Exercise Notes - Chpt 1, Chpt 2

Harley Caham Combest

Fa2025 CS4413 Lecture Notes – Mk1

. . . . . .

Chapter 1: The Role of Algorithms in Computing

. . . . . .

1.1 Exercises.

Exercise 1. ***1.1-1***
Describe your own real-world example that requires sorting. Describe one that requires finding the shortest distance between two points.

Proof. Solution.

Intuition: Sorting is like putting exam papers in alphabetical order so the teacher can enter grades quickly. Finding the shortest distance is like using Google Maps to walk from the dorm to the library.

Rigorous: Sorting takes a sequence of items with keys and produces a permutation in nondecreasing order. Examples include insertion sort, merge sort, or quicksort. The shortest distance problem can be formalized as finding a minimum-weight path in a graph $G = (V, E, w)$. This is solvable with Dijkstra's algorithm for nonnegative weights or Bellman–Ford if negative edges are possible. □

Exercise 2. ***1.1-2***
Other than speed, what other measures of efficiency might you need to consider in a real-world setting?

Proof. Solution.

Intuition: A program that runs fast but crashes, drains the battery, or uses all the memory is not efficient.

Rigorous: Additional efficiency measures include:

- Space complexity (auxiliary and peak memory usage).
- I/O cost and cache locality.
- Energy/power consumption.
- Scalability, throughput, and latency.
- Numerical stability and precision in scientific computing.
- Reliability, fault tolerance, and maintainability.

These factors often determine the practicality of an algorithm. □

Exercise 3. ***1.1-3***
Select a data structure that you have seen, and discuss its strengths and limitations.

Proof. Solution.

Intuition: A hash table is like cubbies with labels. Put something in by label and you can grab it instantly.

Rigorous: *Strengths:* $O(1)$ expected time for insert/search/delete; ideal for membership tests and frequency counts. *Limitations:* No ordering (can't support range queries or "next"); worst-case $O(n)$ under many collisions; memory overhead and resizing; performance depends on hash function quality. Balanced BSTs give $O(\log n)$ operations with ordering guarantees. □

Exercise 4. ***1.1-4***
How are the shortest-path and traveling-salesperson problems given above similar? How are they different?

Proof. Solution.

Intuition: Both are about "finding the best route." The shortest path is one trip, the TSP is visiting every city exactly once.

Rigorous: The shortest-path problem finds a minimum-weight path between two vertices and is solvable in polynomial time (Dijkstra, Bellman–Ford). The traveling-salesperson problem (TSP) seeks a minimum Hamiltonian cycle and is NP-hard. Thus SP has efficient algorithms; TSP requires exponential-time exact methods or approximation schemes. □

Exercise 5. ***1.1-5***
Suggest a real-world problem in which only the best solution will do. Then come up with one in which "approximately" the best solution is good enough.

Proof. Solution.

Exact best required: Radiation therapy planning—constraints on dose to organs-at-risk must be satisfied exactly. Gate assignment in airports also requires exact conflict-free scheduling.

Approximate is fine: Delivery routing (TSP-like) where a route within a few percent of optimal still saves time and fuel. Search engine ranking, where nearly-best relevance is sufficient for users. □

Exercise 6. ***1.1-6***
Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advance and arrives over time.

Proof. Solution.

Intuition: Ride-sharing requests after a concert may arrive in one big batch (offline), but on a normal night they arrive one at a time (online).

Rigorous: Offline algorithms assume complete knowledge of input before computation. Online algorithms must decide without knowing future inputs. Competitive analysis is used to compare online algorithms to the offline optimal. □

. . . . . .

1.2 Exercises.

Exercise 7. ***1.2-1***
Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

Proof. Solution.

Intuition: A navigation app like Google Maps cannot just draw streets; it must use algorithms to compute routes, estimate traffic delays, and suggest alternatives.

Rigorous: At the application level, shortest-path algorithms (e.g., Dijkstra's or A*) are required to compute routes in a weighted graph model of the road network. Additional algorithms like dynamic programming (for travel-time prediction), flow algorithms (for congestion modeling), or data structures (priority queues for fast edge relaxations) are essential to ensure practical, efficient service. □

Exercise 8. ***1.2-2***
Suppose that for inputs of size n on a particular computer, insertion sort runs in $8n^2$ steps and merge sort runs in $64n \lg n$ steps. For which values of n does insertion sort beat merge sort?

Proof. Solution.

