

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Department of Mathematics, Informatics and Geosciences

Course 298SM

ALGORITHMIC DESIGN

Afternotes

Author

Antonio Emanuele Cinà

Academic Year 2026–2027

Contents

A	Foundations and Algorithmic Reasoning	1
1	From problems to algorithms and programs	2
1.1	Problems, algorithms, and programs	2
1.2	The Fibonacci sequence	4
1.3	Chapter exercises	13
2	Asymptotic computational complexity	15
2.1	Cost criteria	15
2.2	Input size	16
2.3	Best, worst, and average cases	17
2.4	Asymptotic notation	18
2.5	Properties of asymptotic bounds	21
2.6	Chapter exercises	27
3	Recurrences	28
3.1	From recursive code to a recurrence	28
3.2	The iteration method	29
3.3	Recursion trees	31
3.4	The master theorem	33
3.5	Binary search	39
3.6	Chapter exercises	40
4	Sorting	41
4.1	The sorting problem	42
4.2	Bubble sort	44
4.3	Insertion sort	47
4.4	Merge sort	48
4.5	Quicksort	50
4.6	Comparing the four methods	54
4.7	Chapter exercises	55

5	Heaps, priority queues, and Heapsort	56
5.1	The shape of a binary heap	57
5.2	Heap order: max-heaps and min-heaps	59
5.3	Restoring heap order	60
5.4	Building a heap	62
5.5	Priority queues	63
5.6	Heapsort	63
5.7	Where Heapsort fits	65
5.8	Chapter exercises	65
6	The limits of comparison sorting	66
6.1	The comparison model	66
6.2	Counting sort	68
6.3	Radix sort	71
6.4	Choosing a sorting method	75
6.5	Chapter exercises	76

Part A

Foundations and Algorithmic Reasoning

Chapter 1

From problems to algorithms and programs

Sources for further study

Book. Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 3rd ed.; Chapters 1–2.

Book. Dasgupta, Papadimitriou, and Vazirani, *Algorithms*; Prologue, Sections 0.1–0.2.

Book. Horowitz, Sahni, and Rajasekaran, *Computer Algorithms*; Sections 1.1–1.2.

Before comparing algorithms, we need to separate three objects that are easily confused: the mathematical task to be solved, the method used to solve it, and the program that realizes that method on a machine. This distinction gives us a precise language for stating what correctness means without prematurely committing to Python or pseudocode.

1.1 Problems, algorithms, and programs

An informal request such as “find a large value” is not yet suitable for analysis. We must first decide which inputs are valid and which outputs count as solutions. Once that contract is fixed, different methods can be compared against the same task.

Definition 1.1.1 (Computational problem). A *computational problem* is a triple

$$\mathcal{P} = (I, S, R),$$

where I is a set of admissible instances, S is a set of possible solutions, and $R \subseteq I \times S$ is the relation of acceptable instance–solution pairs. For $x \in I$, an output $y \in S$ solves the instance exactly when $(x, y) \in R$.

Using a relation rather than a function allows a problem to have several acceptable answers. If the task is to return an index of a maximum in $[4, 9, 9, 2]$, either occurrence of 9 may be valid unless the contract specifically asks for the first or last one.

Definition 1.1.2 (Algorithm). An *algorithm* for $\mathcal{P} = (I, S, R)$ is a finite, unambiguous procedure that, for every $x \in I$, terminates and produces some $y \in S$ such that $(x, y) \in R$.

Definition 1.1.3 (Program). A *program* is a concrete expression of one or more algorithms in a programming language, together with choices of representations, interfaces, and machine-level operations. Several programs may implement the same algorithm, and the same problem may admit fundamentally different algorithms.

1.1.1 A formal running example: maximum

The maximum-value problem is small enough to specify completely, yet rich enough to show why the preceding distinctions matter. Let D be a totally ordered domain and let the input be a nonempty finite sequence over D . Define

$$I = \bigcup_{n \geq 1} D^n, \quad S = D,$$

and

$$(A, m) \in R_{\max} \iff (\exists i \in \{0, \dots, |A| - 1\}) A[i] = m \\ \wedge (\forall j \in \{0, \dots, |A| - 1\}) A[j] \leq m.$$

The first condition requires the result to be an input value; without it, an artificial value larger than every record would satisfy only the inequality.

Definition 1.1.4 (Precondition and postcondition). A *precondition* identifies the initial states for which a routine is required to work. A *postcondition* relates the initial state, final state, and returned result. For maximum, the precondition is $|A| \geq 1$ and the postcondition is precisely $(A, m) \in R_{\max}$.

The specification deliberately says nothing about how the answer is obtained. A sequential algorithm can maintain the best value seen so far:

Linear scan for a maximum

```
Maximum(A)
  require length(A) >= 1 # precondition
  best <- A[0]
  for i <- 1 to length(A)-1
    if A[i] > best
      best <- A[i]
  return best
```

On $A = [4, 1, 7, 7, 2]$, the successive values of **best** are 4, 4, 7, 7, 7. The trace explains one execution; the following invariant explains all valid executions. Immediately before the iteration with index i , **best** is the maximum of $A[0], \dots, A[i-1]$. It is true initially because the processed prefix contains only $A[0]$. Comparing $A[i]$ preserves it for the extended prefix, and at termination, the prefix is the whole input. Thus, the returned value satisfies the postcondition. The scan performs exactly $n - 1$ comparisons and uses constant auxiliary storage.

A parallel reduction could solve the same relation by comparing several pairs simultaneously. Its implementation and cost would differ, but its admissible inputs and correctness obligations would not. This is the practical value of separating a problem from an algorithm and a program.

The rest of the chapter uses Fibonacci numbers to make the comparison more evident: four methods return the same mathematical value but perform very different amounts of work. That comparison motivates the cost notation introduced in Chapter 2.

1.2 The Fibonacci sequence

We introduce the Fibonacci sequence¹ because one short recurrence generates several contrasting algorithms, providing a controlled example in which the problem remains fixed while the method and representation change.

The Fibonacci sequence is defined by

$$F_1 = F_2 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad \text{for } n \geq 3.$$

Its first terms are

$$1, 1, 2, 3, 5, 8, 13, 21, \dots$$

The computational problem is simple: given an integer $n \geq 1$, compute F_n . Actually, given this computational problem, one can derive at least four different methods from the same definition. They return the same mathematical value, but they use very different representations and amounts of work.

1.2.1 A numerical formula

The first method asks whether an algebraic identity can replace an iterative or recursive computation. We therefore look for a formula that determines F_n directly from n .

In doing so, we first look for exponential sequences a^n that satisfy the Fibonacci recurrence. We start from the Fibonacci recurrence:

$$F_n = F_{n-1} + F_{n-2}.$$

We then suppose that a sequence satisfying the same recurrence takes the exponential form

$$X_n = a^n.$$

Then, by construction, we have that:

$$X_{n-1} = a^{n-1} \quad \text{and} \quad X_{n-2} = a^{n-2}.$$

Substituting these expressions into $X_n = X_{n-1} + X_{n-2}$ gives $a^n = a^{n-1} + a^{n-2}$.

If $a \neq 0$, we can divide by a^{n-2} and obtain

$$a^2 = a + 1,$$

or equivalently

$$a^2 - a - 1 = 0.$$

¹Wikipedia - Fibonacci sequence

The two roots are:

$$\varphi = \frac{1 + \sqrt{5}}{2}, \quad \hat{\varphi} = \frac{1 - \sqrt{5}}{2}.$$

Both φ^n and $\hat{\varphi}^n$ satisfy the recurrence, and so does every linear combination of them, meaning that any sequence of the form

$$A\varphi^n + B\hat{\varphi}^n$$

also satisfies it. Choosing A and B so that the initial conditions $F_1 = 1$ and $F_2 = 1$ are satisfied yields Binet's formula.

$$F_n = \frac{\varphi^n - \hat{\varphi}^n}{\sqrt{5}}. \quad (1.1)$$

Proposition 1.2.1 (Correctness of Binet's formula). *For every integer $n \geq 1$, Equation (1.1) equals the recursively defined Fibonacci number F_n .*

Proof. The two roots satisfy $x^2 = x + 1$. Multiplying by x^{n-2} shows that both sequences φ^n and $\hat{\varphi}^n$ satisfy $X_n = X_{n-1} + X_{n-2}$, and their linear combination does too. Moreover, $\varphi - \hat{\varphi} = \sqrt{5}$, so the formula gives $F_1 = 1$; since $\varphi + \hat{\varphi} = 1$, the factorization $\varphi^2 - \hat{\varphi}^2 = (\varphi - \hat{\varphi})(\varphi + \hat{\varphi}) = \sqrt{5}$ also gives $F_2 = 1$. The recurrence and these two values determine every later term uniquely. $\square$

However, Binet's formula illustrates how mathematical equivalence does not necessarily guarantee numerical reliability in a finite representation. Indeed, the formula is exact over the real numbers. A direct implementation on a digital system (like your laptop) would use finite-precision approximations of $\sqrt{5}$, φ , and $\hat{\varphi}$. Rounding may therefore produce an incorrect integer for large n . This is our first example of a distinction that will recur throughout the course, i.e., a correct mathematical identity does not automatically give a correct implementation under every representation.

1.2.2 The direct recursive algorithm

The Fibonacci recurrence does more than define the sequence mathematically. It also describes a direct computational procedure. To compute F_n , we first compute F_{n-1} and F_{n-2} and then add the two results. When the index reaches one of the initial values, the computation stops and returns 1. This reasoning leads to the following recursive algorithm, defined for every integer $n \geq 1$.

Direct recursive Fibonacci

```
Fibonacci-recursive(n)
  require n >= 1
  if n <= 2
    return 1
  return Fibonacci-recursive(n-1) + Fibonacci-recursive(n-2)
```

The algorithm is a literal translation of the mathematical definition. The base case implements $F_1 = F_2 = 1$, while the recursive case implements $F_n = F_{n-1} + F_{n-2}$. Its correctness is therefore immediate from the recurrence.

However, its weakness appears when we draw the calls. The two recursive calls are evaluated independently, even when they require the same Fibonacci number.

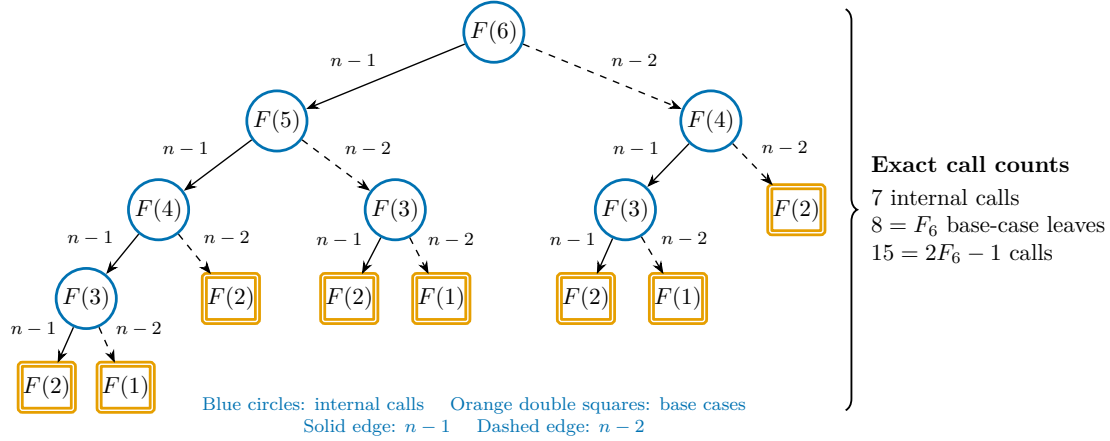


Figure 1.1: The call tree of the direct recursive algorithm for F_6 . Equal subproblems are recomputed in different branches.

For example, computing F_6 evaluates F_4 once inside the F_5 call and again as the second child of the root; the same duplication continues at lower levels. This leads to a fundamental distinction. Correctness guarantees that an algorithm produces an admissible output for every valid input, but it does not guarantee that the required time or memory is acceptable. In short, the *correctness of an algorithm does not imply its efficiency*.

The tree permits an exact count. Let L_n be its number of leaves. Since a call with $n = 1$ or $n = 2$ is a leaf,

$$L_1 = L_2 = 1, \quad L_n = L_{n-1} + L_{n-2}.$$

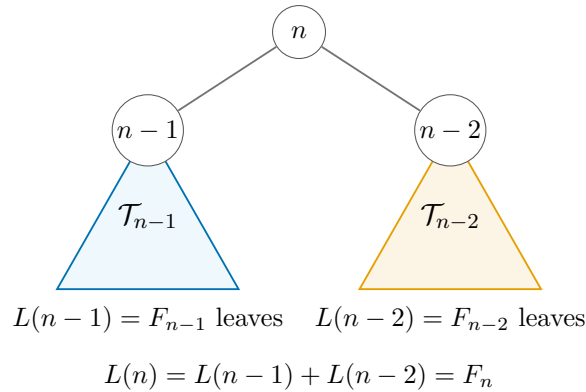


Figure 1.2: The recursion tree for n is composed of the recursion trees for $n-1$ and $n-2$.

Lemma 1.2.2 (Leaves in the Fibonacci recursion tree). *The recursion tree of $\text{Fibonacci-recursive}(n)$ has exactly F_n leaves.*

Proof. Let $L(n)$ denote the number of leaves in the recursion tree generated when computing F_n via the naive recursive algorithm. We show $L(n) = F_n$ for all $n \geq 1$ by *induction* on n .

Base cases ($n = 1, 2$). For $n = 1$ and $n = 2$, the algorithm returns immediately without making any recursive call. Its recursion tree is therefore a single node, which is trivially a leaf. Hence

$$L(1) = 1 = F_1, \quad L(2) = 1 = F_2,$$

so the claim holds for both base cases.

Induction hypothesis. Fix $n > 2$, and suppose that for every k with $1 \leq k < n$,

$$L(k) = F_k.$$

In particular, this gives us

$$L(n-1) = F_{n-1} \quad \text{and} \quad L(n-2) = F_{n-2}.$$

Inductive step. For input $n > 2$, the algorithm makes exactly two recursive calls: one on $n-1$ and one on $n-2$. Consequently, as shown in Figure 1.2, the recursion tree for n is obtained by attaching the recursion tree for $n-1$ and the recursion tree for $n-2$ as the two subtrees of a new root node labeled n .

Since $n > 2$, this root always makes recursive calls, so it is never itself a leaf. Every leaf of the full tree therefore lies in exactly one of the two subtrees $\mathcal{T}_{n-1}$ or $\mathcal{T}_{n-2}$, the shaded regions in the figure, and no leaf is shared between them. Counting leaves is thus a simple matter of addition:

$$L(n) = L(n-1) + L(n-2).$$

Applying the induction hypothesis to substitute $L(n-1)$ and $L(n-2)$ gives

$$L(n) = F_{n-1} + F_{n-2} = F_n,$$

where the last equality is precisely the Fibonacci recurrence.

Conclusion. By induction, $L(n) = F_n$ for all $n \geq 1$: the recursion tree for computing F_n has exactly F_n leaves. $\square$

Lemma 1.2.3 (Internal nodes in the Fibonacci recursion tree). *Let $I(n)$ denote the number of internal nodes in the recursion tree for computing F_n . Then $I(n) = L(n) - 1$, i.e. the tree has one fewer internal node than leaves.*

Proof. We prove it by induction on n , following the same decomposition used for $L(n)$.

Base cases ($n = 1, 2$). As before, the recursion tree for $n = 1$ or $n = 2$ is a single node with no recursive calls. A single node is a leaf, not an internal node, so

$$I(1) = 0 = L(1) - 1, \quad I(2) = 0 = L(2) - 1.$$

Induction hypothesis. Fix $n > 2$, and suppose that for every k with $1 \leq k < n$,

$$I(k) = L(k) - 1.$$

In particular,

$$I(n-1) = L(n-1) - 1 \quad \text{and} \quad I(n-2) = L(n-2) - 1.$$

Inductive step. As shown in Figure 1.2, the recursion tree for n consists of the trees for $n-1$ and $n-2$ joined below a new root. That root is itself an internal node because it makes the two recursive calls, and every other internal node of the full tree lies in exactly one of the two subtrees. Hence

$$I(n) = I(n-1) + I(n-2) + 1.$$

Substituting the induction hypothesis,

$$I(n) = (L(n-1) - 1) + (L(n-2) - 1) + 1 = (L(n-1) + L(n-2)) - 1 = L(n) - 1,$$

where the last step uses $L(n) = L(n-1) + L(n-2)$ from the previous lemma.

Conclusion. By strong induction, $I(n) = L(n) - 1$ for all $n \geq 1$. □

Lastly, the lemmas give $L_n = F_n$ leaf nodes and $L_n - 1$ internal nodes. The total number of calls is consequently

$$C_n = 2F_n - 1.$$

The number of calls made by the recursive Fibonacci algorithm grows exponentially because it repeatedly solves the same overlapping subproblems.

1.2.3 Iterative algorithms

The recursion tree exposes repeated subproblems, so the natural next step is to retain previous results instead of recomputing them. The two iterative versions below also separate time improvement from memory improvement.

The recursive tree shows what should be reused. Once F_{i-1} and F_{i-2} are known, F_i needs one addition. An initial iterative version stores every value.

Array-based iterative Fibonacci

```
Fibonacci-array(n)
  require n >= 1
  if n <= 2
    return 1
  allocate F[1 .. n]
  F[1] <- 1
  F[2] <- 1
  for i <- 3 to n
    F[i] <- F[i-1] + F[i-2]
  return F[n]
```

For $n \geq 3$, the loop performs $n - 2$ additions. The array occupies a number of memory cells proportional to n . Yet, at iteration i , only the two preceding values are used. Keeping only these values gives a space-efficient version.

Space-efficient iterative Fibonacci

```

Fibonacci-iterative(n)
    require n >= 1
    if n <= 2
        return 1
    previous <- 1
    current <- 1
    for i <- 3 to n
        next <- previous + current
        previous <- current
        current <- next
    return current

```

The two iterative algorithms perform essentially the same arithmetic work. Their memory use differs because the second representation retains only the state required by the next step. Under the usual unit-cost model, the array version takes linear time and linear auxiliary space, while the optimized version takes linear time and constant auxiliary space. Chapter 2 will make the words “linear” and “constant” precise.

For $n = 6$, the pair $(\text{previous}, \text{current})$ evolves as

$$(1, 1), (1, 2), (2, 3), (3, 5), (5, 8).$$

This trace is useful evidence, but it is not a proof for every n . A complete argument uses the loop invariant that, immediately before iteration i , $\text{previous} = F_{i-2}$ and $\text{current} = F_{i-1}$. The update assigns their sum F_i to current , so the invariant advances by one index. At termination, $\text{current} = F_n$.

The following short experiment compares Binet’s formula with an exact iterative computation. Run the script and observe how the difference changes as n increases.

Experiment: finite-precision Binet formula

```

from math import sqrt

def fibonacci(n):
    """Compute F_n with exact iterative Fibonacci algorithm."""
    if not isinstance(n, int) or n < 1:
        raise ValueError("n must be a positive integer")

    previous = 0
    current = 1

    for _ in range(n):
        previous, current = current, previous + current
    return previous

def binet(n):
    """Compute F_n using floating point arithmetic."""
    if not isinstance(n, int) or n < 1:

```

```

        raise ValueError("n must be a positive integer")

    phi = (1 + sqrt(5)) / 2
    phi_hat = (1 - sqrt(5)) / 2
    return round((phi**n - phi_hat**n) / sqrt(5))

for n in [10, 50, 71, 80]:
    exact = fibonacci(n)
    approximate = binet(n)

    print(
        f"n={n:3d}  exact={exact}  "
        f"Binet={approximate}  "
        f"difference={approximate - exact}"
    )

```

1.2.4 Fast matrix powering

The iterative algorithm avoids repeated computations, but it still reaches F_n by generating all the preceding Fibonacci numbers in order. It therefore performs a number of updates proportional to n . We now ask whether several consecutive updates can be combined so that the index grows more quickly.

The key idea is to represent two consecutive Fibonacci numbers as a single state vector. A fixed matrix transforms the state containing F_k and F_{k-1} into the state containing F_{k+1} and F_k . Repeating this transformation corresponds to raising the matrix to a power.

