

# DATA STRUCTURES SORTING & SEARCHING

---

## Sorting Methods एवं Search Algorithms

Hindi-English explanations के साथ

14 अध्यायों में Unit IV की पूर्ण तैयारी

SIX SORTS

LINEAR / BINARY

MCQs / DRY RUNS

# पुस्तक का परिचय

## Unit IV / How to Use This Book

यह पुस्तक दिए syllabus के Sorting और Searching topics cover करती है: types of sorting, Selection Sort, Insertion Sort, Bubble Sort, Quick Sort, Merge Sort, Radix Sort, Linear Search और Binary Search.

14 अध्यायों में Hindi-English explanations, vector diagrams, complete pseudocode, solved dry runs, complexity/stability comparisons, exact-count questions और 40 MCQs with answer key हैं. Binary Search duplicate/boundary variants तथा algorithm lab core topics को practical understanding से जोड़ते हैं.

### Learning method

हर pass की array लिखें. Comparisons, shifts, swaps और midpoint probes अलग count करें. Stability के लिए equal-key records पर original order देखें. Complexity answer के साथ algorithm version और assumptions बताएँ.

## Conventions और Assumptions

Programming language specify नहीं है; examples language-independent pseudocode हैं. Assignment  $\leftarrow$ , arrays zero-based और ascending order. For empty loop range, zero iterations. Error/status मिलने पर invalid index access न करें. Numeric examples key-based comparisons assume करते हैं; record sorting में full record move करें.

Quick Sort uses last-pivot Lomuto partition. Bubble uses early-exit flag. Merge is standard array version with reusable temp array. LSD Radix assumes nonnegative integers and base  $b \geq 2$ . Binary Search requires sorted input; basic version inclusive bounds और lower/upper bounds half-open intervals use करते हैं.

## Syllabus Mapping

| Topic                                     | Chapters |
|-------------------------------------------|----------|
| Sorting definition, types and properties  | 1-2      |
| Selection, Insertion and Bubble           | 3-5      |
| Quick, Merge and Radix                    | 6-8      |
| Comparison and algorithm choice           | 9        |
| Linear and Binary Search; useful variants | 10-12    |
| Solved counts, lab, revision and MCQs     | 13-14    |

“In-place” definition और recursion memory अलग से explained हैं. Exact comparisons depend on counting convention; radix bounds include digit count and base.

# विषय-सूची

## Contents / Chapters 1-14

|                                               |    |
|-----------------------------------------------|----|
| 01. Sorting का परिचय                          | 4  |
| 02. Types एवं Properties of Sorting           | 5  |
| 03. Selection Sort                            | 6  |
| 04. Insertion Sort                            | 8  |
| 05. Bubble Sort                               | 10 |
| 06. Quick Sort एवं Partition                  | 11 |
| 07. Merge Sort                                | 13 |
| 08. Radix Sort                                | 15 |
| 09. Sorting Methods का Comparison             | 17 |
| 10. Linear Search                             | 19 |
| 11. Binary Search                             | 21 |
| 12. Binary Search Variants एवं Boundary Cases | 23 |
| 13. Algorithm Lab एवं Solved Cost Questions   | 25 |
| 14. Complete Revision एवं Unit Test           | 27 |

## UNIT IV / CHAPTER 01

# अध्याय 1: Sorting का परिचय

## Unordered records से meaningful order तक

**Hindi समझ:** Sorting में data को किसी key के अनुसार क्रम में रखते हैं। Marks ascending, names alphabetical और dates chronological examples हैं। सही algorithm चुनने से time, memory तथा equal-key records का order प्रभावित होता है।

### 1.1 Sorting की Definition

Sorting records को chosen ordering relation के अनुसार arrange करने की प्रक्रिया है। Ascending numerical order में  $A[i] \leq A[i+1]$ ; descending में  $A[i] \geq A[i+1]$ । Key वह field है जिसके अनुसार comparison होता है। Entire record move होना चाहिए, केवल key नहीं, यदि उससे associated information है।

**Example:** Student record(name, marks). Marks पर sorting करते समय name उसी student's marks के साथ रहना चाहिए। [40, 10, 30, 20] ascending → [10, 20, 30, 40].

### 1.2 Why Sorting?

Sorted data में ordered reporting, rank lists, grouping और binary search आसान होती है। Searching और sorting अलग operations हैं: search required item ढूँढती है; sorting पूरे collection का order बदलती है। पहले sorting करने की cost को query benefit से अलग evaluate करें।

### 1.3 Input Size एवं Cost Model

$n$  logical elements. Time complexity comparison/move operations के growth को बताती है; actual seconds hardware, key comparison cost और implementation पर depend करते हैं। Auxiliary space input array के अतिरिक्त working memory है। Output copy भी extra memory है जब representation अलग हो।

| Measure         | What to count                         |
|-----------------|---------------------------------------|
| Comparisons     | Keys compare कितनी बार हुए            |
| Swaps           | Two elements exchange                 |
| Moves/writes    | One location पर assignment            |
| Passes          | Algorithm का specified round          |
| Auxiliary space | Temporary arrays, recursion, counters |

### 1.4 Correctness Conditions

Output non decreasing होना चाहिए और input का permutation होना चाहिए: same elements, same multiplicities. केवल sorted order check करना पर्याप्त नहीं; accidentally deleted elements से भी sorted-looking output बन सकती है। Records की associated payload integrity भी check करें।

### 1.5 Book Conventions

Arrays zero-based; ascending order;  $n$  may be 0. Pseudocode language-independent, directly runnable source नहीं। ← assignment; = comparison; loop range empty हो तो zero iterations. Negative indices allowed नहीं। Radix chapter nonnegative integers assume करता है; other comparison sorts negative values handle कर सकते हैं।

**Practice:** [3, 1, 3, 2] का sorted result [1, 2, 3, 3]. Explain why duplicate 3 हटाना sorting नहीं है।

## UNIT IV / CHAPTER 02

# अध्याय 2: Types एवं Properties of Sorting

## Internal, external, stable और in-place

**Hindi समझ:** “Fast” अकेला selection criterion नहीं है. Data RAM में है या disk पर, equal keys का पुराना order चाहिए या नहीं, और extra memory कितनी है-इन बातों से algorithm choice बदलती है.

### 2.1 Internal versus External

Internal sorting: data fits working main memory. External sorting: data memory से बड़ा, secondary-storage I/O प्रमुख cost है; sorted runs और merge passes common technique. External mergesort uses blocks/files, not identical to recursive in-memory code though merge idea shared.

### 2.2 Stability

Stable sort equal-key records का relative input order preserve करती है. Input[(A, 70), (B, 60), (C, 70)] marks ascending stable output[(B, 60), (A, 70), (C, 70)]. A, C samekey70; A remains before C.

| Property         | Meaning                                   |
|------------------|-------------------------------------------|
| Stable           | Equal-key input order preserved           |
| In-place         | Here strict $\Theta(1)$ auxiliary storage |
| Adaptive         | Existing order can reduce work            |
| Comparison-based | Keys order comparisons से decide          |
| Non-comparison   | Key structure/digits exploit करें         |

Some texts call array-rearranging Quick Sort “in-place” despite recursion storage. This book distinguishes array work space from total auxiliary space and includes recursion in complexity tables.

### 2.3 Adaptive Examples

Insertion Sort nearly sorted data पर fewer shifts करती है. Optimized Bubble Sort sorted input पर early stop करती है. Standard Selection Sort comparisons input order से नहीं घटती. Standard Merge Sort always divides/merges; special optimized versions can exploit order, but table basic version के लिए है.