Intuition: For small n, the n^2 method can be faster, but as n grows, $n \lg n$ grows more slowly, so merge sort eventually wins.

Rigorous: We compare:

$$8n^2 < 64n \lg n \quad \Longleftrightarrow \quad n < 8 \lg n.$$

The inequality $n < 8 \lg n$ holds only for small n. Checking values:

- $n = 2$: $2 < 8 \cdot 1 = 8$ (true).
- $n = 10$: $10 < 8 \cdot \lg 10 \approx 26.6$ (true).
- $n = 40$: $40 < 8 \cdot \lg 40 \approx 42.6$ (true).
- $n = 43$: $43 < 8 \cdot \lg 43 \approx 42.6$ (false).

Thus insertion sort beats merge sort for $n \leq 43$ on this machine. □

Exercise 9. ***1.2-3***
What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

Proof. Solution.

Intuition: Exponentials eventually outgrow quadratics. The question is where the crossover happens.

Rigorous: We want $100n^2 < 2^n$. Checking values:

- $n = 10$: $100 \cdot 100 = 10{,}000$ vs $2^{10} = 1{,}024$ (false).
- $n = 15$: $100 \cdot 225 = 22{,}500$ vs $2^{15} = 32{,}768$ (true).
- $n = 14$: $100 \cdot 196 = 19{,}600$ vs $2^{14} = 16{,}384$ (false).

Therefore the smallest n is $n = 15$. □

. . . . . .

Chpt 1 Problems.

Exercise 10. ***1-1 Comparison of running times***
For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds.

$f(n)$	*1 second*	*1 minute*	*1 hour*	*1 day*	*1 month*	*1 year*
$\lg n$						
$\sqrt{n}$						
n						
$n \lg n$						
n^2						
n^3						
2^n						
$n!$						

Proof. Solution.

Intuition: Think of each row as "how big an input can you handle" in a given amount of time. Fast-growing functions (like 2^n and $n!$) explode quickly, so they only allow very small n. Slow-growing functions (like $\lg n$) let you solve astronomically large instances.

Rigorous: We assume 1 second $= 10^6$ microseconds, 1 minute $= 6 \cdot 10^7$, 1 hour $= 3.6 \cdot 10^9$, 1 day $= 8.64 \cdot 10^{10}$, 1 month $\approx 2.6 \cdot 10^{12}$, 1 year $\approx 3.15 \cdot 10^{13}$. Solving $f(n) \leq T$ for each case yields:

$f(n)$	1 second	1 minute	1 hour	1 day	1 month	1 year
$\lg n$	2^{10^6}	$2^{6 \cdot 10^7}$	$2^{3.6 \cdot 10^9}$	$2^{8.64 \cdot 10^{10}}$	$2^{2.6 \cdot 10^{12}}$	$2^{3.15 \cdot 10^{13}}$
$\sqrt{n}$	10^{12}	$3.6 \cdot 10^{15}$	$1.3 \cdot 10^{19}$	$7.5 \cdot 10^{21}$	$6.7 \cdot 10^{25}$	$9.9 \cdot 10^{26}$
n	10^6	$6 \cdot 10^7$	$3.6 \cdot 10^9$	$8.6 \cdot 10^{10}$	$2.6 \cdot 10^{12}$	$3.2 \cdot 10^{13}$
$n \lg n$	$\approx 6.3 \cdot 10^4$	$\approx 2.8 \cdot 10^6$	$\approx 1.3 \cdot 10^8$	$\approx 2.7 \cdot 10^9$	$\approx 7.7 \cdot 10^{10}$	$\approx 8.8 \cdot 10^{11}$
n^2	10^3	$7.7 \cdot 10^3$	$6 \cdot 10^4$	$2.9 \cdot 10^5$	$1.6 \cdot 10^6$	$5.6 \cdot 10^6$
n^3	100	390	$1.5 \cdot 10^3$	$4.4 \cdot 10^3$	$1.4 \cdot 10^4$	$3.2 \cdot 10^4$
2^n	20	26	32	36	41	45
$n!$	9	11	12	13	15	16

Thus, functions with polynomial or slower growth scale to very large n, while exponential or factorial growth makes even modest n infeasible. $\square$

Chapter 2: Getting Started

2.1 Exercises.

Exercise 11. ***2.1-1***

Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on an array initially containing the sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$.

