



UNIVERSITY
OF TRIESTE

Algorithmic Design

University of Trieste – a.a. 2026/2027

Prof. Antonio Emanuele Cinà

`antonio.cina@units.it`

Department of Mathematics, Informatics and Geosciences

University of Trieste

a.a. 2026/2027

Chapter 4:

Sorting

Sorting connects specifications, structure, and cost

Sorting has a precise specification, several very different algorithms, and costs that depend on representation.

- **Incremental methods:** Bubble sort and Insertion sort.
- **Divide and conquer:** Merge sort and Quicksort.

For each method, identify the useful structure already present, how it grows, the resulting cost, and the inputs that change the analysis.

Each method grows a different useful structure

Method	Guiding question	Structure produced
Bubble	Is this adjacent pair in the wrong order?	A final sorted suffix.
Insertion	Where does the next record belong?	A growing sorted prefix.
Merge	How can two sorted halves be combined?	Larger sorted runs.
Quick	Which records belong on each side of the pivot?	Two independent subarrays and one final pivot.

The loops and recursive calls implement these ideas. Identify the structure before reading the code.

Sorted order must preserve every record

Definition (Sorting problem)

Let $A = [a_0, a_1, \dots, a_{n-1}]$ be a sequence of records, each with a key drawn from a totally ordered domain. A sorting algorithm must return a permutation $A' = [a'_0, a'_1, \dots, a'_{n-1}]$ of the input such that

$$\text{key}(a'_0) \leq \text{key}(a'_1) \leq \dots \leq \text{key}(a'_{n-1}).$$

A nondecreasing sequence is not a valid answer if it loses, duplicates, or introduces a record.

Stability, storage, and adaptiveness are independent

Properties of a concrete implementation

- *Stable*: records with equal keys retain their relative input order.
- *In place*: only a constant amount of auxiliary storage is used, apart from the convention adopted for a recursion stack.
- *Adaptive*: running time improves when the input is already partly ordered.

These properties answer three different questions.

- What happens to records with equal keys?
- How much additional memory grows with the input size?
- Does existing order reduce the work performed?

Stability is visible only when equal keys are tagged

The number is the sorting key; the subscript identifies the record.

Input		$[4_A, 2_B, 4_C, 1_D]$
Stable output	✓	$[1_D, 2_B, 4_A, 4_C]$
Unstable output	×	$[1_D, 2_B, 4_C, 4_A]$

Both outputs are ordered by key, but only the first preserves the input order of 4_A and 4_C .

Why it matters

After sorting by surname, a stable sort by city preserves surname order inside each city.

In place refers to auxiliary storage

The input array already occupies n record positions. Ask how much *additional* memory the implementation needs.

Constant additional memory

Both Bubble variants and Insertion sort need only a few variables. Early exit adds a flag; insertion saves one record.

Auxiliary space: $\Theta(1)$.

Growing additional memory

The standard array Merge sort allocates an output buffer containing n records.

Auxiliary space: $\Theta(n)$.

Quicksort partitioning uses $\Theta(1)$ array storage, but recursive calls also consume stack frames. State whether the stack is counted.

Adaptiveness converts existing order into less work

Definition (Inversion)

An inversion is a pair (i, j) with $i < j$ but $\text{key}(A[i]) > \text{key}(A[j])$.

Input	$I(A)$	Insertion-sort shifts
[1, 2, 3, 4, 5, 6]	0	0
[1, 2, 3, 5, 4, 6]	1	1
[6, 5, 4, 3, 2, 1]	15	15

For Insertion sort,

$$T(A) = \Theta(n + I(A)).$$

Fewer inversions therefore produce less work. The displayed Merge sort still takes $\Theta(n \log n)$ on all three inputs.

The representation changes the operations

Arrays support constant-time indexed access and direct exchanges.

Method	Representation	Consequence
Bubble sort	Array or sequential container	Adjacent comparisons and exchanges.
Insertion sort	Array or linked sequence	Arrays shift blocks; links avoid shifts only after finding the insertion position.
Merge sort	Array plus buffer, or linked sequence	Merging is sequential; arrays normally need auxiliary storage.
Quicksort	Random-access array; stack	Partitioning exchanges distant positions.