We set $F_0 = 0$, which is consistent with $F_2 = F_1 + F_0$.

Lemma 1.2.4 (Fibonacci matrix identity). *Let $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. For every $n \geq 2$,*

$$A^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}.$$

Proof. We proceed by induction on n .

Base case ($n = 2$). Since $A^1 = A$, we must check that

$$A = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}.$$

Indeed $F_2 = 1$, $F_1 = 1$, and $F_0 = 0$ (using the convention $F_0 = 0$ fixed above), so the right-hand side is $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = A$, matching the left-hand side.

Induction hypothesis. Fix $n > 2$, and suppose the identity holds for $n - 1$ in place of n :

$$A^{n-2} = \begin{pmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{pmatrix}.$$

Inductive step. Write $A^{n-1} = A^{n-2} A$, and substitute the induction hypothesis for the first factor:

$$A^{n-1} = \begin{pmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_{n-1} + F_{n-2} & F_{n-1} \\ F_{n-2} + F_{n-3} & F_{n-2} \end{pmatrix}.$$

Every entry in the top row and left column is now a sum of two consecutive Fibonacci numbers, so the Fibonacci recurrence collapses each one:

$$F_{n-1} + F_{n-2} = F_n, \quad F_{n-2} + F_{n-3} = F_{n-1}.$$

Substituting gives

$$A^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix},$$

which is exactly the claimed identity for n .

Conclusion. By induction, $A^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}$ for every $n \geq 2$. □

If A^n were computed by multiplying A by itself repeatedly, the matrix representation would not provide any improvement. In fact,

$$A^n = \underbrace{A \cdot A \cdots A}_{n \text{ factors}}$$

requires $n - 1$ matrix multiplications. The algorithm would still proceed through a linear number of stages, just as the iterative Fibonacci algorithm advances through the indices one at a time. To make it faster, we can use repeated squaring.

If n is even, we can write $n = 2m$. Applying the exponent law gives:

$$A^n = A^{2m} = A^m A^m = (A^m)^2.$$

Thus, instead of computing A^n through $n - 1$ successive multiplications, we can compute the smaller power A^m once and then square it. If the exponent is odd, we can write $n = 2m + 1$ and use

$$A^n = A^{2m+1} = A(A^m)^2.$$

Combining the two cases gives

$$A^n = \begin{cases} (A^{n/2})^2, & n \text{ even,} \\ A(A^{\lfloor n/2 \rfloor})^2, & n \text{ odd.} \end{cases}$$

The important point is that both cases require the computation of only one smaller power. Its exponent is approximately half of the original exponent.

For example, the recursive computation of A^{13} considers the exponents

$$13, 6, 3, 1.$$

Once A is known, the required powers can be reconstructed as

$$A^3 = A(A)^2, \quad A^6 = (A^3)^2, \quad A^{13} = A(A^6)^2.$$

The following procedure implements this idea. The symbol I denotes the identity matrix with the same dimensions as A .

Matrix powering by repeated squaring

```

Matrix-power(A, k)
  require k >= 0
  if k == 0
    return I
  if k == 1
    return A

  half <- Matrix-power(A, floor(k / 2))
  result <- half * half

  if k is odd
    result <- result * A

  return result

```

The procedure makes only one recursive call. After that call returns, it squares the resulting matrix and performs one additional multiplication when the exponent is odd. The Fibonacci algorithm applies this procedure to the matrix

$$Q = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}.$$

By the matrix identity proved above, the entry in position $(0,0)$ of Q^{n-1} is F_n .

Fibonacci by fast matrix powering

```

Fibonacci-matrix(n)
  require n >= 0
  if n == 0
    return 0

  Q <- [[1, 1],
        [1, 0]]

  P <- Matrix-power(Q, n - 1)
  return P[0, 0]

```

For $n = 0$, the algorithm returns the value fixed by convention. For $n \geq 1$, it computes Q^{n-1} and extracts F_n from its upper-left entry. The general matrix-powering procedure is kept separate because repeated squaring is not specific to Fibonacci numbers and can be reused whenever a large power must be computed.

At each recursive level, the exponent is replaced by its integer half. Under the unit-cost assumption, multiplying two fixed-size matrices requires constant work. The running time therefore satisfies

$$T(n) = \begin{cases} \Theta(1), & n \leq 1, \\ T(\lfloor n/2 \rfloor) + \Theta(1), & n > 1. \end{cases}$$

After j recursive levels, the remaining exponent is at most

$$\left\lfloor \frac{n}{2^j} \right\rfloor.$$

The recursion stops when this value reaches 1, which occurs when 2^j is at least n . Hence, the recursion has $\Theta(\log n)$ levels and

$$T(n) = \Theta(\log n).$$

This is the essential advantage of repeated squaring. The iterative Fibonacci algorithm reduces the remaining index from n to $n - 1$, while fast matrix powering reduces the remaining exponent from n to approximately $n/2$.

The cost statement still depends on the arithmetic model. The matrices have fixed dimension, but their entries become larger as n increases. Under the unit-cost model, each arithmetic operation is counted as one step, giving $\Theta(\log n)$ matrix multiplications. With arbitrary-precision arithmetic, operations on large integers require more time because the entries contain an increasing number of bits. The unit-cost result therefore describes the number of matrix operations, while a complete bit-complexity analysis must also account for the size of the numbers being manipulated.

The following plot recalls the motivation for seeking a faster representation. It compares the exponential number of calls made by direct recursion with the linear number of additions made by iteration. Matrix powering is not included as a third curve because a matrix multiplication and a scalar addition are different charged operations. Under the unit-cost model, its number of matrix multiplications grows logarithmically.

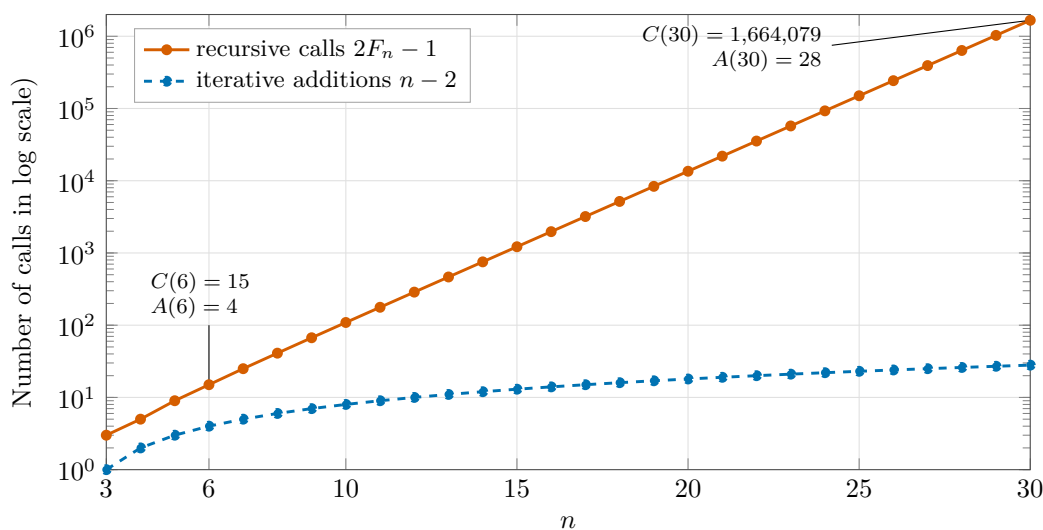


Figure 1.3: Under the stated unit-cost counters, direct Fibonacci recursion uses exponentially many calls, whereas iterative evaluation uses linearly many additions. The curves compare growth rates under different charged operations rather than equal execution times.

1.3 Chapter exercises

Exercise 1.3.1 (Designing a data contract). A sensor file contains a timestamp, a station identifier, and a temperature, with some readings missing. Specify a computational problem that returns the maximum valid temperature and its timestamp. State how ties

and missing values are handled, then implement a one-pass algorithm and test its invariant on an all-negative dataset.

Exercise 1.3.2 (Measuring repeated work). Implement recursive, memoized, and iterative Fibonacci. Instrument recursive calls, cache hits, additions, and maximum recursion depth rather than reporting only elapsed time. Run a range of feasible inputs, plot the counters, and explain which measured quantity supports each analytical claim.

Exercise 1.3.3 (Finite representation experiment). Compare the floating-point Binet implementation with exact integer iteration. Find the first input on your Python installation for which their results differ, report both values, and explain why this is an implementation failure rather than a failure of Equation (1.1).

Exercise 1.3.4 (Reusable fast powering). Implement iterative or recursive repeated squaring for 2×2 integer matrices. Validate it against repeated multiplication on small exponents, use it to compute F_n , and report both the number of matrix multiplications and the bit length of the largest entry. Explain why these two measurements correspond to different cost models.

Chapter 2

Asymptotic computational complexity

Sources for further study

Book. Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 3rd ed.; Sections 2.2 and 3.1–3.2.

Book. Kleinberg and Tardos, *Algorithm Design*; Sections 2.1–2.4.

Book. Horowitz, Sahni, and Rajasekaran, *Computer Algorithms*; Section 1.3.

The algorithms studied in Chapter 1 solve the same Fibonacci problem but use very different amounts of computational work. Direct recursion makes an exponentially growing number of calls, iteration performs a linear number of additions, and repeated squaring uses a logarithmic number of matrix multiplications under the unit-cost model.

These comparisons already rely on several choices. We decided which operations to count, used the Fibonacci index n as the size parameter, and specified how arithmetic operations were charged. We now make these choices explicit and introduce a mathematical language for comparing the growth of cost functions. In the rest of these notes, we will write $T(n)$ for algorithmic running time and $S(n)$ for its auxiliary space when the input size is n .

2.1 Cost criteria

The execution time observed on a computer depends on the processor, programming language, compiler, operating system, and details of the implementation. A faster measurement does not necessarily identify the better algorithm because changing the machine may change the measurement.

Algorithm analysis, therefore, replaces elapsed time with an abstract count of selected operations. Before stating a cost, we must specify which resource is being measured and how each operation is charged.

Definition 2.1.1 (Uniform-cost model). In the *uniform-cost model*, elementary operations on machine-sized values are charged a constant cost. Comparisons, assignments, arithmetic operations, and array accesses are typical examples.

This model is deliberately simple and serves as a useful first approximation when the operands fit into a fixed machine word. It also keeps the analysis focused on the structure of the algorithm.

Definition 2.1.2 (Logarithmic-cost model). In the *logarithmic-cost model*, the cost of an operation depends on the number of bits needed to represent its operands.

For example, adding two b -bit integers requires examining a number of bits proportional to b under the usual addition algorithm. The operation cannot be regarded as independent of the operand size.

The distinction appeared at the end of Chapter 1. Under the uniform-cost model, multiplying two fixed-size 2×2 matrices takes constant time. Repeated squaring consequently computes the required matrix power in $\Theta(\log n)$ matrix multiplications. Under the logarithmic-cost model, the matrix entries are Fibonacci numbers whose representations grow with n , so the arithmetic operations do not have a constant bit cost. Neither model is universally correct. A useful analysis states its model and uses it consistently. Under uniform cost, the optimized iterative Fibonacci algorithm of Chapter 1 performs $n - 2$ constant-cost additions. Under bit cost, the additions become more expensive as their operands grow. The apparent disagreement is resolved once the charged resource is made explicit.

2.2 Input size

After choosing a cost model, we must express the cost as a function of the amount of information given to the algorithm. This quantity is called the input size.

Definition 2.2.1 (Input size). The *input size* is the length of the representation of an input, or a declared collection of representation parameters, relative to which the resource usage of an algorithm is measured.

For an array, n usually denotes the number of elements. For a string, it usually denotes the number of characters. The size of a graph is generally described using both $|V|$ and $|E|$ because the number of vertices alone does not determine the amount of graph data.

When the input is a number, its numerical value and the length of its representation must be distinguished. A positive integer m represented in binary requires

$$\lfloor \log_2 m \rfloor + 1$$

bits. If this length is denoted by b , then

$$2^{b-1} \leq m < 2^b.$$

An algorithm performing m iterations is linear in the numerical value m , but its worst-case running time is exponential in the binary input length b .

In Chapter 1, the Fibonacci index n was used directly as the parameter of the analysis. This is a valid and useful convention, but it must be stated. Calling an algorithm linear is incomplete unless we also say which input parameter appears in the linear bound.

2.3 Best, worst, and average cases

Input size alone does not determine an execution. Two arrays of the same length may cause an algorithm to perform different numbers of operations. Different instances of the same size may cause different executions. We therefore distinguish different costs among instances of the same size.

Let $\mathcal{I}_n$ be the set of valid inputs of size n , and let $T(I)$ be the cost of the algorithm on input I .

Definition 2.3.1 (Best-case cost). The best-case cost is

$$T_{\text{best}}(n) = \min_{I \in \mathcal{I}_n} T(I).$$

It is the smallest cost incurred by an input of size n .

Definition 2.3.2 (Worst-case cost). The worst-case cost is

$$T_{\text{worst}}(n) = \max_{I \in \mathcal{I}_n} T(I).$$

It is the largest cost incurred by an input of size n .

Definition 2.3.3 (Average-case cost). Let $p_n(I)$ be the probability that the input of size n is I . The average-case cost is

$$T_{\text{avg}}(n) = \sum_{I \in \mathcal{I}_n} p_n(I) T(I).$$

An average-case statement is incomplete unless the probability distribution p_n is specified. Sequential search provides the standard example. The algorithm compares the key with each array entry from left to right and returns only after finding a match or exhausting the array.

Sequential search

```
Sequential-search(A, key)
  for i <- 0 to length(A)-1
    if A[i] = key
      return i
  return Not-found
```

If the first entry contains the key, the algorithm makes one comparison. This is the best case. If the key occurs in the final position or does not occur, the algorithm makes n comparisons. This is the worst case.

Suppose that the search is successful and that each of the n positions is equally likely to contain the key. The expected number of comparisons is

$$\frac{1}{n} \sum_{i=1}^n i = \frac{n+1}{2}.$$

The best, worst, and average costs answer different questions. Unless another case or probability model is stated, this course uses the worst case as the default.

2.4 Asymptotic notation

Asymptotic analysis focuses on the behavior of a cost function for sufficiently large inputs. It preserves the rate of growth while abstracting from constant factors (e.g., additions) and behavior restricted to a finite initial part of the domain. Exact expressions such as $7n^2 + 3n + 10$ contain implementation-dependent constants and lower-order terms. Asymptotic notation retains the growth that *dominates* for sufficiently large inputs.

Definition 2.4.1 (Asymptotic complexity). The *asymptotic complexity* describes the eventual growth of $T(n)$ for an algorithm by comparing it with a reference function up to positive constant factors.

In other words, let $f, g: \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, and assume that $g(n) > 0$ for all sufficiently large n . The function f is being studied, while g represents a reference growth rate.

Definition 2.4.2 (Asymptotic upper bound).

$$O(g(n)) = \{ z(n) : \exists c > 0 \wedge n_0 \geq 0 : z(n) \leq c g(n), \forall n \geq n_0 \}$$

The relation $f(n) = O(g(n))$ means that f grows *no faster than* g , up to a constant factor, once n is sufficiently large. The function g is therefore an asymptotic upper bound for f .

Definition 2.4.3 (Asymptotic lower bound).

$$\Omega(g(n)) = \{ z(n) : \exists c > 0 \wedge n_0 \geq 0 : c g(n) \leq z(n), \forall n \geq n_0 \}$$

The relation $f(n) = \Omega(g(n))$ means that f grows *at least* as fast as g , up to a constant factor, once n is sufficiently large.

Definition 2.4.4 (Asymptotically tight bound).

$$\Theta(g(n)) = \{ z(n) : \exists c_1, c_2 > 0 \wedge n_0 \geq 0 : c_1 g(n) \leq z(n) \leq c_2 g(n), \forall n \geq n_0 \}$$

The relation $f(n) = \Theta(g(n))$ means that f and g have the *same asymptotic order* of growth. The function f is trapped between two positive constant multiples of g from some threshold onward. The asymptotically tight bound satisfies the following property:

$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

The constants c , c_1 , and c_2 must not depend on n . The threshold n_0 allows the functions to behave differently on a finite number of small inputs. Asymptotic notation describes eventual growth, not necessarily the behavior observed at the first few values.

Importantly, $O(g)$, $\Omega(g)$, and $\Theta(g)$ are sets of functions. The expression $f(n) = O(g(n))$ is a conventional abbreviation for $f \in O(g)$. It should not be manipulated as an ordinary symmetric equality. For example, $n \in O(n^2)$, whereas $n^2 \notin O(n)$.

Example 2.4.5 (Finding the crossover point). Consider the two modeled cost functions

$$T_{\text{linear}}(n) = 100n \quad \text{and} \quad T_{\text{quadratic}}(n) = n^2.$$

A crossover occurs when the two functions have the same value. We therefore solve

$$100n = n^2.$$

Moving all terms to one side and factoring gives

$$n^2 - 100n = 0 \quad \Longleftrightarrow \quad n(n - 100) = 0.$$

The two solutions are $n = 0$ and $n = 100$. The value $n = 0$ is the trivial intersection at the origin. The relevant crossover for positive input sizes is therefore $n = 100$. At this point, both models have cost: $T_{\text{linear}}(100) = T_{\text{quadratic}}(100) = 10,000$.

To determine which function is smaller on either side of the crossover, consider their difference:

$$T_{\text{quadratic}}(n) - T_{\text{linear}}(n) = n^2 - 100n = n(n - 100).$$

For positive input sizes, the sign depends only on $n - 100$. Therefore,

$$\begin{cases} n^2 < 100n, & 0 < n < 100, \\ n^2 = 100n, & n = 100, \\ n^2 > 100n, & n > 100. \end{cases}$$

Thus, the quadratic model is smaller before the crossover, while the linear model is smaller after the crossover.

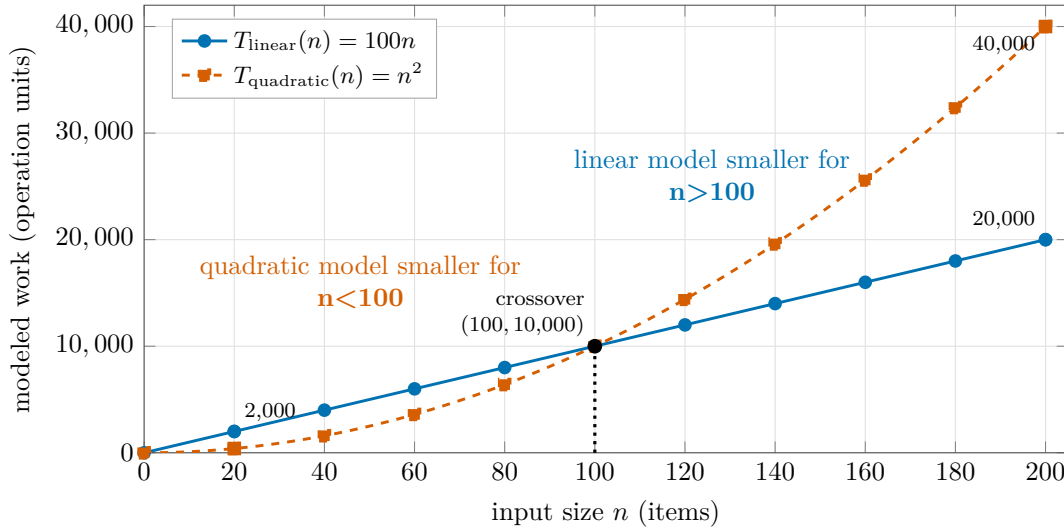


Figure 2.1: The linear model $100n$ becomes smaller than n^2 only after the crossover at $n = 100$, despite its better asymptotic growth.

Example 2.4.6 (Example for $\Theta(n^2)$). Consider $f(n) = \frac{1}{2}n^2 - 3n$. We show step by step how the constants c_1 , c_2 , and n_0 used above can be derived, and we tie each step to Figure 2.2.

