

UNIT 8

Sorting

(5 Hours) — Data Structures and Algorithms (CACS201)

Sorting is one of the most fundamental and most studied operations in computer science. It is estimated that a very large share of all computing time spent worldwide goes into arranging data into order. This unit develops the classical internal sorting algorithms — insertion, selection, exchange, merge, distribution and tree-based methods — analyses their running time, and introduces the asymptotic notation used to describe algorithmic efficiency.

8.1 Introduction to Sorting

Sorting is the process of rearranging a collection of data items into a specified order — either **ascending** (non-decreasing) or **descending** (non-increasing) — according to the value of a designated field called the **sort key**.

Formally, given a list of n elements $a_1, a_2, a_3, \dots, a_n$, a sorting algorithm produces a permutation (rearrangement) $a'_1, a'_2, \dots, a'_n$ of those same elements such that:

$$a'_1 \leq a'_2 \leq a'_3 \leq \dots \leq a'_n \text{ (for ascending order)}$$

8.1.1 Why Sorting Matters

- **Faster searching.** Binary search runs in $O(\log n)$ time but requires sorted data; on unsorted data we are limited to $O(n)$ linear search.
- **Easier duplicate detection.** Equal items become adjacent after sorting.
- **Simpler merging.** Two sorted lists can be merged in linear time.
- **Better presentation.** Reports, rankings, directories and indexes must be ordered to be readable.
- **Building block.** Many algorithms — closest-pair, convex hull, Kruskal's minimum spanning tree — begin with a sorting step.

8.1.2 Basic Terminology

Term	Meaning
Sort key	The field of a record on which the ordering is based (e.g. roll number in a student record).

Term	Meaning
Record	A collection of related fields treated as one unit; the key is one of its fields.
Pass	One complete traversal of the list (or of the unsorted part of it) by the algorithm.
In-place sort	Requires only $O(1)$ or $O(\log n)$ extra memory beyond the input array itself.
Stable sort	Preserves the relative order of records having equal keys.
Comparison sort	Determines order only by comparing pairs of keys (insertion, selection, bubble, quick, merge, heap).
Non-comparison sort	Uses the internal structure of the key instead of comparisons (radix, counting, bucket).
Adaptive sort	Runs faster when the input is already partially sorted (insertion, bubble with flag).

8.1.3 Stability of Sorting

A sorting algorithm is **stable** if two records with the same key appear in the output in the same relative order as they appeared in the input. Stability matters when data is sorted repeatedly on different keys.

Example 8.1 — Why stability matters

A list of students is already sorted by **name**. It is then sorted again by **marks**. Input (name, marks):

(Anil, 80), (Bina, 75), (Chetan, 80), (Deepa, 75)

Result

Stable sort by marks gives: (Bina, 75), (Deepa, 75), (Anil, 80), (Chetan, 80) — within each marks group the names are still alphabetical.

Unstable sort might give: (Deepa, 75), (Bina, 75), (Chetan, 80), (Anil, 80) — the name ordering inside each group is lost.

Stable: insertion, bubble, merge, radix. **Unstable:** selection, quick, heap, shell.

8.2 Internal and External Sorting

Sorting methods are broadly classified by **where the data resides** during the sorting process.

8.2.1 Internal Sorting

In **internal sorting** the entire collection of data to be sorted is small enough to be held in the computer's main memory (RAM) at one time. Because any element can be accessed directly in constant time, internal sorting algorithms are free to move elements around arbitrarily. All of the algorithms studied in the remainder of this unit — insertion, selection, bubble, quick, merge, shell, radix and heap sort — are internal sorts.

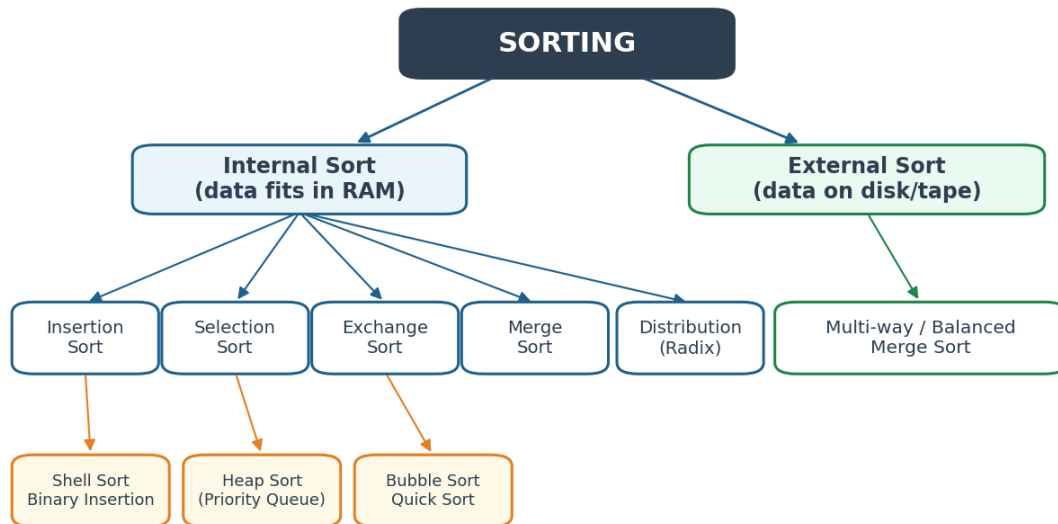
8.2.2 External Sorting

In **external sorting** the data is too large to fit into main memory and must remain on secondary storage such as a hard disk or magnetic tape. Only a small portion — a **block** or **run** — can be brought into memory at any instant. Since disk access is several orders of magnitude slower than memory access, the efficiency of an external sort is measured by the **number of disk I/O operations** rather than the number of comparisons.

The standard technique is the **external merge sort**, carried out in two phases:

- **Run-creation phase:** read as much data as memory allows, sort it internally (often with quick sort or heap sort), and write the sorted block back to disk as a *run*. Repeat until all data has been processed into runs.
- **Merge phase:** repeatedly merge groups of runs together using a k-way merge until a single sorted file remains. Each round of merging is called a *merge pass*.

Basis	Internal Sorting	External Sorting
Data location	Entirely in main memory (RAM)	On disk / tape; only blocks in RAM
Data size	Small — fits in available memory	Very large — exceeds memory capacity
Cost measure	Number of comparisons and swaps	Number of disk read/write operations
Access pattern	Random access, constant time	Mostly sequential block access
Speed	Fast	Comparatively very slow
Examples	Quick, merge, heap, insertion, shell	External merge sort, polyphase merge, balanced merge



Variants shown in yellow are refinements of the parent method

Figure 8.1 — Classification of sorting techniques

Note: Merge sort appears in both families. Internally it is a divide-and-conquer array algorithm; externally the same merging idea is applied to sorted runs stored on disk.

