]$.
 [End of loop.]
 [End of If structure.]
4. Exit.

Time complexity of Merging = $O(r+s) = O(n)$

Merging: improvement when $r \ll s$

- ▶ When number of elements in A is much smaller than number of elements in B , A can be merged into B more efficiently using binary search
- ▶ For each element of A , perform binary search on B to find its proper location: $\log s$ comparisons
- ▶ For r elements of A : $r \log s$ comparisons
- ▶ Since $r \ll s$, $r \log s < r + s$

Merging: Further improvement

- Reducing the target set
 - Say, first element is inserted after $B[16]$, then we need to perform binary search on $B[17]$ onwards for inserting next element
- Tabbing
 - Find the tab using linear search and perform binary search on the selected tab

Example

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:

33, 66 22, 40 55, 88 11, 60 20, 80 44, 50 30, 70

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

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

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

11, 22, 33, 40, 55, 60, 66, 88 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

MERGE(A,R,LBA,B,S,LBB,C,LBC)

This procedure merges sorted arrays A and B into array C.

1. Set $NA=LBA$, $NB=LBB$, $PTR=LBC$, $UBA=LBA+R-1$, $UBB=LBB+S-1$
2. Same as MERGING except R is replaced by UBA and S by UBB.
3. Return

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

Merge Sort

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.

Complexity

- ▶ Time- execution is independent of order of elements
 - ▶ Best: $O(n \log n)$
 - ▶ Worst: $O(n \log n)$
 - ▶ Average: $O(n \log n)$
- ▶ Space- an auxiliary array of same size as input array is required
 - ▶ $O(n)$

Counting Sort

- ▶ Counting sort is an integer sorting method
- ▶ effective when the difference between different values are not large
- ▶ Method
 - ▶ Use element value as index to a count array
 - ▶ count the number of times each element occurs
 - ▶ using arithmetic on those counts to determine the positions of each value in the output sequence

Example: Counting Sort

Consider the data in the range 0 to 9.

Input data: 1, 4, 1, 2, 7, 5, 2

1. Take a count array to store the count of each unique element

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 2 0 1 1 0 1 0 0

2. Modify the count array such that each element at each index stores the sum of previous counts

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 4 4 5 6 6 7 7 7

3. Output each value from the input data followed by decreasing its count by 1

Put data 1 at index 2 in output. Decrease count by 1 to place next data 1 at an index 1 smaller than this index