Goal. We wish to show $f(n) = \Theta(n^2)$, i.e. to exhibit constants $c_1, c_2 > 0$ and n_0 such that

$$c_1 n^2 \leq f(n) \leq c_2 n^2 \quad \forall n \geq n_0.$$

The dominant-term intuition suggests $g(n) = n^2$, but this intuition alone is not a proof.

Step 1 - The upper bound. Since $-3n \leq 0$ for every $n \geq 0$,

$$f(n) = \frac{1}{2}n^2 - 3n \leq \frac{1}{2}n^2.$$

Thus $c_2 = \frac{1}{2}$ works for *all* $n \geq 0$, with no threshold required. In Figure 2.2 this is the dash-dotted curve $c_2 g(n) = \frac{1}{2}n^2$. The curve $f(n)$ never rises above it anywhere on the plot, including near the origin.

Step 2 - The lower bound. We look for c_1 such that $c_1 n^2 \leq \frac{1}{2}n^2 - 3n$. Treating c_1 as an unknown and rearranging,

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \iff n^2 \left(\frac{1}{2} - c_1 \right) \geq 3n \iff n \left(\frac{1}{2} - c_1 \right) \geq 3 \quad (\text{dividing by } n > 0).$$

This last step only isolates n on one side—giving a bound of the form $n \geq n_0$ —when the coefficient $\frac{1}{2} - c_1$ is *strictly positive*. If instead $c_1 \geq \frac{1}{2}$, the coefficient is zero or negative and no threshold exists: the term $c_1 n^2$ would grow at least as fast as the leading term of $f(n)$ itself, and $f(n)$ could never stay ahead of it for large n . Hence a necessary condition on c_1 is

$$c_1 < \frac{1}{2},$$

and for any such admissible c_1 , solving for n gives the corresponding threshold

$$n_0 = \left\lceil \frac{3}{\frac{1}{2} - c_1} \right\rceil.$$

Any c_1 in this range yields a valid witness, paired with its own n_0 : a smaller c_1 produces a smaller (earlier) threshold, while a c_1 closer to $\frac{1}{2}$ pushes the threshold further out.

We now fix a specific admissible value, $c_1 = \frac{1}{4}$ chosen simply because it yields a clean integer threshold. Substituting $c_1 = \frac{1}{4}$ into the general derivation above (or repeating the computation directly),

$$\frac{1}{4}n^2 \leq \frac{1}{2}n^2 - 3n \iff \frac{1}{4}n^2 \geq 3n \iff \frac{1}{4}n \geq 3 \iff n \geq 12,$$

where we again divided through by $n > 0$. So $c_1 = \frac{1}{4}$ does *not* hold for every n : it only becomes valid once $n \geq 12$, matching the general formula $n_0 = 3/(\frac{1}{2} - \frac{1}{4}) = 12$. This is precisely the role of n_0 in the definition of Θ : it is not chosen freely in advance, but derived as the point from which the chosen c_1 becomes valid.

In the figure this is the dashed curve $c_1 g(n) = \frac{1}{4}n^2$: for small n it lies *above* $f(n)$, so the bound genuinely fails there. Only from $n = 12$ onward does it drop below $f(n)$ and stay below.

Step 3 - Conclusion. Collecting the constructed constants,

$$c_1 = \frac{1}{4}, \quad c_2 = \frac{1}{2}, \quad n_0 = 12,$$

satisfies $c_1 n^2 \leq f(n) \leq c_2 n^2$ for every $n \geq n_0$, exactly as required by the definition of Θ . Hence

$$f(n) = \Theta(n^2).$$

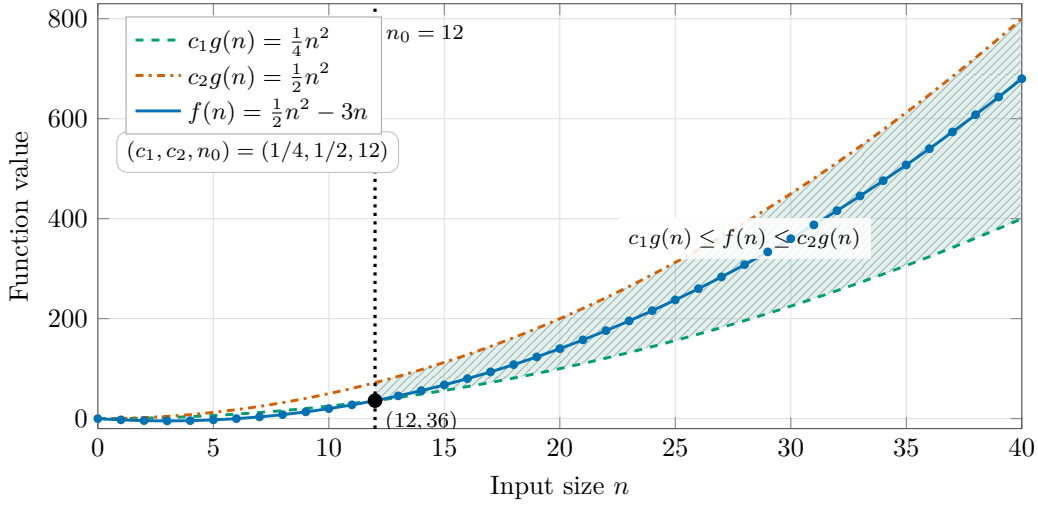


Figure 2.2: Explicit witnesses trap $\frac{1}{2}n^2 - 3n$ between two constant multiples of n^2 from $n_0 = 12$ onward.

2.5 Properties of asymptotic bounds

The definitions of O , Ω , and Θ provide precise criteria for comparing two functions. Applying their quantified definitions from the beginning every time, however, would make even simple analyzes unnecessarily long. We therefore establish a small collection of properties that allows us to combine previously proven bounds. These properties play the same role as familiar algebraic rules. Reflexivity allows a function to be compared with itself, transitivity allows bounds to be chained, and the dominant-term rule allows a sum to be simplified to its fastest-growing component.

Proposition 2.5.1. *Let f , g , and h be eventually non-negative functions, and let $a > 0$ be a constant. The following properties hold.*

1. Reflexivity

$$f(n) = O(f(n)), \quad f(n) = \Omega(f(n)), \quad f(n) = \Theta(f(n)).$$

2. Constant scaling

$$af(n) = \Theta(f(n)).$$

3. Transitivity

$$\begin{aligned}
f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) &\implies f(n) = O(h(n)), \\
f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) &\implies f(n) = \Omega(h(n)), \\
f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) &\implies f(n) = \Theta(h(n)).
\end{aligned}$$

4. Upper and lower bound duality

$$f(n) = O(g(n)) \iff g(n) = \Omega(f(n)).$$

5. Characterization of a tight bound

$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

6. Symmetry of a tight bound

$$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n)).$$

7. Dominant term in a sum

$$f(n) + g(n) = \Theta(\max\{f(n), g(n)\}).$$

Proof. We verify each property directly from the definitions.

Reflexivity and constant scaling. The inequalities

$$f(n) \leq f(n) \leq f(n)$$

hold whenever $f(n) \geq 0$. Therefore, choosing both multiplicative constants equal to 1 proves $f(n) = \Theta(f(n))$ and consequently also the corresponding O and Ω bounds.

For any constant $a > 0$, we have

$$af(n) \leq af(n) \leq af(n).$$

Thus $af(n) = \Theta(f(n))$. A positive constant factor changes the numerical value of a cost but not its asymptotic rate of growth.

Transitivity. Suppose that $f(n) = O(g(n))$ and $g(n) = O(h(n))$. By definition, there exist positive constants c_1, c_2 and thresholds n_1, n_2 such that

$$f(n) \leq c_1 g(n) \quad \forall n \geq n_1,$$

and

$$g(n) \leq c_2 h(n) \quad \forall n \geq n_2.$$

Both inequalities hold simultaneously when

$$n \geq \max\{n_1, n_2\}.$$

For these values of n ,

$$f(n) \leq c_1 g(n) \leq c_1 c_2 h(n).$$

Thus the positive constant c_1c_2 , together with the threshold $\max\{n_1, n_2\}$, satisfies the definition of $f(n) = O(h(n))$.

The proof for Ω is obtained by reversing the inequalities. Since a Θ bound consists of both an upper and a lower bound, transitivity for Θ follows by applying the two preceding results.

Upper and lower bound duality. Suppose that $f(n) = O(g(n))$. Then, for some $c > 0$ and all sufficiently large n ,

$$f(n) \leq cg(n).$$

Dividing by the positive constant c gives

$$\frac{1}{c}f(n) \leq g(n).$$

This is precisely an asymptotic lower bound for g in terms of f , and therefore $g(n) = \Omega(f(n))$. The converse follows by applying the same reasoning in the opposite direction.

Characterization and symmetry of a tight bound. If $f(n) = \Theta(g(n))$, then there exist $c_1, c_2 > 0$ and n_0 such that

$$c_1g(n) \leq f(n) \leq c_2g(n) \quad \forall n \geq n_0.$$

The right inequality proves $f(n) = O(g(n))$, while the left inequality proves $f(n) = \Omega(g(n))$. Conversely, suppose that $f(n) = O(g(n))$ holds with constant $c_2 > 0$ from threshold n_2 onward, and that $f(n) = \Omega(g(n))$ holds with constant $c_1 > 0$ from threshold n_1 onward. For every

$$n \geq \max\{n_1, n_2\},$$

both bounds therefore hold simultaneously, giving

$$c_1g(n) \leq f(n) \leq c_2g(n).$$

These are precisely the lower and upper inequalities required by the definition of Θ . Hence,

$$f(n) = \Theta(g(n)).$$

The same inequalities can be rewritten as

$$\frac{1}{c_2}f(n) \leq g(n) \leq \frac{1}{c_1}f(n).$$

They show that $g(n) = \Theta(f(n))$, proving symmetry.

Dominant term in a sum. Let

$$m(n) = \max\{f(n), g(n)\}.$$

Since both functions are eventually non-negative, the larger term cannot exceed their sum:

$$m(n) \leq f(n) + g(n).$$

Moreover, both $f(n)$ and $g(n)$ are at most $m(n)$, so

$$f(n) + g(n) \leq 2m(n).$$

Consequently,

$$m(n) \leq f(n) + g(n) \leq 2m(n),$$

which proves

$$f(n) + g(n) = \Theta(m(n)) = \Theta(\max\{f(n), g(n)\}).$$

□

The dominant-term property formalizes a simplification used throughout algorithm analysis. If one term grows asymptotically faster than all the others, then it determines the growth of the entire sum. For example,

$$n^2 + 7n + 4 = \Theta(n^2),$$

because n^2 eventually dominates both $7n$ and the constant term.

More generally, if $f(n) = O(g(n))$ and both functions are eventually non-negative, then

$$f(n) + g(n) = \Theta(g(n)).$$

The smaller term may affect the running time for concrete input sizes, but it does not change the asymptotic growth class.

2.5.1 Comparison by limits

The definitions based on explicit constants and thresholds remain the formal basis of asymptotic notation. For many familiar functions, however, limits provide a faster way to determine and prove their asymptotic relationship. The central idea is to study the ratio:

$$\frac{f(n)}{g(n)}.$$

This ratio measures the size of $f(n)$ relative to $g(n)$. If it approaches zero, then f becomes negligible compared with g . If it approaches a positive constant, the two functions grow at the same asymptotic rate. If it grows without bound, then f eventually dominates g .

Theorem 2.5.2 (Comparison by a ratio limit). *Let f and g be eventually positive functions.*

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \implies f(n) = O(g(n)), \\ L \in (0, \infty) & \implies f(n) = \Theta(g(n)), \\ +\infty & \implies f(n) = \Omega(g(n)). \end{cases}$$

Proof. Suppose first that:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

By the definition of the limit, there exists a threshold n_0 such that

$$\frac{f(n)}{g(n)} \leq 1 \quad \forall n \geq n_0.$$

Since $g(n) > 0$ for sufficiently large n , multiplying by $g(n)$ gives

$$f(n) \leq g(n).$$

Thus $f(n) = O(g(n))$, with multiplicative constant $c = 1$.

Suppose instead that:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty.$$

The ratio eventually exceeds every fixed positive value. In particular, there exists n_0 such that

$$\frac{f(n)}{g(n)} \geq 1 \quad \forall n \geq n_0.$$

It follows that

$$f(n) \geq g(n),$$

which proves $f(n) = \Omega(g(n))$.

Finally, suppose that

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L$$

for some constant L with $0 < L < \infty$. Choose $\varepsilon = L/2$. For every sufficiently large n ,

$$L - \frac{L}{2} \leq \frac{f(n)}{g(n)} \leq L + \frac{L}{2}.$$

Therefore,

$$\frac{L}{2} \leq \frac{f(n)}{g(n)} \leq \frac{3L}{2}.$$

Multiplying by the eventually positive function $g(n)$ gives

$$\frac{L}{2}g(n) \leq f(n) \leq \frac{3L}{2}g(n).$$

The constants $L/2$ and $3L/2$ are positive witnesses for $f(n) = \Theta(g(n))$. □

Example 2.5.3. Consider

$$f(n) = n \log n \quad \text{and} \quad g(n) = n^2.$$

Their ratio is

$$\frac{f(n)}{g(n)} = \frac{n \log n}{n^2} = \frac{\log n}{n}.$$

Since

$$\lim_{n \rightarrow \infty} \frac{\log n}{n} = 0,$$

the theorem gives

$$n \log n = O(n^2).$$

The limit provides more information than a non-strict upper bound. It shows that $n \log n$ becomes an increasingly small fraction of n^2 as n grows.

The ratio test is sufficient but not necessary for a tight bound. A ratio may fail to converge even when two functions have the same order. For example, $n(2 + \sin n) = \Theta(n)$, although its ratio to n oscillates between 1 and 3. In that situation, the original constant-and-threshold definition remains decisive.

The ratio method also explains several standard simplifications. Let

$$P(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0$$

be a polynomial with $a_k > 0$. Dividing by its highest-degree term gives

$$\frac{P(n)}{n^k} = a_k + \frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \cdots + \frac{a_0}{n^k}.$$

Every fraction after a_k tends to zero, so

$$\lim_{n \rightarrow \infty} \frac{P(n)}{n^k} = a_k.$$

Since a_k is positive and finite, it follows that

$$P(n) = \Theta(n^k).$$

Logarithms in any two fixed bases greater than one also differ only by a positive constant factor. Indeed,

$$\log_a n = \frac{\log_b n}{\log_b a}.$$

Consequently,

$$\log_a n = \Theta(\log_b n),$$

which is why the base is normally omitted in asymptotic expressions.

2.5.2 Ranking functions

For eventually positive functions, we write

$$f \prec g$$

when

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

This notation means that f grows strictly more slowly than g . For fixed constants $0 < \varepsilon < k$ and $c > 1$, the standard growth hierarchy is

$$1 \prec \log n \prec n^\varepsilon \prec n^k \prec c^n \prec n! \prec n^n.$$

The constants ε , k , and c are fixed independently of n . Allowing them to vary with the input would change the comparisons and invalidate the stated hierarchy.

2.6 Chapter exercises

The exercises use measured data only after the size parameter and analytical model have been declared.

Exercise 2.6.1 (A reproducible cost experiment). Instrument sequential search on arrays with the key at the first position, at the last position, and absent. Record comparisons for increasing n , then add an explicit probability model and compare the measured mean with the derived expected value. Keep operation counts separate from elapsed time.

Exercise 2.6.2 (Explicit asymptotic bounds). Find constants $c_1, c_2 > 0$ and a threshold n_0 proving $3n^2 + 7n + 4 = \Theta(n^2)$. Write a small program that checks the inequalities over a large finite range, and explain why the check supports debugging but does not replace the proof.

Exercise 2.6.3 (The size parameter changes the conclusion). A data-cleaning routine scans all integers from 1 through an input value m . Describe its cost as a function of m and as a function of the bit length of the binary representation of m . Implement the routine only for safe small inputs and show how the two horizontal axes produce different-looking plots.

Exercise 2.6.4 (Growth-rate audit). Create a table containing $\log n$, n , $n \log n$, $n^{3/2}$, n^2 , and 2^n over a justified range of n . Order the functions analytically, identify any finite crossovers caused by constants, and state why a log-scale plot is evidence about the model rather than a proof of asymptotic membership.

Chapter 3

Recurrences

Sources for further study

Book. Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 3rd ed.; Section 4.3 and the recurrence-analysis material in Chapter 4.

Book. Kleinberg and Tardos, *Algorithm Design*; Sections 5.1–5.2.

Book. Dasgupta, Papadimitriou, and Vazirani, *Algorithms*; Sections 2.2–2.3 and the binary-search box.

Chapter 1 introduced recursive algorithms whose calls reduce an input to smaller instances. Chapter 2 supplied the cost models and asymptotic notation needed to compare their running times. We now connect these two ideas. The recursive structure of an algorithm will be translated into a recurrence relation, and the recurrence will then be solved to obtain a nonrecursive asymptotic bound.

The chapter develops two complementary techniques. Iteration exposes the pattern of repeated calls, and the Master Theorem summarizes the most common balanced divide-and-conquer cases. Binary search provides a concrete running example and later supports the introduction of formal correctness arguments.

3.1 From recursive code to a recurrence

The running time of a recursive algorithm includes both the work performed by the current call and the work performed by every recursive call that it executes. A recurrence records this dependency explicitly.

Definition 3.1.1 (Running-time recurrence). A *running-time recurrence* expresses the cost of solving an instance of size n in terms of the costs of solving strictly smaller instances, together with the work performed outside the recursive calls and one or more base cases.

Deriving a recurrence requires three pieces of information.

1. How many recursive calls are executed?
2. What is the size of the input passed to each call?
3. How much work is performed by the current call outside recursion?

The third quantity includes tests, index calculations, input decomposition, and the work required to combine the returned results. A complete recurrence must also state the cost of the base cases.

For example, the direct Fibonacci algorithm executes two recursive calls and does a constant amount of additional work. Its running time satisfies

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

Binary search executes only one recursive call, on approximately half of the active array, and performs constant work outside that call:

$$T(n) = T(\lfloor n/2 \rfloor) + \Theta(1).$$

Merge sort instead executes two calls on inputs of half the original size and spends linear time merging their results:

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

when n is a power of two. These examples show why the recurrence must be derived from the calls that are actually executed. Similar-looking algorithms can generate very different cost functions.

3.1.1 The divide-and-conquer form

Many recursive algorithms follow the divide-and-conquer paradigm. They divide an instance into smaller subproblems, solve those subproblems recursively, and combine their solutions. If the subproblem sizes are $n_1, \dots, n_k$, the corresponding running time has the general form

$$T(n) = \sum_{i=1}^k T(n_i) + f(n),$$

where $f(n)$ contains the cost of dividing the input and combining the answers.

When the algorithm creates a subproblems of equal size n/b , this becomes

$$T(n) = aT(n/b) + f(n). \quad (3.1)$$

Here a is the number of recursive calls, b describes the reduction in problem size, and $f(n)$ is the local work of one call. This form will be used by the recursion-tree analysis and the Master Theorem. It does not cover every recurrence. For example, the two Fibonacci subproblems have different sizes, $n-1$ and $n-2$.

Floors and ceilings are often necessary when n/b is not an integer. We will first solve several recurrences under the simplifying assumption that n is an exact power of b . This makes the algebra easier to read. Floors and ceilings will then be restored when they affect the proof or the algorithm.

3.2 The iteration method

The first technique repeatedly expands the recurrence until the argument reaches a base case. This process is called *iteration* or *unrolling*. Its purpose is not merely to display the

first few expansions. The essential step is to recognize the expression produced after an arbitrary number k of expansions.

The method follows five steps.

1. Write the recurrence together with its base cases.
2. Expand the recursive term at least twice.
3. Express the result after k expansions.
4. Determine the value of k at which a base case is reached.
5. Substitute this value and simplify the resulting sums.