8.3 Insertion Sort

Insertion sort builds the sorted list one element at a time. It maintains a sorted region at the left end of the array; in each pass it takes the first element of the unsorted region (the **key**), scans backwards through the sorted region shifting larger elements one position to the right, and drops the key into the gap that opens up. This is exactly how most people arrange a hand of playing cards.

8.3.1 Algorithm

Algorithm: INSERTION-SORT(A, n)

```

for i = 1 to n-1 do
  key = A[i]
  j = i - 1
  while j >= 0 and A[j] > key do
    A[j+1] = A[j]      // shift right
    j = j - 1
  end while
  A[j+1] = key        // place key in the gap
end for
  
```

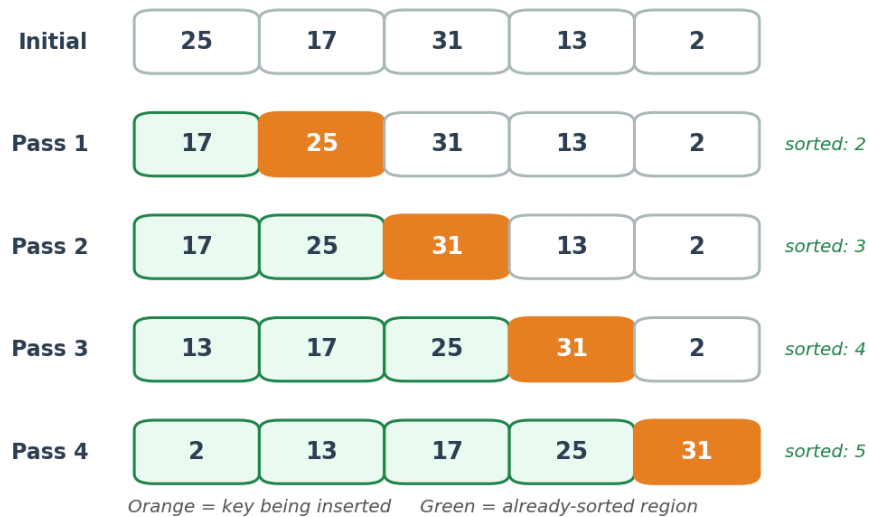


Figure 8.2 — Insertion sort on the list 25, 17, 31, 13, 2

Example 8.2

Sort the list **25, 17, 31, 13, 2** using insertion sort and show every pass.

Solution

Pass 1 (key = 17): $17 < 25$, so 25 shifts right → **17, 25, 31, 13, 2**

Pass 2 (key = 31): $31 > 25$, already in place → **17, 25, 31, 13, 2**

Pass 3 (key = 13): 31, 25, 17 all shift right → **13, 17, 25, 31, 2**

Pass 4 (key = 2): 31, 25, 17, 13 all shift right → **2, 13, 17, 25, 31**

Sorted list: **2, 13, 17, 25, 31**. Total comparisons = $1 + 1 + 3 + 4 = 9$.

8.3.2 Analysis

The number of passes is always $n - 1$. The work done inside each pass depends on the data:

Case	Condition	Comparisons	Complexity
Best	List already sorted — the while loop exits immediately	$n - 1$	$O(n)$
Average	Random order — about half of the sorted region is scanned	$\approx n^2/4$	$O(n^2)$
Worst	List sorted in reverse order — every element shifts fully	$n(n-1)/2$	$O(n^2)$

- **Space complexity:** $O(1)$ — in-place, needs only the single variable *key*.
- **Stable:** Yes, because shifting stops as soon as $A[j] \leq \text{key}$.

- **Adaptive:** Yes — it is the fastest simple sort on nearly-sorted data, and is commonly used to finish off the small sublists produced by quick sort.

8.4 Selection Sort

Selection sort works by repeatedly *selecting* the smallest element from the unsorted portion of the array and exchanging it with the element at the first unsorted position. After the i -th pass, the first i positions hold the i smallest values in their final sorted order.

8.4.1 Algorithm

Algorithm: SELECTION-SORT(A, n)

```

for i = 0 to n-2 do
    min = i
    for j = i+1 to n-1 do
        if A[j] < A[min] then
            min = j
    end for
    if min != i then
        swap A[i] and A[min]
    end for

```

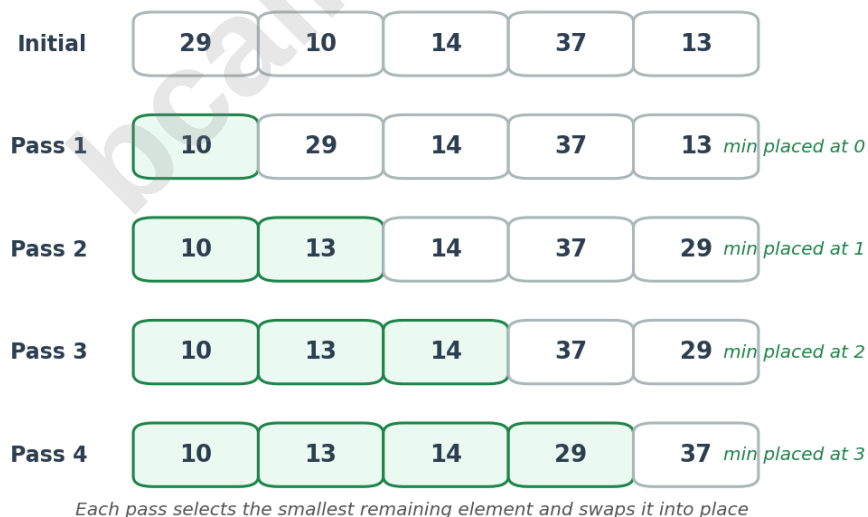


Figure 8.3 — Selection sort on the list 29, 10, 14, 37, 13

Example 8.3

Sort 29, 10, 14, 37, 13 using selection sort.

Solution

Pass 1: minimum of the whole list is 10 → swap with 29 → **10**, 29, 14, 37, 13

Pass 2: minimum of {29, 14, 37, 13} is 13 → swap with 29 → **10, 13**, 14, 37, 29

Pass 3: minimum of {14, 37, 29} is 14 → already in place → **10, 13, 14**, 37, 29

Pass 4: minimum of {37, 29} is 29 → swap with 37 → **10, 13, 14, 29, 37**

Sorted list: **10, 13, 14, 29, 37**. Swaps used: only 3.