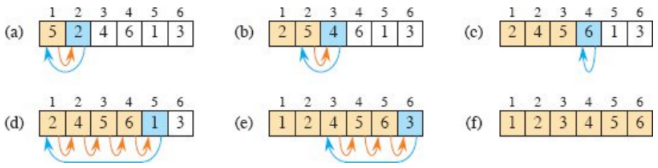

Figure 2.2 The operation of INSERTION-SORT(A, n), where A initially contains the sequence $\langle 5, 2, 4, 6, 1, 3 \rangle$ and $n = 6$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the blue rectangle holds the key taken from $A[i]$, which is compared with the values in tan rectangles to its left in the test of line 5. Orange arrows show array values moved one position to the right in line 6, and blue arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

Proof. Solution.

Intuition: Treat the left prefix as your "sorted hand." At step j, pull out the key $A[j]$, slide any larger elements in the prefix one position right, and drop the key in the hole that opens.

Rigorous (full trace matching Fig. 2.2): Let the outer loop index be $j = 2, \ldots, 6$ and the key be $A[j]$ at each step.

$$
\begin{aligned}
&\text{Initial} && [\,5,\ 2,\ 4,\ 6,\ 1,\ 3\,] \\
j=2,\ \text{key}=2: \quad & [\,\underline{5},\ \boxed{2},\ 4,\ 6,\ 1,\ 3\,] && \Rightarrow\ [\,2,\ 5,\ 4,\ 6,\ 1,\ 3\,] \\
j=3,\ \text{key}=4: \quad & [\,2,\ \underline{5},\ \boxed{4},\ 6,\ 1,\ 3\,] && \Rightarrow\ [\,2,\ 4,\ 5,\ 6,\ 1,\ 3\,] \\
j=4,\ \text{key}=6: \quad & [\,2,\ 4,\ 5,\ \boxed{6},\ 1,\ 3\,] && \Rightarrow\ [\,2,\ 4,\ 5,\ 6,\ 1,\ 3\,] \quad \text{(no shifts)} \\
j=5,\ \text{key}=1: \quad & [\,2,\ 4,\ 5,\ 6,\ \boxed{1},\ 3\,] && \Rightarrow\ \text{shift } 6,5,4,2 \text{ right } \Rightarrow\ [\,1,\ 2,\ 4,\ 5,\ 6,\ 3\,] \\
j=6,\ \text{key}=3: \quad & [\,1,\ 2,\ 4,\ 5,\ 6,\ \boxed{3}\,] && \Rightarrow\ \text{shift } 6,5,4 \text{ right } \Rightarrow\ [\,1,\ 2,\ \boxed{3},\ 4,\ 5,\ 6\,]
\end{aligned}
$$

Final (sorted): [1, 2, 3, 4, 5, 6].

Why this is correct: The standard loop invariant holds: before processing index j, the prefix $A[1..j-1]$ is sorted in nondecreasing order. The inner while-loop maintains the invariant by shifting all elements greater than *key* one position to the right and then placing *key* in the unique position that preserves order. At termination ($j = 7$), the entire array $A[1..6]$ is sorted. □

Exercise 12. ***2.1-2***
Consider the procedure SUM-ARRAY. It computes the sum of the n numbers in array $A[1:n]$. State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the procedure returns the sum of the numbers in $A[1:n]$.

Proof. Solution.

Intuition: After $i-1$ steps, the running total is exactly the sum of the first $i-1$ items. Each step adds the next item, so the invariant "slides" forward until all items are included.

Rigorous (loop invariant): At the start of each iteration with index i ($1 \le i \le n+1$),

$$\texttt{sum} \;=\; \sum_{k=1}^{i-1} A[k].$$

Initialization ($i = 1$): Before the first iteration, `sum`$= 0$, which equals the empty sum $\sum_{k=1}^{0} A[k] = 0$.
Maintenance: Assuming the invariant holds at the start of iteration $i \le n$, the body executes `sum = sum + A[i]`, yielding

$$\texttt{sum} \;=\; \sum_{k=1}^{i-1} A[k] + A[i] \;=\; \sum_{k=1}^{i} A[k],$$

which matches the invariant for the next iteration ($i \leftarrow i+1$).
Termination: The loop ends when $i = n+1$. By the invariant,

$$\texttt{sum} \;=\; \sum_{k=1}^{n} A[k],$$

so the returned value is the sum of $A[1:n]$. □

Exercise 13. ***2.1-3***
Rewrite the INSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.

Proof. Solution.