3.2.1 Constant work at each recursive level

Consider

$$T(n) = \begin{cases} d, & n = 1, \\ T(n/2) + c, & n > 1, \end{cases} \quad (3.2)$$

where $c, d > 0$, and initially assume that n is a power of two. The first expansions are

$$\begin{aligned} T(n) &= c + T(n/2) \\ &= c + (c + T(n/2^2)) \\ &= 2c + T(n/2^2) \\ &= 3c + T(n/2^3). \end{aligned}$$

Each expansion contributes one additional copy of c and divides the remaining argument by another factor of two. After k expansions,

$$T(n) = kc + T(n/2^k). \quad (3.3)$$

The expansion stops when the remaining argument is 1. Therefore,

$$\frac{n}{2^k} = 1 \quad \Longleftrightarrow \quad 2^k = n \quad \Longleftrightarrow \quad k = \log_2 n.$$

Substituting this value into Equation (3.3) gives

$$T(n) = c \log_2 n + T(1) = c \log_2 n + d = \Theta(\log n).$$

The logarithm appears because the input can be halved only logarithmically many times before its size becomes one. The argument does not depend on a particular logarithm base because all fixed bases differ by a constant factor.

3.2.2 When unrolling produces a sum

The local work need not be constant. Consider

$$T(n) = \begin{cases} 1, & n = 1, \\ 9T(n/3) + n, & n > 1, \end{cases} \quad (3.4)$$

and assume that n is a power of three. The first two expansions give

$$\begin{aligned} T(n) &= 9T(n/3) + n \\ &= 9(9T(n/3^2) + n/3) + n \\ &= 9^2T(n/3^2) + 3n + n, \end{aligned}$$

and one more expansion gives

$$T(n) = 9^3T(n/3^3) + 3^2n + 3n + n.$$

After k expansions, the pattern is

$$T(n) = 9^kT(n/3^k) + n \sum_{j=0}^{k-1} 3^j. \quad (3.5)$$

The stopping condition $n/3^k = 1$ gives $k = \log_3 n$. We can therefore simplify the recursive term as

$$9^k = 9^{\log_3 n} = (3^2)^{\log_3 n} = n^2,$$

and the geometric sum as

$$\sum_{j=0}^{k-1} 3^j = \frac{3^k - 1}{3 - 1} = \frac{n - 1}{2}.$$

Using $T(1) = 1$ in Equation (3.5), we obtain

$$T(n) = n^2 + n \frac{n - 1}{2} = \frac{3n^2 - n}{2} = \Theta(n^2).$$

This example also explains why stopping after two or three substitutions is not a proof. The parameter k , the stopping condition, and the resulting finite sum must all be determined. Iteration is especially useful because it reveals where the work accumulates and often suggests a bound.

3.3 Recursion trees

When a recurrence contains several recursive calls, repeated algebraic expansion can obscure how the work is distributed. A recursion tree represents each call as a node and each recursive invocation as an edge. The label of a node records the local work of that call, excluding the work performed by its descendants.

For a balanced recurrence

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

the structure at depth i is described by four quantities:

$$\begin{aligned} \text{number of nodes} &= a^i, \\ \text{subproblem size} &= n/b^i, \\ \text{local work per node} &= f(n/b^i), \\ \text{total work at the depth} &= a^i f(n/b^i). \end{aligned}$$

The final expression is the important one. Looking only at the cost of one node can give the wrong asymptotic conclusion because the number of nodes changes with depth.

Consider

$$T(n) = 3T(n/4) + n^2, \quad (3.6)$$

with $T(1) = 1$. Figure 3.1 shows only the recursive structure for the concrete input $n = 64$. The root creates three subproblems of size 16, each of those creates three subproblems of size 4, and the next expansion produces 27 base cases of size 1.

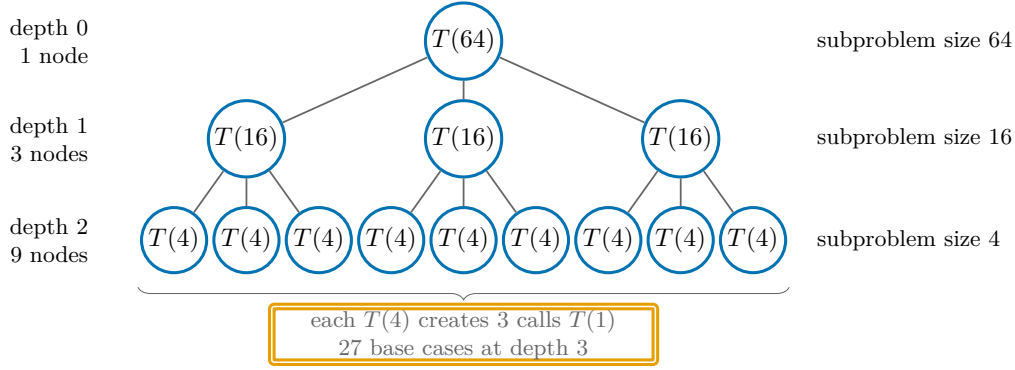


Figure 3.1: The recursion structure of $T(n) = 3T(n/4) + n^2$ for $n = 64$. The number of nodes triples while the subproblem size is divided by four.

At depth i , the tree contains 3^i subproblems of size $n/4^i$. The local work at each node is $(n/4^i)^2$, so the total work at that depth is

$$W_i = 3^i \left(\frac{n}{4^i} \right)^2 = n^2 \left(\frac{3}{16} \right)^i. \quad (3.7)$$

Successive levels therefore contribute only $3/16$ as much work as the preceding level. Figure 3.2 plots this single quantity, normalized by the root cost n^2 . Separating the plot from the tree keeps the recursive structure and the cost calculation visually distinct.

The tree has depth $k = \log_4 n$. Summing the internal levels and the leaves gives

$$T(n) = n^2 \sum_{i=0}^{k-1} \left(\frac{3}{16} \right)^i + 3^k T(1).$$

The geometric series is bounded by the infinite sum

$$\sum_{i=0}^{\infty} \left(\frac{3}{16} \right)^i = \frac{1}{1 - 3/16} = \frac{16}{13},$$

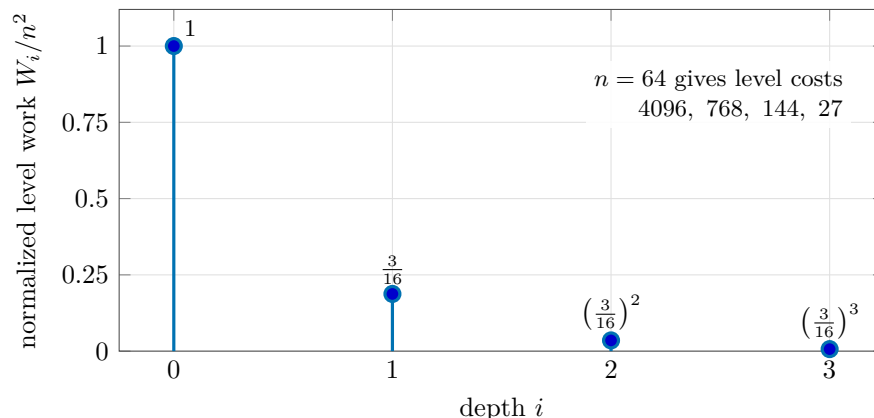


Figure 3.2: Normalized work by depth for $T(n) = 3T(n/4) + n^2$. Each level contributes $3/16$ of the work of the preceding level.

so the internal work is $O(n^2)$. The leaf contribution is

$$3^k = 3^{\log_4 n} = n^{\log_4 3} = O(n^2).$$

Since the root alone performs n^2 work, it also supplies a matching lower bound. Therefore,

$$T(n) = \Theta(n^2).$$

The same calculation explains two other recurring patterns. For $T(n) = 2T(n/2) + n$, every depth contributes exactly n , and there are $\Theta(\log n)$ depths, giving $\Theta(n \log n)$. If the total work increases by a fixed factor from one depth to the next, the leaves dominate instead. A recursion tree thus reveals whether the root, all depths equally, or the leaves provide the largest contribution.

3.4 The master theorem

The recursion-tree calculation of the previous section appears so often that it is useful to summarize it in a theorem. The result applies to balanced divide-and-conquer recurrences of the form

$$T(n) = aT(n/b) + f(n), \tag{3.8}$$

with $a \geq 1$, $b > 1$, and an eventually non-negative function f .

The notation has a direct algorithmic meaning. The value a is the number of recursive calls actually executed by one invocation. The value b is the factor by which the input size is reduced in each call. Finally, $f(n)$ is the work performed outside recursion, including the work needed to divide the instance and combine the returned solutions. For example, binary search examines two halves but recursively visits only one of them, so its recurrence has $a = 1$, not $a = 2$.

For the moment, assume that n is an exact power of b and that recursion stops at size 1. These assumptions make the tree easy to read and do not change the asymptotic conclusions in the usual applications.

3.4.1 Reading the recursion tree

At depth 0 there is one problem of size n . At depth 1 there are a subproblems of size n/b . More generally, at depth i there are a^i subproblems, each of size n/b^i . Since one node of that depth performs $f(n/b^i)$ local work, the total local work of the entire depth is

$$W_i = a^i f\left(\frac{n}{b^i}\right). \quad (3.9)$$

The leaves are reached at the depth h for which the subproblem size is 1:

$$\frac{n}{b^h} = 1 \iff h = \log_b n.$$

The number of leaves is therefore

$$a^h = a^{\log_b n} = n^{\log_b a}.$$

Figure 3.3 collects these quantities in one picture.

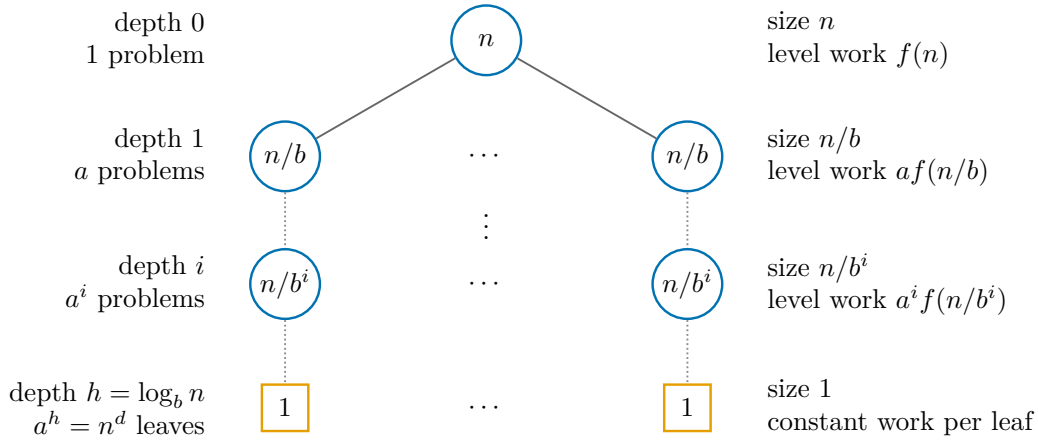


Figure 3.3: The recursion tree represented by $T(n) = aT(n/b) + f(n)$. At depth i , the number of subproblems is a^i , their size is n/b^i , and their total local work is $a^i f(n/b^i)$.

Definition 3.4.1 (Critical function). For the recurrence in Equation (3.8), define

$$d = \log_b a \quad \text{and} \quad g(n) = n^d = n^{\log_b a}.$$

The function $g(n)$ measures the number of leaves, up to the constant cost of each base case.

The Master Theorem compares $f(n)$, which is already paid at the root, with $g(n)$, which measures the work generated by recursive branching. This comparison tells us where most of the work lies. It may be concentrated near the leaves, distributed across all depths, or concentrated near the root.

The comparison to remember

Do not begin by memorizing three formulas. First compute $d = \log_b a$, write $g(n) = n^d$, and compare $f(n)$ with $g(n)$. The three cases express which of these two sources of work dominates the recursion tree.

3.4.2 The three cases

Theorem 3.4.2 (Master theorem). *Let*

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

where $a \geq 1$, $b > 1$, and f is eventually non-negative. Set $d = \log_b a$.

1. If $f(n) = O(n^{d-\varepsilon})$ for some $\varepsilon > 0$, then

$$T(n) = \Theta(n^d).$$

2. If $f(n) = \Theta(n^d)$, then

$$T(n) = \Theta(n^d \log n).$$

3. If $f(n) = \Omega(n^{d+\varepsilon})$ for some $\varepsilon > 0$ and

$$af(n/b) \leq cf(n)$$

for some constant $c < 1$ and all sufficiently large n , then

$$T(n) = \Theta(f(n)).$$

The conditions involving ε mean that the difference must be polynomial. In the first case, $f(n)$ is smaller than $g(n)$ by at least a factor n^ε . In the third case, it is larger by at least that factor. A mere logarithmic difference does not satisfy either condition.

The regularity condition in the third case also has a concrete interpretation. The quantity $af(n/b)$ is the total local work one depth below the root. The inequality

$$af(n/b) \leq cf(n), \quad c < 1,$$

says that this work is at most a fixed fraction of the root work. Repeating the argument makes the work decrease geometrically as the tree descends.

Case	Comparison	Where the work lies	Result
1	$f(n)$ is polynomially smaller than $g(n)$	near the leaves	$\Theta(g(n))$
2	$f(n)$ and $g(n)$ have the same order	every depth contributes comparably	$\Theta(g(n) \log n)$
3	$f(n)$ is polynomially larger than $g(n)$ and is regular	near the root	$\Theta(f(n))$

3.4.3 Why the three cases follow from the tree

Expanding the recurrence to the base cases gives

$$T(n) = a^h T(1) + \sum_{i=0}^{h-1} a^i f\left(\frac{n}{b^i}\right), \quad h = \log_b n. \quad (3.10)$$

The first term is the cost of all leaves. Since $a^h = n^d$, that term is $\Theta(n^d) = \Theta(g(n))$. The summation adds the local work of all preceding depths.

In Case 1, the hypothesis bounds the level costs by an increasing geometric sequence whose total is $O(n^d)$; the actual costs need not increase monotonically. The leaves supply the matching lower bound. In Case 2, each depth contributes $\Theta(n^d)$ and there are $\Theta(\log n)$ depths. In Case 3, regularity bounds successive depth costs by a decreasing geometric sequence, while the root supplies the matching lower bound.

3.4.4 A procedure for applying the theorem

The following order of operations makes the calculation both shorter and safer.

1. Rewrite the recurrence as $T(n) = aT(n/b) + f(n)$.
2. Verify that $a \geq 1$, $b > 1$, and $f(n)$ are eventually non-negative.
3. Compute $d = \log_b a$ and write $g(n) = n^d$ explicitly.
4. Compare the local work $f(n)$ with $g(n)$. Check the polynomial separation required by Case 1 or Case 3.
5. For Case 3, also verify $af(n/b) \leq cf(n)$ for some $c < 1$.
6. State the applicable case and then write the resulting bound.

Common sources of error

- Compare $f(n)$ with $n^{\log_b a}$, not with n^a or n^b .
- Count recursive calls, not the number of regions into which the data is conceptually divided.
- Do not select Case 1 or Case 3 from a strict inequality alone. The classical theorem requires a polynomial gap.
- Do not omit the regularity check in Case 3.

3.4.5 Worked examples

Example 3.4.3. Consider

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

Here $a = 1$, $b = 2$, $f(n) = \Theta(1)$, and

$$d = \log_2 1 = 0, \quad g(n) = n^0 = 1.$$

Thus $f(n) = \Theta(g(n))$. Case 2 gives

$$T(n) = \Theta(\log n).$$

The additional logarithm counts the number of times the active interval can be halved before its size becomes 1.

Example 3.4.4. For

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

we have $a = 2$, $b = 2$, $d = 1$, and $g(n) = n$. The merge work has the same order as $g(n)$, so this is again Case 2:

$$T(n) = \Theta(n \log n).$$

In the recursion tree, every depth contains a total of $\Theta(n)$ merge work.

Example 3.4.5 (The leaves dominate). Consider

$$T(n) = 9T(n/3) + n.$$

Here $a = 9$, $b = 3$, $d = \log_3 9 = 2$, and $g(n) = n^2$. Since

$$f(n) = n = O(n^{2-1}),$$

Case 1 applies with $\varepsilon = 1$, giving

$$T(n) = \Theta(n^2).$$

Although one call performs only linear local work, recursive branching creates n^2 leaves.

Example 3.4.6 (The root dominates). Consider

$$T(n) = 3T(n/2) + n^2.$$

Here $d = \log_2 3 < 2$, so $f(n) = n^2$ is polynomially larger than $g(n)$. The regularity condition is verified by

$$3f(n/2) = 3 \left(\frac{n}{2} \right)^2 = \frac{3}{4} f(n).$$

We may choose $c = 3/4 < 1$, and Case 3 gives

$$T(n) = \Theta(n^2).$$

The local work decreases by a factor $3/4$ from one depth to the next, so the root sets the asymptotic scale.

3.4.6 The boundary of the classical theorem

The recurrence

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

has the required divide-and-conquer shape, but it does not fit any case of the classical theorem stated above. Here $d = 1$ and $g(n) = n$. The function $n \log n$ is larger than n , but it is not polynomially larger, because $\log n = o(n^\varepsilon)$ for every fixed $\varepsilon > 0$.

Some references state an extended second case. If

$$f(n) = \Theta\left(n^d \log^k n\right)$$

for a fixed $k \geq 0$, then

$$T(n) = \Theta\left(n^d \log^{k+1} n\right).$$

The classical statement corresponds to $k = 0$. The recurrence above has $k = 1$, so the extended form gives $T(n) = \Theta(n \log^2 n)$. Alternatively, the same result follows directly by summing the costs of the recursion-tree depths.

The theorem also does not apply directly to recurrences with unequal subproblem sizes, such as $T(n-1) + T(n-2)$, or to recurrences in which a or b depends on n . In those situations, iteration or a direct recursion-tree analysis remains necessary.

3.5 Binary search

Binary search turns sorted order into a reduction in problem size. A comparison with the middle element determines which half of the current interval can still contain the requested value. This makes the algorithm a concrete example of the recurrence $T(n) = T(n/2) + \Theta(1)$. The following version searches the half-open interval $[\text{low}, \text{high})$ and returns the first occurrence of the key.

Binary search for the first occurrence

```
Binary-search(A, key)
  low <- 0
  high <- length(A)
  while low < high
    middle <- low + floor((high-low)/2)
    if A[middle] < key
      low <- middle + 1
    else
      high <- middle
  if low < length(A) and A[low] = key
    return low
  return Not-found
```

The loop maintains the half-open candidate interval $[\text{low}, \text{high})$ with two boundary facts: every position before **low** contains a value smaller than **key**, and every position at or after **high** contains a value at least **key**. The update preserves these facts because the array is sorted. When the interval becomes empty, **low** is the first position whose value can equal the key. The final test distinguishes an occurrence from insertion at the end or between two different values.

If the current interval has length n , one comparison and a constant number of assignments reduce it to an interval of length at most $\lceil n/2 \rceil$. Thus,

$$T(n) = T(\lceil n/2 \rceil) + \Theta(1) = \Theta(\log n).$$

The iterative implementation uses constant auxiliary space. A recursive version has the same asymptotic running time but uses $\Theta(\log n)$ stack frames.

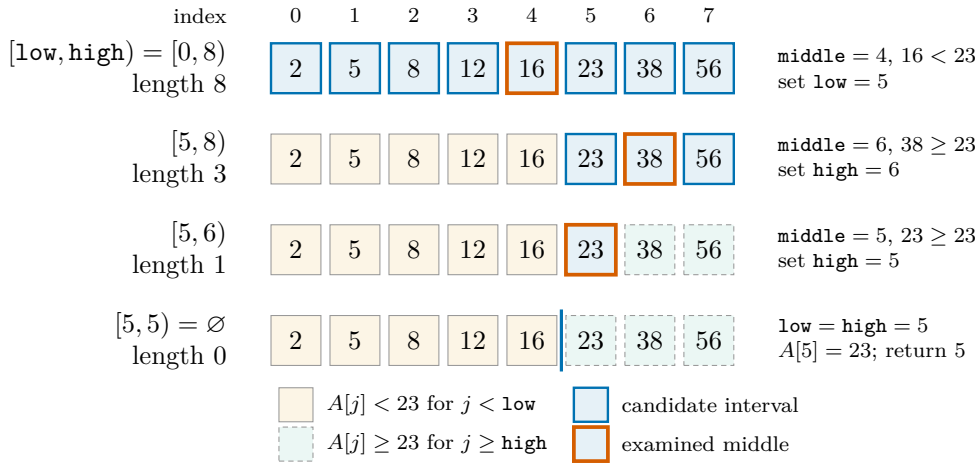


Figure 3.4: Binary search repeatedly reduces the half-open candidate interval while preserving the boundary between values smaller than the key and values not smaller than the key.