8.4.2 Analysis

The inner loop always runs to completion regardless of the data, so the number of comparisons is fixed:

$$(n-1) + (n-2) + \dots + 2 + 1 = n(n-1)/2 \approx n^2/2 \text{ comparisons}$$

- **Time complexity:** $O(n^2)$ in the best, average *and* worst case — it is not adaptive.
- **Space complexity:** $O(1)$ — in-place.
- **Swaps:** at most $n - 1$, the fewest of any simple sort. This makes selection sort attractive when a write/swap is far more expensive than a comparison (for example on flash memory with limited write cycles).
- **Stable:** No — a long-distance swap can jump one equal key over another. A linked-list variant that moves rather than swaps can be made stable.

Note: Insertion sort and selection sort are both $O(n^2)$, but insertion sort usually wins in practice because it can terminate its inner loop early, whereas selection sort never can.

8.5 Exchange Sort — Bubble Sort

Exchange sorting is the family of methods that work by comparing pairs of elements and exchanging (swapping) them whenever they are out of order. The two principal members of this family are **bubble sort** (simple, $O(n^2)$) and **quick sort** (a partition-exchange method, $O(n \log n)$ on average).

8.5.1 Bubble Sort

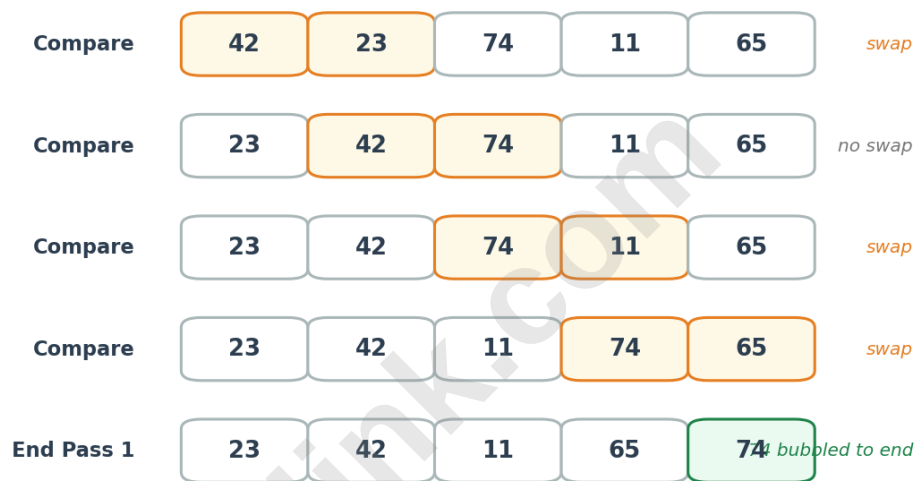
Bubble sort repeatedly steps through the list comparing each adjacent pair and swapping them if they are in the wrong order. On each pass the largest remaining element is carried, or "bubbles", to the end of the unsorted region. After $n - 1$ passes the entire list is sorted.

Algorithm: BUBBLE-SORT(A, n)

```

for i = 0 to n-2 do
  swapped = false
  for j = 0 to n-2-i do
    if A[j] > A[j+1] then
      swap A[j] and A[j+1]
      swapped = true
    end for
  if swapped == false then
    break          // list is already sorted - early exit
  end for

```



After every pass the largest unsorted value settles at the rightmost free slot

Figure 8.4 — First pass of bubble sort on 42, 23, 74, 11, 65

Example 8.4

Sort **42, 23, 74, 11, 65** using bubble sort. Show the state after each pass.

Solution

Pass 1: (42,23) swap; (42,74) no; (74,11) swap; (74,65) swap → 23, 42, 11, 65, **74**

Pass 2: (23,42) no; (42,11) swap; (42,65) no → 23, 11, 42, **65, 74**

Pass 3: (23,11) swap; (23,42) no → 11, 23, **42, 65, 74**

Pass 4: (11,23) no swap → flag stays false, algorithm stops early.

Sorted list: **11, 23, 42, 65, 74**

Case	Condition	Comparisons	Swaps	Complexity
Best	Already sorted (with the swapped flag)	$n - 1$	0	$O(n)$
Average	Random order	$\approx n^2/2$	$\approx n^2/4$	$O(n^2)$

Case	Condition	Comparisons	Swaps	Complexity
Worst	Reverse sorted	$n(n-1)/2$	$n(n-1)/2$	$O(n^2)$

- **Space complexity:** $O(1)$ — in-place.
- **Stable:** Yes — only strictly-greater adjacent pairs are swapped.
- Bubble sort performs the most swaps of all the simple sorts, which makes it the slowest in practice. It survives mainly as a teaching example.

8.6 Quick Sort

Quick sort, invented by C. A. R. Hoare in 1960, is a divide-and-conquer exchange sort. It chooses one element as the **pivot** and **partitions** the list so that every element smaller than the pivot lies to its left and every larger element lies to its right. The pivot is then in its final sorted position, and the two sublists on either side are sorted recursively by the same method.

8.6.1 Algorithm

Algorithm: QUICK-SORT(A, low, high)

```

if low < high then
    p = PARTITION(A, low, high)
    QUICK-SORT(A, low, p-1)    // left sublist
    QUICK-SORT(A, p+1, high)   // right sublist

PARTITION(A, low, high)       // Lomuto scheme
    pivot = A[high]
    i = low - 1
    for j = low to high-1 do
        if A[j] <= pivot then
            i = i + 1
            swap A[i] and A[j]
    end for
    swap A[i+1] and A[high]
    return i + 1

```

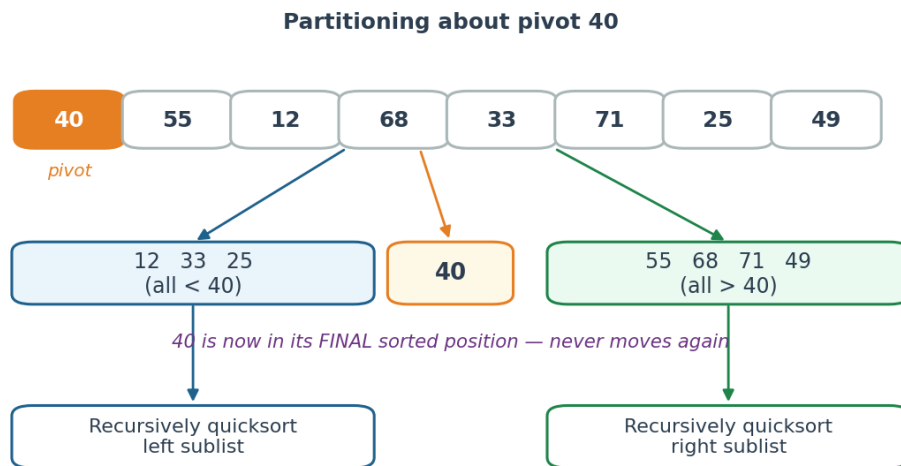


Figure 8.5 — One partitioning step of quick sort

Example 8.5

Sort **40, 55, 12, 68, 33, 71, 25, 49** using quick sort, taking the first element of each sublist as the pivot.

