Sorting, Searching, and Recurrences

Algorithmics study and revision notes

Jaak Vilo

2026-08-22

Contents

1 Why ordering matters

Ordering pays an up-front cost to make later questions structured. A sorted array supports logarithmic search; a sorted stream can be merged in one pass; a partition can isolate one rank without sorting everything. These algorithms also introduce the central divide-and-conquer questions: how are subproblems formed, why does recombination preserve correctness, and how is work distributed across recursive levels?

2 Learning goals

After studying this note, you should be able to:

- derive and prove the invariant of `lower_bound`;
- specify sorting requirements such as stability and auxiliary space;
- prove the comparison-sorting lower bound;
- explain insertion sort, merge sort, and randomized three-way quicksort;
- solve standard divide-and-conquer recurrences with trees and the Master theorem;
- explain when counting, radix, bitwise, and bucket sorting avoid the lower bound;
- select an order statistic without fully sorting; and
- search and update a skip list under a stated random model.

3 Prerequisites

You should understand arrays, half-open ranges, loops, recursion, probability as expectation, asymptotic notation, and arithmetic/geometric sums. The note restates the invariants and cost assumptions needed for each algorithm.

4 Models and definitions

A sorting input is a sequence of **records** with **keys**. A comparator must behave consistently as a total order: comparisons cannot contradict one another. Sorting rearranges records into nondecreasing key order while preserving exactly the input multiset.

A sort is **stable** if equal-key records retain their original relative order. It is **in-place** when it uses only a small amount of auxiliary storage beyond the input; state whether recursion-stack space is included. An **adaptive** algorithm benefits from existing order.

The **comparison model** learns about arbitrary keys only through comparisons. Integer-key algorithms may inspect digits or bits and therefore use a stronger model. Claims of expected time must identify whether randomness comes from the input distribution or the algorithm.

5 Binary search as boundary finding

For sorted `A`, define the lower bound of `x` as the smallest index j with `A[j] >= x`, or n if no such index exists.

```
lower_bound(A, x):
    lo = 0
    hi = length(A)
    while lo < hi:
        mid = lo + floor((hi - lo) / 2)
        if A[mid] < x:
            lo = mid + 1
        else:
            hi = mid
    return lo
```

The loop invariant is:

- $0 \le lo \le hi \le n$;
- every index below `lo` contains a value less than `x`; and
- every array index from `hi` through `n-1` contains a value at least `x`.

The unknown boundary lies in the closed set of candidate positions from `lo` through `hi`. Initially both value claims are vacuous. If `A[mid] < x`, positions through `mid` cannot be the boundary. Otherwise `mid` and everything after it remain on the at-least side. Each update preserves the invariant and strictly shortens `hi-lo`. At termination `lo==hi`, so the unique remaining position is the lower bound.

Membership is a separate question: `x` occurs exactly when the returned position satisfies `lo < n` and `A[lo] == x`. The interval length at least halves, so time is $\Theta(\log n)$ and iterative auxiliary space is $\Theta(1)$.

6 Worked trace: a duplicate key

Let `A = [2,4,4,7,9]` and `x = 4`.

lo	hi	mid	Decision
0	5	2	`A[2] >= 4`, set `hi=2`
0	2	1	`A[1] >= 4`, set `hi=1`
0	1	0	`A[0] < 4`, set `lo=1`

The answer is position 1, the first of the equal keys. Notice that the maintained object is an answer **position**, possibly n, not an interval containing every occurrence.

7 Why comparison sorting needs $\Omega(n \log n)$

Assume all n keys are distinct. A deterministic comparison sort induces a binary decision tree: each internal node compares two keys and each leaf identifies one input ordering. Correctness requires at least $n!$ leaves.

A binary tree of height h has at most 2^h leaves, so $2^h \ge n!$ and

$$h \ge \log_2(n!).$$

At least half the factors in $n!$ are at least $n/2$, hence

$$n! \ge (n/2)^{n/2}$$

and therefore $\log_2(n!) = \Omega(n \log n)$. This is a worst-case lower bound for comparison sorting, not for algorithms allowed to inspect restricted integer keys.