The precondition that the array is sorted is essential. Without sortedness, a comparison at the midpoint provides no information about the discarded half. Efficient random access is also required. Locating the middle of a linked list can itself require linear time, removing the advantage of binary search.

3.6 Chapter exercises

Exercise 3.6.1 (Instrument a divide-and-conquer routine). Implement a recursive routine with recurrence $T(n) = 4T(n/2) + n$ for power-of-two inputs. Count calls and local-work units by depth, compare the measurements with a recursion-tree table, and verify the Master-Theorem prediction without using elapsed time as the only measurement.

Exercise 3.6.2 (Choose the correct analysis tool). For $T_1(n) = 16T_1(n/4) + n$, $T_2(n) = 4T_2(n/2) + n^2$, and $T_3(n) = 2T_3(n/2) + n \log n$, identify the applicable theorem or direct summation. Implement a recurrence evaluator for small powers of two and compare normalized values with the predicted growth. Explain why finite measurements do not establish the proof.

Exercise 3.6.3 (Binary search as a data service). Implement the first-occurrence binary search and validate it against a linear scan on random sorted arrays containing duplicates. Tests must include an empty array, an absent key, and keys at both boundaries. Record comparisons and verify the two boundary facts after every update in a debugging version.

Exercise 3.6.4 (Break-even analysis). A static collection receives q membership queries. Compare a linear scan of the unsorted data with sorting once and then using binary search. Include sorting, queries, and the requested output in the cost model, derive a symbolic break-even condition for q , and test it experimentally on one controlled dataset.

Chapter 4

Sorting

Sources for further study

Book. Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 3rd ed.; Chapters 2 and 7 for insertion sort, merge sort, and Quicksort.

Book. Kleinberg and Tardos, *Algorithm Design*; Section 5.1 for divide-and-conquer sorting.

Book. Dasgupta, Papadimitriou, and Vazirani, *Algorithms*; Sections 2.2–2.3 for recurrence-guided sorting algorithms.

Sorting is a useful meeting point for the ideas developed so far. It has a precise input–output specification, admits algorithms with very different running times, and makes the influence of the underlying data representation easy to observe. This chapter begins with two incremental methods and then returns to the divide-and-conquer techniques of Chapter 3 through Merge sort and Quicksort.

Sorting is also a frequent preprocessing step in data analysis. Ordered records make duplicate detection, grouped aggregation, ranking, interval queries, and merge-based joins easier. The intended downstream operation therefore matters alongside the running time of the sort itself.

The purpose is not to collect four recipes. For each method we identify the part of the array that already has a useful structure, the operations used to extend that structure, the resulting cost, and the input patterns on which the analysis changes.

The four algorithms can first be distinguished by the question that guides their next step.

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

This overview should be kept in mind when reading the pseudocode. The loops and

recursive calls implement these ideas; they are not the ideas themselves.

4.1 The sorting problem

Before studying an algorithm, we must state what is being sorted. Real applications usually sort records by a key rather than moving bare numbers.

Definition 4.1.1 (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

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

The word *permutation* is essential. An increasing sequence is not a valid answer if it loses a record, duplicates a record, or introduces a new one.

Definition 4.1.2 (Stable, in-place, and adaptive sorting). A concrete sorting implementation may have the following properties.

- It is *stable* if records with equal keys retain their relative input order.
- It is *in place* if it uses only a constant amount of auxiliary storage, apart from the convention adopted for a recursion stack.
- It is *adaptive* if its running time improves when the input is already partly ordered.

These are independent properties. Stability matters, for example, when a table that is already ordered by surname is subsequently sorted by city. An in-place algorithm may be desirable when memory is restricted, while an adaptive algorithm can exploit nearly sorted data.

The three terms describe an implementation rather than the sorting problem itself. The same correctly sorted sequence may be produced by implementations with different stability, memory use, or sensitivity to existing order.

4.1.1 Stability preserves information hidden by the keys

To observe stability, equal keys must be attached to distinguishable records. Suppose the subscript identifies a record and the number is its sorting key:

$$[4_A, 2_B, 4_C, 1_D].$$

A stable result is

$$[1_D, 2_B, 4_A, 4_C],$$

because 4_A appeared before 4_C in the input. The sequence

$$[1_D, 2_B, 4_C, 4_A]$$

is sorted by key, but it is not a stable result for this input. This example shows why sortedness alone cannot be used to test stability.

Stability is particularly useful for successive sorts. If records are first sorted by surname and then stably sorted by city, records belonging to the same city remain in surname order. An unstable second sort may silently destroy that earlier ordering.

4.1.2 In-place sorting limits additional memory

The input array already occupies n record positions. Auxiliary storage is the additional memory used while transforming that array. The Bubble sort implementations and Insertion sort need only a few variables. The early-exit Bubble variant adds one Boolean flag, while Insertion sort adds one saved record. Their auxiliary storage is therefore $\Theta(1)$, independently of n .

The standard array implementation of Merge sort allocates an output buffer containing n records, so it uses $\Theta(n)$ auxiliary storage and is not in place. Lomuto partition uses only a pivot value and a few indices, but recursive Quicksort also uses a call stack. For this reason, a statement that Quicksort is in place should say whether the recursion stack is being counted.

In-place does not mean that no records move, and it does not mean that the algorithm uses no memory. It means that the amount of auxiliary storage remains bounded by a constant under the stated stack convention.

4.1.3 Adaptiveness turns existing order into less work

To make partial order measurable, we use inversions.

Definition 4.1.3 (Inversion). An inversion of A is a pair of indices (i, j) such that $i < j$ but $\text{key}(A[i]) > \text{key}(A[j])$.

For six distinct keys, compare

Input	Number of inversions
$[1, 2, 3, 4, 5, 6]$	0
$[1, 2, 3, 5, 4, 6]$	1
$[6, 5, 4, 3, 2, 1]$	15

Insertion sort performs no shifts on the first input, one shift on the second, and fifteen shifts on the third. Its running time is $\Theta(n + I(A))$, where $I(A)$ is the inversion count. This is concrete evidence of adaptiveness because decreasing disorder decreases the amount of work.

By contrast, the Merge sort implementation in this chapter still divides and merges an already sorted array, so its running time remains $\Theta(n \log n)$. The basic Bubble sort algorithm also performs all of its planned passes, even when the input is already sorted.

Adding an early-exit test produces a limited adaptive variant: it can detect a sorted input after one pass, but its comparison count does not generally satisfy the same precise $\Theta(n + I(A))$ bound as Insertion sort.

Three different questions

Stability asks what happens to equal-key records. In-place sorting asks how much additional memory grows with n . Adaptiveness asks whether a measurable form of existing order reduces the performed work. None of these properties follows merely from the output being correctly sorted.

4.1.4 The representation matters

The algorithms in this chapter are presented on arrays. Arrays provide constant-time indexed access and allow two positions to be exchanged directly. The same abstract sorting method can behave differently on another representation.

Method	Main representation	Structural consequence
Bubble sort	array or sequential container	adjacent records are compared and exchanged
Insertion sort	array or linked sequence	arrays shift a block; links avoid shifts only after the insertion position is found
Merge sort	arrays plus a buffer, or linked sequences	merging needs sequential access; arrays normally require auxiliary storage
Quicksort	random-access array plus a recursion stack	in-place partitioning depends on exchanging distant positions

We count key comparisons and record movements when this distinction is informative. The two counts need not have the same order of growth on every input.

Cost convention for this chapter

The input size n is the number of records. Unless another case is stated, a running-time bound refers to the worst case in the uniform-cost model of Chapter 2. A comparison of two machine-sized keys, an assignment, and an array access each have constant cost. When comparisons and movements tell different stories, both are reported explicitly.

4.2 Bubble sort

Bubble sort is a useful first example because a global objective is reached through simple local repairs. During a left-to-right pass, each record is compared with its right neighbour. An inverted pair is exchanged immediately. The larger of the two records then continues to the next comparison, so the largest record encountered travels to the right end of the active region.

High-level idea

Treat one pass as a sequence of local competitions. The larger key wins each competition and moves one position to the right. By the end of the pass, the maximum active key has won every necessary comparison and has reached its final position. The algorithm repeats the idea on the shorter prefix that remains.

For example, one pass over $[5, 1, 4, 2]$ produces

$$[5, 1, 4, 2] \rightarrow [1, 5, 4, 2] \rightarrow [1, 4, 5, 2] \rightarrow [1, 4, 2, 5].$$

Only the final 5 is guaranteed to be in its final position after this pass. The prefix $[1, 4, 2]$ is still a sorting problem.

4.2.1 The basic algorithm

The initial design follows the pass structure directly. After every pass, the active region becomes one position shorter because its maximum has reached the right boundary. No test is made to determine whether the remaining prefix is already sorted.

Basic Bubble sort

```
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]
```

After the first complete pass, the maximum input record is in position $n - 1$. After the second pass, the two largest records occupy the last two positions in sorted order. More generally, after p passes the suffix of length p is final.

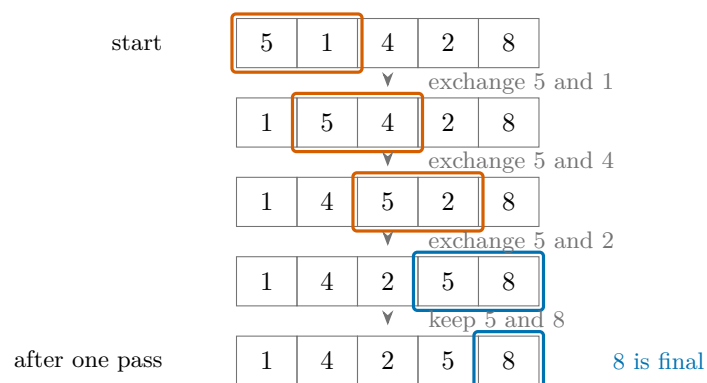


Figure 4.1: One Bubble sort pass moves the largest active key to the right boundary by successive adjacent exchanges.

The loop bounds do not depend on the input values. The first pass performs $n - 1$ comparisons, the second performs $n - 2$, and the final pass performs one. Therefore the

basic algorithm always performs

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

adjacent comparisons. Its best-case, worst-case, and input-independent comparison cost are all $\Theta(n^2)$. A sorted input causes no exchanges, but the algorithm does not inspect that fact at the level of a complete pass and therefore continues.

Bubble sort is in place. With the strict test $A[i] > A[i+1]$, equal-key records are never exchanged and the algorithm is stable. Each exchange removes exactly one adjacent inversion. Consequently, the number of exchanges equals $I(A)$ even though the basic algorithm always performs a quadratic number of comparisons. This is an important distinction between data movements and comparison cost.

Proposition 4.2.1 (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.*

The proposition is the structural reason the active boundary can move left after every pass. Students should be able to verify it directly on a trace before using it as a formal invariant.

4.2.2 Early exit and adaptivity

The basic algorithm may complete a pass without exchanging any pair. In that case, every adjacent pair in the active prefix is already in nondecreasing order. The prefix is sorted, while the suffix produced by earlier passes is already final, so the complete array is sorted. Further passes cannot change the output.

This observation suggests recording whether the current pass made at least one exchange.

Bubble sort with early exit

```

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

```

The flag does not change the exchanges performed during a pass, the final-suffix invariant, stability, or the $\Theta(1)$ auxiliary-space bound. It changes only the decision to begin another pass.

On an already sorted array, the first pass performs $n-1$ comparisons, makes no exchange, and terminates. The best-case running time is therefore $\Theta(n)$. In a decreasing array, every active pass still performs exchanges, so all passes are required and the running time remains $\Theta(n^2)$.

This is a simple form of adaptivity because existing order can reduce the performed work. It is weaker than the inversion-sensitive behaviour of Insertion sort. For example, the array

$$[2, 3, \dots, n, 1]$$

has only $n - 1$ inversions, but the final record can move left by only one position per Bubble sort pass. The early-exit version therefore still requires $\Theta(n)$ passes and $\Theta(n^2)$ comparisons, whereas Insertion sort runs in $\Theta(n + I(A)) = \Theta(n)$ time on the same input.

What the optimization teaches

The basic and early-exit versions solve the same sorting problem and use the same local repair. The additional Boolean flag gives the algorithm information about the observed execution. That information leaves the worst case unchanged but improves the best case from $\Theta(n^2)$ to $\Theta(n)$.

4.3 Insertion sort

Bubble sort develops a sorted suffix by repeated local exchanges. Insertion sort uses a different incremental structure. It maintains a sorted prefix and inserts one new record into that prefix at every iteration. The familiar analogy is arranging playing cards in one hand.

High-level idea

Assume the records already held in the left prefix are sorted. Remove the next record from the unsorted suffix, open a gap by shifting larger prefix records to the right, and place the saved record in that gap. One iteration increases the sorted prefix by exactly one record.

For example, if the current prefix is $[2, 4, 5, 6]$ and the saved key is 1, all four prefix records shift one position right before 1 is placed at the beginning. If the saved key is 7, no shift is needed. This contrast anticipates the input-sensitive cost analysis.

Insertion sort

```
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
```

At the beginning of iteration j , the prefix $A[0 \dots j - 1]$ is already sorted and contains exactly the records originally stored there. The variable **saved** protects the next record while larger prefix records are shifted one position to the right. When the loop stops, position $i + 1$ is the only place where the saved record can be inserted without breaking the order.

For an already sorted array, every iteration performs one order check and no shift, giving $\Theta(n)$ time. For a decreasing array, the prefix has length j and every one of its records must move. The total work is

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

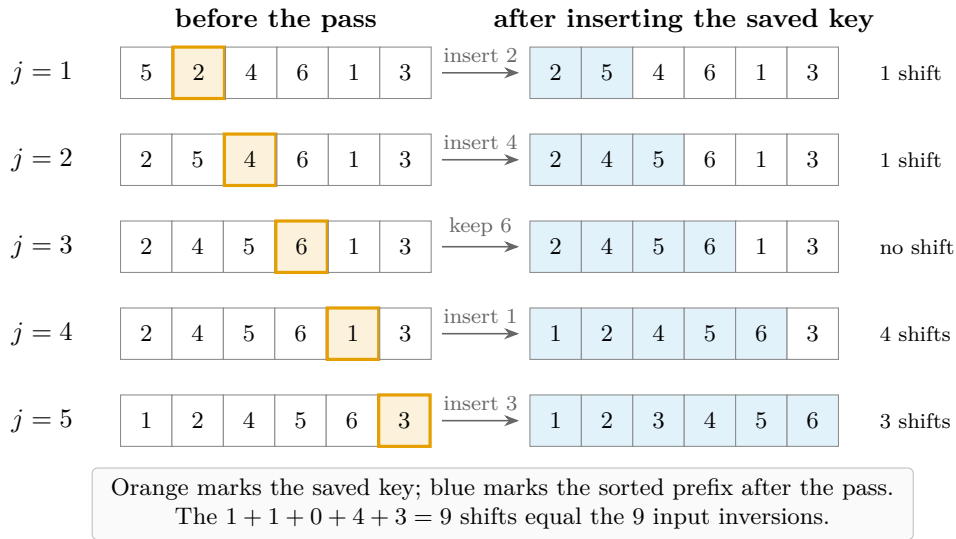


Figure 4.2: Each Insertion sort pass saves one key and inserts it into the sorted prefix. The number of displaced records is the number of shifts performed by that pass.

Thus the best case is $\Theta(n)$ and the worst case is $\Theta(n^2)$.

Each shift performed by Insertion sort removes one inversion involving the saved record. If $I(A)$ is the number of input inversions, the running time can be written more informatively as

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

This makes the meaning of adaptiveness precise. The algorithm is fast not because an array merely looks almost sorted, but because it contains few inversions.

Insertion sort is in place and stable when the while condition uses a strict comparison. Changing $>$ to $\geq$ permits an equal record to move in front of an earlier one and destroys stability.

4.4 Merge sort

The quadratic methods move records through the array one position or one exchange at a time. Merge sort changes the organization of the computation. It divides the input into two halves, sorts them recursively, and combines two sorted sequences in linear time.

High-level idea

Divide by position until every subarray is trivially sorted. Then reverse direction and repeatedly merge neighbouring sorted runs. The difficult step is not splitting the array; it is exploiting the order inside two runs so that they can be combined in one forward scan.

For example, once the two halves have become $[2, 5]$ and $[1, 4]$, the algorithm does not reconsider every pair of records. It compares the two front records, outputs 1, then

compares 2 with 4, and continues until both runs are exhausted. Each front comparison permanently determines the next output position.

4.4.1 Merging two sorted sequences

If L and R are sorted, their smallest remaining records are at their current fronts. Comparing only those two records is enough to decide the next output position.

Stable merge of two sorted arrays

```
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
```

The two indices move only forward. Every record is copied once, so merging sequences of lengths n_1 and n_2 takes $\Theta(n_1 + n_2)$ time. Selecting the left record when the keys are equal makes the operation stable.

For arrays, the simple implementation needs an auxiliary output buffer. Linked sequences can be merged by reconnecting links, but splitting them and obtaining random positions has a different cost. The representation therefore changes the implementation even though the merge principle remains the same.

4.4.2 The recursive algorithm and its cost

The merge operation supplies the linear-time combination step required by a divide-and-conquer algorithm. The recursion stops when a subarray has at most one record, since such an array is already sorted.

Merge sort

```
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)
```

Figure 4.3 shows the complete organization of the computation. The upper half repeatedly divides the input until every subarray contains one record. The lower half merges neighbouring sorted runs until the complete sorted array is reconstructed.

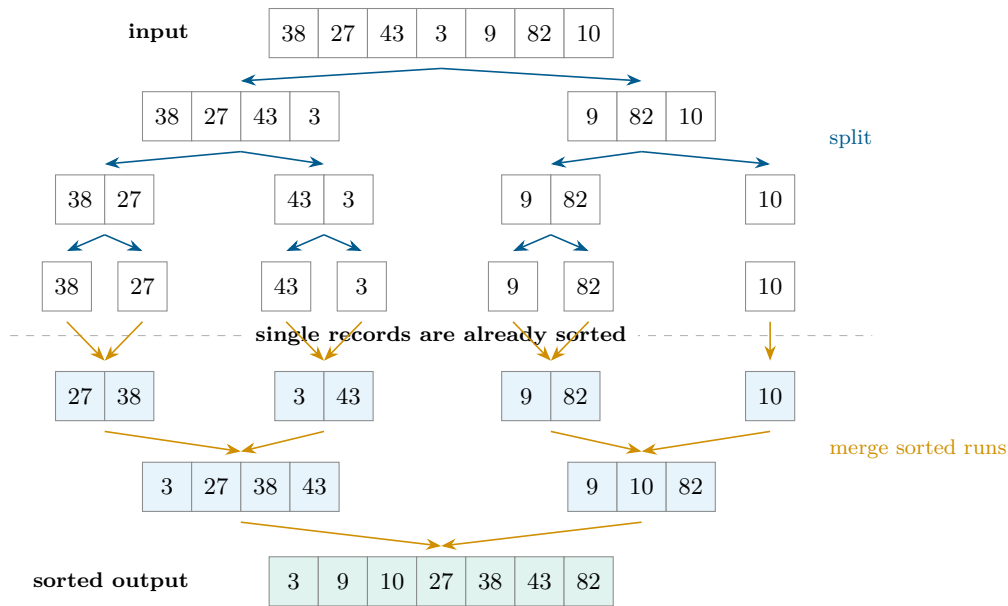


Figure 4.3: Merge sort first divides the input into one-record subarrays and then merges sorted runs in the reverse direction.

For a power-of-two input size, the running time satisfies

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

At each recursion depth the subarrays are disjoint and their lengths sum to n , so the total merge work at that depth is $\Theta(n)$. The tree has $\Theta(\log n)$ depths. Consequently,

$$T(n) = \Theta(n \log n)$$

in the best, average, and worst cases.