Solution

Step 1 — pivot = 40. Partition gives [12, 33, 25] **40** [55, 68, 71, 49]. 40 is now in its final place.

Step 2 — left sublist [12, 33, 25], pivot = 12 → [] **12** [33, 25]. Then pivot = 33 → [25] **33** [] → gives 12, 25, 33.

Step 3 — right sublist [55, 68, 71, 49], pivot = 55 → [49] **55** [68, 71]. Then pivot = 68 → [] **68** [71] → gives 49, 55, 68, 71.

Result: 12, 25, 33, **40**, 49, 55, 68, 71

8.6.2 Analysis

The cost of quick sort is dominated by how evenly the pivot splits the list. Each partitioning step itself costs $O(n)$ comparisons.

Case	Pivot behaviour	Recurrence	Complexity
Best	Pivot always splits the list into two equal halves	$T(n) = 2T(n/2) + n$	$O(n \log n)$
Average	Pivot splits reasonably well on random data	$T(n) \approx 2T(n/2) + n$	$O(n \log n)$
Worst	Pivot is always the smallest or largest element (already-sorted input with first/last pivot)	$T(n) = T(n-1) + n$	$O(n^2)$

- **Space complexity:** $O(\log n)$ average for the recursion stack; $O(n)$ in the worst case. No auxiliary array is needed, so it is regarded as in-place.
- **Stable:** No — partitioning swaps elements across long distances.
- **Avoiding the worst case:** choose the pivot as the *median-of-three* (first, middle, last) or pick it at random. Switching to insertion sort for sublists shorter than about 10 elements also speeds it up noticeably.

Note: Quick sort is usually the fastest general-purpose internal sort in practice, despite its $O(n^2)$ worst case, because its inner loop is extremely simple and it has excellent cache locality.

8.7 Merge Sort

Merge sort is a divide-and-conquer algorithm that splits the list into two halves, sorts each half recursively, and then **merges** the two sorted halves into a single sorted list. The recursion bottoms out at sublists of length one, which are trivially sorted.

8.7.1 Algorithm

Algorithm: MERGE-SORT(A, low, high)

```

if low < high then
    mid = (low + high) / 2
    MERGE-SORT(A, low, mid)
    MERGE-SORT(A, mid+1, high)
    MERGE(A, low, mid, high)

MERGE(A, low, mid, high)
    i = low; j = mid+1; k = 0; create temp[]
    while i <= mid and j <= high do
        if A[i] <= A[j] then temp[k++] = A[i++]
        else temp[k++] = A[j++]
    copy any remaining elements of the left half into temp
    copy any remaining elements of the right half into temp
    copy temp[] back into A[low..high]

```

DIVIDE ↓

↑ CONQUER (merge)

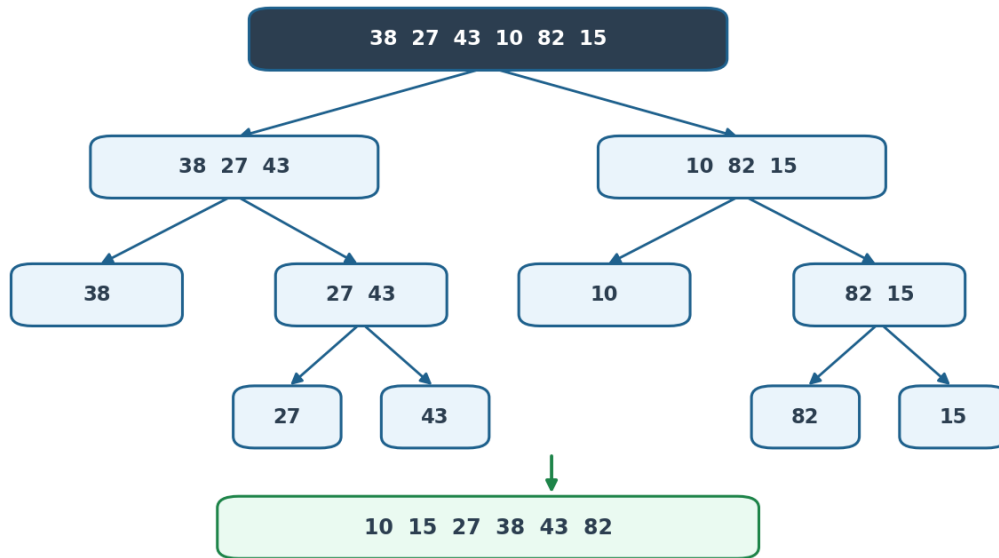


Figure 8.6 — Merge sort on 38, 27, 43, 10, 82, 15; divide then merge

Example 8.6

Trace merge sort on 38, 27, 43, 10, 82, 15.

Solution

Divide: [38, 27, 43] and [10, 82, 15] → [38] [27, 43] and [10] [82, 15] → [38] [27] [43] [10] [82] [15]

Merge level 1: [27] + [43] → [27, 43]; [82] + [15] → [15, 82]

Merge level 2: [38] + [27, 43] → [27, 38, 43]; [10] + [15, 82] → [10, 15, 82]

Merge level 3: [27, 38, 43] + [10, 15, 82] → [10, 15, 27, 38, 43, 82]

8.7.2 Analysis

The list is halved repeatedly, so the depth of the recursion tree is $\log_2 n$. Each level performs a total of n comparisons and movements to merge all its sublists. The recurrence is $T(n) = 2T(n/2) + O(n)$, whose solution is:

$T(n) = O(n \log n)$ — in the best, average and worst case

- **Space complexity:** $O(n)$ — the temporary array is the chief drawback; merge sort is *not* in-place.
- **Stable:** Yes, provided the merge step takes from the left half when the two keys are equal.
- **Predictable:** its running time does not depend on the input arrangement, which makes it the method of choice when worst-case guarantees are required, and the natural basis for

external sorting.

8.8 Radix Sort

Radix sort is a non-comparison, distribution sort. Instead of comparing whole keys it examines the keys digit by digit, distributing the records into **buckets** (also called pockets) — one bucket for each possible digit value — and then collecting the buckets back in order. The process is repeated once for each digit position. The number of distinct digit values is called the **radix** or base: 10 for decimal numbers, 26 for alphabetic keys, 2 for binary.