8 A baseline: insertion sort

Insertion sort maintains a sorted prefix and inserts the next record into it.

```
insertion_sort(A):
    for i = 1 to length(A) - 1:
        x = A[i]
        j = i
        while j > 0 and x.key < A[j-1].key:
            A[j] = A[j-1]
            j = j - 1
        A[j] = x
```

At the start of iteration `i`, `A[0..i)` is a sorted permutation of the first i original records. Shifting larger records right and placing `x` in the gap preserves sortedness and the multiset. At the final iteration the prefix is the whole array.

Worst-case time is $\Theta(n^2)$, because a reverse-sorted input shifts $1 + 2 + \cdots + (n-1)$ records. An already sorted input takes $\Theta(n)$ comparisons. Using `<` rather than `<=` in the shift test makes this version stable. It is in-place and often effective for small or nearly sorted ranges.

9 Merge sort

Merge sort recursively sorts two halves and combines them. The merge step repeatedly takes the smaller first unconsumed record:

```
merge(L, R):
    i = 0; j = 0; out = empty sequence
    while i < length(L) and j < length(R):
        if L[i].key <= R[j].key:
            append L[i] to out; i = i + 1
        else:
            append R[j] to out; j = j + 1
    append the unconsumed suffix of L or R to out
    return out
```

The merge invariant says `out` is sorted, contains exactly the consumed records, and no unconsumed record is smaller than its last record. Since `L` and `R` are sorted, choosing their smaller front preserves this invariant. Taking from the left on equality preserves stability.

Correctness of merge sort follows by induction on input length. Arrays of length at most one are sorted. The induction hypothesis sorts both smaller halves, and the merge argument combines them correctly. Its recurrence is

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n).$$

Standard array merging uses $\Theta(n)$ auxiliary storage. Linked sequences can be merged by relinking nodes, but recursive stack and representation costs still need counting.

10 Randomized three-way quicksort

Quicksort partitions a range around a pivot, then recursively sorts the unequal sides. A three-way partition handles duplicate keys explicitly:

```
partition3(A, lo, hi, pivot):       // range [lo, hi)
    lt = lo; i = lo; gt = hi
    while i < gt:
        if A[i].key < pivot:
            swap A[lt], A[i]; lt = lt + 1; i = i + 1
        else if A[i].key > pivot:
            gt = gt - 1; swap A[i], A[gt]
        else:
            i = i + 1
    return (lt, gt)
```

The invariant partitions the range into `< pivot`, `== pivot`, unknown, and `> pivot` regions: `[lo,lt)`, `[lt,i)`, `[i,gt)`, and `[gt,hi)`. Each branch moves one unknown record to a known region. At termination the unknown region is empty. Quicksort recurses only on `[lo,lt)` and `[gt,hi)`.

A pivot chosen uniformly from the current range gives expected $O(n \log n)$ time for every fixed input and expected $\Theta(n \log n)$ when the keys are distinct. Three-way partitioning can be faster when many keys are equal; an all-equal range is finished in $\Theta(n)$ time. Repeated extreme pivots still give a $\Theta(n^2)$ worst case.

Time and recursion depth need separate safeguards. Ordinary recursive quicksort has expected $O(\log n)$ stack depth and worst-case $\Theta(n)$ depth. Recursing on the smaller side and iterating over the larger side limits stack depth to $O(\log n)$ even in the worst case. Typical in-place quicksort is not stable.

11 Recurrences and the Master theorem

Start with a recurrence tree: label work per node, multiply by nodes per level, determine depth, and sum. The following common polynomial-gap form of the Master theorem then summarises recurrences

$$T(n) = aT(n/b) + f(n),$$

with constants $a \geq 1$, $b > 1$, comparable subproblem sizes, and a base case. Let $q = \log_b a$.

1. If $f(n) = O(n^{q-\varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^q)$.
2. If $f(n) = \Theta(n^q)$, then $T(n) = \Theta(n^q \log n)$.
3. If $f(n) = \Omega(n^{q+\varepsilon})$ and $af(n/b) \leq cf(n)$ eventually for some $c < 1$, then $T(n) = \Theta(f(n))$.