### 2.4 Comparison Lower Bound

General comparison sorting of arbitrary distinct keys needs  $\Omega(n \log n)$  comparisons in worst case. Decision tree must distinguish  $n!$  permutations;  $\text{height} \geq \lceil \log_2(n!) \rceil$ . This does not imply every algorithm best case  $\Omega(n \log n)$ , nor prohibit linear-time digit/range-based sorts under extra key assumptions.

### 2.5 Multi-key Sorting

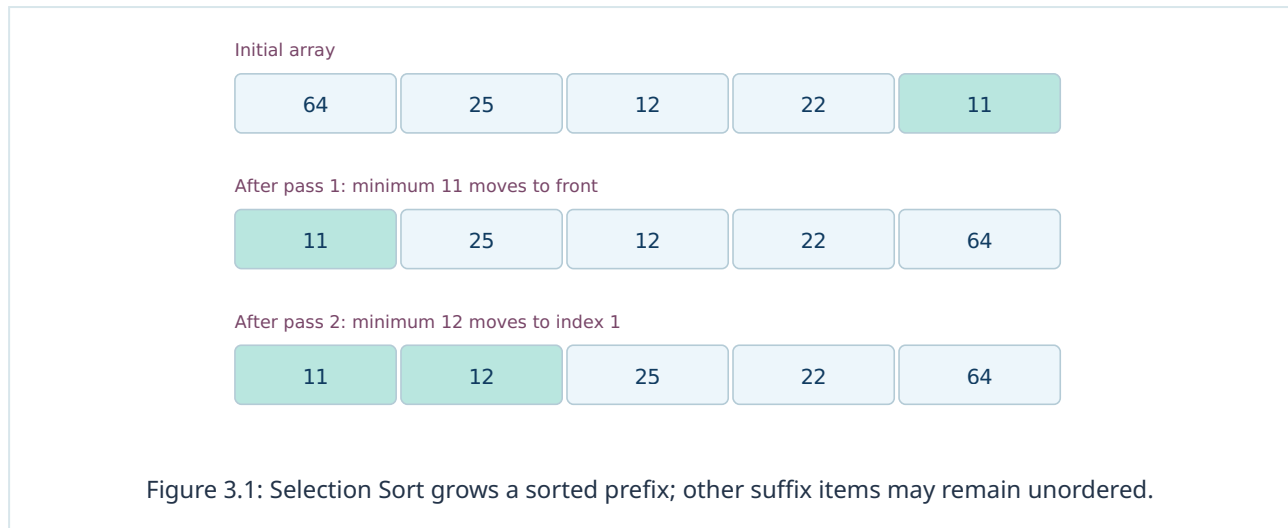
To order by department then marks, stable sort by secondary key marks first, then primary key department. Or compare tuple(department, marks) directly. Stability useful when earlier ordering must survive later equal-primary-key sorting.

**Practice:** Stable versus unstable is about equal keys, not ascending versus descending. A stable algorithm can produce either direction under suitable comparison rule.

## UNIT IV / CHAPTER 03

# अध्याय 3: Selection Sort

## Select minimum और sorted prefix बढ़ाएँ



**Hindi समझ:** हर pass में unsorted भाग का smallest element खोजकर उसे सही position पर लाते हैं। Comparisons कई होती हैं, लेकिन swaps कम-अधिकतम  $n-1$  successful outer passes में एक-एक swap.

### 3.1 Algorithm

```

SELECTION_SORT(A, n)
  for i ← 0 to n-2
    smallest ← i
    for j ← i+1 to n-1
      if A[j] < A[smallest]
        smallest ← j
    if smallest ≠ i
      swap A[i], A[smallest]

```

After pass  $i$ , indices  $0..i$  contain  $i+1$  smallest keys in sorted order. Remaining suffix retains remaining items. Empty/singleton arrays do no work.

### 3.2 Dry Run: [64, 25, 12, 22, 11]

| Pass | Minimum chosen        | Array after pass     |
|------|-----------------------|----------------------|
| i=0  | 11                    | [11, 25, 12, 22, 64] |
| i=1  | 12                    | [11, 12, 25, 22, 64] |
| i=2  | 22                    | [11, 12, 22, 25, 64] |
| i=3  | 25 already positioned | [11, 12, 22, 25, 64] |

Comparisons  $4+3+2+1=10$ . Actual swaps 3 because final minimum already at index 3. Maximum swaps  $n-1$ , not always exactly  $n-1$  when swap skipped.

### 3.3 Complexity

Total comparisons  $n(n-1)/2$  regardless initial order. Best, average, worst  $\Theta(n^2)$ , auxiliary  $\Theta(1)$ . Standard swap-based Selection Sort is not stable and not adaptive.

### 3.4 Stability Counterexample

$[(2, A), (2, B), (1, C)]$  keys ascending. First swap minimum 1 with first 2  $\rightarrow [(1, C), (2, B), (2, A)]$ . Equal key 2 records reverse order. Choosing first minimum in inner loop doesn't prevent this crossing swap.

A stable shifting-based variant can exist: remove minimum and shift intervening items. That variant changes move behavior; table refers to standard swaps above.

### 3.5 When Useful?

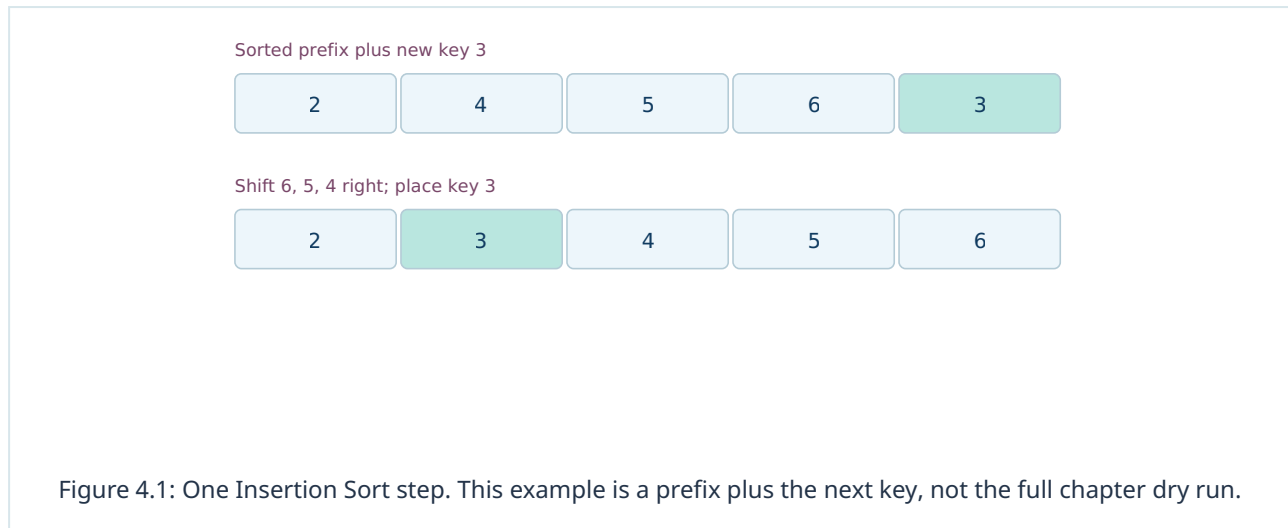
Simple, predictable, low swap count. Useful teaching baseline; may suit circumstances writes expensive relative comparisons, but actual choice depends implementation. Usually inefficient for large arrays compared  $O(n \log n)$  methods.

**Practice:**  $n=6$  comparisons 15. Sorted input still 15 comparisons but zero swaps with skip condition. Trace  $[3, 2, 1]$ : first pass  $[1, 2, 3]$ , next no swap.