State what the sorting bound measures

Cost convention

The input size n is the number of records. Unless another case is stated, bounds refer to worst-case time in the uniform-cost model.

Comparisons of machine-sized keys, assignments, and array accesses each cost $\Theta(1)$.

Count key comparisons and record movements separately when informative: their growth need not be the same on every input.

One adjacent repair removes one inversion

Write $I(A)$ for the inversion count introduced earlier in the chapter.

- A sorted array has no inversions.
- Equal keys do not form an inversion.
- Swapping adjacent records whose keys are in decreasing order removes exactly one inversion.

Bubble sort removes inversions by adjacent exchanges. Insertion sort accounts for them through shifts.

Bubble sort repairs neighbouring disorder

Treat one left-to-right pass as a sequence of local competitions. In each adjacent pair, the larger key moves to the right.

$$[5, 1, 4, 2] \rightarrow [1, 5, 4, 2] \rightarrow [1, 4, 5, 2] \rightarrow [1, 4, 2, \boxed{5}].$$

- The maximum active key reaches the right boundary.
- Only that final record is guaranteed to be fixed after one pass.
- The remaining prefix is a smaller instance of the same problem.

Grow a final suffix with adjacent exchanges

A left-to-right pass exchanges adjacent inverted records. The largest active key moves to the right boundary.

Proposition (Bubble sort invariant)

At the end of a pass whose active boundary is end , the record in $A[\text{end}]$ has maximum key among $A[0 \dots \text{end}]$ and will not move again.

After one pass, position $n - 1$ is final. After two passes, the last two positions are final and sorted.

After p passes, the suffix of length p is final, so the active boundary can move left.

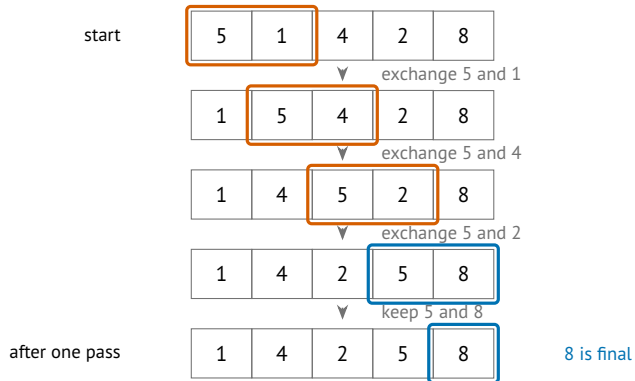
Basic Bubble sort completes every planned pass

```
Bubble-sort(A)
  for end <- length(A)-1 downto 1
    for i <- 0 to end-1
      if key(A[i]) > key(A[i+1])
        exchange A[i] and A[i+1]
```

Each pass shortens the active prefix by one. There is no test for an already sorted prefix.

For $n \leq 1$, the outer loop is empty.

One pass moves the active maximum to the boundary



Basic Bubble sort always makes quadratic comparisons

The loop bounds depend on n , not on the input values:

$$C(n) = (n - 1) + (n - 2) + \dots + 1 = \frac{n(n - 1)}{2}.$$

Thus both best-case and worst-case running time are $\Theta(n^2)$.

Comparisons and movements tell different stories

A sorted input causes no exchanges, but all comparisons still occur. Every adjacent exchange removes one inversion, so the total number of exchanges is exactly $I(A)$.

The strict $>$ test preserves equal-key order. Auxiliary storage is $\Theta(1)$.

A pass with no exchanges certifies sortedness

Suppose a complete pass makes no exchange.

1. Every adjacent pair in the active prefix is in nondecreasing order, so that prefix is sorted.
2. Earlier passes have already placed the largest records in the final sorted suffix.
3. The prefix followed by that suffix is the complete sorted array.

Turn the observation into a stopping rule

Reset a Boolean flag before each pass. Set it whenever a pair is exchanged. If it stays false, return.

The local repair and final-suffix invariant stay the same.