Intuition: The only change is the comparison direction: shift elements that are *smaller* than the key to the right so that larger elements stay to the left.

Rigorous (pseudocode):

```
DECREASING-INSERTION-SORT(A, n)
1  for j = 2 to n
2      key = A[j]
3      i = j - 1
4      while i >= 1 and A[i] < key       // flip the comparison
5          A[i+1] = A[i]
6          i = i - 1
7      A[i+1] = key
```

Correctness sketch. The standard loop invariant adapts: at the start of each outer iteration j, the prefix $A[1..j-1]$ is sorted in *nonincreasing* order. The inner loop shifts elements $<$ `key` rightward, and places `key` in the unique spot preserving nonincreasing order. □

Exercise 14. ***2.1-4***
Write pseudocode for linear search, which scans through the array from beginning to end, looking for x. Using a loop invariant, prove that your algorithm is correct.

Proof. Solution.

Intuition: Walk left-to-right; if you see x, return its index; if you finish the walk without seeing it, return NIL.

Rigorous (pseudocode):

```
LINEAR-SEARCH(A, n, x)
1  for i = 1 to n
2      if A[i] == x
3          return i
4  return NIL
```

Loop invariant: At the start of each iteration with index i ($1 \leq i \leq n+1$), x does not occur in $A[1..i-1]$.
Initialization: For $i = 1$, the prefix $A[1..0]$ is empty, so the invariant holds.
Maintenance: If $A[i] = x$, we return i and are correct. If not, then $x \notin A[1..i]$, so at the next iteration ($i \leftarrow i+1$) the invariant holds.
Termination: If the loop completes, $i = n+1$ and the invariant gives $x \notin A[1..n]$, so returning NIL is correct. □

Exercise 15. ***2.1-5***
Consider adding two n-bit binary integers a and b, stored in arrays $A[0:n-1]$ and $B[0:n-1]$ (LSB at index 0). The sum $c = a+b$ should be stored in an $(n+1)$-element array $C[0:n]$. Write ADD-BINARY-INTEGERS.

Proof. Solution.

Intuition: Add bit-by-bit from least significant to most significant, carrying a 0/1 as needed (exactly like pencil-and-paper addition in base 2).

Rigorous (pseudocode & correctness):

```
ADD-BINARY-INTEGERS(A, B, n)
1  carry = 0
2  for i = 0 to n-1
3      s = A[i] + B[i] + carry       // s IN {0,1,2,3}
4      C[i] = s mod 2
5      carry = floor(s / 2)          // 0 or 1
6  C[n] = carry
7  return C
```

Correctness invariant: After processing bit $i-1$ (i.e., at loop index i),

$$\sum_{k=0}^{i-1} C[k] \cdot 2^k \;+\; \texttt{carry} \cdot 2^i \;=\; \sum_{k=0}^{i-1} A[k] \cdot 2^k \;+\; \sum_{k=0}^{i-1} B[k] \cdot 2^k.$$

Initialization holds with all sums 0. Maintenance follows from base-2 addition: $s = A[i] + B[i] + \texttt{carry}$ writes $C[i] = s \bmod 2$ and updates $\texttt{carry} = \lfloor s/2 \rfloor$, preserving the weighted-sum identity. At termination ($i = n$), carry is stored as $C[n]$, so C encodes $a + b$ in binary. □

. . . . . .

2.2 Exercises.

Exercise 16. ***2.2-1***
Express the function $n^3/1000 + 100n^2 - 100n + 3$ *in terms of* Θ*-notation.*

Proof. Solution.

Intuition: For large n, the highest-degree term "wins." Cubic beats quadratic and linear, so the function grows like n^3.

Rigorous: There exist $c_1, c_2 > 0$ and n_0 such that for all $n \geq n_0$,

$$c_1 n^3 \ \leq \ \frac{n^3}{1000} + 100n^2 - 100n + 3 \ \leq \ c_2 n^3.$$

For example, take $n_0 = 1000$, $c_1 = \frac{1}{2000}$ and $c_2 = \frac{1}{1000} + 100 + \frac{100}{n_0} + \frac{3}{n_0^3}$. Hence $n^3/1000 + 100n^2 - 100n + 3 \in \Theta(n^3)$. □

Exercise 17. ***2.2-2***
Consider sorting n numbers stored in array $A[1:n]$ by repeatedly finding the smallest remaining element and swapping it into place (selection sort). Write pseudocode for this algorithm. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than all n? Give the worst-case running time of selection sort in Θ-notation. Is the best-case running time any better?