Examples:

- $T(n) = T(n/2) + 1$ has $q = 0$ and gives $\Theta(\log n)$.
- $T(n) = 2T(n/2) + n$ has equal $\Theta(n)$ work per level and gives $\Theta(n \log n)$.
- $T(n) = 8T(n/2) + n^2$ has $q = 3$; leaves dominate and give $\Theta(n^3)$.

The theorem does not apply directly to $T(n) = T(n-1) + n$, unequal irregular splits, or recurrences missing a shrinking factor. Expand or use substitution instead of forcing a case.

12 Sorting restricted keys

The following methods obtain stronger bounds by using structure beyond arbitrary key comparisons. For each one, identify the extra assumption before quoting its running time.

Counting sort assumes integer keys in `0..k-1`. Count each key, convert counts to ending positions with prefix sums, and scan the input from right to left while placing records into an output array. The reverse scan plus prefix positions makes the method stable. Time is $\Theta(n + k)$ and auxiliary space is $\Theta(n + k)$. It is useful only when the key range is manageable.

LSD radix sort applies a stable inner sort from the least significant digit to the most significant. After pass i, records are ordered by their last i digits; stability preserves the order established by earlier passes. With d digits and radix k, time is $\Theta(d(n + k))$.

Bitwise MSD sorting partitions fixed-width unsigned keys by a mask, beginning with the highest bit, then recursively partitions the zero and one groups using the next bit.

Each record participates at most once per bit. For w bits, time is $O(wn)$; this is linear only when w is treated as fixed, and is $O(n \log U)$ when keys come from a universe of size U. Signed integers and floating-point encodings require a deliberately chosen ordering transformation.

Bucket sort maps keys into ordered buckets, sorts each bucket, and concatenates. Expected linear time needs a distribution assumption that keeps total within-bucket work linear. A highly concentrated input can destroy that bound.

13 Selection and order statistics

The element of rank k is the record that would occupy index k after sorting. Quickselect uses the same three-way partition but continues only in the region containing the desired rank:

```
quickselect(A, k):
    require 0 <= k < length(A)
    lo = 0; hi = length(A)
    while true:
        choose a uniformly random pivot from A[lo..hi)
        (lt, gt) = partition3(A, lo, hi, pivot)
        if k < lt:       hi = lt
        else if k >= gt: lo = gt
        else:            return A[k]
```

Partition correctness ensures that a rank outside the equal region lies in exactly one remaining side. A random pivot yields expected $\Theta(n)$ time because only one side is visited. Sorting first would cost $\Theta(n \log n)$. Deterministic median-of-medians pivoting gives $\Theta(n)$ worst-case time, but with more machinery and larger constants.

14 Skip lists

A skip list implements an ordered dictionary as levels of sorted linked lists. Level 0 contains every key. Independently promote each inserted key to the next level with probability 1/2, repeating until the first failure. Higher levels act as express lanes.

Search starts at the top-left sentinel. At each level, move right while the next key is smaller than the target; otherwise drop one level. Sortedness guarantees that moving down cannot skip the target. At level 0, the next node is either the target or its insertion position.

Insertion records the last node visited at every level, chooses a random height, and splices the new tower after those predecessors. Deletion unlinks the target at each level where it occurs. With promotion probability 1/2, the expected number of stored node occurrences is

$$n + n/2 + n/4 + \cdots < 2n.$$

The expected height and expected search, insertion, and deletion times are $O(\log n)$; the worst case remains $\Theta(n)$. These expectations are over the promotion choices. Skip lists trade deterministic rotation rules for a simple randomized representation.