The early-exit variant observes each pass

```
Adaptive-Bubble-sort(A)
  for end <- length(A)-1 downto 1
    exchanged <- false
    for i <- 0 to end-1
      if key(A[i]) > key(A[i+1])
        exchange A[i] and A[i+1]
        exchanged <- true
    if not exchanged
      return
```

Early exit improves the best case

Already sorted input

The first pass makes $n - 1$ comparisons and no exchanges, then returns: $\Theta(n)$ time.

Distinct keys in decreasing order

Every pass makes exchanges, so all passes run:

$$C(n) = \frac{n(n-1)}{2}, \quad T(n) = \Theta(n^2).$$

The flag changes only whether another pass begins. Stability, the final-suffix invariant, and $\Theta(1)$ auxiliary space are preserved.

Few inversions do not guarantee fast Bubble sort

For $A = [2, 3, \dots, n, 1]$, only pairs involving the final 1 are inverted:

$$I(A) = n - 1.$$

Early-exit Bubble sort

The 1 moves left by only one position per left-to-right pass. All $n - 1$ passes make an exchange, so the comparison count is

$$(n - 1) + (n - 2) + \dots + 1 = \Theta(n^2).$$

Insertion sort shifts each larger record once when it inserts 1:

$$\Theta(n + I(A)) = \Theta(n).$$

Both exploit some existing order, but their sensitivity to that order differs.

Insertion sort grows a sorted prefix

Assume the left prefix is already sorted. Save the next record, shift larger records right, and place the saved record in the resulting gap.

Sorted prefix and next key $[2, 4, 5, 6] \mid 1$

Open a gap $[_, 2, 4, 5, 6]$

Insert the saved key $[1, 2, 4, 5, 6]$

If the next key were 7, no record would shift. This difference is the source of adaptiveness.

Insert the next record into a sorted prefix

Like arranging playing cards, Insertion sort maintains a sorted prefix.

At the beginning of iteration j

$A[0 \dots j - 1]$ is sorted and contains exactly the records originally stored there.

1. Save $A[j]$ before shifting any records.
2. Shift larger prefix records one position right.
3. Insert the saved record at $i + 1$, after any equal keys.

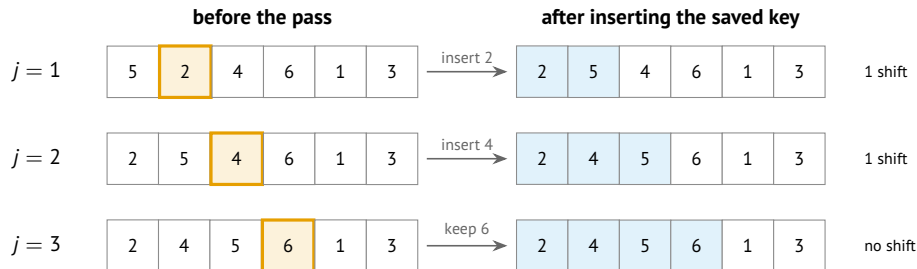
Placing the saved record after equal keys preserves their relative order.

Protect the saved record while shifting

```
Insertion-sort(A)
  for j <- 1 to length(A)-1
    saved <- A[j]
    i <- j-1
    while i >= 0 and key(A[i]) > key(saved)
      A[i+1] <- A[i]
      i <- i-1
    A[i+1] <- saved
```

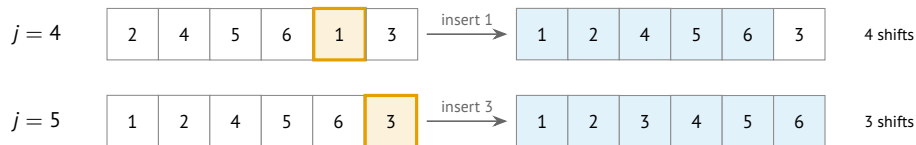
Evaluate and from left to right: when $i < 0$, stop without accessing $A[i]$.

Insert 2 and 4, then keep 6



At pass j , the first $j + 1$ cells of the result form the sorted prefix. Orange outlines mark the saved record.

Insert 1 and 3 into the growing prefix



Orange marks the saved key; blue marks the sorted prefix after the pass.
 The $1 + 1 + 0 + 4 + 3 = 9$ shifts equal the 9 input inversions.