Proof. Solution.

Intuition: On pass i, scan the tail to find the minimum and swap it into position i. After pass i, the first i entries are the i smallest in order.

Rigorous (pseudocode):

```
SELECTION-SORT(A, n)
1  for i = 1 to n-1
2      minIdx = i
3      for j = i+1 to n
4          if A[j] < A[minIdx]
5              minIdx = j
6      swap A[i] and A[minIdx]
```

Loop invariant: At the start of each outer iteration i ($1 \le i \le n$), the prefix $A[1..i-1]$ consists of the $i-1$ smallest elements of A in nondecreasing order, and every element in $A[i..n]$ is at least $A[i-1]$.

Initialization: For $i = 1$, the prefix $A[1..0]$ is empty—trivially sorted and smallest.
Maintenance: The inner loop finds the minimum of $A[i..n]$ and swaps it into $A[i]$, so after the swap the invariant holds for $i+1$.
Termination: When $i = n$, the invariant implies $A[1..n-1]$ is the $n-1$ smallest in order, so the last element must be the largest; the array is sorted.
Why only to $n-1$: After placing the first $n-1$ elements, the nth is determined.
Running time: Comparisons are $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} \in \Theta(n^2)$; swaps are $n-1$. This holds in the worst and *also* the best case (the minimum search still scans the tail), so best-case time is not better: $\Theta(n^2)$. □

Exercise 18. ***2.2-3***
Consider linear search again (see Exercise 2.1-4). How many elements of the input array need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? Using Θ*-notation, give the average-case and worst-case running times of linear search. Justify your answers.*

Proof. Solution.

Intuition: If the target is equally likely to be anywhere, you'll find it roughly halfway through on average, but sometimes you might need to inspect all n elements.

Rigorous: Let X be the index where the key occurs; assume $\Pr[X = i] = 1/n$ for $i = 1, \ldots, n$. The algorithm checks exactly X elements. Thus

$$\mathbb{E}[X] = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n+1}{2}.$$

So the average number checked is $(n+1)/2$, giving average time $\Theta(n)$. In the worst case, the key is at position n (or absent, if allowed), requiring n checks: worst-case time $\Theta(n)$. □

Exercise 19. ***2.2-4***
How can you modify any sorting algorithm to have a good best-case running time?

Proof. Solution.

Intuition: Detect when the input is already sorted (or a pass makes no changes) and stop early.

Rigorous: Augment the algorithm with an $O(n)$ precheck (e.g., a single left-to-right scan verifying $A[i] \leq A[i+1]$ for all i) *or* an early-exit condition within passes (e.g., a "no-swap" flag in bubblesort, or halting insertion sort immediately when no shifts occur during a pass). If the input is already sorted, the algorithm returns after $O(n)$ work, so the *best-case* time becomes $\Theta(n)$, while the *worst-case* and *average-case* complexities remain unchanged. □

. . . . . .

2.3 Exercises.

Exercise 20. ***2.3-1***

Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence $\langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

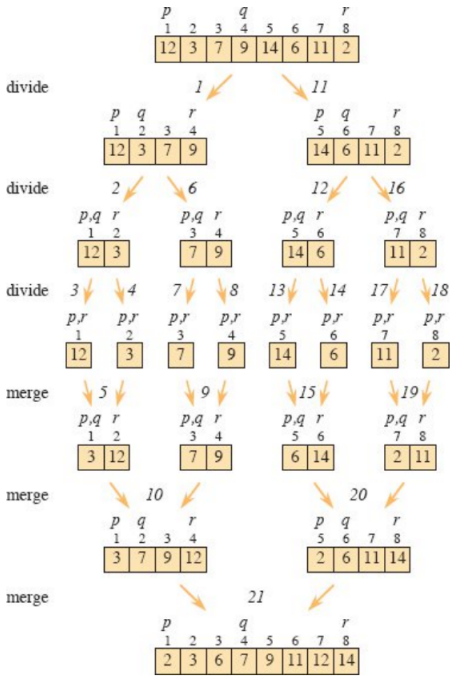

Figure 2.4 The operation of merge sort on the array A with length 8 that initially contains the sequence ⟨12, 3, 7, 9, 14, 6, 11, 2⟩. The indices p, q, and r into each subarray appear above their values. Numbers in italics indicate the order in which the MERGE-SORT and MERGE procedures are called following the initial call of MERGE-SORT(A, 1, 8).