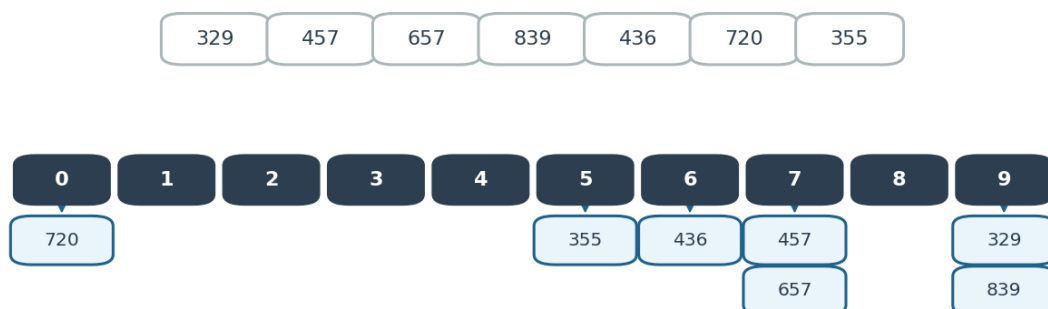
The usual form is **LSD radix sort**, which processes the *least significant digit first* and works upward to the most significant digit. LSD radix sort is correct only because the distribution step is stable — the ordering achieved on earlier (less significant) digits is preserved while later digits are processed.

8.8.1 Algorithm

Algorithm: RADIX-SORT(A, n, d) // d = number of digits

```
for pos = 1 to d do                                // pos = 1 is the units digit
  create 10 empty buckets B[0..9]
  for each element x in A, in order do
    digit = (x / 10^(pos-1)) mod 10
    append x to bucket B[digit]
  end for
  concatenate B[0], B[1], ..., B[9] back into A
end for
```

Pass 1 — distribute on the units digit



Collect buckets 0→9 in order:



Repeat for the tens digit, then the hundreds digit — 3 passes in total

Figure 8.7 — Radix sort: first pass on the units digit

Example 8.7

Sort **329, 457, 657, 839, 436, 720, 355** using LSD radix sort.

Solution

Pass 1 (units digit): buckets → 0:720, 5:355, 6:436, 7:457 657, 9:329 839

Collected: **720, 355, 436, 457, 657, 329, 839**

Pass 2 (tens digit): buckets → 2:720 329, 3:436 839, 5:355 657, 7:457

Collected: **720, 329, 436, 839, 355, 657, 457**

Pass 3 (hundreds digit): buckets → 3:329 355, 4:436 457, 6:657, 7:720, 8:839

Collected: **329, 355, 436, 457, 657, 720, 839** ← sorted

8.8.2 Analysis

If there are n keys, each of d digits drawn from a radix of size r , then each pass costs $O(n + r)$ and there are d passes:

$T(n) = O(d \cdot (n + r))$, which is $O(n)$ when d and r are constants

- **Space complexity:** $O(n + r)$ for the buckets — it is not in-place.
- **Stable:** Yes, and it must be, otherwise the algorithm would be incorrect.
- **Limitation:** it applies only to keys that can be decomposed into digits or characters of bounded length. It is useless for arbitrary objects with a user-supplied comparison function, and it becomes inefficient when the keys are very long (d large).

Note: Radix sort does not contradict the $\Omega(n \log n)$ lower bound for sorting, because that bound applies only to algorithms that determine order by comparing keys. Radix sort never compares two keys with each other.

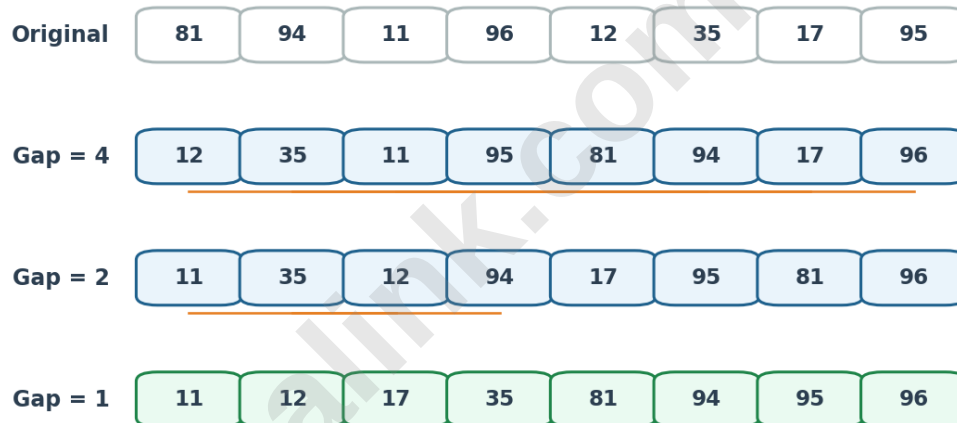
8.9 Shell Sort

Shell sort, devised by Donald L. Shell in 1959, is a generalisation of insertion sort — it is also called **diminishing increment sort**. Ordinary insertion sort moves elements only one position at a time, so an element far from its final place needs many shifts. Shell sort fixes this by first sorting elements that are far apart, using a **gap** (increment) h : the array is treated as h interleaved sublists, each of which is insertion-sorted. The gap is then reduced and the process repeated, finishing with a gap of 1 — a plain insertion sort on a list that is by then almost sorted, and therefore very fast.

8.9.1 Algorithm

Algorithm: SHELL-SORT(A, n)

```
gap = n / 2
while gap > 0 do
  for i = gap to n-1 do
    temp = A[i]
    j = i
    while j >= gap and A[j-gap] > temp do
      A[j] = A[j-gap]
      j = j - gap
    end while
    A[j] = temp
  end for
  gap = gap / 2
end while
```



Orange links join elements compared within the same gap-sublist

Figure 8.8 — Shell sort with the gap sequence 4, 2, 1

Example 8.8

Sort 81, 94, 11, 96, 12, 35, 17, 95 using shell sort with gaps 4, 2, 1.

Solution

Gap = 4 — sublists {81,12}, {94,35}, {11,17}, {96,95} are each sorted:

→ 12, 35, 11, 95, 81, 94, 17, 96

Gap = 2 — sublists {12,11,81,17} and {35,95,94,96} are each sorted:

→ 11, 35, 12, 94, 17, 95, 81, 96

Gap = 1 — a final insertion sort on the nearly-ordered list:

→ 11, 12, 17, 35, 81, 94, 95, 96

8.9.2 Analysis

The running time of shell sort depends entirely on the gap sequence chosen, and its exact average-case behaviour is still an open problem in mathematics.