The saved keys are 1 and 3; the sorted prefixes have lengths 5 and 6.

Count how many records each insertion shifts

Already sorted input

Each iteration checks the order and performs no shift: $\Theta(n)$ time.

Distinct keys in decreasing order

At iteration j , all j prefix records move one position right:

$$1 + 2 + \cdots + (n - 1) = \frac{n(n - 1)}{2}.$$

The worst-case time is therefore $\Theta(n^2)$.

Each shift corresponds to one input inversion: an earlier record with a larger key than the saved record.

Few inversions give fast insertion sorting

If $I(A)$ is the number of input inversions,

$$T(A) = \Theta(n + I(A)).$$

The n term accounts for the loop overhead and stopping checks; the $I(A)$ term counts shifts. Few inversions therefore give nearly linear time.

- Only constant auxiliary storage is needed.
- The strict $>$ comparison preserves the order of equal-key records.
- Replacing $>$ by $\geq$ shifts earlier equal records right, placing the saved record before them and breaking stability.

Merge sort builds order from small sorted runs

1. Divide the array by position until every subarray has at most one record.
2. Recursively obtain two sorted halves.
3. Merge the halves with one forward scan.

For example, after recursion has produced $L = [2, 5]$ and $R = [1, 4]$,

$$[2, 5] + [1, 4] \longrightarrow [1, 2, 4, 5].$$

The split itself creates no order. Efficiency comes from combining two *already sorted* runs without reconsidering every pair.

Merge sorted sequences from the front

Merge sort divides the input, sorts both halves recursively, and combines them in linear time.

If L and R are sorted, the smallest remaining records are at their fronts. Comparing those records determines the next output record.

Stability rule

Choose the record from L when the keys are equal.

When one input is exhausted, append the remaining suffix of the other.

Stable merge into an output buffer

```
Merge(L, R)
  allocate output[0 .. length(L)+length(R)-1]
  i <- 0; j <- 0; k <- 0
  while i < length(L) and j < length(R)
    if key(L[i]) <= key(R[j])
      output[k] <- L[i]; i <- i+1
    else
      output[k] <- R[j]; j <- j+1
    k <- k+1
  copy the remaining suffix of L into output
  copy the remaining suffix of R into output
  return output
```

Copy each remaining suffix at the next free output position; do not overwrite the prefix already produced.

Forward scans make merging linear

Each index moves only forward and every record is copied once.

$$T_{\text{merge}}(n_1, n_2) = \Theta(n_1 + n_2).$$

- Arrays use an auxiliary output buffer.
- Linked sequences can merge by reconnecting links.
- Splitting linked sequences and accessing random positions have different costs.

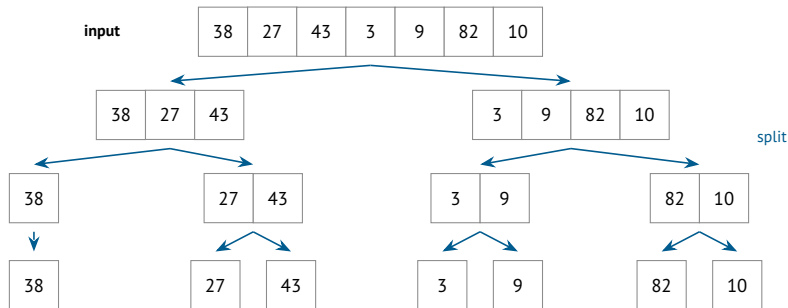
The merge principle is the same, but its implementation depends on representation.