Proof. Solution.

Intuition: Repeatedly split the array in half until singletons, then merge adjacent sorted subarrays back together.

Rigorous (trace matching Fig. 2.4):

$$A = [12, 3, 7, 9, 14, 6, 11, 2]$$

Divide to singletons:

$$[12, 3, 7, 9]\,[14, 6, 11, 2] \;\Rightarrow\; [12, 3]\,[7, 9]\,[14, 6]\,[11, 2] \;\Rightarrow\; [12]\,[3]\,[7]\,[9]\,[14]\,[6]\,[11]\,[2]$$

Merge pairs:

$$[12] \circ [3] \to [3, 12]\,, \quad [7] \circ [9] \to [7, 9]\,, \quad [14] \circ [6] \to [6, 14]\,, \quad [11] \circ [2] \to [2, 11]$$

Merge level-2:

$$[3, 12] \circ [7, 9] \to [3, 7, 9, 12]\,, \qquad [6, 14] \circ [2, 11] \to [2, 6, 11, 14]$$

Final merge:

$$[3, 7, 9, 12] \circ [2, 6, 11, 14] \;\to\; [2, 3, 6, 7, 9, 11, 12, 14].$$

By the merge invariant (see 2.3-3), each merge produces a sorted array from two sorted halves, hence the final array is sorted. □

Exercise 21. ***2.3-2***
The test in line 1 of MERGE-SORT *reads "if $p \geq r$" rather than "if $p \neq r$." If* MERGE-SORT *is called with $p > r$, then $A[p : r]$ is empty. Argue that as long as the initial call* MERGE-SORT$(A, 1, n)$ *has $n \geq 1$, the test "if $p \neq r$" suffices to ensure that no recursive call has $p > r$.*

Proof. Solution.

Intuition: We only ever split valid nonempty ranges in half; both children remain valid ranges with $p \leq r$.

Rigorous: Assume the initial call has $1 \leq p \leq r = n$. If $p \neq r$, the code sets $q = \lfloor (p+r)/2 \rfloor$ and recurses on $[p : q]$ and $[q+1 : r]$. Because $p \leq q < r$, both satisfy $p \leq r$ and are nonempty or singletons. Inductively, any call with $p \leq r$ generates children with $p \leq q \leq r - 1$ and $p + 1 \leq q + 1 \leq r$, hence $p \leq r$ persists and $p > r$ never occurs. Thus the condition "$p \neq r$" is sufficient. □

Exercise 22. ***2.3-3***
State a loop invariant for the `while` *loop of lines 12–18 of the* MERGE *procedure. Show how to use it, along with the* `while` *loops of lines 20–23 and 24–27, to prove that* MERGE *is correct.*

Proof. Solution.

Intuition: At each step, we place the smallest first element among the two input lists into the output; the prefix written so far is the globally smallest set of elements in sorted order.

Rigorous (main invariant): Let i, j index the current heads of the two sorted lists L and R, and let k index the output segment $A[p..r]$. *Invariant:* At the start of each iteration of lines 12–18,

$A[p..k-1]$ contains the $(k-p)$ smallest elements of $L[i..] \cup R[j..]$ in nondecreasing order,

and all elements in $A[p..k-1]$ are $\leq$ every element remaining in $L[i..]$ and $R[j..]$.
Initialization: $k = p$, so the prefix is empty and trivially sorted.
Maintenance: The loop compares $L[i]$ and $R[j]$, writes the smaller to $A[k]$, and advances the corresponding index. This preserves that the output prefix contains exactly the next smallest element and remains sorted.
Termination: The loop stops when either list is exhausted. The cleanup loops (20–23 and 24–27) copy the remainder of the nonempty list into A. Since all remaining elements are $\geq$ the last written element, the final array $A[p..r]$ is a stable, sorted merge of the two inputs. □

Exercise 23. ***2.3-4***
Use mathematical induction to show that when $n \geq 2$ is an exact power of 2, the solution of the recurrence $T(n) = 2T(n/2) + n$ is $T(n) = n \lg n$.

Proof. Solution.

Intuition: The recursion tree has $\lg n + 1$ levels with total work n per internal level; summing gives $n \lg n$ plus linear leaves.