Output data: 1, 4, 1, 2, 7, 5, 2

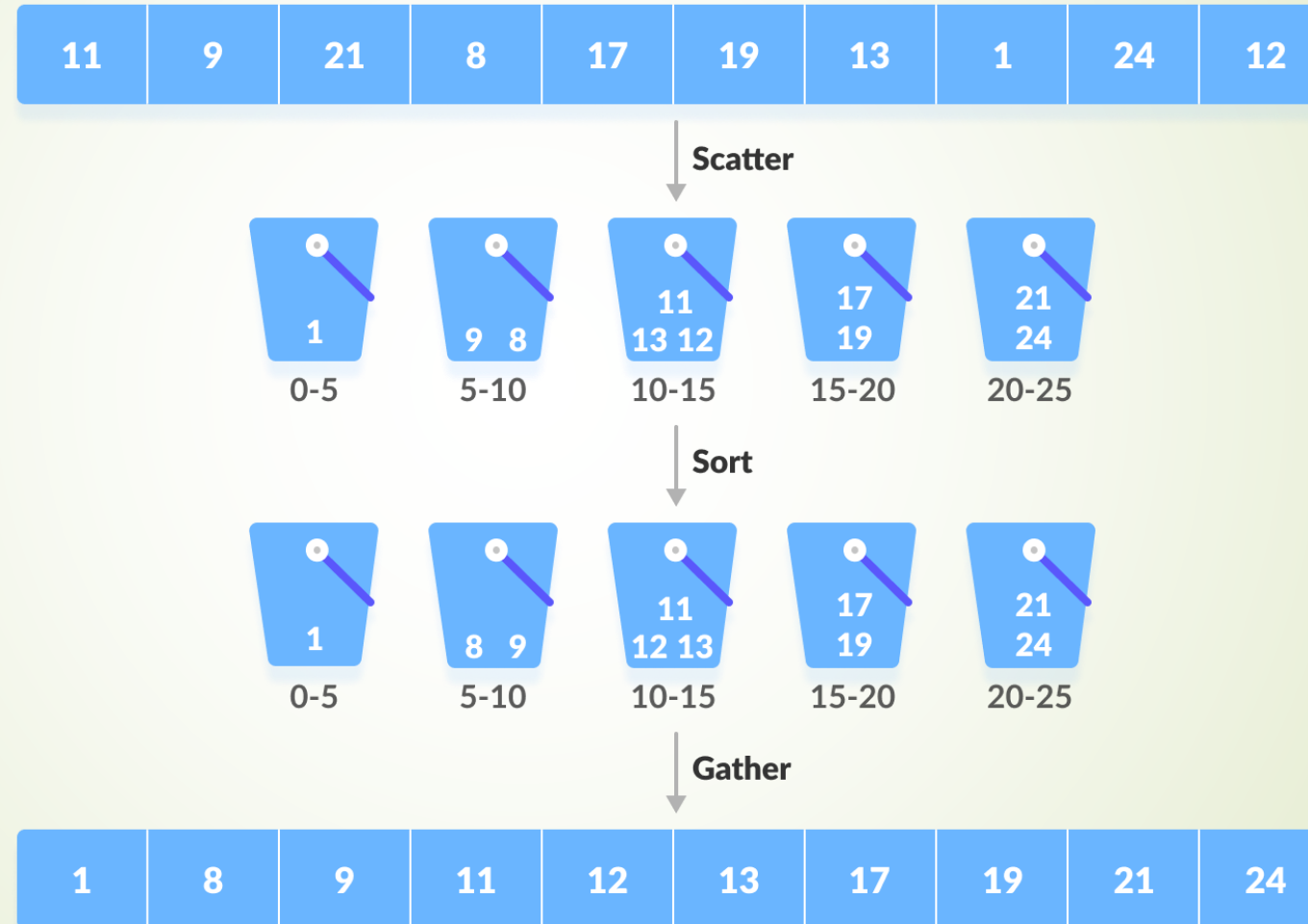
Complexity of Counting Sort

- ▶ Time Complexity: $O(n+k)$ where n is the number of elements in input array and k is the range of input
- ▶ Auxiliary Space: $O(n+k)$

Bucket Sort (or Bin Sort)

- Method
 - distribute the elements of input array into a fixed number of buckets
 - Buckets are implemented usually using linked lists
 - Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm
- generalization of counting sort
 - if each bucket has size 1 then bucket sort degenerates to counting sort

Example: Bucket Sort



Bucket Sort: Complexity

- mainly useful when input is uniformly distributed over a range
- The worst-case scenario occurs when all the elements are placed in a single bucket
 - When the input contains several values that are close to each other, those elements are likely to be placed in the same bucket
 - overall performance would then be dominated by the algorithm used to sort each bucket ($O(n^2)$ or $O(n \log n)$)
- When input values are distributed uniformly among the buckets, time complexity is $O(n)$
 - Average case: $O(n + n^2/k + k)$ and $k = \theta(n)$

Radix sort

- Radix = “The base of a number system”
- Idea: Counting (Bucket) Sort on each digit, bottom up (from LSD to MSD)
- Example: base 2

2	0 1 0	0 1 0	0 0 0	0 0 0	0
0	0 0 0	0 0 0	1 0 0	0 0 1	1
5	1 0 1	1 0 0	1 0 1	0 1 0	2
1	0 0 1	1 1 0	0 0 1	0 1 1	3
7	1 1 1	1 0 1	0 1 0	1 0 0	4
3	0 1 1	0 0 1	1 1 0	1 0 1	5
4	1 0 0	1 1 1	1 1 1	1 1 0	6
6	1 1 0	0 1 1	0 1 1	1 1 1	7