Divide, sort both halves, and merge

```
Merge-sort(A)
  if length(A) <= 1
    return A
  middle <- floor(length(A)/2)
  left <- Merge-sort(A[0 .. middle-1])
  right <- Merge-sort(A[middle .. length(A)-1])
  return Merge(left, right)
```

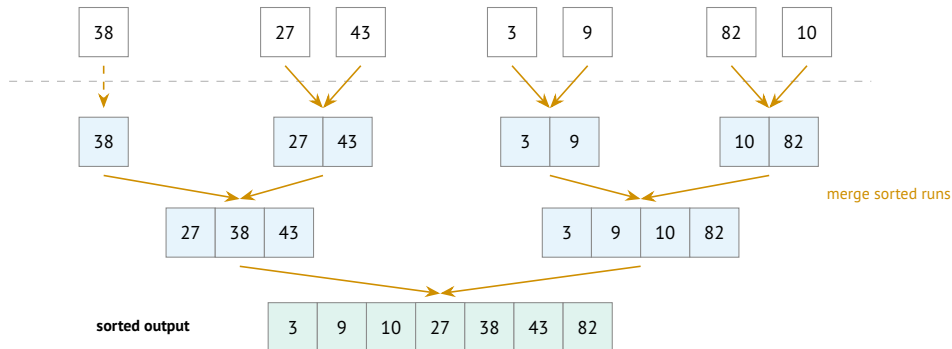
An array of length zero or one is already sorted. The two recursive calls cover disjoint halves.

Split into one-record subarrays



As in the code, split after $\lfloor n/2 \rfloor$ records. For odd lengths the right half receives the extra record. Dashed lines carry a singleton unchanged.

Merge sorted runs in the reverse direction



Merge neighbouring sorted runs. The dashed line carries the singleton 38 unchanged; every other arrow feeds a merge.

Every depth contributes linear merge work

For powers of two,

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

At each merging depth the subarrays are disjoint and their lengths sum to n . There are $\Theta(\log n)$ depths, hence

$$T(n) = \Theta(n \log n) \quad \text{in the best, average, and worst cases.}$$

- Stable, with $\Theta(n)$ auxiliary array space.
- An additional recursion stack has depth $\Theta(\log n)$.
- This version does not exploit an already sorted input.

Quicksort divides by value around a pivot

Merge sort combines after recursion. Quicksort first partitions around a pivot, then sorts the two resulting parts in place.

$$[7, 2, 9, 4, \boxed{6}] \xrightarrow{\text{partition}} [2, 4] \boxed{6} [9, 7].$$

- The pivot is now in its final sorted position.
- The two groups are separated, but they are not necessarily sorted.
- Recursion sorts the groups in their current locations, so no merge follows.

The sizes of the two groups determine whether the recursion is balanced or expensive.

Separate records around a pivot

Quicksort rearranges an array around a pivot and sorts both sides independently. No merge step is needed.

Bounds are inclusive. The last record, at `high`, is the pivot. The scan index is j ; i marks the end of the region with keys at most the pivot.

Regions before scanning position j

$A[\text{low} \dots i]$	$\text{keys} \leq \text{pivot_key}$
$A[i + 1 \dots j - 1]$	$\text{keys} > \text{pivot_key}$
$A[j \dots \text{high} - 1]$	unclassified
$A[\text{high}]$	pivot

Initially $i = \text{low} - 1$ and $j = \text{low}$: both classified regions are empty. The pivot stays at `high` while the scan stops at `high - 1`.

Classify one record while preserving the four regions

If $\text{key}(A[j]) > \text{pivot_key}$

Make no exchange. Advancing j adds this record to the greater-than region.

If $\text{key}(A[j]) \leq \text{pivot_key}$

Increment i , then exchange $A[i]$ and $A[j]$. The accepted record extends the no-larger region.

If $i < j$, the displaced record was in the greater-than region and can safely remain behind the scan.

If $i = j$ after the increment, the exchange is with the same position and changes nothing. In either case, advance j .

Lomuto partition places the pivot

```
Partition(A, low, high)
  pivot_key ← key(A[high])
  i ← low-1
  for j ← low to high-1
    if key(A[j]) ≤ pivot_key
      i ← i+1
      exchange A[i] and A[j]
  exchange A[i+1] and A[high]
  return i+1
```

After the final exchange, the pivot is in its final sorted position. The two surrounding regions still need sorting.

Place the pivot between the classified regions

At the end of the scan, $j = \text{high}$. Every non-pivot record has been classified.

Exchange $A[i + 1]$ with $A[\text{high}]$ and return $q = i + 1$.