Rigorous (induction): Let $n = 2^m$, $m \geq 1$. Claim: $T(2^m) = 2^m m$. *Base* $m = 1$ *($n = 2$):* $T(2) = 2T(1) + 2$. With $T(1) = 0$ (or any constant), $T(2) = 2$, which equals $2^1 \cdot 1$.
Step: Assume $T(2^{m-1}) = 2^{m-1}(m-1)$. Then

$$T(2^m) = 2T(2^{m-1}) + 2^m = 2 \cdot 2^{m-1}(m-1) + 2^m = 2^m(m-1) + 2^m = 2^m m.$$

Thus $T(n) = n \lg n$ for $n = 2^m$. □

Exercise 24. ***2.3-5***
Think of insertion sort as a recursive algorithm. To sort $A[1:n]$, recursively sort $A[1:n-1]$ and then insert $A[n]$. Write pseudocode for this recursive version. Give a recurrence for its worst-case running time.

Proof. Solution.

Intuition: Sort the first $n-1$ items, then bubble the last item left until it lands in place.

Rigorous (pseudocode & cost):

```
REC-INSERTION-SORT(A, n)
1  if n <= 1: return
2  REC-INSERTION-SORT(A, n-1)
3  key = A[n]; i = n-1
4  while i >= 1 and A[i] > key
5      A[i+1] = A[i]; i = i-1
6  A[i+1] = key
```

In the worst case, line 4 shifts through the entire sorted prefix, costing $\Theta(n)$ after a recursive call on $n-1$ elements. Hence

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

□

Exercise 25. ***2.3-6***
Referring to the searching problem, write pseudocode (iterative or recursive) for binary search. Argue that the worst-case running time is $\Theta(\lg n)$.

Proof. Solution.

Intuition: Repeatedly compare to the midpoint and discard half the remaining range each time.

Rigorous (iterative binary search):

```
BINARY-SEARCH(A, n, x)
1  lo = 1; hi = n
2  while lo <= hi
3      mid = floor((lo + hi)/2)
4      if A[mid] == x: return mid
5      else if A[mid] < x: lo = mid + 1
6      else: hi = mid - 1
7  return NIL
```

Each iteration halves the interval length, so after t iterations the size is $\leq n/2^t$. The loop stops after $t = \lceil \lg n \rceil$ iterations, giving worst-case time $\Theta(\lg n)$. □

Exercise 26. ***2.3-7***
What if insertion sort used a binary search (see 2.3-6) instead of a linear search to find where to insert? Would the overall worst-case running time improve to $\Theta(n \lg n)$*?*

Proof. Solution.

Intuition: Finding the insertion spot can be $O(\lg n)$, but you still have to *move* elements to make space, which costs $O(n)$ per pass.

Rigorous: At outer iteration j, binary search locates the position in $O(\lg j)$ time, but shifting the larger elements right costs $\Theta(j)$ in the worst case. Summing $\Theta(j)$ over $j = 1..n$ gives $\Theta(n^2)$, so the asymptotic worst-case running time does *not* improve. □

Exercise 27. ***2.3-8***
Describe an algorithm that, given a set S of n integers and another integer x, determines whether S contains two elements that sum to exactly x. Your algorithm should take $\Theta(n \lg n)$ time in the worst case.

Proof. Solution.

Intuition: Sort the numbers, then use two pointers that squeeze toward the middle: too small $\rightarrow$ move left pointer right; too large $\rightarrow$ move right pointer left.

Rigorous (algorithm & analysis):

1. Sort S in $\Theta(n \lg n)$ time to get $a_1 \leq \cdots \leq a_n$.
2. Set $i = 1$, $j = n$. While $i < j$:
   - If $a_i + a_j = x$, return TRUE.
   - If $a_i + a_j < x$, set $i \leftarrow i + 1$.
   - Else ($a_i + a_j > x$), set $j \leftarrow j - 1$.
3. If the loop finishes, return FALSE.

The two-pointer sweep performs at most $n - 1$ pointer moves, i.e. $O(n)$ work after sorting. Total worst-case time is $\Theta(n \lg n)$. □

. . . . . .

Chpt 2 Problems.

Exercise 28. ***2-1 Insertion sort on small arrays in merge sort***
Consider the hybrid: split the input into n/k sublists of length k, sort each sublist by Insertion-Sort*, then merge using the standard merging mechanism.*

(a) Show that sorting the n/k sublists takes $\Theta(nk)$ time.

(b) Show that merging them takes $\Theta\big(n \lg(n/k)\big)$ time.