## UNIT IV / CHAPTER 04

# अध्याय 4: Insertion Sort

## Sorted prefix में next key insert करें



**Hindi समझ:** Cards व्यवस्थित करने की तरह हर नया key पहले से sorted prefix में सही जगह डालते हैं. बड़े elements को right shift करते हैं. Nearly sorted arrays में shifts कम होती हैं.

### 4.1 Algorithm

```

INSERTION_SORT(A, n)
  for i ← 1 to n-1
    key ← A[i]
    j ← i-1
    while j ≥ 0 and A[j] > key
      A[j+1] ← A[j]
      j ← j-1
    A[j+1] ← key

```

Short-circuit evaluation आवश्यक:  $j \geq 0$  check before  $A[j]$ . Pseudocode's left condition fail होने पर right not accessed. Prefix 0..i sorted after each pass.

### 4.2 Dry Run: [5, 2, 4, 6, 1, 3]

| i / key | Array after insertion |
|---------|-----------------------|
| 1 / 2   | [2, 5, 4, 6, 1, 3]    |
| 2 / 4   | [2, 4, 5, 6, 1, 3]    |
| 3 / 6   | [2, 4, 5, 6, 1, 3]    |
| 4 / 1   | [1, 2, 4, 5, 6, 3]    |

| i / key | Array after insertion |
|---------|-----------------------|
| 5 / 3   | [1, 2, 3, 4, 5, 6]    |

Shifts  $1+1+0+4+3=9$ . Key assignments not counted as shifts here. Comparisons and writes are different measures.

### 4.3 Complexity एवं Inversions

Sorted input:  $n-1$  failed key-order checks, no shifts  $\rightarrow \Theta(n)$ . Reverse distinct input: shifts  $n(n-1)/2 \rightarrow \Theta(n^2)$ . Average random distinct order  $\Theta(n^2)$ . Auxiliary  $\Theta(1)$ .

Inversion = pair  $iA[j]$ . This insertion algorithm performs exactly one right shift per inversion. Work  $\Theta(n+I)$  under unit-cost operations,  $I$  inversion count. Equal keys not strict inversions.

### 4.4 Stability

Condition  $A[j] > \text{key}$  shifts strictly larger keys; equal earlier key stays before later key. Replacing  $>$  with  $\geq$  can reverse equal records and lose stability.

### 4.5 Binary Insertion Variant

Binary search may reduce position-search comparisons to  $O(\log i)$  per insert, but shifts still worst  $\Theta(n^2)$  total. For stable variant insert after existing equals (upper bound). Fewer comparisons does not automatically mean linear movement cost.

**Practice:** [1, 2, 3, 4] best case  $\Theta(n)$ ; [4, 3, 2, 1] shifts 6. Count shifts on [2, 1, 3]: 1.

## UNIT IV / CHAPTER 05

# अध्याय 5: Bubble Sort

## Adjacent swaps और early exit

**Hindi समझ:** साथ-साथ रखे गलत-order elements swap करते हैं। Left-to-right pass का सबसे बड़ा remaining element अंत में पहुँचता है। यदि पूरे pass में swap न हो, array sorted है और algorithm रुक सकता है।

### 5.1 Optimized Algorithm

```
BUBBLE_SORT(A, n)
  for end ← n-1 downto 1
    swapped ← false
    for j ← 0 to end-1
      if A[j] > A[j+1]
        swap A[j], A[j+1]
        swapped ← true
    if swapped = false
      break
```

Sorted suffix grows from right. Swapped reset for each pass. First j range empty for  $n \leq 1$  so arrays safe.

### 5.2 Dry Run: [5, 1, 4, 2, 8]

| Pass / end | Array after pass | Swapped?  |
|------------|------------------|-----------|
| 1 / 4      | [1, 4, 2, 5, 8]  | Yes       |
| 2 / 3      | [1, 2, 4, 5, 8]  | Yes       |
| 3 / 2      | [1, 2, 4, 5, 8]  | No → stop |

Comparisons  $4+3+2=9$  for shown optimized run. Plain shrinking-loop version without flag would also run  $end-1:10$  comparisons. State algorithm before quoting exact counts.

### 5.3 Complexity एवं Stability

Optimized  $\text{best } \Theta(n)$ , average/worst  $\Theta(n^2)$ , auxiliary  $\Theta(1)$ . Without early exit,  $\text{best } \Theta(n^2)$ . Swap only when  $>$  preserves equal order, so stable. Adjacent inverted pair swap reduces inversion count by 1; total actual swaps equals initial inversions.

### 5.4 Invariant

After pass  $\text{end}=n-1$ , maximum at last index. Next pass needn't compare that final lot. On no-swap pass, every checked adjacent pair ordered; suffix already final, hence whole array ordered.

### 5.5 Comparison with Insertion

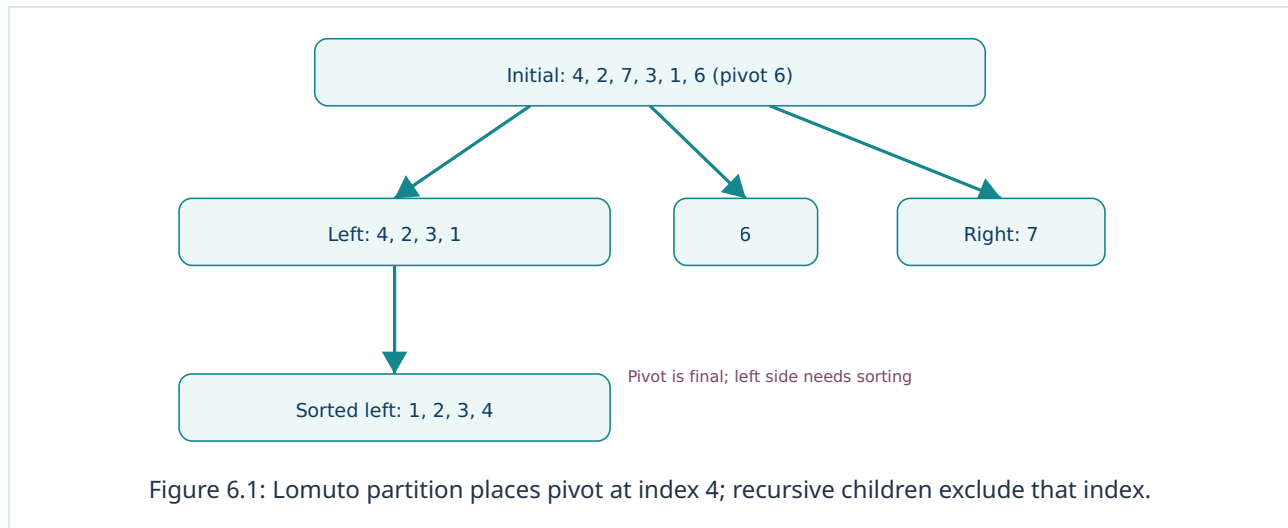
Both stable, strict constant auxiliary space and adaptive in common forms. Both quadratic worst case. Bubble uses adjacent swaps; insertion shifts larger prefix items. In practice insertion often preferred for small/nearly sorted segments; measurement depends workload.

**Practice:** Sorted 5 elements optimized first pass 4 comparisons, zero swaps then stop. Reverse 5 elements has 10 inversions and 10 swaps.