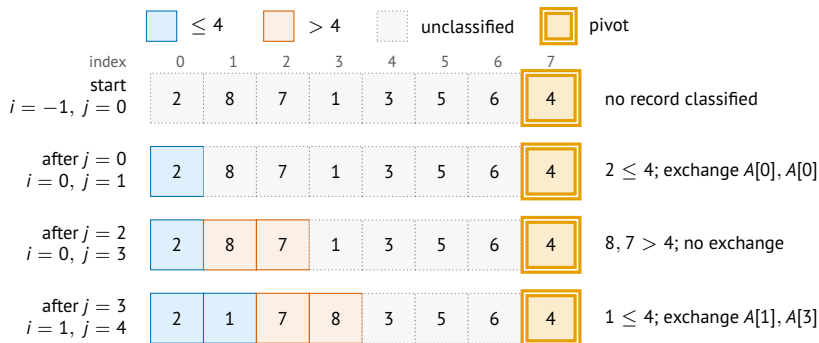
What partition guarantees

- Every key in $A[\text{low} \dots q - 1]$ is no larger than $\text{key}(A[q])$.
- Every key in $A[q + 1 \dots \text{high}]$ is larger than $\text{key}(A[q])$.
- The subarray contains exactly the same records as before.

The pivot occupies a valid final position. Neither side is necessarily sorted.

Lomuto partition on pivot 4: classify the first records

After notes Figure 4.4, first part: low = 0, high = 7.

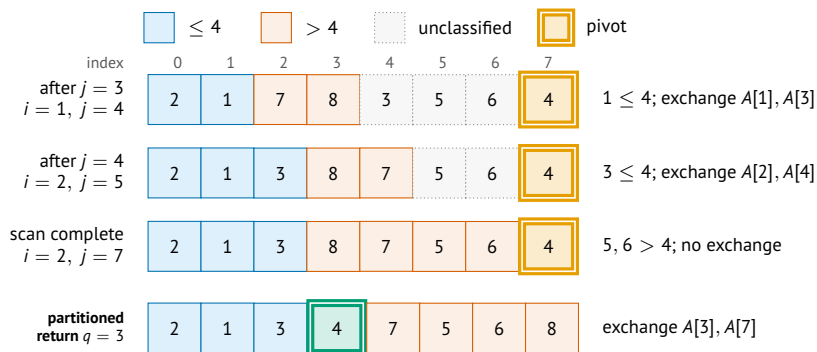


The first exchange leaves 2 in place. Accepting 1 exchanges it with 8; the displaced 8 stays in the greater-than region.

Each displayed j is the next scan position; i ends the region with keys ≤ 4 .

Lomuto partition on pivot 4: finish the scan

After notes Figure 4.4, continued: the first row repeats the previous state.



Accepting 3 exchanges it with 7. Rejecting 5 and 6 completes the scan.

The final exchange places pivot 4 at $q = i + 1 = 3$ and moves 8 to the right region. Neither side is yet sorted.

One partition scans the subarray once

Let $m = \text{high} - \text{low} + 1$ be the subarray length.

- The scan makes exactly $m - 1$ comparisons with the pivot key.
- Each scanned record causes at most one exchange, possibly with itself.
- One final exchange places the pivot.

$$T_{\text{partition}}(m) = \Theta(m), \quad \text{auxiliary space} = \Theta(1).$$

Distant exchanges do not generally preserve equal-key order: this partition is not stable.

Partition solves the separation problem. Quicksort must still sort both sides.

Recurse on the two sides, excluding the pivot

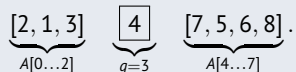
```

Quicksort(A, low, high)
  if low < high
    q ← Partition(A, low, high)
    Quicksort(A, low, q-1)
    Quicksort(A, q+1, high)