Gap sequence	Worst-case complexity
Shell's original: $n/2, n/4, \dots, 1$	$O(n^2)$
Hibbard: $1, 3, 7, 15, \dots, 2^k - 1$	$O(n^{1.5})$
Knuth: $1, 4, 13, 40, \dots, (3^k - 1)/2$	$O(n^{1.5})$
Sedgewick	$O(n^{4/3})$

- **Space complexity:** $O(1)$ — in-place.
- **Stable:** No — comparing and moving elements across a gap can reorder equal keys.
- Shell sort is simple to code, needs no recursion and no extra memory, and performs very well on medium-sized arrays, which makes it popular in embedded systems.

8.10 Binary Sort

The term **binary sort** is used for two related techniques, both of which apply the idea of binary division to sorting.

8.10.1 Binary Insertion Sort

In plain insertion sort the correct position of the key inside the sorted region is found by a *linear* backward scan. Since that region is already sorted, the position can instead be located by **binary search**, cutting the number of comparisons for each key from $O(i)$ down to $O(\log i)$.

Algorithm: BINARY-INSERTION-SORT(A, n)

```

for i = 1 to n-1 do
    key = A[i]
    // binary search for the insertion point in A[0..i-1]
    low = 0; high = i - 1
    while low <= high do
        mid = (low + high) / 2
        if key < A[mid] then high = mid - 1
        else low = mid + 1
    end while
    // shift A[low..i-1] one position right, then insert
    for j = i-1 downto low do A[j+1] = A[j]
    A[low] = key
end for

```

The total number of **comparisons** falls to $O(n \log n)$, which is a real improvement. However, the elements must still be physically shifted to open the gap, and that shifting still costs $O(n^2)$ **movements**. The overall time complexity therefore remains $O(n^2)$. Binary insertion sort is worthwhile only when comparing two keys is much more expensive than moving them — for instance when the keys are long strings.

Example 8.9

Insert the key **29** into the sorted region **11, 18, 24, 35, 47, 52** using binary search.

Solution

low = 0, high = 5 $\rightarrow$ mid = 2, $A[2] = 24$; $29 > 24 \rightarrow$ low = 3

low = 3, high = 5 $\rightarrow$ mid = 4, $A[4] = 47$; $29 < 47 \rightarrow$ high = 3

low = 3, high = 3 $\rightarrow$ mid = 3, $A[3] = 35$; $29 < 35 \rightarrow$ high = 2

low (3) > high (2) $\rightarrow$ loop ends. Insertion position = index 3.

Shift 35, 47, 52 right and place 29 $\rightarrow$ **11, 18, 24, 29, 35, 47, 52**

8.10.2 Binary Tree Sort

Binary tree sort inserts all n elements into a **binary search tree** (BST) one at a time, and then performs an **inorder traversal** of the tree. Because an inorder traversal of a BST always visits the nodes in ascending key order, the traversal output is the sorted list.

- Each insertion costs $O(\log n)$ in a balanced tree, so building the tree costs $O(n \log n)$; the traversal costs $O(n)$. Total: **$O(n \log n)$** .
- If the input is already sorted, the BST degenerates into a linked list of depth n and the cost rises to **$O(n^2)$** . Using a self-balancing tree (AVL or red-black) restores the $O(n \log n)$ guarantee.
- **Space complexity:** $O(n)$ for the tree nodes — not in-place.

8.11 Heap Sort and the Priority Queue

8.11.1 The Heap Data Structure

A **heap** is a **complete binary tree** (every level is full except possibly the last, which is filled from the left) that satisfies the **heap property**:

- **Max-heap:** the key of every node is greater than or equal to the keys of its children, so the largest key sits at the root.

- **Min-heap:** the key of every node is less than or equal to the keys of its children, so the smallest key sits at the root.

Because the tree is complete, a heap needs no pointers — it is stored compactly in an array. For the node at index i (0-based indexing):

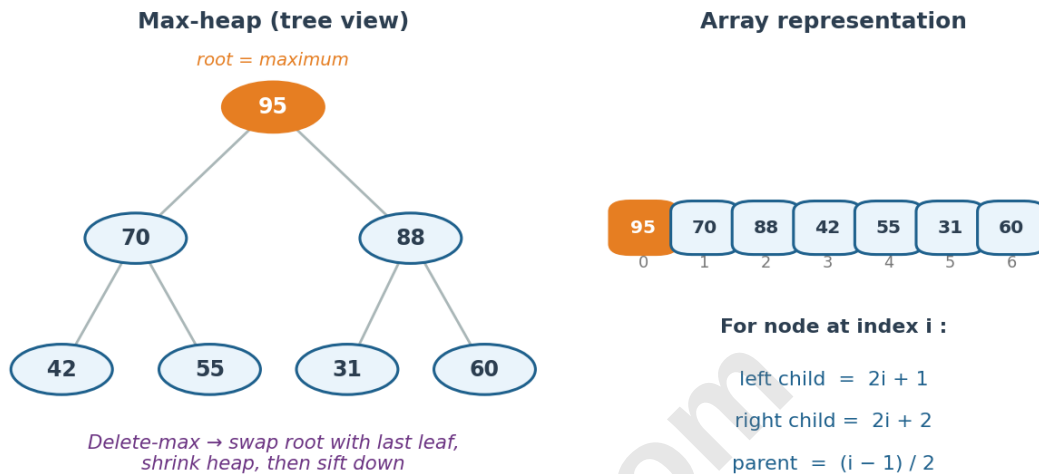


Figure 8.9 — A max-heap, its array representation and the index formulas

8.11.2 Heap as a Priority Queue

A **priority queue** is an abstract data type in which every element carries a priority, and the element removed is always the one with the highest (or lowest) priority rather than the one that arrived first. It is the natural structure for CPU process scheduling, printer spooling, Dijkstra's shortest-path algorithm and Huffman coding.

The heap is the standard implementation of a priority queue because both of its core operations cost only $O(\log n)$:

Operation	What it does	Method	Cost
insert(x)	Add a new element with its priority	Place x at the next free array slot, then sift up (compare with parent and swap while it is larger)	$O(\log n)$
delete-max()	Remove and return the highest-priority element	Take the root, move the last leaf to the root, shrink the heap, then sift down (swap with the larger child while smaller)	$O(\log n)$
peek()	Look at the highest-priority element	Read $A[0]$	$O(1)$
build-heap()	Turn an arbitrary array into a heap	Sift down from index $n/2 - 1$ down to 0	$O(n)$

Note: Building a heap by sifting down from the middle of the array costs only $O(n)$, not $O(n \log n)$, because most of the nodes are near the bottom of the tree where the sift-down distance is short.