15 Complexity and trade-offs

Method	Expected/best information	Worst-case time	Auxiliary space	Stable?
Insertion sort	$\Theta(n)$ when sorted	$\Theta(n^2)$	$\Theta(1)$	yes
Merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$ for arrays	yes
Randomized three-way quicksort	expected $O(n \log n)$; $\Theta(n)$ when all keys are equal	$\Theta(n^2)$	ordinary recursion: expected $O(\log n)$ stack, worst $\Theta(n)$	usually no
Counting sort	$\Theta(n + k)$	$\Theta(n + k)$	$\Theta(n + k)$ stable version	yes
Quickselect	expected $\Theta(n)$	$\Theta(n^2)$	implementation-dependent	n/a

No row is universally best. Stability, key model, existing order, memory, worst-case guarantees, record size, and implementation environment all affect the decision.

16 Connections

- Binary search depends on the array and half-open-range foundations of the preceding note.
- Merge sort and quicksort are templates for divide-and-conquer design.

- Skip lists provide a randomized alternative to the balanced trees in the next note.
- Heap sort and priority-based selection appear with heaps.
- String indexing later uses counting, radix, and lexicographic ordering ideas.

17 Common mistakes

- Binary-searching unsorted data or mixing closed and half-open interval rules.
- Treating the return value of `lower_bound` as guaranteed membership.
- Losing stability by taking equal merge keys from the right.
- Describing quicksort without a policy for equal keys or without a random model.
- Applying the comparison lower bound to algorithms that inspect integer digits.
- Calling $O(wn)$ "linear" without stating whether word width w is fixed.
- Applying the Master theorem to a recurrence outside its conditions.
- Quoting expected skip-list or quickselect time as a per-operation worst-case guarantee.

18 Self-check

1. State and prove the three clauses of the `lower_bound` invariant.
2. Why does `lower_bound` need to return n for some inputs?
3. Prove the insertion-sort prefix invariant.
4. Why does taking the left record on a merge tie preserve stability?
5. State the four-region invariant of three-way partitioning.
6. Solve $T(n) = 4T(n/2) + n$ and justify the applicable Master-theorem case.
7. Why must LSD radix sort use a stable inner sort?
8. When is Quickselect preferable to sorting?
9. Derive the expected linear space bound for promotion probability $1/2$ in a skip list.

19 Revision summary

- Binary search locates a boundary in a sorted range; its invariant concerns candidate positions.
- Comparison sorting needs $\Omega(n \log n)$ comparisons in the worst case.
- Insertion sort is adaptive; merge sort is stable with linear extra array space; randomized quicksort is practical but retains a quadratic worst case.
- Recurrence trees explain the Master theorem rather than merely supplying cases to memorise.
- Counting, radix, bitwise, and bucket methods rely on key or distribution structure outside the comparison model.
- Partitioning can select one rank in expected linear time.
- Skip lists use randomized levels for expected logarithmic dictionary operations.
- Every complexity claim needs its key model, random source, space convention, and case.

20 Implementations and hands-on exploration

- Python's `bisect` implements boundary search, while the sorting HOWTO documents stable `list.sort` and `sorted`. `insort` still shifts a list suffix in linear time, and `bisect` functions are not safe for concurrent mutation of the same sequence.
- The GNU C Library documents `qsort` and `bsearch`. The comparator must define the required order; `qsort` is not stable, and `bsearch` need not return the first equal element.
- Rust slice methods include stable and unstable sorting, binary search, and `partition_point`. A binary search is meaningful only under the same total order used to sort, and an equal match is not necessarily the first.
- Java `ConcurrentSkipListMap` is a concurrent ordered-map implementation with expected average logarithmic basic operations. Iterators are weakly consistent and bulk operations are not atomic snapshots.

20.1 Small experiments

1. Implement `lower_bound` and assert all three clauses of its loop invariant at every iteration. Exhaustively compare the result with Python `bisect_left` on every nondecreasing array of length at most 7 over keys `0,1,2`, including absent and duplicate targets.
2. Count comparisons for insertion sort, merge sort, and seeded three-way quicksort on sorted, reversed, all-equal, and random tagged records. Before interpreting the counts, assert multiset preservation and sortedness for every result, stability for insertion and merge sort, and the four-region invariant inside `partition3`.

21 Sources and further study

- Linear structures, sorting, searching, and recurrences deck, Jaak Vilo.
- (Cormen et al. 2022), Chapters 2, 4, 7-9.

References

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. 4th ed. MIT Press.