```

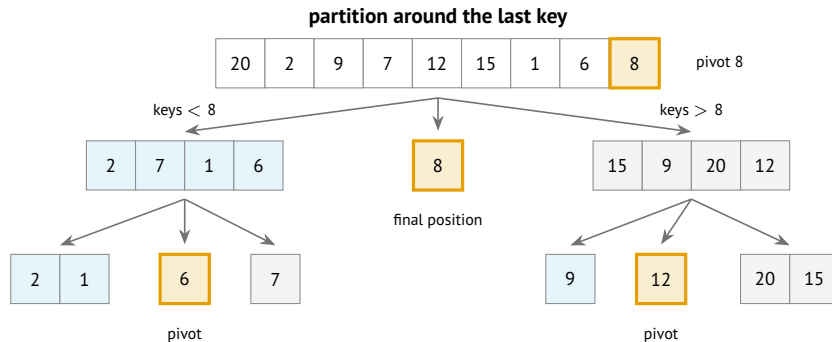
Call initially with $\text{low}=0$ and $\text{high}=\text{length}(A)-1$. Empty and one-record intervals need no work.

Continue the example from Figure 4.4



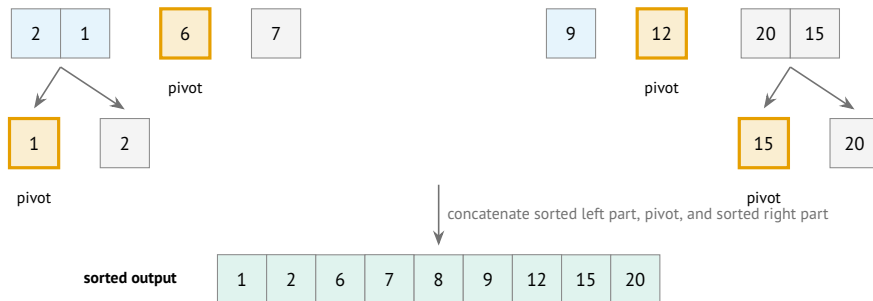
Recurse on positions $0 \dots 2$ and $4 \dots 7$. Position 3 is excluded: its pivot is final. The two results will already lie on the correct sides of 4.

Partition first around 8, then around 6 and 12



Each isolated pivot is final; the remaining groups are independent sorting problems.

Finish the independent sorting problems



“Concatenate” describes the final order: the parts already occupy their array positions, so the code performs no copying merge.

The pivot determines the subproblem sizes

Partitioning n records costs $\Theta(n)$. With k records to the left,

$$T(n) = T(k) + T(n - k - 1) + \Theta(n).$$

Balanced splits

For equal sides, $T(n) = 2T((n - 1)/2) + \Theta(n) = \Theta(n \log n)$. The same bound holds when both sides remain fixed positive fractions of the input.

An extreme pivot at every split

$$T(n) = T(n - 1) + \Theta(n) = \Theta(n^2).$$

With the last record as pivot, sorted distinct keys give the worst case.

Random pivots give an expected guarantee

At each call, choose a record uniformly at random and exchange it with the last position before partitioning.

Expected running time

For every fixed input of *distinct keys*, the expected running time over random pivot choices is $\Theta(n \log n)$.

The expectation is over the algorithm's choices; the input itself need not be random.

The worst case remains $\Theta(n^2)$, but a fixed input order no longer determines the sequence of pivot ranks.

Repeated keys require separate care with the displayed two-way partition.

Equal keys can defeat two-way partitioning

With all keys equal, every comparison in Lomuto partition satisfies $\leq$. The pivot finishes at the last position.

$$T(n) = T(n - 1) + \Theta(n) = \Theta(n^2).$$

Randomly choosing another equal key does not improve the split.

Three-way partitioning

Form regions with keys $<$, $=$, and $>$ the pivot key. Recurse only on the first and third regions: the equal-key region is already final.

In-place partitioning still needs a recursion stack

Partitioning uses $\Theta(1)$ auxiliary array storage. The displayed recursive Quicksort also keeps one stack frame per active call.

Auxiliary stack space

- Balanced calls have depth $\Theta(\log n)$.
- Consistently poor partitions can have depth $\Theta(n)$.

State whether the recursion stack is included when calling Quicksort *in place*.

The ordinary exchange-based algorithm is not stable: distant exchanges can change the relative order of equal-key records.