The standard array implementation is stable and uses $\Theta(n)$ auxiliary array space plus a recursion stack of depth $\Theta(\log n)$. Its guarantee is stronger than that of Insertion sort, but it does not exploit an already sorted array in the version shown here.

4.5 Quicksort

Merge sort first solves two smaller sorting problems and then combines their answers. Quicksort reverses this order. It first rearranges the input around one selected record, called the *pivot*, and only then sorts the two resulting parts. The rearrangement is performed directly inside the array, so no merge buffer is needed.

This change makes one operation central. The *partition* procedure must place the pivot in a position that separates keys no larger than the pivot from keys larger than it. Partitioning does not sort either side. It creates two smaller sorting problems whose answers will already occupy the correct regions of the array.

High-level idea

Choose one pivot record and use it to classify the remaining records into two groups. Place keys no larger than the pivot on its left and larger keys on its right. Once the pivot is placed between these groups, its position is final. Recursively apply the same idea to the two groups; no later merge is necessary because they already occupy the correct sides of the pivot.

The partition step produces separation rather than complete order. For example, partitioning $[7, 2, 9, 4, 6]$ around pivot 6 may produce

$$[2, 4] ; 6 ; [9, 7].$$

The pivot is final, but the right group still needs sorting. The sizes of these two groups, rather than partitioning alone, determine the total running time.

4.5.1 Partitioning around a pivot

Several partition procedures can implement the same idea. We begin with the Lomuto scheme because it exposes the classification process through a single left-to-right scan. For a subarray $A[\text{low} \dots \text{high}]$, the last record is chosen as the pivot and its key is saved in `pivot_key`. The scan stops at $\text{high} - 1$, so the pivot remains untouched until every other record has been classified.

Two indices describe the progress of the scan. Index j identifies the next unclassified record, while i identifies the final position of the region containing keys no larger than the pivot. At the beginning of an iteration, the subarray has the following four regions:

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

An interval may be empty. Initially $i = \text{low} - 1$ and $j = \text{low}$, so both classified regions are empty and all non-pivot records are unclassified.

Each iteration examines $A[j]$ and has one of two effects.

- If $\text{key}(A[j]) > \text{pivot_key}$, no exchange is needed. Advancing j simply extends the greater-than region by one record.
- If $\text{key}(A[j]) \leq \text{pivot_key}$, the no-larger region must grow. The algorithm first increments i and then exchanges $A[i]$ with $A[j]$. The accepted record moves to the end of the no-larger region. Any record displaced from $A[i]$ belongs to the greater-than region, so it can safely occupy the position that the scan is leaving behind.

If $i = j$ after the increment, the exchange is a harmless self-exchange. This occurs when the current record already immediately follows the no-larger region.

Lomuto partition

```

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

```

When the scan ends, $j = \text{high}$ and every non-pivot record has been classified. Position $i + 1$ is the first position after the no-larger region. Exchanging $A[i + 1]$ with $A[\text{high}]$ places the pivot between the two regions. The procedure returns this position as $q = i + 1$.

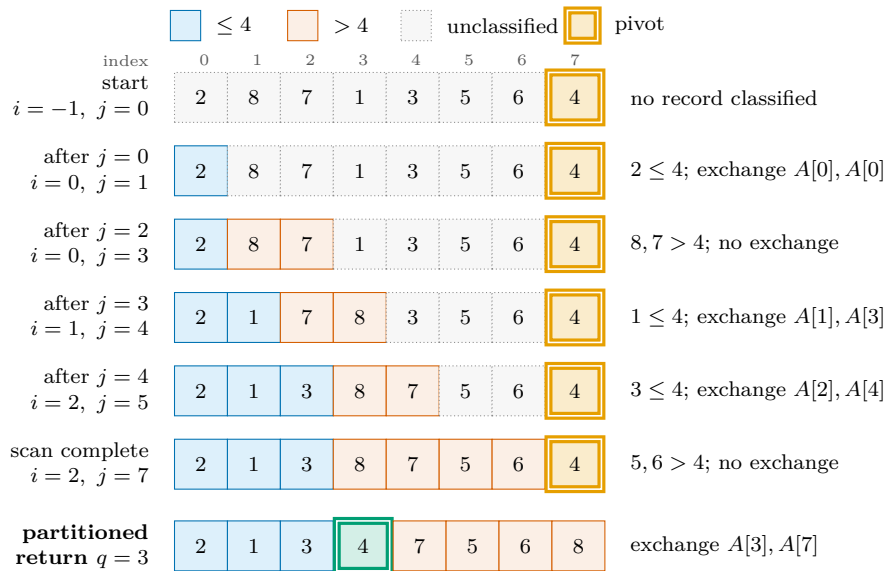


Figure 4.4: Lomuto partition on pivot 4. The scan moves the boundary indices from left to right while preserving the no-larger, greater-than, unclassified, and pivot regions.

Figure 4.4 traces the array $[2, 8, 7, 1, 3, 5, 6, 4]$. The first accepted record, 2, is exchanged with itself. The records 8 and 7 then extend the greater-than region without any exchange. When 1 is encountered, it must cross that region and is exchanged with 8. The same operation moves 3 across 7. After 5 and 6 are rejected, the final exchange places the pivot in position 3 and moves 8 to the right region.

What partition guarantees

If Lomuto partition returns q , then 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 $\text{key}(A[q])$, and the subarray contains exactly the same records as before. The pivot is therefore in a valid final position, although neither surrounding region is necessarily sorted.

For a subarray of length $m = \text{high} - \text{low} + 1$, the loop performs exactly $m - 1$ pivot comparisons. It makes at most one exchange per scanned record and one final pivot exchange, so the running time is $\Theta(m)$. The procedure uses $\Theta(1)$ auxiliary storage. Its distant exchanges do not generally preserve the relative order of equal-key records, so this implementation is not stable.

4.5.2 From one partition to Quicksort

Partition has solved only the separation problem. Quicksort completes the work by recursively sorting the records on the two sides of the returned pivot position. The pivot itself is excluded from both recursive calls because its position is already final.

Quicksort

```

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

```

Figure 4.5 follows this recursive structure. Each pivot is placed in its final position, while the two surrounding groups remain independent sorting problems. Once both recursive calls return, their sorted results already lie on the correct sides of the pivot and no merge step is needed.

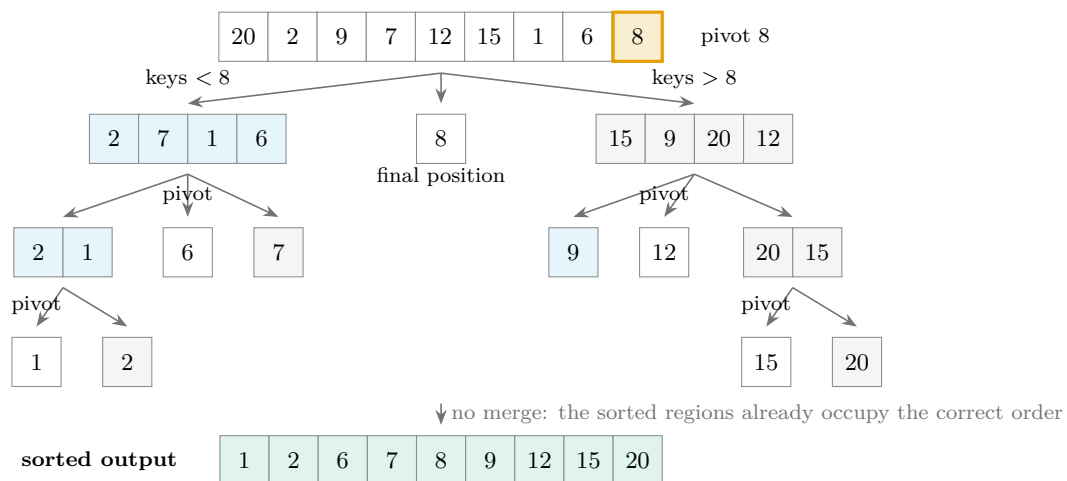


Figure 4.5: Quicksort recursively partitions the array around successive pivots. Each pivot reaches its final position before the two sides are sorted independently.

4.5.3 Why pivot balance controls the cost

Partitioning n records costs $\Theta(n)$. If the pivot leaves k records on the left, the recurrence is

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

The value of k is therefore the central performance question.

If every pivot produces two parts of comparable size, each recursion depth costs $\Theta(n)$ and there are $\Theta(\log n)$ depths. Hence

$$T(n) = \Theta(n \log n).$$

If every pivot is the minimum or maximum record, the recurrence becomes

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

Choosing the last position as pivot therefore gives the worst case on an already sorted array of distinct keys.

A randomized implementation first exchanges a uniformly chosen record with the last position. For every fixed input order, the expected running time over the random pivot choices is $\Theta(n \log n)$. Randomization does not remove the quadratic worst case, but it prevents a fixed input order from determining it.

Many equal keys are another important boundary case. Ordinary two-way partitioning can repeatedly place only one pivot even when every key is equal. A three-way partition groups keys smaller than, equal to, and greater than the pivot, and recurses only on the first and third regions.

Quicksort is in place with respect to array storage, although recursion consumes stack space. Balanced calls use $\Theta(\log n)$ stack depth; consistently poor partitions can use $\Theta(n)$. The ordinary exchange-based algorithm is not stable.

4.6 Comparing the four methods

The asymptotic running time is important, but it is not the only selection criterion. The following table summarizes the implementations developed in this chapter.

Algorithm	Best	Worst	Stable	Extra storage
Basic Bubble sort	$\Theta(n^2)$	$\Theta(n^2)$	yes	$\Theta(1)$
Bubble sort with early exit	$\Theta(n)$	$\Theta(n^2)$	yes	$\Theta(1)$
Insertion sort	$\Theta(n)$	$\Theta(n^2)$	yes	$\Theta(1)$
Merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$	yes	$\Theta(n)$
Quicksort	$\Theta(n \log n)$	$\Theta(n^2)$	no	recursion stack

The $\Theta(n \log n)$ expected running time of randomized Quicksort is measured over its random pivot choices and is not an unqualified average-case claim. The two Bubble sort rows differ only because the second version observes whether a pass made an exchange. These qualifications are part of the claims, not footnotes to be ignored.

For small or nearly sorted arrays, Insertion sort is often the simplest useful choice. Merge sort is appropriate when stability and a worst-case $n \log n$ guarantee justify the buffer. Quicksort is attractive for arrays when locality and in-place partitioning matter, provided pivot selection and recursion depth are handled carefully. Bubble sort remains primarily a transparent teaching example.

Questions to ask before choosing a sorting algorithm

Is the input an array or a linked structure? Must equal-key records preserve their relative order? Is auxiliary memory available? Is the required guarantee worst-case, expected, or input-sensitive? Answering these questions is part of the algorithmic problem; an asymptotic running time alone does not determine the best method.

4.7 Chapter exercises

Every implementation should be checked for sortedness, preservation of all input records, and any claimed stability property.

Exercise 4.7.1 (A sorting experiment on records). Create records (*group*, *timestamp*, *id*) with repeated group keys. Implement Insertion sort, Merge sort, and randomized Quicksort over the group field. Verify sortedness and record preservation, test stability using the timestamp order, and count comparisons and movements separately.

Exercise 4.7.2 (Adaptivity from measured disorder). Generate arrays with controlled inversion counts, including sorted, reversed, and $[2, 3, \dots, n, 1]$ inputs. Compare basic Bubble sort, early-exit Bubble sort, and Insertion sort. Plot comparisons and movements against both n and the inversion count, then explain which algorithm is genuinely inversion-sensitive.

Exercise 4.7.3 (Partition as a testable component). Implement Lomuto partition and use property-based tests to verify that the returned pivot position separates the two regions and preserves the input multiset. Compare last-element and random pivots on sorted, reversed, equal-key, and random datasets, recording recursion depth as well as comparisons.

Exercise 4.7.4 (Select a preprocessing sort). A data pipeline must sort ten million event records before grouped aggregation. Evaluate Insertion sort, Merge sort, and randomized Quicksort under three contracts: stable output, strict auxiliary-memory limits, and nearly sorted daily updates. State the chosen representation, guarantee, measurements, and one reason for rejecting each alternative.

Chapter 5

Heaps, priority queues, and Heapsort

Sources for further study

Book. Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 3rd ed.; Chapter 6.

Book. Kleinberg and Tardos, *Algorithm Design*; Section 2.5 for priority queues.

Book. Horowitz, Sahni, and Rajasekaran, *Computer Algorithms*; Heap and heapsort sections in Chapter 3.

The sorting algorithms in Chapter 4 worked directly with sequences. Heapsort follows a different design path. It first organizes the array as a data structure and then uses that structure to place one element at a time in its final position. The data structure is a *binary heap*.

To see why it is useful, suppose that we must repeatedly remove the largest record from an unsorted array. Finding the maximum once may require inspecting all n positions. Repeating the same scan after every removal leads to quadratic work. A max-heap instead keeps a maximum at a known position and can restore its organization after a removal by following a short path. A min-heap provides the symmetric service for a minimum.

At a high level, a binary heap combines three views of the same data. Its *shape* is a nearly complete binary tree, its *order* is a local comparison between every parent and its children, and its physical representation is a compact array. The tree view explains the relationships between records. The array view is the actual implementation and requires no explicit child pointers.

Heap at a glance

A binary heap is a nearly complete binary tree with a local parent–child ordering rule. In a max-heap, every parent is at least as large as its children, so a maximum is stored at the root. In a min-heap, every parent is at most as large as its children, so a minimum is stored at the root. The definitions below make the terms in this description precise.

In this chapter, the word *heap* always refers to this data structure. It is not the region of memory also called a heap in discussions of dynamic allocation or garbage collection.

We develop the structure in stages. We first introduce the required tree vocabulary and show why the tree can be stored without gaps in an array. We then add heap order, study how to repair and construct a heap, and finally apply it to priority queues and Heapsort.

5.1 The shape of a binary heap

A binary heap is defined by both its shape and its ordering rule. This section focuses on the shape and its array representation. Section 5.2 will add the ordering rule. Keeping the two requirements separate is important because a nearly complete tree is not necessarily a heap.

5.1.1 Binary-tree vocabulary

Definition 5.1.1 (Binary tree). A *binary tree* is a rooted tree in which every node has at most two children, distinguished as its *left child* and *right child*.

The node with no parent is the *root*. A node with no children is a *leaf*; all other nodes are *internal nodes*. The *depth* of a node is the number of edges from the root to that node, so the root has depth zero. The *subtree* rooted at a node contains that node and all of its descendants. The *height* of a node is the number of edges on a longest downward path from that node to a leaf. Leaves therefore have height zero, and the height of the tree is the height of its root.

An arbitrary binary tree may have missing children in many places. Storing such a tree in an array may require empty positions, and the representation becomes wasteful. A binary heap avoids these gaps by imposing a stronger shape.

5.1.2 From a nearly complete tree to an array

Definition 5.1.2 (Nearly complete binary tree). A binary tree is *nearly complete* if every level except possibly the last is full and the nodes of the last level occupy the leftmost available positions.

This shape is precisely what permits compact array storage. Start at the root, read each level from left to right, and append every encountered value to the array. Since only the right end of the last level may be missing, this traversal never needs to reserve an empty array position between two nodes.

For example, consider a nearly complete tree whose levels contain

depth 0 : 16,
depth 1 : 14, 10,
depth 2 : 8, 7, 9, 3.

Concatenating the levels gives

$$A = [16, 14, 10, 8, 7, 9, 3].$$

The root is stored at index 0, its two children at indices 1 and 2, and the next level at indices 3 through 6. The tree and the array contain exactly the same records. They are two views of one representation, not two copies of the data.

We use zero-based indexing throughout this chapter. Whenever a tree node is written as

$$i : v,$$

the first component i is the array index and the second component $v = A[i]$ is the value stored at that position. For example, the label $2 : 10$ means that index 2 contains the value 10. The index is an address in the representation; it is not part of the record's key.

The regular level-order layout makes every parent-child relationship computable from an index. For a node stored at index i , the related positions are

$$\text{left}(i) = 2i + 1, \quad \text{right}(i) = 2i + 2, \quad \text{parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor,$$

where the parent expression is used only for $i > 0$. These formulas describe possible positions. A child exists only if its computed index is smaller than the current logical heap size.

In the running example, the node $2 : 10$ has children at $\text{left}(2) = 5$ and $\text{right}(2) = 6$. These positions contain $5 : 9$ and $6 : 3$. Its parent is at $\lfloor (2-1)/2 \rfloor = 0$, which contains $0 : 16$. No pointer fields are required to recover these relationships.

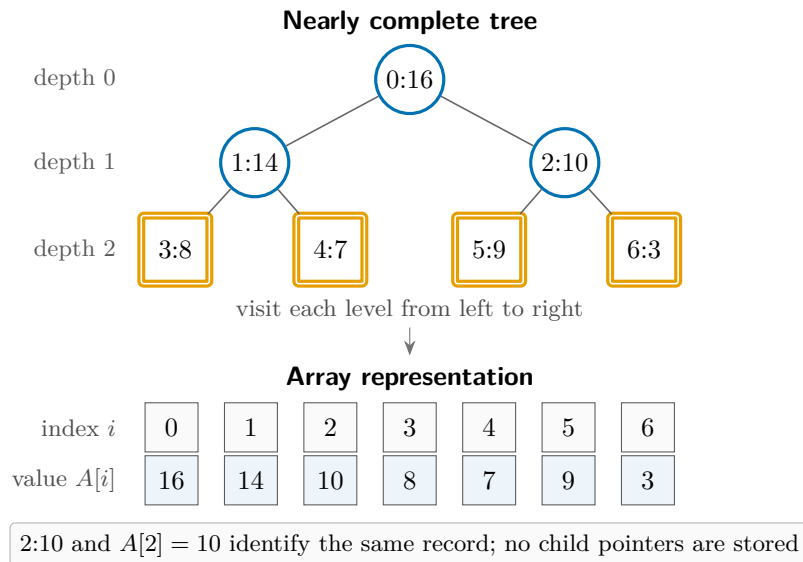


Figure 5.1: A nearly complete tree becomes a compact array by storing its nodes level by level from left to right. Each tree label has the form *index : value*.

The array length and the logical heap size are different quantities. The value $\text{length}(A)$ is the number of physical array positions. The variable `heap_size` counts how many positions currently belong to the heap. Initially they may be equal. During Heapsort, the array length stays fixed while `heap_size` decreases and the released positions form a sorted suffix.

Two views of one structure

The array row gives the physical storage order. A tree node labelled $i : v$ denotes the same record as array cell $A[i] = v$. The lines in the tree make the relationships between indices visible, but they are not pointers stored by the implementation.

Lemma 5.1.3 (Height of a binary heap). *A nearly complete tree containing $n \geq 1$ nodes has height $\lfloor \log_2 n \rfloor$.*

The reason is geometric. All levels above the last are full, so a tree of height h has at least 2^h nodes and at most $2^{h+1} - 1$ nodes. Consequently, doubling the number of nodes increases the height by only about one. This logarithmic height will bound every operation that follows one path from the root to a leaf or in the opposite direction.

5.2 Heap order: max-heaps and min-heaps

The nearly complete shape explains where records are stored, but it says nothing about their values. To obtain a heap, we add a local ordering condition that makes one extreme value immediately accessible.

Definition 5.2.1 (Max-heap and min-heap). A *max-heap* is a nearly complete binary tree, stored as described in Section 5.1, in which every non-root position i satisfies

$$A[\text{parent}(i)] \geq A[i].$$

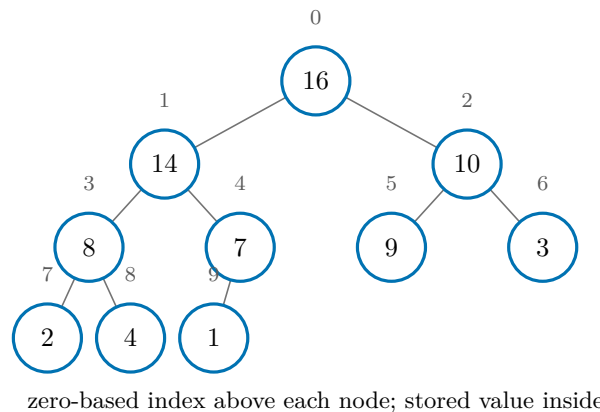
A *min-heap* has the same shape and reverses the inequality.