## UNIT IV / CHAPTER 06

# अध्याय 6: Quick Sort एवं Partition

## Pivot से problem divide करना



**Hindi समझ:** Partition pivot को ऐसी position देता है जिसके बाएँ छोटे/बराबर और दाएँ बड़े keys हों। फिर दोनों parts sort करते हैं। खराब pivot choices से balanced split नहीं मिलता और time quadratic हो सकती है।

### 6.1 Lomuto Partition Convention

Last element pivot; indices low..high inclusive. Other partition schemes differ; this book uses Lomuto. Final pivot index  $p$  belongs neither recursive child range.

```

PARTITION(A, low, high)
    pivot ← A[high]
    i ← low-1
    for j ← low to high-1
        if A[j] ≤ pivot
            i ← i+1
            swap A[i], A[j]
    swap A[i+1], A[high]
    return i+1

QUICK_SORT(A, low, high)
    if low < high
        p ← PARTITION(A, low, high)
        QUICK_SORT(A, low, p-1)
        QUICK_SORT(A, p+1, high)
    
```

Initial call low 0, high  $n-1$ . Empty array high  $= -1$ ; base case prevents access. Partition invariant processed  $A[\text{low}..i] \leq \text{pivot}$ ,  $A[i+1..j-1] > \text{pivot}$ .

## 6.2 Dry Run: [4, 2, 7, 3, 1, 6]

Pivot 6. Scan 4, 2 → accept; 7 → leave; 3 swaps with 7 → [4, 2, 3, 7, 1, 6]; 1 swaps with 7 → [4, 2, 3, 1, 7, 6].  
Final swap pivot with 7 → [4, 2, 3, 1, 6, 7], p 4. Left 0..3 sorts to [1, 2, 3, 4]; right 5..5 already done. Final [1, 2, 3, 4, 6, 7].

## 6.3 Time Complexity

Balanced partitions  $T(n) = 2T(n/2) + \Theta(n) \rightarrow \Theta(n \log n)$ . Expected  $\Theta(n \log n)$  with suitable randomized pivot selection or random-input assumptions. Worst  $T(n) = T(n-1) + \Theta(n) \rightarrow \Theta(n^2)$ .

Last pivot on already ascending distinct input produces extremely unbalanced partitions. All-equal input with shown  $\leq$  Lomuto also produces  $p = \text{high}$  repeatedly:  $\Theta(n^2)$ . Random pivot doesn't fix all-equal Lomuto; three-way partition addresses duplicate-heavy inputs.

## 6.4 Space एवं Stability

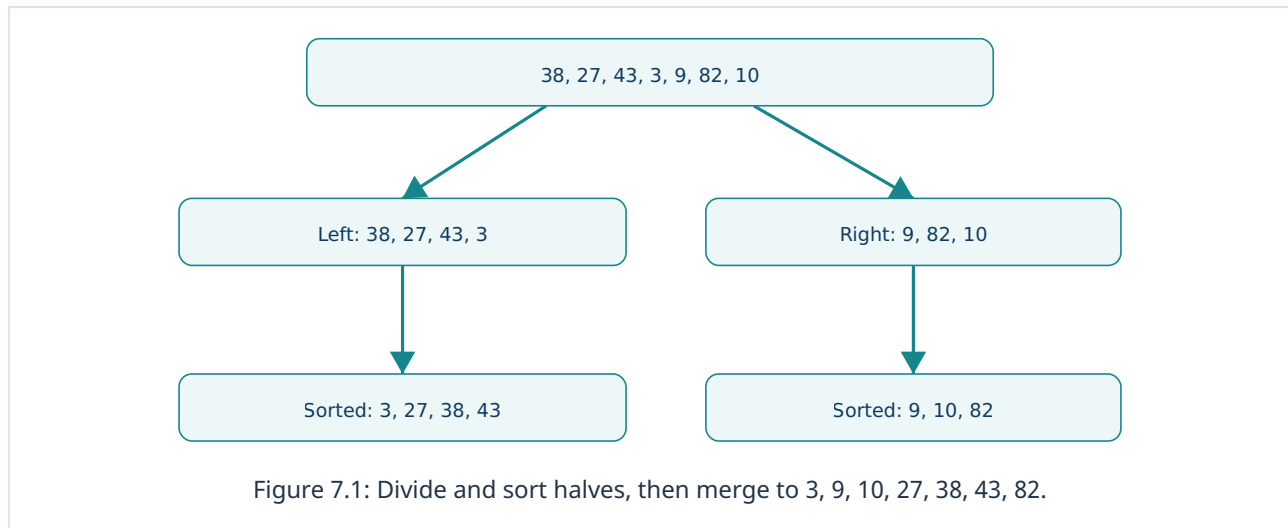
Partition uses  $\Theta(1)$  array work space. Naive recursion expected/balanced  $O(\log n)$ , worst  $O(n)$  call stack. Thus not strict  $\Theta(1)$  total space. Recurse on smaller partition and iterate larger part can limit stack to  $O(\log n)$  worst case. Standard Quick Sort unstable.

**Practice:** [2A, 2B, 1C] first partition with 1 swaps first/last → [1C, 2B, 2A]; equal 2 records reversed. Sortedness is not stability.

## UNIT IV / CHAPTER 07

# अध्याय 7: Merge Sort

## Divide, recursively sort, then merge



**Hindi समझ:** Array को दो parts में बाँटकर separately sort करते हैं; फिर दो sorted parts को merge करते हैं। Equal keys पर left part का item पहले लेने से stability बनी रहती है। Standard array version extra temporary storage उपयोग करती है।

### 7.1 Recursive Algorithm

```

MERGE_SORT(A, left, right, temp)
  if left ≥ right: return
  mid ← left + floor((right-left)/2)
  MERGE_SORT(A, left, mid, temp)
  MERGE_SORT(A, mid+1, right, temp)
  MERGE(A, left, mid, right, temp)

MERGE(A, left, mid, right, temp)
  i ← left; j ← mid+1; k ← left
  while i ≤ mid and j ≤ right
    if A[i] ≤ A[j]
      temp[k] ← A[i]; i ← i+1
    else
      temp[k] ← A[j]; j ← j+1
    k ← k+1
  while i ≤ mid
    temp[k] ← A[i]; i ← i+1; k ← k+1
  while j ≤ right
    temp[k] ← A[j]; j ← j+1; k ← k+1
  for k ← left to right
    A[k] ← temp[k]
  
```

Allocate one temp array length n before initial call 0, n-1. Reuse across recursion. Empty/singleton base case. Do not forget remaining items after one half exhausted.

## 7.2 Dry Run: [38, 27, 43, 3, 9, 82, 10]

Split into [38, 27, 43, 3] and [9, 82, 10] with shown floor-mid rule. Left sorts [3, 27, 38, 43], right [9, 10, 82]. Merge choices 3, 9, 10, 27, 38, 43, then remaining 82. Final [3, 9, 10, 27, 38, 43, 82].

## 7.3 Merge Example

[2, 5, 8] and [1, 3, 7]: compare 2/1 → 1; 2/3 → 2; 5/3 → 3; 5/7 → 5; 8/7 → 7; append 8. At most  $a+b-1$  key comparisons when merging two nonempty sorted lists sizes  $a, b$ ; actual maybe fewer.

## 7.4 Complexity एवं Stability

Basic balanced top-down Merge Sort best/average/worst  $\Theta(n \log n)$ ,  $n \geq 2$ . Merge work  $\Theta(n)$  per level,  $O(\log n)$  levels. Temp  $\Theta(n)$  plus call stack  $O(\log n)$ , total auxiliary  $\Theta(n)$ . Stable when left item chosen on equal  $\leq$ . Usual array implementation not in-place under strict definition.

Linked list Merge Sort can relink nodes and avoid  $\Theta(n)$  temp array; recursion stack still matters. External merging adapted for files/blocks useful for data that doesn't fit RAM. Advanced in-place variants exist, but table does not claim them.