Compare the implementations just studied

Algorithm	Best time	Worst time	Stable	Extra storage
Basic Bubble	$\Theta(n^2)$	$\Theta(n^2)$	yes	$\Theta(1)$
Early-exit Bubble	$\Theta(n)$	$\Theta(n^2)$	yes	$\Theta(1)$
Insertion	$\Theta(n)$	$\Theta(n^2)$	yes	$\Theta(1)$
Merge	$\Theta(n \log n)$	$\Theta(n \log n)$	yes	$\Theta(n)$ plus stack
Quick	$\Theta(n \log n)$	$\Theta(n^2)$	no	recursion stack

These bounds describe the displayed array implementations. For Quicksort, extra storage includes its recursion stack.

Read each bound together with its assumptions

Basic and early-exit Bubble sort

The local exchanges are identical. Only the early-exit variant observes a pass with no exchanges and stops, improving the best case to $\Theta(n)$.

Memory depends on the implementation

Merge sort uses a $\Theta(n)$ array buffer and a logarithmic recursion stack. Quicksort partitions with constant extra array storage, but its stack can grow from $\Theta(\log n)$ to $\Theta(n)$.

With distinct keys, randomized Quicksort has expected $\Theta(n \log n)$ time over random pivot choices. This is not a worst-case guarantee or an average over an unspecified input distribution.

Choose a method for the required guarantee

Insertion sort Simple and useful for small or nearly sorted arrays.

Merge sort Stability and a worst-case $n \log n$ guarantee justify the buffer.

Quicksort Attractive for array locality and in-place partitioning, with careful pivot selection and recursion depth.

Bubble sort Primarily a transparent teaching example.

Before choosing

Array or linked structure? Stable order required? Auxiliary memory available? Worst-case, expected, or input-sensitive guarantee?

Exercises on the two Bubble variants

Distinguish comparisons from record movements in every trace.

Exercise 1

Trace one complete pass of Bubble sort on $[5, 1, 4, 2, 8]$. State the active region after each exchange and identify the record that becomes final at the end of the pass.

Exercise 2

Derive the exact number of adjacent comparisons made by basic Bubble sort on every input of length n . Then analyze the early-exit variant on a sorted array and on a decreasing array. Explain why the optimization changes the best case but not the worst case.

Exercise 3

Compare early-exit Bubble sort and Insertion sort on $[2, 3, \dots, n, 1]$. Compute the inversion count and derive the asymptotic number of comparisons performed by each algorithm. Explain why both algorithms are called adaptive even though their sensitivity to existing order is different.

Exercises on insertion, stability, and merging

Exercise 4

Trace Insertion sort on $[5, 2, 4, 6, 1, 3]$. For every iteration record the saved key, the number of shifts, and the sorted prefix. Count the input inversions independently and compare the two totals.

Exercise 5

Give a tagged input that distinguishes stability from sortedness. Explain why the strict comparison in Insertion sort preserves the order of equal-key records.

Exercise 6

Merge $[2_a, 4, 7]$ and $[2_b, 3, 7_b]$. Show every comparison and identify the two tie decisions required for stability.

Exercises on merge costs and partitioning

Exercise 7

Draw the Merge sort recursion tree for an input of length 8. Write the total amount of merge work at each depth and derive the $\Theta(n \log n)$ bound without invoking the Master Theorem.

Exercise 8

Trace Lomuto partition on $[2, 8, 7, 1, 3, 5, 6, 4]$. After every iteration, record the values of i and j , the exchange performed, if any, and the four regions maintained by the scan.

Exercises on Quicksort costs and algorithm selection

Exercise 9

Derive the balanced and maximally unbalanced Quicksort recurrences. Construct an input that gives the worst case when the last record is always chosen as pivot.

Exercise 10

Choose among Insertion sort, Merge sort, and randomized Quicksort for each situation: a nearly sorted array, stable sorting of large records, and an in-memory array with a strict auxiliary-memory limit. State the representation and performance assumption behind each choice.

Reading pointers

- CLRS, *Introduction to Algorithms*, 3rd edition, Chapters 2 and 7.
- Kleinberg and Tardos, Section 5.1.
- Dasgupta, Papadimitriou, and Vazirani, Sections 2.2–2.3.

Thank you!



UNIVERSITY
OF TRIESTE