Radix Sort

Example: base 10

0	3	2
2	2	4
0	1	6
0	1	5
0	3	1
1	6	9
1	2	3
2	5	2

0	3	1
0	3	2
2	5	2
1	2	3
2	2	4
0	1	5
0	1	6
1	6	9

0	1	5
0	1	6
1	2	3
2	2	4
0	3	1
0	3	2
2	5	2
1	6	9

0	1	5
0	1	6
0	3	1
0	3	2
1	2	3
1	6	9
2	2	4
2	5	2



Radix sort: Complexity

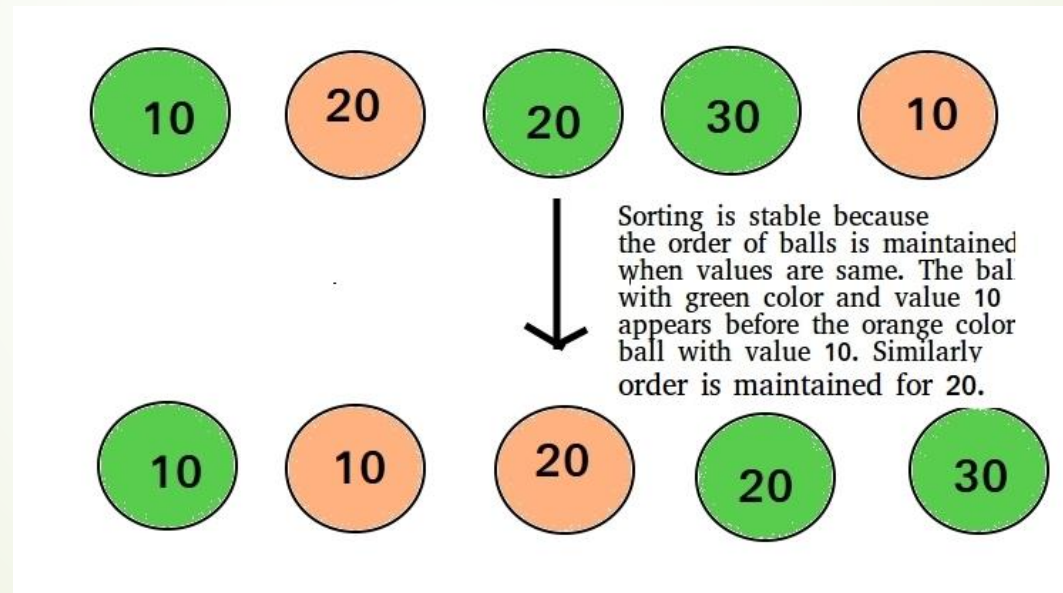
- Each sorting pass can be performed via counting sort
 - It is $O(n)$: counting sort is applied only on a single digit
- If the numbers are m digits long (or max m digits in any number), then there are m sorting steps.
- Hence, radix sort is $O(mn)$

In-place Sorts

- An in-place algorithm is an algorithm that does not need an extra space and produces an output in the same memory that contains the data by transforming the input 'in-place'. However, a small constant extra space (i.e. $O(1)$) used for variables is allowed.
- An algorithm which is not in-place is sometimes called **not-in-place** or **out-of-place**
- In-Place : Bubble sort, Selection Sort, Insertion Sort, Heapsort.
- Not In-Place : Merge Sort.

Stable Sorts

- ▶ Stability is mainly important when we have duplicate input values
- ▶ A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.