**Practice:** Choosing right item on equal can break stability across halves. Explain why left  $\leq$  test matters.

## UNIT IV / CHAPTER 08

# अध्याय 8: Radix Sort

## Stable digit passes और non-comparison sorting

Units pass

|     |    |     |   |    |    |    |    |
|-----|----|-----|---|----|----|----|----|
| 170 | 90 | 802 | 2 | 24 | 45 | 75 | 66 |
|-----|----|-----|---|----|----|----|----|

Tens pass

|     |   |    |    |    |     |    |    |
|-----|---|----|----|----|-----|----|----|
| 802 | 2 | 24 | 45 | 66 | 170 | 75 | 90 |
|-----|---|----|----|----|-----|----|----|

Hundreds pass: final sorted order

|   |    |    |    |    |    |     |     |
|---|----|----|----|----|----|-----|-----|
| 2 | 24 | 45 | 66 | 75 | 90 | 170 | 802 |
|---|----|----|----|----|----|-----|-----|

Figure 8.1: Stable LSD digit passes in base 10. Values are whole records, not isolated digits.

**Hindi समझ:** Radix Sort पूरे number की तुलना करने के बजाय digits पर stable sorting passes करती है। LSD version units digit से शुरू होकर higher digits तक जाती है। हर pass stable होना चाहिए, तभी पहले pass का useful order बचता है।

### 8.1 Assumptions एवं Meaning

यह implementation nonnegative integers, integer base  $b \geq 2$  और exact integer arithmetic assume करती है। Digit at exponent  $\text{exp} = \text{floor}(\text{value}/\text{exp}) \bmod b$ . Decimal example  $b=10$ ; passes units, tens, hundreds. Missing leading digits 0. Negative integers के लिए offset/sign-aware handling अलग चाहिए; shown algorithm directly negative values पर valid नहीं।

LSD = least significant digit first. MSD starts most significant side and uses different grouping/recursion. This chapter's pseudocode LSD है।

### 8.2 Stable Counting Pass

```

COUNTING_DIGIT_PASS(A, n, exp, b)
    count[0..b-1] ← all zeros
    output[0..n-1] ← temporary array
    for i ← 0 to n-1
        digit ← floor(A[i]/exp) mod b
        count[digit] ← count[digit]+1
    for digit ← 1 to b-1
        count[digit] ← count[digit]+count[digit-1]
    for i ← n-1 downto 0
        digit ← floor(A[i]/exp) mod b
        output[count[digit]-1] ← A[i]
        count[digit] ← count[digit]-1
    copy output into A

```

```
RADIX_SORT_LSD(A, n, b)
```

```

require integer  $b \geq 2$  and every  $A[i] \geq 0$  integer
if  $n = 0$ : return
maximum  $\leftarrow \max(A)$ 
exp  $\leftarrow 1$ 
repeat
    COUNTING_DIGIT_PASS( $A, n, \text{exp}, b$ )
    if  $\text{floor}(\text{maximum}/\text{exp}) < b$ : break
    exp  $\leftarrow \text{exp} * b$ 

```

Reverse input scan with cumulative ending positions preserves equal-digit input order. Forward scan with this same placement rule would reverse equals; other correct stable variants can use starting positions instead.

Shown loop executes one pass for all-zero array. Stop check before exponent multiplication avoids unnecessary next power; bounded integer implementations should still define their arithmetic limits and allocations.

### 8.3 Dry Run: [170, 45, 75, 90, 802, 24, 2, 66]

| Pass | Digit key | Array after stable pass           |
|------|-----------|-----------------------------------|
| 1    | Units     | [170, 90, 802, 2, 24, 45, 75, 66] |
| 2    | Tens      | [802, 2, 24, 45, 66, 170, 75, 90] |
| 3    | Hundreds  | [2, 24, 45, 66, 75, 90, 170, 802] |

Units digit0:170 stays before90. Tens digit7:170 stays before75. These examples show stability across passes, not whole-array sorting after first pass.

### 8.4 Complexity

$d$  digits of maximum, treating zero as one digit. Per pass  $\Theta(n+b)$ ; total  $\Theta(d(n+b))$ , auxiliary  $\Theta(n+b)$ . If  $b$  fixed,  $d$  bounded independently of  $n$ , time can be  $\Theta(n)$ . If digit count grows with key range, include  $d$ ; "Radix always  $O(n)$ " incomplete.

Comparison  $\Omega(n \log n)$  bound doesn't apply because algorithm exploits digit/range structure rather than only comparisons. Large base requires larger count array. Strings or negative values need appropriate adaptations, not blind reuse.

**Practice:** [0, 10, 1] units  $\rightarrow$  [0, 10, 1], tens  $\rightarrow$  [0, 1, 10]. All-zero input stays unchanged. State why each digit pass must be stable.

## UNIT IV / CHAPTER 09

# अध्याय 9: Sorting Methods का Comparison

## Best, average, worst एवं practical choice

**Hindi समझ:** Complexity table याद करते समय version और assumptions भी याद रखें. Algorithm stable है या नहीं, memory कितनी लेती है और input पहले से ordered है या नहीं-इन सबका उत्तर अलग होता है.

### 9.1 Time Comparison

For  $n \geq 2$ ; usual random-order assumptions for quadratic average cases:

| Method                 | Best                        | Average / expected                       | Worst                        |
|------------------------|-----------------------------|------------------------------------------|------------------------------|
| Selection, standard    | $\Theta(n^2)$               | $\Theta(n^2)$                            | $\Theta(n^2)$                |
| Insertion, shown       | $\Theta(n)$                 | $\Theta(n^2)$                            | $\Theta(n^2)$                |
| Bubble, early exit     | $\Theta(n)$                 | $\Theta(n^2)$                            | $\Theta(n^2)$                |
| Quick, shown partition | $\Theta(n \log n)$ balanced | $\Theta(n \log n)$ , stated random model | $\Theta(n^2)$                |
| Merge, basic array     | $\Theta(n \log n)$          | $\Theta(n \log n)$                       | $\Theta(n \log n)$           |
| LSD Radix              | $\Theta(d(n+b))$            | $\Theta(d(n+b))$                         | $\Theta(d(n+b))$ , key model |

Radix  $d$ ,  $b$  depend on keys and chosen base; its "average" is not directly same comparison model. Empty/singleton cases handled by base checks; asymptotic tables describe increasing  $n$ .

### 9.2 Space एवं Stability

| Method      | Auxiliary space                                    | Stable in shown form?     |
|-------------|----------------------------------------------------|---------------------------|
| Selection   | $\Theta(1)$                                        | No                        |
| Insertion   | $\Theta(1)$                                        | Yes, strict > shift       |
| Bubble      | $\Theta(1)$                                        | Yes, strict > swap        |
| Quick       | Expected/balanced $O(\log n)$ , worst $O(n)$ stack | No                        |
| Merge array | $\Theta(n)$                                        | Yes, left chosen on equal |
| LSD Radix   | $\Theta(n+b)$                                      | Yes, stable passes        |

"In-place" exact definition differs across sources. Shown Quick Sort uses constant partition work space but nonconstant recursion stack. This distinction is better than one label alone.

### 9.3 Choose by Situation

| Situation                                         | Reasonable choice / consideration         |
|---------------------------------------------------|-------------------------------------------|
| Small/nearly sorted array                         | Insertion; few shifts                     |
| Need stable guaranteed $n \log n$ comparison sort | Merge, memory available                   |
| Generic array, good average performance           | Quick, pivot/duplicates/stack addressed   |
| Nonnegative fixed-digit integers                  | LSD Radix, memory/key assumptions valid   |
| Predictable low swap count                        | Selection, but quadratic comparisons      |
| Data larger than RAM                              | External merge approach with I/O analysis |

No universal faster algorithm; key comparison cost, input distribution, locality and implementation matter. Hybrid libraries may combine strategies, but this book teaches core methods.