Following a parent chain in a max-heap can only move to values that are at least as large. The root must therefore contain a maximum value. This is a partial order, not a complete sorting of the records. The children of a node are no larger than their parent, but there is no required order between siblings or between records in different subtrees. For example, the array

$$[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$$

is a max-heap but is not sorted. In particular, the values 7 and 9 occupy different subtrees and need not appear in numerical order in the array.

The corresponding tree is shown below. The array position is written above each node and the stored value appears inside it. Every edge satisfies the max-heap inequality. For instance, the value 14 at position 1 is at least as large as its children 8 and 7, while the value 10 at position 2 is at least as large as its children 9 and 3. The same check applies to every internal node. The root contains 16, the maximum value, even though the remaining values are not globally sorted.



Two independent requirements

The word *heap* combines a shape requirement and an order requirement. The nearly complete shape gives logarithmic height and compact array storage. The local parent-child inequality exposes the maximum or minimum. A nearly complete tree need not satisfy heap order, and a tree satisfying the inequalities need not have the compact shape required by a binary heap.

The array is already a level-order traversal of the tree. A heap is not a binary search tree, however, because the rule does not tell us whether an arbitrary key lies to the left or to the right. Searching for a particular non-extreme value may require inspecting every occupied position and therefore takes $\Theta(n)$ time in the worst case.

5.3 Restoring heap order

Many heap operations disturb only one position while leaving the two child subtrees valid. The repair problem is then local. This situation arises, for example, after replacing the root of a max-heap with a smaller value. The new root may violate heap order, but the trees rooted at its children have not changed and are still max-heaps.

To repair a position i , compare $A[i]$ with its existing children. If $A[i]$ is at least as large as both of them, the subtree already satisfies heap order and the procedure stops. Otherwise, exchange $A[i]$ with the larger child. Choosing the larger child is essential: after the exchange, the new parent must dominate both children. The smaller, displaced value may now violate heap order one level lower, so the same test continues there. Thus the repair follows one downward path rather than inspecting or rebuilding the entire tree.

Downward repair in a max-heap

```
Max-heapify(A, i, heap_size)
  left <- 2*i+1
  right <- 2*i+2
  largest <- i
  if left < heap_size and key(A[left]) > key(A[largest])
    largest <- left
  if right < heap_size and key(A[right]) > key(A[largest])
    largest <- right
  if largest != i
    exchange(A[i], A[largest])
    Max-heapify(A, largest, heap_size)
```

```

    largest ← right
    if largest ≠ i
        exchange A[i] and A[largest]
        Max-heapify(A, largest, heap_size)

```

The precondition is important. Before the call, the left and right subtrees of i must already be max-heaps. Only the relationship between i and its children may be invalid. After an exchange, every untouched subtree remains a heap, and a violation can continue only inside the subtree that received the smaller record.

Consider the array represented in Figure 5.2:

$$A^{(0)} = [4, 14, 10, 8, 7, 9, 3, 2, 1].$$

The two subtrees below the root are max-heaps, but the root value 4 is smaller than both children. At position 0, the larger child is 14 at position 1, so the first exchange produces

$$A^{(1)} = [14, 4, 10, 8, 7, 9, 3, 2, 1].$$

The root is now correct. Only position 1, which received the displaced value 4, can still violate heap order. Its children contain 8 and 7, so the procedure exchanges 4 with the larger value 8 and obtains

$$A^{(2)} = [14, 8, 10, 4, 7, 9, 3, 2, 1].$$

At position 3, the value 4 is at least as large as its children 2 and 1. The procedure therefore stops, and the whole array represents a max-heap.

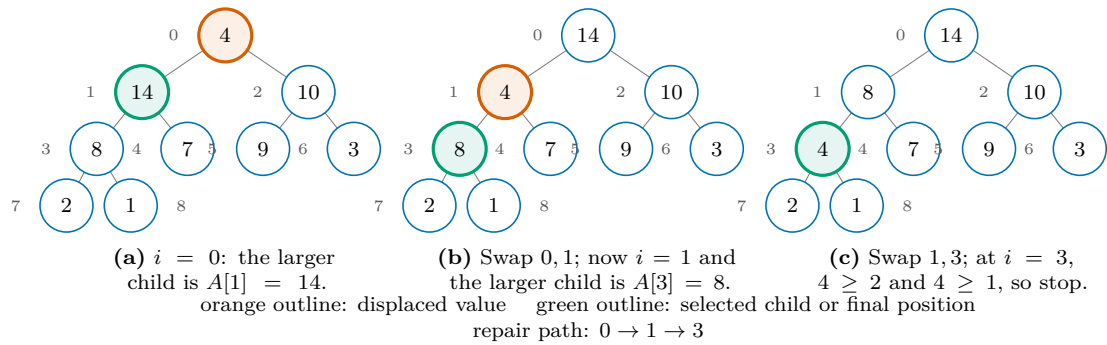


Figure 5.2: Downward repair follows one path: the displaced key is repeatedly exchanged with its larger child until the parent–child order is restored.

Each iteration performs a constant amount of local work and moves the displaced record down by one level. A heap with n records has height $\lfloor \log_2 n \rfloor$, so no repair path can contain more than $O(\log n)$ exchanges. The running time of **Max-heapify** is therefore $O(\log n)$. The recursive version may use $O(\log n)$ stack space in the worst case, whereas an equivalent iterative implementation uses constant auxiliary space.

The symmetric operation, often called **Heapify-up**, repairs a key that has become larger than its parent. It repeatedly exchanges the record with its parent and also takes $O(\log n)$ time.

5.4 Building a heap

An arbitrary array may contain violations at many internal nodes. Calling **Max-heapify** once at the root is not enough because the two child subtrees may not yet be heaps. Bottom-up construction processes the nodes in an order that makes the repair precondition true whenever it is used.

With zero-based indexing, all positions from $\lfloor n/2 \rfloor$ through $n - 1$ are leaves. Each leaf is a one-element heap. Construction therefore begins at the last internal node, whose index is $\lfloor n/2 \rfloor - 1$, and moves backward toward the root.

Bottom-up max-heap construction

```
Build-max-heap(A)
  heap_size ← length(A)
  for i ← floor(length(A)/2)-1 downto 0
    Max-heapify(A, i, heap_size)
  return heap_size
```

When position i is processed, both children have larger indices. They have already been processed or are leaves, so they are roots of heaps. The call can combine those two heaps with $A[i]$ to form one larger heap. When the loop reaches index zero, the complete array is a max-heap.

A first estimate charges each of fewer than n calls $O(\log n)$ time and gives $O(n \log n)$. This bound is valid but not tight. It treats every node as if it could travel from the root to the bottom, whereas most nodes lie close to the leaves and can move only one or two levels.

The geometric distribution of node heights gives the tighter analysis. At most $\lceil n/2^{h+1} \rceil$ nodes have height h , and a repair beginning there descends at most h levels. Therefore the total work is bounded by

$$\sum_{h=0}^{\lfloor \log n \rfloor} O\left(\frac{nh}{2^{h+1}}\right) = O(n) \sum_{h=0}^{\infty} \frac{h}{2^{h+1}} = O(n).$$

The series converges to a constant. Reading the input already costs $\Omega(n)$, so bottom-up heap construction takes $\Theta(n)$ time.

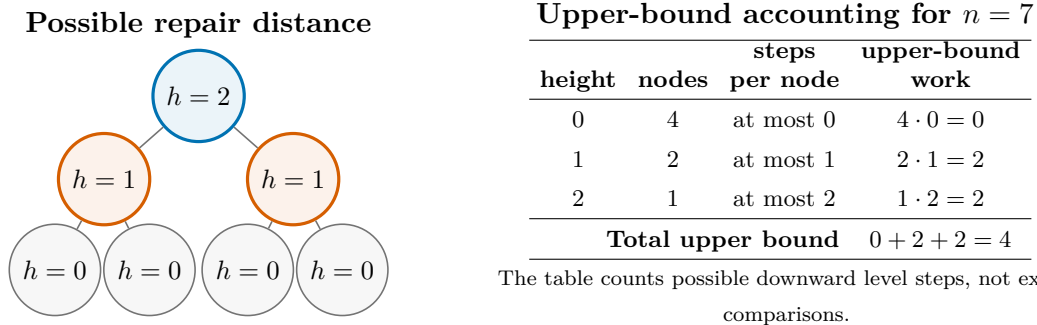


Figure 5.3: A small height-accounting example. Many nodes have zero or one possible downward step, while only a few nodes can travel far.

5.5 Priority queues

Before applying the heap to sorting, it is useful to identify the more general abstraction that it implements. A priority queue stores records that must be processed according to priority rather than arrival time. Examples include selecting the next event in a simulation and choosing the next tentative vertex in a graph algorithm.

Definition 5.5.1 (Priority queue). A max-priority queue supports the operations

Maximum, Extract-max, Insert, Increase-key.

A min-priority queue provides the symmetric minimum-oriented operations.

In a max-heap, Maximum reads the root in $\Theta(1)$ time. To extract it, we move the last occupied record to the root, reduce the logical heap size, and repair downward.

Extracting the maximum

```

Heap-extract-max(A, heap_size)
  if heap_size = 0
    error Empty-heap
  maximum <- A[0]
  A[0] <- A[heap_size-1]
  heap_size <- heap_size-1
  if heap_size > 0
    Max-heapify(A, 0, heap_size)
  return (maximum, heap_size)

```

Insertion places the new record in the next free leaf position and repairs upward. Increasing an existing key uses the same upward path. These operations take $O(\log n)$ time.

Operation	Unsorted array	Sorted array	Binary heap
Find maximum	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Extract maximum	$\Theta(n)$	$\Theta(1)$	$O(\log n)$
Insert	$\Theta(1)$ with free slot	$\Theta(n)$	$O(\log n)$
Search arbitrary key	$\Theta(n)$	$O(\log n)$	$\Theta(n)$

The unsorted-array insertion bound assumes that an unused array position is already available. Resizing a full dynamic array requires amortized analysis, which is developed in Module B.

The comparison shows why the operation mix determines the representation. A heap is excellent for repeated access to an extreme key but is not a general ordered-search structure. Applications that update priorities by record identifier often maintain a separate map from identifiers to heap positions. Every heap exchange must then update that map.

5.6 Heapsort

We can now return to the sorting problem that motivated the chapter. Heapsort first builds a max-heap, so the largest remaining value is always at index zero. It then repeats the same three actions.

1. Exchange the root with the last position of the active heap.
2. Reduce `heap_size`, making the moved maximum part of the sorted suffix.
3. Repair the new root of the remaining heap.

The physical array length never changes. Only the boundary between the active heap and the final sorted suffix moves to the left.

Heapsort

```

Heapsort(A)
  heap_size ← Build-max-heap(A)
  for end ← length(A)-1 downto 1
    exchange A[0] and A[end]
    heap_size ← heap_size-1
    Max-heapify(A, 0, heap_size)

```

At the start of each iteration, the prefix $A[0 \dots \text{heap_size} - 1]$ is a max-heap and the remaining suffix contains the largest records in final order. The root is therefore the largest record not yet placed. Exchanging it with the last heap position extends the sorted suffix by one.

Figure 5.4 follows this process on five values. The blue prefix is the active heap. The green suffix no longer belongs to the heap and will not be changed again. Each row shows the state after the exchange, the boundary update, and the necessary root repair.

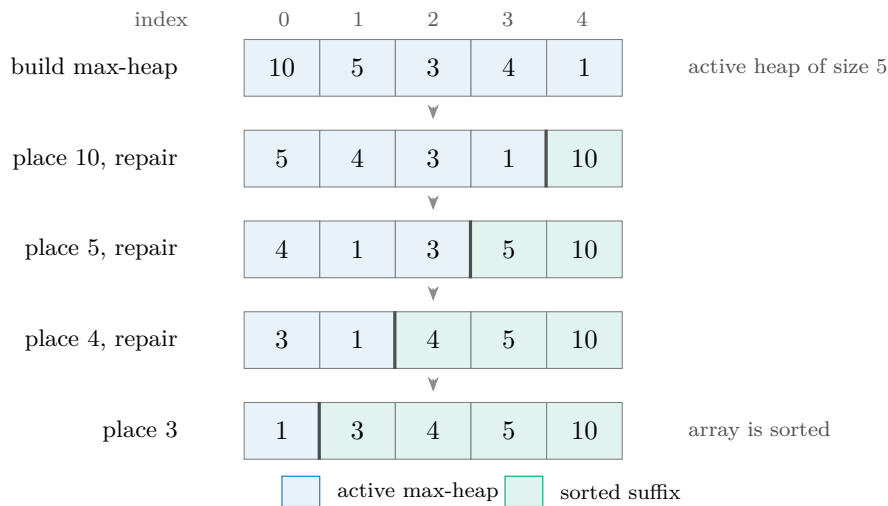


Figure 5.4: Heapsort repeatedly moves the root maximum to the boundary and repairs the smaller heap. The active heap shrinks while the sorted suffix grows.

Bottom-up construction costs $\Theta(n)$. The extraction loop runs $n - 1$ times. Each iteration performs a constant-time exchange and at most one downward path of length $O(\log n)$. Therefore

$$T(n) = \Theta(n \log n)$$

in the best, average, and worst cases. With iterative repair, the algorithm uses constant auxiliary array space and is therefore in place. It is not stable because the long-distance exchanges between the root and the heap boundary can reverse the relative order of equal-key records.

5.7 Where Heapsort fits

Heapsort, Merge sort, and Quicksort all have an $n \log n$ regime, but their data structures and guarantees differ.

- Merge sort uses an auxiliary buffer, is stable, and guarantees $\Theta(n \log n)$ time.
- Heapsort reuses the input array, guarantees $\Theta(n \log n)$ time, and is not stable.
- Quicksort also partitions in place and often has excellent locality, but its ordinary deterministic version has a quadratic worst case.

The heap is therefore not merely an implementation detail of another sorting algorithm. It is a reusable representation whose deliberately limited order supports both sorting and priority-driven algorithms.

5.8 Chapter exercises

In every implementation, keep the logical heap size separate from the physical array length and test the parent-child invariant after each update.

Exercise 5.8.1 (Build and validate a heap). Implement iterative **Max-heapify** and bottom-up construction for zero-based arrays. Build a heap from $[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$, print the array and tree-index view, and write an invariant checker that reports the first violating parent-child pair.

Exercise 5.8.2 (A priority queue for data processing). Use a heap to process records $(priority, id, payload)$ in descending priority. Define a deterministic tie policy, implement insertion and extraction, and compare the results with a reference implementation based on sorting the complete batch. Explain why the record identifier may be needed in a Python heap tuple.

Exercise 5.8.3 (Why heap construction is linear). Instrument bottom-up construction by recording the number of downward levels traveled from every starting height. Run random, sorted, and reversed arrays, aggregate work by height, and compare it with the bound $\sum_{h \geq 0} nh/2^{h+1}$. Distinguish the valid $O(n \log n)$ estimate from the tighter $\Theta(n)$ result.

Exercise 5.8.4 (Heapsort audit). Implement Heapsort and validate after every iteration that the active prefix is a max-heap and the inactive suffix is sorted and contains the largest records. Compare its comparisons, movements, memory use, and stability with Merge sort on a dataset containing repeated keys.

Chapter 6

The limits of comparison sorting

Sources for further study

Book. Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 3rd ed.; Chapter 8.

Book. Dasgupta, Papadimitriou, and Vazirani, *Algorithms*; Section 2.3 and the sorting lower-bound box.

Book. Horowitz, Sahni, and Rajasekaran, *Computer Algorithms*; Lower-bound and linear-time sorting material.

The sorting algorithms studied so far discover order by comparing pairs of keys. Bubble sort and Insertion sort may require quadratic time, while Merge sort and Heapsort achieve $\Theta(n \log n)$ time in the worst case. Quicksort achieves the same bound in expectation under suitable assumptions. The repeated appearance of $n \log n$ raises two natural questions. Is this growth unavoidable, and if so, how can algorithms such as Counting sort and Radix sort run in linear time?

The answer depends on the information available to the algorithm. A comparison can distinguish only two possible relative orders. An integer key in a small known range, on the other hand, can be used directly as an array index. This chapter first proves the lower bound for comparison sorting and then studies two algorithms that escape it by using additional structure in the keys.

6.1 The comparison model

Before proving a lower bound, we must state which algorithms it concerns. The essential restriction is how the algorithm is allowed to learn the relative order of the input keys.

Definition 6.1.1 (Comparison-sorting model). A sorting algorithm works in the *comparison model* if it obtains all information about relative order by comparing two keys. It may move records and use auxiliary storage, but it does not inspect individual digits or use key values as array positions.

For example, Merge sort may ask whether $a_i \leq a_j$, but it does not use the value of a_i to jump directly to an output position. If the input contains n distinct keys, those keys may

arrive in any of $n!$ relative orders. A correct deterministic comparison sort must be able to distinguish every one of them.

The comparisons performed on all possible inputs can be represented by a *decision tree*.

- Each internal node represents one comparison.
- Its two outgoing edges represent the two possible outcomes.
- A root-to-leaf path records the comparisons made for one input order.
- Each reachable leaf identifies one of the possible sorted orders.

Figure 6.1 shows the idea for three distinct records a_1, a_2, a_3 . Each internal node asks an explicit question. For example, the root asks whether $a_1 \leq a_2$. Following the “yes” or “no” edge leads to the next comparison until the relative order of all three records is known. There are $3! = 6$ possible orders, so the tree needs at least six reachable leaves. Some path must therefore contain at least three comparisons, because a binary tree of height two can contain at most four leaves.

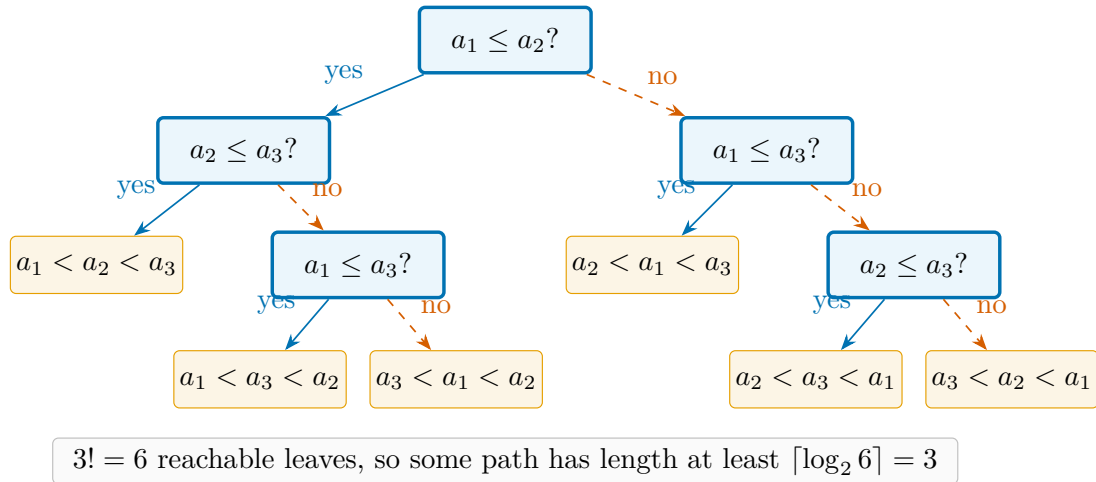


Figure 6.1: A comparison decision tree for three distinct records. Every internal node asks a yes–no comparison question, while each leaf states the relative order that has been determined. The longest path gives the worst-case number of comparisons.

Theorem 6.1.2 (Comparison-sorting lower bound). *Every deterministic comparison-sorting algorithm performs $\Omega(n \log n)$ comparisons in the worst case on n distinct keys.*

The proof generalizes this counting argument. A binary tree of height h has at most 2^h leaves. Since the decision tree needs at least $n!$ reachable leaves,

$$2^h \geq n! \quad \implies \quad h \geq \log_2(n!).$$

For $n \geq 2$, at least $n/2$ factors in the product $n!$ are at least $n/2$. Consequently,

$$n! \geq \left(\frac{n}{2}\right)^{n/2}.$$

Taking logarithms gives

$$\log_2(n!) \geq \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n).$$

The height of the decision tree is the maximum number of comparisons made on any input. It follows that some input requires $\Omega(n \log n)$ comparisons.