8.11.3 Heap Sort

Heap sort is simply the repeated use of the priority-queue delete-max operation. It is best understood as a refinement of selection sort: instead of scanning the whole unsorted region linearly to find the maximum ($O(n)$), it keeps that region organised as a heap so that the maximum is available at the root and can be extracted in $O(\log n)$.

Algorithm: HEAP-SORT(A, n)

```
// Phase 1 - build a max-heap out of the whole array
for i = n/2 - 1 downto 0 do
    SIFT-DOWN(A, i, n)

// Phase 2 - repeatedly extract the maximum
for end = n-1 downto 1 do
    swap A[0] and A[end]      // largest goes to its final place
    SIFT-DOWN(A, 0, end)     // restore the heap on the shrunken range
end for

SIFT-DOWN(A, i, size)
    largest = i;  l = 2*i+1;  r = 2*i+2
    if l < size and A[l] > A[largest] then largest = l
    if r < size and A[r] > A[largest] then largest = r
    if largest != i then
        swap A[i] and A[largest]
        SIFT-DOWN(A, largest, size)
```

Example 8.10

Sort **4, 10, 3, 5, 1** using heap sort.

Solution

Build max-heap: start at index 1 (value 10) — already larger than child 1. Then index 0 (value 4) — larger child is 10, swap; 4 sifts down again past 5.

Heap array: **10, 5, 3, 4, 1**

Extract 1: swap 10 with 1 → 1, 5, 3, 4 | **10**; sift down → 5, 4, 3, 1 | 10

Extract 2: swap 5 with 1 → 1, 4, 3 | **5**, 10; sift down → 4, 1, 3 | 5, 10

Extract 3: swap 4 with 3 → 3, 1 | **4**, 5, 10; sift down → 3, 1 | 4, 5, 10

Extract 4: swap 3 with 1 → 1 | **3**, 4, 5, 10

Sorted list: **1, 3, 4, 5, 10**

8.11.4 Analysis

Phase 1 (build-heap) costs $O(n)$. Phase 2 performs $n - 1$ delete-max operations, each costing $O(\log n)$, giving $O(n \log n)$. The total is therefore:

$$T(n) = O(n) + O(n \log n) = O(n \log n) \text{ — in all cases}$$

- **Space complexity:** $O(1)$ — heap sort is in-place, unlike merge sort.
- **Stable:** No.
- It offers merge sort's worst-case guarantee with quick sort's memory footprint. In practice it is somewhat slower than quick sort because its memory access pattern jumps around the array and uses the cache poorly.

8.12 Efficiency of Sorting

The efficiency of a sorting algorithm is judged on several dimensions, not on speed alone:

- **Time complexity** — the number of key comparisons and element movements, expressed as a function of n . The best, average and worst cases must be distinguished.
- **Space complexity** — the amount of auxiliary memory required beyond the input itself. An algorithm using $O(1)$ extra space is *in-place*.
- **Stability** — whether the relative order of equal keys is preserved.
- **Adaptivity** — whether the algorithm exploits existing order in the input.
- **Nature of the data** — size of n , how nearly sorted it already is, the cost of a comparison versus the cost of a move, and whether the data is in an array or a linked list.

8.12.1 The Comparison-Sort Lower Bound

Any algorithm that sorts purely by comparing keys can be modelled as a **decision tree** in which each internal node is one comparison and each leaf is one of the $n!$ possible orderings. A binary tree with $n!$ leaves must have height at least $\log_2(n!)$, and by Stirling's approximation $\log_2(n!) \approx n \log_2 n - 1.44n$. Hence:

No comparison-based sorting algorithm can do better than $\Omega(n \log n)$ comparisons in the worst case.

Merge sort and heap sort achieve this bound and are therefore asymptotically optimal. Radix and counting sort beat it only because they are not comparison sorts.

8.12.2 Comparison of Sorting Algorithms

Algorithm	Best	Average	Worst	Space	Stable	In-place
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Yes
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Shell Sort	$O(n \log n)$	$O(n^{1.25})$	$O(n^2)$	$O(1)$	No	Yes
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	No
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Yes
Binary Tree Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$	Yes	No
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n + r)$	Yes	No

Note: In the radix sort row, k is the number of digits in the longest key and r is the radix (the number of buckets).

8.12.3 Choosing the Right Algorithm

Situation	Recommended method
n is very small (roughly under 20)	Insertion sort — lowest overhead
Data is almost already sorted	Insertion sort or bubble sort with the early-exit flag
General-purpose, average speed matters most	Quick sort with median-of-three pivot
A worst-case guarantee is required	Merge sort or heap sort
Stability is required	Merge sort (or insertion sort for small n)
Memory is very tight	Heap sort or shell sort — both $O(1)$ extra space
Keys are short integers or fixed-length strings	Radix sort
Data is a linked list	Merge sort — no random access needed, no extra array
Data will not fit in main memory	External merge sort

8.13 Big-O Notation

Asymptotic notation describes how the running time or memory use of an algorithm grows as the input size n becomes large. It deliberately ignores constant factors and lower-order terms, because these depend on the machine, the compiler and the programmer, while the growth rate

is a property of the algorithm itself.

8.13.1 Definition of Big-O

Let $f(n)$ and $g(n)$ be functions of the input size n . We say that

$f(n) = O(g(n))$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

In words: beyond some threshold n_0 , the function $f(n)$ never grows faster than a constant multiple of $g(n)$. Big-O therefore gives an **asymptotic upper bound** — a guarantee that the algorithm is "no worse than" this growth rate.

Example 8.11

Show that $f(n) = 3n^2 + 5n + 7$ is $O(n^2)$.

Solution

For every $n \geq 1$ we have $n \leq n^2$ and $1 \leq n^2$, so:

$$3n^2 + 5n + 7 \leq 3n^2 + 5n^2 + 7n^2 = 15n^2$$

Choosing $c = 15$ and $n_0 = 1$ satisfies the definition $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Therefore **$f(n) = O(n^2)$** .

The lower-order term $5n$ and the constant 7 become negligible as n grows, and the coefficient 3 is absorbed into c .