### 9.4 Exam Traps

Average and amortized are different terms. Sorting stability doesn't guarantee smallest memory. Worst  $n \log n$  guarantee not given by ordinary Quick Sort. Sorted input is good for insertion but bad for last pivot Lomuto with distinct keys. Radix stable pass essential.

**Practice:** Choose for payroll records need equal-salary original order: stable method, e.g. Merge if memory available; explain constraints rather than only name.

## UNIT IV / CHAPTER 10

# अध्याय 10: Linear Search

## Sequential comparisons और first-match policy



**Hindi समझ:** Linear Search एक-एक element check करती है. Data sorted होना जरूरी नहीं. यदि duplicate keys हों, नीचे वाला algorithm first matching index return करता है.

### 10.1 Algorithm

```
LINEAR_SEARCH(A, n, target)
  for i ← 0 to n-1
    if A[i] = target
      return i
  return NOT_FOUND
```

NOT\_FOUND distinct status/sentinel; if using -1 for index, valid indices 0..n-1 so ambiguity not with stored value -1. A[-1] access after failure must be avoided. Empty array returns failure without comparison.

### 10.2 Dry Run

A=[18, 7, 25, 7, 30], target 7: compare 18 then 7 → return index 1, two comparisons. Target 30 → index 4, five comparisons. Target 9 → NOT\_FOUND, five comparisons. To collect all matches, continue scanning instead of early return.

### 10.3 Complexity

Best successful  $\Theta(1)$ , worst successful/unsuccessful  $\Theta(n)$ , auxiliary  $\Theta(1)$ . If successful target equally likely at each of  $n$  positions, average comparisons  $(n+1)/2$ . This assumes exactly selected occurrence position model; duplicate target first match can change distribution.

Uniform successful plus failure probability  $p$ : expected comparisons  $= (1-p)(n+1)/2 + pn$  understated uniform model. Without probability assumptions average exact count not determined.

#### 10.4 Sorted-data Early stop Variant

For ascending sorted array, if  $A[i] > \text{target}$ , target cannot appear later; may return failure early. Worst still  $\Theta(n)$ . Plain algorithm above requires no sorting and performs  $n$  comparisons for absent target.

#### 10.5 Arrays एवं Linked Lists

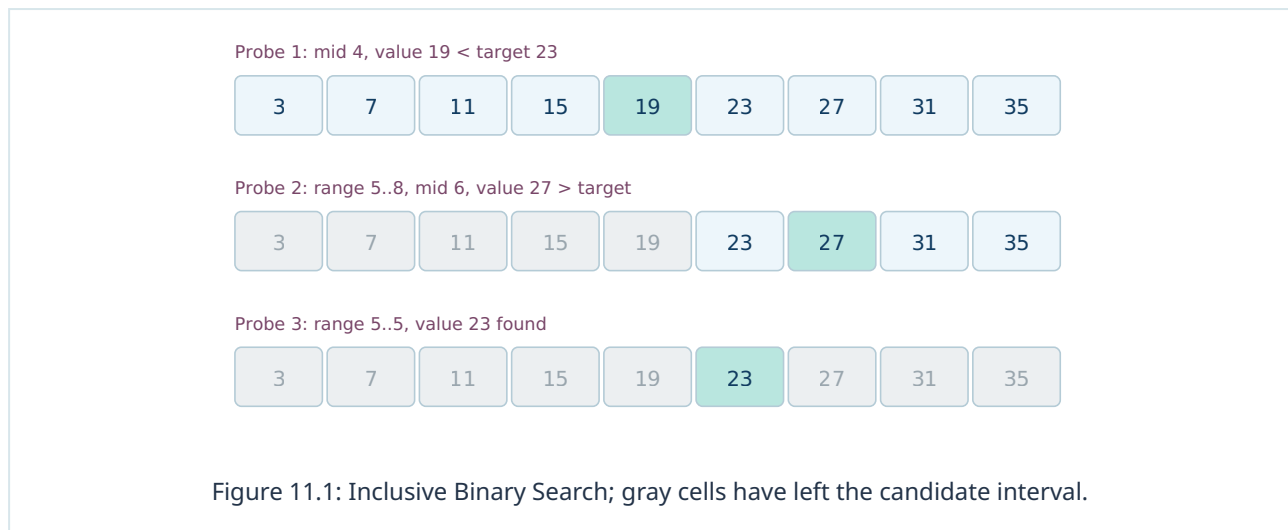
Sequential scan works with arrays or linked nodes. Linked search visits nodes via references; can't use direct  $A[i]$  notation unless representation array. Linear search good for few queries, small collections or unsorted data.

**Practice:**  $n=9$ , successful uniform average 5 comparisons. Absent target plain linear search 9. Negative value target does not confuse index failure status.

## UNIT IV / CHAPTER 11

# अध्याय 11: Binary Search

## Sorted array का search interval halve करें



**Hindi समझ:** Binary Search के लिए ordering आवश्यक है। Middle key से comparison करके आधा candidate interval छोड़ते हैं। Sorted array पर fast search मिलती है; unsorted array पर वही code सही answer की guarantee नहीं देता।

### 11.1 Preconditions

Ascending sorted array; efficient index access; comparison consistent with sort order. Duplicates allowed, but basic version may return any matching index. Descending order needs reversed comparison updates.

```
BINARY_SEARCH(A, n, target)
  low ← 0
  high ← n-1
  while low ≤ high
    mid ← low + floor((high-low)/2)
    if A[mid] = target
      return mid
    else if A[mid] < target
      low ← mid+1
    else
      high ← mid-1
  return NOT_FOUND
```

Inclusive bounds. Equality low=high still one candidate. Use mid+1/mid-1 to ensure progress; keeping mid can infinite-loop. Safer mid formula avoids low+high overflow for valid bounded indices.

### 11.2 Solved Trace

A=[3, 7, 11, 15, 19, 23, 27, 31, 35], target 23:

| Step | low | high | mid | A[mid] | Decision      |
|------|-----|------|-----|--------|---------------|
| 1    | 0   | 8    | 4   | 19     | Search5..8    |
| 2    | 5   | 8    | 6   | 27     | Search5..5    |
| 3    | 5   | 5    | 5   | 23     | Found index 5 |

Target4: mid4→high3;mid1→high0;mid0→low1; low>high→not found. Three midpoint probes in this example.

### 11.3 Complexity एवं Counting Convention

Best $\Theta(1)$ , average/worst $\Theta(\log n)$  for  $n \geq 2$ , iterative auxiliary $\Theta(1)$ . Recursive version uses  $O(\log n)$  call stack. Maximum midpoint probes nonempty  $n = \lceil \log_2 n \rceil + 1 = \lceil \log_2(n+1) \rceil$  under this inclusive halving model. Empty  $n$  0 probes0.

A midpoint probe may perform one equality test and one order test in shown code. “Probes” and “individual key comparison operations” are different counts. Some MCQs say comparisons intending probes; define your count.

### 11.4 Sorted Precondition Trap

Unsorted[10, 1, 8, 3, 6]can mislead interval discard. Sort once if many queries justify cost, or linear search without reordering. Sorting changes indices; if original positions needed, keep(index, key)records or mapping.

### 11.5 Linked-list Limitation

A plain linked list lacks $\Theta(1)$  midpoint access. Repeated position traversals remove array binary search's usual benefit; an ordered container does not automatically give $\Theta(\log n)$  time. Use appropriate indexed structure/BST for fast navigation.