Summary of Sorting Algorithms

81


Algorithm	Best TC	Avg TC	Worst TC	Space Com	In-place?	Stable?	Remark
Bubble	n	n^2	n^2	1	Yes	Yes	Best case with early termination
Insertion	n	n^2	n^2	1	Yes	Yes	
Selection	n^2	n^2	n^2	1	Yes	No	
Merge	$n \log n$	$n \log n$	$n \log n$	n	No	Yes	
Quick	$n \log n$	$n \log n$	n^2	$\log n$	Yes	No	Extra space is for recursive calls
Heap	$n \log n$	$n \log n$	$n \log n$	n	Yes	No	
Counting	$n+k$	$n+k$	$n+k$	$n+k$	No	Yes	k is range of input values. If k is $O(n)$, time complexity is $O(n)$
Radix	mn	mn	mn	$n+k$	No	Yes	m is max number of digits in any input value. k is range of digits
Bucket	$n+k$	$n+k$	n^2	$n+k$	No	Yes	Buckets are implemented using arrays, counting sort is used for individual buckets

Other Sorting Algorithms

- Comparison Sorts
 - Shell Sort, Gnome sort, Tournament Sort, Tim Sort, Block Sort, Cube Sort, Tree Sort, Block Sort, Library Sort, Smooth Sort, odd-even Sort, Strand Sort etc.