The scope of the theorem is important. It is a worst-case lower bound for algorithms that learn order only through comparisons. It does not say that every input requires $n \log n$ comparisons. It also does not apply when the key representation supports operations that reveal more information, such as direct indexing or digit extraction. Counting sort and Radix sort exploit precisely these additional operations.

6.2 Counting sort

Counting sort is our first example of an algorithm that does not discover order by comparing records. It applies when each record has an integer key in a small known range. The key is used directly as an index into an auxiliary array.

Definition 6.2.1 (Bounded integer-key instance). Counting sort accepts n records whose keys belong to

$$\{0, 1, \dots, k\}.$$

For keys in an interval $[L, U]$, subtracting L gives the equivalent range $\{0, \dots, U - L\}$. The size of this range is part of the cost of the algorithm.

6.2.1 The central idea

Assume that the input is

$$A = [2, 5, 3, 0, 2, 3, 0, 3].$$

Instead of repeatedly asking which of two values is smaller, we first count how many times each value occurs. For keys from 0 to 5, the frequency array is

$$C = [2, 0, 2, 3, 0, 1].$$

This already tells us which values must appear in the output, but it does not yet tell us the position of each individual record.

We therefore replace each frequency with a cumulative count. After this step,

$$C = [2, 2, 4, 7, 7, 8],$$

and $C[v]$ is the number of input records whose key is at most v . For example, $C[2] = 4$ means that the four records with keys at most 2 must occupy output positions 0, 1, 2, 3. Hence the last available position for a record with key 2 is $C[2] - 1 = 3$.

Array convention used in this chapter

The input and output arrays use zero-based positions, so $A[0 \dots n-1]$ and $B[0 \dots n-1]$. The counter array is indexed by key value, so $C[0 \dots k]$. After cumulative counting, a record with key v is placed at position $C[v] - 1$. This convention agrees with the Python implementations used in the course.

6.2.2 The algorithm

The stable form of Counting sort uses three arrays. Array A contains the input records, array C contains counts and cumulative counts, and array B receives the sorted output. The algorithm proceeds through three simple scans.

Stable Counting sort

```

Counting-sort(A, k)
  require k >= 0 and 0 <= key(record) <= k for every record in A
  C[0 .. k] <- all zeroes
  for j <- 0 to length(A)-1
    C[key(A[j])] <- C[key(A[j])] + 1
  for value <- 1 to k
    C[value] <- C[value] + C[value-1]
  B[0 .. length(A)-1] <- empty
  for j <- length(A)-1 downto 0
    value <- key(A[j])
    position <- C[value]-1
    B[position] <- A[j]
    C[value] <- C[value]-1
  return B

```

The meaning of C changes during the algorithm.

1. After the first scan, $C[v]$ is the number of records whose key is exactly v .
2. After the cumulative scan,

$$C[v] = |\{x \in A \mid \text{key}(x) \leq v\}|.$$

Thus $C[v] - 1$ is the rightmost output position currently available for key v .

3. During the final scan, each record is written directly to that position. The corresponding counter is then decreased so that the next equal key uses the preceding position.

Figure 6.2 follows the same running example through these stages. Panel (a) shows the frequency counts, and panel (b) shows the cumulative counts. Panels (c)–(e) show the first three iterations of the reverse scan.

In panel (c), $A[7] = 3$ and $C[3] = 7$, so the algorithm writes 3 to $B[6]$ and decreases $C[3]$ to 6. In panel (d), $A[6] = 0$ is written to $B[1]$. In panel (e), $A[5] = 3$ is written to $B[5]$. Panel (f) shows the final sorted array.

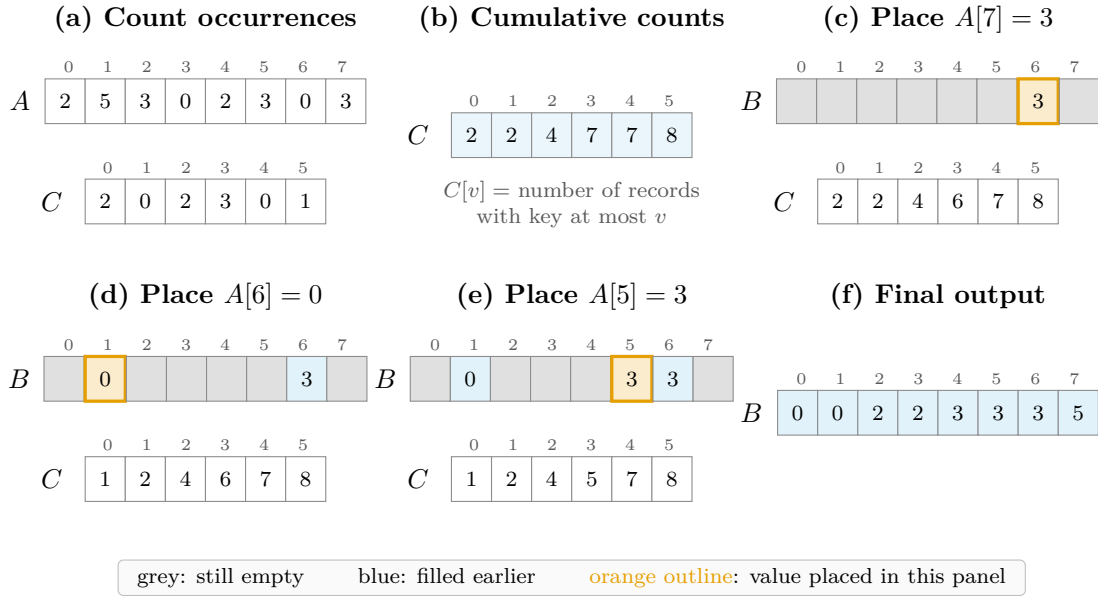


Figure 6.2: Counting sort first counts equal keys, then computes cumulative counts, and finally places records from right to left. Light cells in B have already been filled; the orange outline marks the most recent placement.

6.2.3 Why the reverse scan matters

A sorting algorithm is *stable* if records with equal keys retain their input order. Stability matters when records contain additional information. Two students may have the same mark, for example, while still carrying different names and identifiers.

Suppose the three occurrences of key 3 in the running example are tagged $3_a, 3_b, 3_c$ according to their input order. The reverse scan encounters them as $3_c, 3_b, 3_a$, but it fills their reserved block from right to left. It therefore places 3_c last, 3_b before it, and 3_a first. Their output order remains $3_a, 3_b, 3_c$.

If the placement scan moved from left to right while still filling positions from right to left, equal records would be reversed. The numerical values would remain sorted, but the algorithm would no longer be stable. This distinction becomes essential in Radix sort.

6.2.4 Cost and applicability

Initializing and accumulating the counter array costs $\Theta(k)$, while counting and placing the records costs $\Theta(n)$. Therefore

$$T(n, k) = \Theta(n + k), \quad S(n, k) = \Theta(n + k)$$

when the output array is included. The counter array alone requires $\Theta(k)$ additional positions.

The two parameters must remain visible. For ten records whose keys are 0 and 10^9 , a direct counter array requires roughly one billion positions. Counting sort is therefore useful when k is reasonably small compared with n and direct indexing is available.

Why the parameter k cannot be hidden

The input is described by both the number of records n and the largest normalized key k . Calling Counting sort “linear” is justified only after stating that $k = O(n)$. Under this assumption, $\Theta(n + k) = \Theta(n)$. Counting sort does not contradict the comparison lower bound because it uses key values as array indices and performs no comparisons between input records.

6.3 Radix sort

Counting sort is unattractive when the complete key range is very large. A decimal number may be large even though each of its digits belongs to the small set $\{0, \dots, 9\}$. Radix sort exploits this representation by sorting one digit position at a time.

The surprising choice is to begin with the least significant digit. After sorting by the units digit, the complete array is sorted again by the tens digit, then by the hundreds digit, and so on. Every digit sort must be stable. When two records have the same current digit, stability preserves the order already established by the less significant digits.

Definition 6.3.1 (LSD Radix sort requirements). Least-significant-digit Radix sort, abbreviated LSD Radix sort, assumes that every key has d digit positions and that each digit belongs to $\{0, \dots, b-1\}$. It applies a stable sorting method to positions $0, 1, \dots, d-1$, beginning with the least significant position 0.

Fixed-width nonnegative integers satisfy this requirement directly. Shorter numbers can be padded with leading zeroes. Other data types require a representation whose lexicographic digit order agrees with the desired key order.

6.3.1 The algorithm and a complete trace

LSD Radix sort

```
LSD-radix-sort(A, d, b)
  require d >= 0 and b >= 2
  require every key has d base-b digits
  for position <- 0 to d-1
    A <- Stable-digit-sort(A, position, b)
  return A
```

Consider the seven three-digit values

[329, 457, 657, 839, 436, 720, 355].

Figure 6.3 shows the complete execution. The highlighted digit is the digit used to produce the current column.

The first pass sorts by units and produces

[720, 355, 436, 457, 657, 329, 839].

Notice that 457 remains before 657, since both have units digit 7, and 329 remains before 839, since both have units digit 9.

The second stable pass sorts by tens and produces

[720, 329, 436, 839, 355, 457, 657].

The values are now ordered by their two-digit suffixes 20, 29, 36, 39, 55, 57, 57. The final pass sorts by hundreds and yields the complete numeric order.

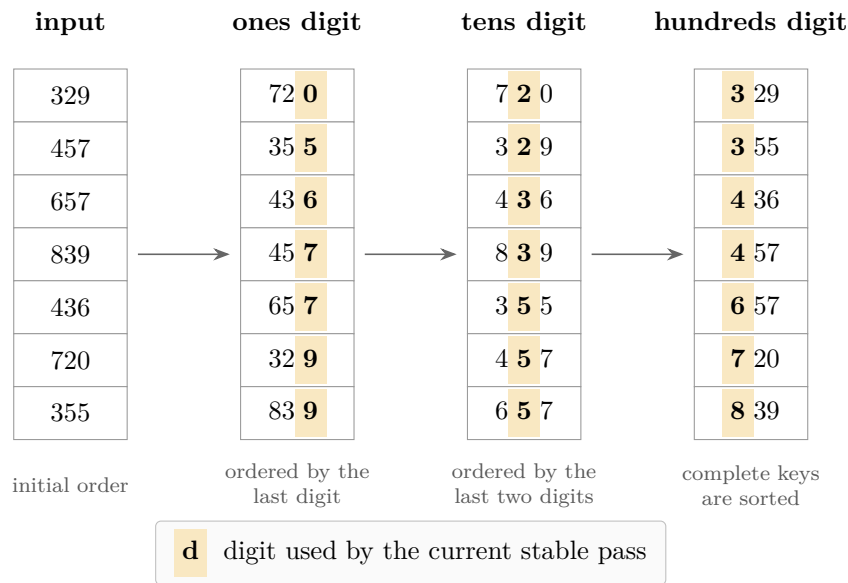


Figure 6.3: LSD Radix sort on seven three-digit values. Each column is obtained by a stable sort on the highlighted digit of the preceding column.

6.3.2 Why the method works

After the units pass, the array is sorted by its last digit. After the tens pass, it is sorted by its last two digits. More generally, after processing position i , the array is sorted by the suffix formed by positions $i, i - 1, \dots, 0$.

To see why stability is essential, compare 21 and 22. The units pass places 21 before 22. Their tens digits are equal. A stable tens-digit pass preserves their order, while an unstable pass may reverse them and destroy the ordering obtained from the units digits.

This reasoning gives the following invariant.

After pass i , the records are sorted by their $i + 1$ least significant digits.

The invariant is true after the first pass. During the next pass, records with different active digits are put in the correct order by that digit. Records with the same active digit retain their previous suffix order because the digit sort is stable. After d passes, the suffix contains the entire key, so the array is sorted.

The same idea applies to records with several fields. Dates can be ordered by three stable sorts, first by day, then by month, and finally by year. The most important field is processed last.

6.3.3 Cost and practical conditions

If Counting sort is used for each digit, one pass costs $\Theta(n + b)$ because the digit has b possible values. There are d passes, and therefore

$$T(n, d, b) = \Theta(d(n + b)), \quad S(n, b) = \Theta(n + b).$$

When d is constant and $b = O(n)$, the total running time is $\Theta(n)$. For example, every integer in $[0, n^3 - 1]$ has at most three base- n digits. Three Counting-sort passes each cost $\Theta(n + n)$, giving $\Theta(n)$ total time under the unit-cost word-RAM model.

The radix b creates a trade-off. Increasing b reduces the number of digits but enlarges the counter array used during each pass. Moreover, the usual stable implementation is not in place. Radix sort can be asymptotically faster than a comparison sort without being faster for every dataset or implementation.

As with Counting sort, the comparison lower bound does not apply. Radix sort inspects individual digits and uses them as indices during the stable digit sorts.

6.3.4 When Radix sort is not directly applicable

Radix sort is not a general replacement for comparison sorting. Its input keys must have a positional representation, and the order of those representations must agree with the order that the algorithm is expected to produce. In addition, every digit pass must be efficient and stable. If one of these requirements is absent, the standard LSD algorithm cannot be applied directly.

It is useful to distinguish two different situations. In the first, the data can be transformed into a suitable representation, but that transformation must be stated and justified. In the second, Radix sort is technically possible but offers no computational or practical advantage.

No suitable positional order has been defined

Suppose that records are ordered by an arbitrary comparison procedure. A university may, for example, order applications first by an eligibility rule, then by a score with several exceptional cases, and finally by a locale-dependent comparison of names. Unless this rule is decomposed into a finite sequence of ordered fields, there is no digit position on which Radix sort can operate. A comparison sort can call the rule directly, whereas Radix sort first needs an order-preserving encoding.

The same difficulty occurs when the visible representation does not have the desired order. Sorting the strings 2, 11, and 100 lexicographically gives

$$100 < 11 < 2,$$

which is different from their numerical order

$$2 < 11 < 100.$$

Treating these strings as sequences of characters would therefore solve a different sorting problem. For nonnegative integers, padding with leading zeroes repairs the representation. With three positions, the strings become 002, 011, and 100, whose lexicographic and numerical orders agree.

Keys do not share the assumed format

The simple LSD algorithm assumes exactly d positions for every key. Variable-length strings therefore require an additional convention. For ordinary lexicographic order, shorter strings may be padded on the right with a special end marker that is smaller than every alphabet symbol. Thus **a**, **aa**, and **b** may be represented as

$$\mathbf{a}\perp, \quad \mathbf{aa}, \quad \mathbf{b}\perp,$$

where $\perp$ precedes every letter. An LSD implementation can then process the positions from right to left. The convention is part of the algorithm, however, and the number of passes is the maximum string length. An MSD string sort, which begins with the most significant character and recursively processes equal-prefix groups, may be a more natural alternative.

Signed integers also require care. Applying an unsigned digit order directly to the written sign and magnitude does not place negative values correctly. For example, ordering magnitudes would put -2 before -10 , even though

$$-10 < -2.$$

Possible remedies include translating a known bounded interval to nonnegative keys, or treating negative and nonnegative values separately with a carefully specified recombination rule. The transformation must preserve the desired numerical order.

Floating-point values present a similar problem. Their printed decimal strings may have signs, decimal points, exponents, and different lengths. Their raw machine bit patterns also do not behave as ordinary unsigned integers across negative and positive values. Specialized order-preserving transformations exist for fixed floating-point formats, but the basic integer version of Radix sort cannot be used without such a transformation.

The digit procedure is not stable

LSD Radix sort relies on earlier passes when it processes a more significant digit. Suppose the units pass has correctly placed 21 before 22. During the tens pass, both records have digit 2. If that pass is unstable, it may reverse them and return 22, 21. The final array is then not sorted. Consequently, replacing the stable digit sort with an arbitrary in-place comparison sort does not in general produce a valid LSD Radix sort.

Stability is not merely an implementation preference in this setting. It is the mechanism that preserves the ordering already established by the less significant positions.

The digit range or the number of positions is too large

Even when Radix sort is correct, its cost may be unattractive. Consider $n = 1,000$ unsigned 64-bit keys. Treating each complete key as one digit would give $b = 2^{64}$. A Counting-sort pass would then require an impossible counter array with 2^{64} entries. Dividing each key into bytes gives $b = 256$ and $d = 8$, producing the much more reasonable bound

$$\Theta(8(n + 256)).$$

This example shows why the radix must be chosen rather than simply made as large as possible.

A large number of positions can cause the opposite problem. If the keys are strings of maximum length $d = 10^6$, an LSD implementation may perform one million full-array passes even when most strings are short. The nominal bound

$$\Theta(d(n + b))$$

then exposes the difficulty directly. The algorithm is valid, but a comparison sort, an MSD variant, or a trie-based method may examine much less irrelevant data.

Memory may rule out the method as well. A stable Counting-sort digit pass normally uses an output array of size n and a counter array of size b , for $\Theta(n + b)$ auxiliary space. When additional storage is tightly constrained, in-place Heapsort or another comparison-based method may be preferable despite its $\Theta(n \log n)$ running time.

Radix-sort applicability checklist

Before choosing Radix sort, verify that each key has an order-preserving positional representation, that the number of positions d is manageable, that every position has a manageable range b , and that the digit-sorting procedure is stable. If a conversion is needed for strings, signs, floating-point values, or compound fields, state it explicitly and prove that it preserves the intended order.

6.4 Choosing a sorting method

The input representation and the required guarantees should determine the algorithm. The following table places all methods studied in Chapters 4–6 in one decision framework.

Algorithm	Running time	Extra space	Most relevant condition
Insertion sort	worst $\Theta(n^2)$	$\Theta(1)$	small or nearly sorted input; stable
Merge sort	worst $\Theta(n \log n)$	$\Theta(n)$	stable, predictable running time
Quicksort	expected $\Theta(n \log n)$; worst $\Theta(n^2)$	expected $\Theta(\log n)$ stack	fast average-case array sorting
Heapsort	worst $\Theta(n \log n)$	$\Theta(1)$	low memory; worst-case guarantee
Counting sort	$\Theta(n + k)$	$\Theta(n + k)$	small integer-key range
Radix sort	$\Theta(d(n + b))$	$\Theta(n + b)$	fixed-width digit keys

There is no universally fastest row. A claim that Counting sort is linear must state the key-range assumption. A claim that Quicksort is $n \log n$ must distinguish expected

performance from worst-case performance. A claim that Merge sort is in-place must identify a nonstandard implementation. Algorithm selection is a contract between the data, the representation, and the required guarantee.

The lower bound and its boundary

The $\Omega(n \log n)$ result applies when relative order can be learned only through pairwise comparisons. Counting sort and Radix sort do not violate it: they assume a structured key representation and use operations, such as digit extraction and direct indexing, that are unavailable in the comparison model.

6.5 Chapter exercises

The exercises emphasize the representation assumptions that permit a non-comparison sort.

Exercise 6.5.1 (Stable Counting sort on records). Implement Counting sort for records $(score, id)$ with scores in $[0, 100]$. Validate stability using repeated scores, compare the output with a trusted stable library sort, and report separately the time spent initializing counters, counting records, accumulating counts, and placing output.

Exercise 6.5.2 (Normalize a bounded key range). One million transaction records have integer codes in $[10^9, 10^9 + 100]$. Design the translation used by Counting sort, prove that it preserves order, and compare the counter size with the incorrect design indexed from zero to $10^9 + 100$. Include validation for an out-of-range input record.

Exercise 6.5.3 (Radix-sort implementation audit). Implement LSD Radix sort for fixed-width nonnegative identifiers and retain a trace after every digit. Use tagged records to verify that the digit routine is stable. Then replace it deliberately with an unstable routine, construct a failing input, and explain which pass invariant is lost.

Exercise 6.5.4 (Choose a sort from a data contract). Select and justify a method for examination marks in $[0, 100]$, signed 64-bit sensor readings, locale-dependent surnames, fixed-width identifiers, and arbitrary records requiring stable order. For every dataset, state the key representation, parameters n , k or (d, b) , auxiliary-memory limit, and the validation tests that the implementation must pass.