8.13.2 Related Asymptotic Notations

Notation	Name	Meaning	Bound
$O(g(n))$	Big-O	$f(n) \leq c \cdot g(n)$ for all $n \geq n_0$	Upper bound (at most)
$\Omega(g(n))$	Big-Omega	$f(n) \geq c \cdot g(n)$ for all $n \geq n_0$	Lower bound (at least)
$\Theta(g(n))$	Big-Theta	$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$	Tight bound (exactly)
$o(g(n))$	Little-o	$f(n)$ becomes negligible compared to $g(n)$	Strict upper bound
$\omega(g(n))$	Little-omega	$f(n)$ grows strictly faster than $g(n)$	Strict lower bound

Note: Big-O is often used loosely to mean the tight bound. Strictly, saying merge sort is $O(n^2)$ is a true but weak statement; $\Theta(n \log n)$ is the precise one.

8.13.3 Common Growth Rates

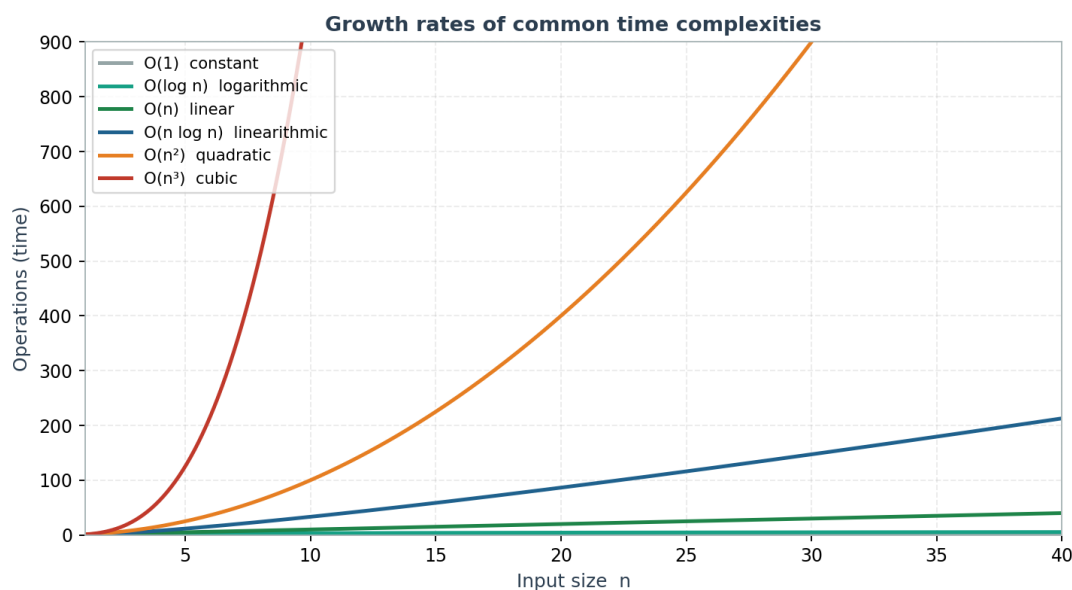


Figure 8.10 — Growth of common complexity classes with input size

The complexity classes ordered from fastest to slowest growth are:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

Complexity	Name	Typical example	Steps when $n = 1000$
$O(1)$	Constant	Accessing $A[i]$; pushing onto a stack	1
$O(\log n)$	Logarithmic	Binary search; heap insert	≈ 10
$O(n)$	Linear	Linear search; one pass of radix sort	1,000
$O(n \log n)$	Linearithmic	Merge sort, heap sort, average quick sort	$\approx 10,000$
$O(n^2)$	Quadratic	Bubble, selection, insertion sort	1,000,000
$O(n^3)$	Cubic	Naive matrix multiplication	1,000,000,000
$O(2^n)$	Exponential	Naive subset enumeration	astronomically large

8.13.4 Rules for Computing Big-O

- **Drop constants:** $O(5n) = O(n)$; $O(n/2) = O(n)$.
- **Keep only the dominant term:** $O(n^2 + n + 100) = O(n^2)$.
- **Sum rule (sequential code):** if block A is $O(f)$ and block B is $O(g)$, running them one after the other is $O(\max(f, g))$.
- **Product rule (nested loops):** a loop that is $O(f)$ containing a loop that is $O(g)$ costs $O(f \cdot g)$.

- **Different inputs use different variables:** two independent loops over arrays of size m and n cost $O(m + n)$, not $O(n)$.

Example 8.12

Determine the time complexity of the following fragment.

```
for i = 0 to n-1:
    for j = 0 to n-1:
        sum = sum + A[i][j]
for k = 0 to n-1:
    print B[k]
```

Solution

The nested loops execute $n \times n = n^2$ times $\rightarrow O(n^2)$ by the product rule.

The single loop executes n times $\rightarrow O(n)$.

The two parts run one after the other, so by the sum rule the total is $O(n^2 + n) = O(n^2)$.

Key Takeaways

✓ **Sorting** rearranges records into ascending or descending order of a chosen **sort key**. **Internal** sorts hold all data in RAM; **external** sorts work on data too large for memory and are measured in disk I/O.

✓ **Insertion sort** inserts each key into a growing sorted prefix — $O(n^2)$, but $O(n)$ on nearly-sorted data, stable and adaptive.

✓ **Selection sort** repeatedly selects the minimum — always $O(n^2)$, unstable, but uses at most $n - 1$ swaps.

✓ **Bubble sort** swaps adjacent out-of-order pairs — $O(n^2)$, stable, the slowest of the simple sorts because of its heavy swapping.

✓ **Quick sort** partitions about a pivot — $O(n \log n)$ average, $O(n^2)$ worst, in-place, unstable, and fastest in practice.

✓ **Merge sort** divides, sorts recursively and merges — $O(n \log n)$ in every case and stable, but needs $O(n)$ extra space.

✓ **Radix sort** distributes keys into buckets digit by digit — $O(nk)$, stable, and not a comparison sort, so the $n \log n$ lower bound does not apply to it.

✓ **Shell sort** is insertion sort with diminishing gaps — in-place, no recursion, and much faster than plain insertion sort on medium arrays.

✓ **Binary insertion sort** locates the insertion point by binary search, cutting comparisons to $O(n \log n)$ but leaving movements at $O(n^2)$.

✓ A **heap** is a complete binary tree stored in an array; it implements a **priority queue** with $O(\log n)$ insert and delete-max. **Heap sort** builds a heap then repeatedly extracts the maximum — $O(n \log n)$, in-place, unstable.

✓ Every **comparison-based** sorting algorithm needs $\Omega(n \log n)$ comparisons in the worst case; merge sort and heap sort meet this bound.

✓ **Big-O** gives an asymptotic upper bound: $f(n) = O(g(n))$ if $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. Drop constants, keep the dominant term.

— *End of Unit 8* —

bcalink.com