- Non-Comparison Sorts
 - Pigeonhole Sort, Flash Sort, Spread Sort, Burst Sort, Postman Sort etc.
- Others
 - Bitonic, Poll Sort, Bogosort, Stooge Sort etc.



Lower Bound for Comparison Sorting

- Merge sort and Heap sort
 - worst-case running time is $O(N \log N)$
 - Are there better algorithms?
 - Goal: Prove that any sorting algorithm based on only comparisons takes $\Omega(N \log N)$ comparisons in the worst case (worst-case input) to sort N elements.
- 



Lower Bound for Sorting

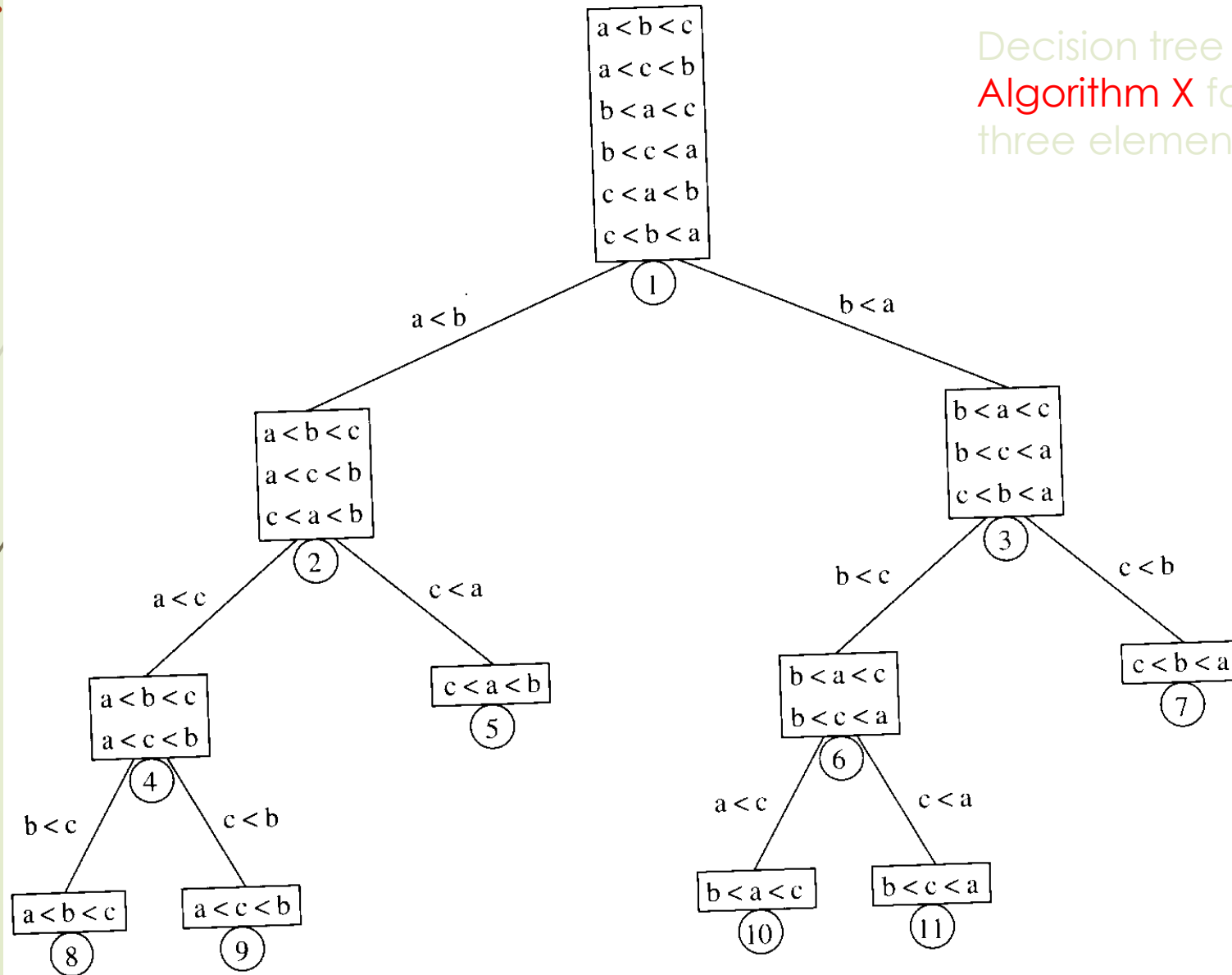
- ▶ Suppose we want to sort N distinct elements
- ▶ How many possible orderings do we have for N elements?
- ▶ We can have $N!$ possible orderings (e.g., the sorted output for a, b, c can be $a b c$, $b a c$, $a c b$, $c a b$, $c b a$, $b c a$.)