**Practice:**  $n=16$  maximum5 midpoint probes.  $n=15$  maximum4. Explain why not always exactly  $\log_2 n$  for successful search.

## UNIT IV / CHAPTER 12

# अध्याय 12: Binary Search Variants एवं Boundary Cases

## First/last occurrence एवं insertion position

**Hindi समझ:** Duplicate keys में basic Binary Search कोई matching index दे सकती है. First या last occurrence चाहिए तो मिलने के बाद भी एक तरफ search जारी रखें. Lower/upper bound insertion positions भी देते हैं.

### 12.1 Lower Bound: Half-open Interval

$\text{lowerBound}(x)$  = first index with  $A[\text{index}] \geq x$ , or  $n$  if none. Candidate interval  $[\text{low}, \text{high})$ ,  $\text{low} = 0$ ,  $\text{high} = n$ .  $\text{high}$  excluded.

```
LOWER_BOUND(A, n, x)
  low ← 0; high ← n
  while low < high
    mid ← low + floor((high-low)/2)
    if A[mid] < x
      low ← mid+1
    else
      high ← mid
  return low
```

Return  $n$  is valid insertion position, not valid access index. First occurrence:  $p = \text{lowerBound}(x)$ ; if  $p < n$  and  $A[p] = x$ , found  $p$ ; else absent.

### 12.2 Upper Bound

$\text{upperBound}(x)$  = first index with  $A[\text{index}] > x$ , or  $n$  if none. Same algorithm except if  $A[\text{mid}] \leq x \rightarrow \text{low} \leftarrow \text{mid} + 1$ ; else  $\text{high} \leftarrow \text{mid}$ . Last occurrence =  $\text{upperBound}(x) - 1$ , after checking that  $\text{index} \geq 0$  and  $A[\text{index}] = x$ . Count occurrences =  $\text{upperBound} - \text{lowerBound}$ .

### 12.3 Solved Duplicate Example

$A = [1, 2, 2, 2, 4, 7]$ ,  $x_2$ : lower1, upper4, first1, last3, count3.  $x_3$ : lower4, upper4, count0.  $x_8$ : lower6, upper6; never read  $A[6]$ . Empty array both bounds0.

### 12.4 Inclusive versus Half-open

| Version               | Initial high | Loop                          | Left update                    |
|-----------------------|--------------|-------------------------------|--------------------------------|
| Basic inclusive       | $n-1$        | $\text{low} \leq \text{high}$ | $\text{high} = \text{mid} - 1$ |
| Lower bound half open | $n$          | $\text{low} < \text{high}$    | $\text{high} = \text{mid}$     |

Mixing conditions creates off-by-one errors. Two schemes both correct when internally consistent. Half-open interval updates always shrinks while keeping insertion candidate boundary.

## 12.5 Search-cost Planning

One unsorted query: sorting  $\Theta(n \log n)$  + binary  $O(\log n)$  typically not justified over linear scan  $\Theta(n)$  asymptotically.  $q$  queries: repeated linear  $O(qn)$ ; sort once + search  $O(n \log n + q \log n)$ . Factors and updates matter; asymptotic expression not fixed wall clock crossover.

## 12.6 Boundary Tests

Test  $n=0, n=1$ ; key before first, after last; missing between values; all keys identical; negative values; first/last found; duplicate  $d$  key. Verify both returned status and range before indexing.

**Practice:** Stable binary insertion uses upperBound for new key so existing equals stay before it. Find lower/upper on  $[5, 5, 5]$ :  $x=5 \rightarrow 0, 3$ .

## UNIT IV / CHAPTER 13

# अध्याय 13: Algorithm Lab एवं Solved Cost Questions

## Dry runs, comparisons और verification

**Hindi समझ:** Algorithm याद करने के साथ उसके steps की संख्या भी समझें। Comparisons, swaps और shifts अलग counts हैं। Test cases में sorted, reverse, duplicates, empty और single-element inputs जरूर रखें।

### 13.1 Exact Count Examples

| Question                       | Answer / assumption                          |
|--------------------------------|----------------------------------------------|
| Selection n 8 comparisons      | $8 \times 7 / 2 = 28$                        |
| Reverse insertion n 5 shifts   | $5 \times 4 / 2 = 10$ , distinct keys        |
| Reverse bubble n 4 swaps       | $4 \times 3 / 2 = 6$ , strict adjacent swaps |
| Sorted optimized bubble n 6    | 5 comparisons, 0 swaps, 1 pass               |
| Linear uniform successful n 11 | 6 comparisons average                        |
| Binary n 31 maximum probes     | 5                                            |
| Binary n 32 maximum probes     | 6                                            |
| Radix decimal max 802          | 3 digit passes                               |

### 13.2 Growth Numerical

For quadratic leading term, n doubles  $\rightarrow$  roughly  $4 \times$  work. For  $n \log_2 n$  from n 1024 to 2048:  $(2048 \times 11) / (1024 \times 10) = 2.2 \times$  leading work. This is not exact wall clock prediction or proof that constants irrelevant.

### 13.3 Merge Numerical

Merging 4 and 3 nonempty sorted items uses at most 6 key comparisons; writes 7 merge d items then may copy back 7. If [1, 2, 3, 4] and [5, 6, 7], key comparisons 4 before left exhausted, then append remaining. Sorting complexity is not just one final merge.

### 13.4 Quick Sort Partition Exercise

A = [9, 3, 7, 1, 5], last pivot 5: scan 9 skip; 3 swap  $\rightarrow$  [3, 9, 7, 1, 5]; 7 skip; 1 swap  $\rightarrow$  [3, 1, 7, 9, 5]; final pivot  $\rightarrow$  [3, 1, 5, 9, 7], p2. Pivot finally correct; both sides not yet fully sorted. Partition is not entire Quick Sort.

### 13.5 Practical Project: Sorting-Searching Workbench

Menu-based implementation in your chosen language: load integer array; choose one of six sorts; display sorted output; count defined comparisons/shifts/swaps; linear search on original; binary search on sorted copy. Radix option reject negative values or use clearly documented adaptation.

Store original copy to verify permutation and report original positions. For stable methods use records(key, originalIndex) and compare only key. Check equal key originalIndex ascending after stable output. Don't sort by tuple including index if you want to test algorithm's intrinsic stability.

### 13.6 Boundary Testing

| Input          | Expected property                           |
|----------------|---------------------------------------------|
| []             | No error; searches not found                |
| [5]            | Same array; found index 0for5               |
| [1, 2, 3, 4]   | Sorted preserved                            |
| [4, 3, 2, 1]   | Ascending output                            |
| [2, 1, 2, 1]   | Multiplicities preserved                    |
| [-3, 0, -1, 2] | Comparison sorts valid; shown Radix rejects |
| [0, 0, 0]      | Radix one pass, stable                      |

Correctness check: output ordered AND same value frequencies. Avoid comparing timings on very tiny runs and drawing universal faster claims; operation counts are clearer teaching evidence.