Lower Bound for Sorting

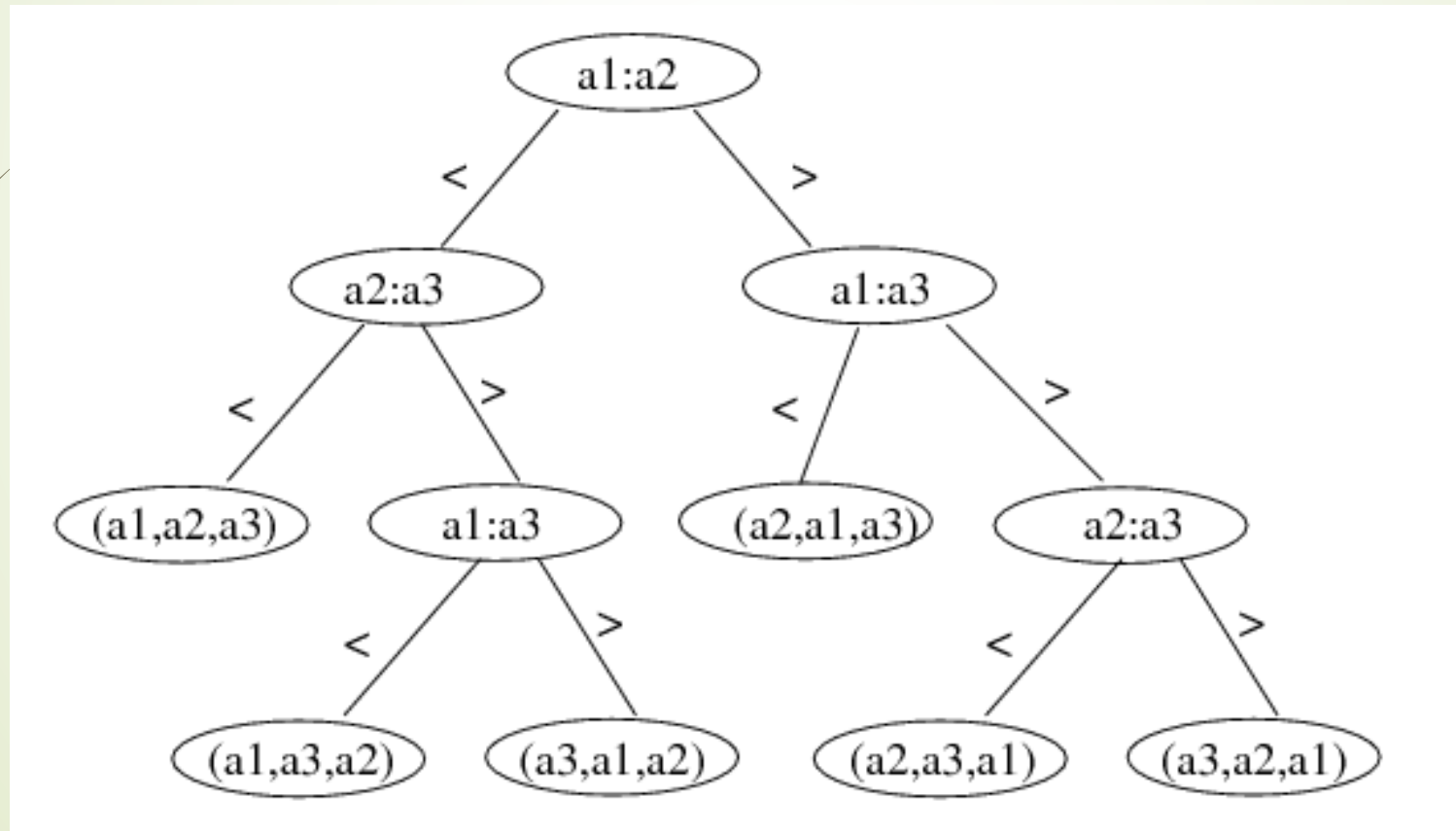
- Any comparison-based sorting process can be represented as a binary **decision tree**.
- Each node represents a set of possible orderings, consistent with all the comparisons that have been made
- The tree edges are results of the comparisons

Decision tree for
Algorithm X for sorting
three elements a, b, c



Lower Bound for Sorting

- A different algorithm would have a different decision tree
- Decision tree for **Insertion Sort** on 3 elements:



Lower Bound for Sorting

- The worst-case number of comparisons used by the sorting algorithm is equal to the **depth of the deepest leaf**
 - The average number of comparisons used is equal to the average depth of the leaves
- A decision tree to sort N elements must have **$N!$ leaves**
 - a binary tree of depth d has at most 2^d leaves
 - \Rightarrow the tree must have depth at least $\lceil \log_2 (N!) \rceil$
- Therefore, any sorting algorithm based on only comparisons between elements requires at least $\lceil \log_2(N!) \rceil$ comparisons in the worst case.

Lower Bound for Sorting

$$\begin{aligned}\log_2(N!) &= \log(N(N-1)(N-2)\cdots(2)(1)) \\ &= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1 \\ &\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log(N/2) \\ &\geq \frac{N}{2} \log \frac{N}{2} \\ &= \frac{N}{2} \log N - \frac{N}{2} \\ &= \Omega(N \log N)\end{aligned}$$

- Any sorting algorithm based on comparisons between elements requires $\Omega(N \log N)$ comparisons.

Better than $n \log n$

- Theoretical computer scientists have detailed other sorting algorithms that provide better than $O(n \log n)$ time complexity assuming additional constraints, including:
 - Thorup's algorithm, a randomized algorithm for sorting keys from a domain of finite size, taking $O(n \log \log n)$ time and $O(n)$ space
 - A randomized integer sorting algorithm taking $O(n \log \log n)$ expected time and $O(n)$ space