## UNIT IV / CHAPTER 14

# अध्याय 14: Complete Revision एवं Unit Test

## Methods, assumptions और exam preparation

**Hindi समझ:** Exam में algorithm के नाम के साथ उसकी version बताएं। Quick Sort की pivot policy, Bubble का early-exit flag, Radix का key/base model और Binary Search की sorted-input condition answer का हिस्सा हैं।

### 14.1 One-line Revision

Selection selects minimum; insertion shifts sorted prefix; bubble swaps adjacent inversions; Quick partitions around pivot; Merge divides and merges; LSD Radix stable-sorts from low digits. Linear checks sequentially; Binary halves sorted search range. Stability equal-key order है, sortedness नहीं।

### 14.2 Key Formulas

| Concept                           | Formula                                                   |
|-----------------------------------|-----------------------------------------------------------|
| Selection comparisons             | $n(n-1)/2$                                                |
| Reverse insertion shifts          | $n(n-1)/2$ , distinct keys                                |
| Bubble actual swaps               | Initial inversion count                                   |
| Uniform linear successful average | $(n+1)/2$                                                 |
| Inclusive binary maximum probes   | $\lfloor \log_2 n \rfloor + 1, n \geq 1$                  |
| Safe midpoint                     | $\text{low} + \text{floor}((\text{high} - \text{low})/2)$ |
| Occurrence count                  | $\text{upperBound} - \text{lowerBound}$                   |
| LSD Radix time                    | $\Theta(d(n+b))$                                          |

### 14.3 Unit Test: 40 MCQs

One best answer; algorithms and conventions as in the book.

- Sorting arranges records by: A. Chosen key order B. Always input index C. Deleting duplicates D. Only memory address
- Stability preserves: A. Key values must all unique B. Equal-key relative order C. Every old index D. Descending only
- Internal sorting data fits: A. Only network B. Nowhere C. Main memory D. Only DVD
- General comparison sort worst lower bound: A.  $\Omega(1)$  only B. Exactly  $\Theta(n^2)$  for all C. Always  $\Theta(n)$  D.  $\Omega(n \log n)$

5. Selection Sort chooses: A. Minimum remaining key B. Middle always C. Random target D. Last without comparison
6. Standard selection best time: A.  $\Theta(1)$  B.  $\Theta(n^2)$  C.  $\Theta(n)$  D.  $\Theta(\log n)$
7. Selection n 5 comparisons: A. 25 B. 4 C. 10 D. 5
8. Standard selection stability: A. Stable always B. No comparisons C. Only FIFO D. Unstable
9. Insertion best on sorted input: A.  $\Theta(n)$  B.  $\Theta(n^2)$  C.  $\Theta(n \log n)$  D.  $\Theta(\log n)$
10. Stable insertion loop condition: A. No index check B.  $A[j] > \text{key}$  C.  $A[j] \geq \text{key}$  D.  $A[j] = \text{key}$  only
11. Insertion right shift count equals: A. Number duplicates B. Array capacity C. Initial inversions D.  $n^2$  always
12. Optimized bubble stops if: A. One swap occurs B. n not zero C. First comparison false only D. No swap in pass
13. Sorted bubble with flag best: A.  $\Theta(n)$  B.  $\Theta(n^2)$  C.  $\Theta(n \log n)$  D.  $\Theta(2^n)$
14. Bubble equal key swapping with  $>$  is: A. Impossible B. Stable C. Unstable always D. Priority queue
15. Lomuto pivot in this book: A. Middle always B. No key C. Last key D. First always
16. Partition return p means: A. Whole array sorted always B. Minimum always C. Failure always D. Pivot final position
17. Quick Sort worst time: A.  $\Theta(n^2)$  B.  $\Theta(n)$  C.  $\Theta(\log n)$  D.  $\Theta(1)$
18. Last pivot ascending distinct input: A. Linear always B. Unbalanced splits C. Guaranteed balanced D. No recursion
19. All-equal shown  $\leq$  Lomuto: A. No partition B. Always stable C. Can be quadratic D. Guaranteed linear
20. Quick Sort naive recursion worst space: A.  $\Theta(1)$  total always B.  $\Theta(n^2)$  C. No stack D.  $O(n)$
21. Stable merge equal keys select: A. Left first B. Right first always C. Randomly D. Discard one
22. Basic array Merge Sort aux space: A.  $\Theta(2^n)$  B.  $\Theta(n)$  C.  $\Theta(1)$  total D.  $\Theta(n^2)$
23. Basic Merge Sort worst time: A.  $\Theta(n^3)$  B.  $\Theta(1)$  C.  $\Theta(n \log n)$  D.  $\Theta(n^2)$
24. Merge 4 and 3 items max key comparisons: A. 7 B. 12 C. 3 D. 6
25. LSD Radix starts with: A. Least significant digit B. Most always C. Median D. All keys comparison
26. Each LSD digit pass must be: A. Random B. Stable C. Unstable D. Descending only
27. Shown Radix supports: A. Arbitrary objects without keys B. No zero C. Nonnegative integers D. All negative inputs unchanged code
28. Radix complexity with d, b: A. Always  $\Theta(\log n)$  B. Always  $\Theta(n^2)$  C.  $\Theta(1)$  D.  $\Theta(d(n+b))$
29. Linear Search needs: A. No sorted precondition B. Sorted always C. Perfect BST D. Unique keys mandatory
30. Plain linear absent target comparisons: A. 0 always B. n C. 1 always D.  $\log n$  always
31. Uniform successful linear average n 9: A. 4 B. 1 C. 5 D. 9
32. Binary Search requires: A. Unsorted always B. Duplicate free always C. No array D. Matching order precondition
33. Inclusive binary initial high: A. n-1 B. n C. 0 always D. n+1
34. Binary when  $A[\text{mid}] < \text{target}$ : A. Return failure always B.  $\text{low} = \text{mid} + 1$  C.  $\text{high} = \text{mid} - 1$  D.  $\text{low} = \text{mid}$  always
35. Binary n 16 maximum probes: A. 16 B. 8 C. 5 D. 4
36. Basic binary duplicate key returns: A. First always B. Last always C. Count always D. Any matching index possible

37. Lower bound  $x$  returns: A.  $\text{First} \geq x$  position B.  $\text{First} > x$  always C.  $\text{Last} < x$  always D. Minimum key
38. Upper bound  $x$  returns: A. Mid always B.  $\text{First} > x$  position C.  $\text{First} \geq x$  always D.  $\text{Last} = x$  always
39. Occurrence count in sorted array: A.  $\text{mid} + 1$  B.  $\text{high} + \text{low}$  C.  $\text{upperBound} - \text{lowerBound}$  D.  $n - \text{lowerBound}$  always
40. Sorting then binary search original indices: A. Original indices always unchanged B. Duplicates must delete C. Cannot store indices D. Need mapping if original positions required

### उत्तर-कुंजी:

1-A, 2-B, 3-C, 4-D, 5-A, 6-B, 7-C, 8-D, 9-A, 10-B, 11-C, 12-D, 13-A, 14-B, 15-C, 16-D, 17-A, 18-B, 19-C, 20-D, 21-A, 22-B, 23-C, 24-D, 25-A, 26-B, 27-C, 28-D, 29-A, 30-B, 31-C, 32-D, 33-A, 34-B, 35-C, 36-D, 37-A, 38-B, 39-C, 40-D.

## 14.4 Written Exam Practice

**5 marks:** Stability and in-place; internal/external; inversion; pivot; merge operation; radix digit pass; linear versus binary search.

### 10 marks:

1. Selection, Insertion and Bubble algorithms with dry runs and costs.
2. Quick Sort partition, recursion, pivot policy and worst case example.
3. Merge Sort algorithm, merge example, stability and space analysis.
4. Radix Sort stable counting pass with decimal dry run and assumptions.
5. Compare all six sorts: time, space, stability, adaptivity.
6. Linear/Binary Search algorithms, trace, complexity and preconditions.
7. Duplicate binary search first/last occurrence and boundary cases.

## 14.5 Final Checklist

Can you sort the example by hand? Distinguish comparison from swap? Explain why  $>$  preserves equals? Trace partition with pivot position? Complete both merge-tail copies? Keep digit passes stable? Reject unsorted binary input? Handle empty/singleton/duplicates? Preserve original record payload? State versions and assumptions before quoting complexity? These checks link theory with correct implementation.