(c) Given total $\Theta\big(nk + n \lg(n/k)\big)$, find the largest $k = k(n)$ such that the hybrid matches merge sort's $\Theta(n \lg n)$.

(d) How should k be chosen in practice?

Proof. Solution.

(a) Intuition: Each small array of length k costs $\Theta(k^2)$ with insertion sort. There are n/k of them.
Rigorous: $(n/k) \cdot \Theta(k^2) = \Theta(nk)$.

(b) Intuition: Merge in rounds; each round scans all n items, and there are $\lg(n/k)$ rounds.
Rigorous: A pairwise tournament of n/k runs has height $\lceil \lg(n/k) \rceil$; each level's total merge work is $\Theta(n)$. Hence $\Theta\big(n \lg(n/k)\big)$.

(c) Intuition: Keep the total on the order of $n \lg n$. The nk term must not dominate.
Rigorous: We need $nk = O(n \lg n)$, i.e. $k = O(\lg n)$. Taking $k = \Theta(\lg n)$ yields

$$nk + n \lg(n/k) = \Theta\big(n \lg n\big) + \Theta\big(n(\lg n - \lg \lg n)\big) = \Theta(n \lg n).$$

Thus the largest asymptotically valid choice is $k = \Theta(\lg n)$.

(d) Practice: Choose k empirically to exploit caches/branch prediction; small constants like $k \in [16, 64]$ are common on modern hardware. Benchmark on target inputs and fix k near the observed crossover. □

Exercise 29. ***2-2 Correctness of bubblesort***
BUBBLESORT*:*

```
BUBBLESORT(A, n)
1  for i = 1 to n-1
2      for j = n downto i+1
3          if A[j] < A[j-1]
4              exchange A[j] with A[j-1]
```

(a) Besides A' being sorted, what else must be shown to prove correctness?

(b) Give and prove a loop invariant for lines 2–4.

(c) Using (b), state a loop invariant for lines 1–4 that proves the algorithm sorts.

(d) What is the worst-case running time? Compare to Insertion-Sort*.*

Proof. Solution.

(a) That A' is a *permutation* of the input A (no elements lost/duplicated). Swaps preserve the multiset.

(b) Inner-loop invariant (fixed i): At the start of each iteration with index j ($i+1 \le j \le n$), the suffix $A[j+1..n]$ contains the $(n-j)$ *largest* elements of $A[i..n]$ in nondecreasing order. *Initialization:* Before the first inner iteration ($j = n$), the suffix is empty—trivially true. *Maintenance:* Comparing $A[j-1]$ and $A[j]$ and swapping if out of order ensures that after the iteration, the larger of the two resides in $A[j]$, keeping the largest elements packed at the right in order; decrementing j preserves the claim. *Termination:* When $j = i+1$, the suffix $A[i+2..n]$ holds the $(n-i-1)$ largest elements of $A[i..n]$; thus $A[i+1]$ is at least as large as any element to its right, and the minimum of $A[i..n]$ must be at $A[i]$ after one more comparison/swap step. Equivalently, by the usual bubble variant, after completing lines 2–4, $A[i]$ is the i-th smallest element.

(c) Outer-loop invariant: At the start of iteration i of lines 1–4, the prefix $A[1..i-1]$ contains the $(i-1)$ smallest elements in nondecreasing order. By (b), the inner pass places the next smallest at $A[i]$, preserving the invariant. When $i = n$, the whole array is sorted.

(d) Time: The inner loop performs $\sum_{i=1}^{n-1}(n-i) = \Theta(n^2)$ comparisons; in the worst case, a constant fraction cause swaps, so time is $\Theta(n^2)$. Insertion-Sort also has $\Theta(n^2)$ worst-case time, though it typically performs fewer moves and is faster in practice. □

Exercise 30. ***2-3 Correctness of Horner's rule***
Evaluate $P(x) = \sum_{i=0}^{n} a_i x^i$ *using* HORNER*:*

```
HORNER(A, n, x)
1  p = 0
2  for i = n downto 0
3      p = A[i] + x * p
4  return p
```

(a) Give the running time. (b) Write naive pseudocode and compare runtimes. (c) Prove Horner's correctness via a loop invariant.

Proof. Solution.

(a) Time. The loop executes $n+1$ iterations; each does $O(1)$ work, so $\Theta(n)$.
(b) Naive evaluation.

```
NAIVE-POLY(A, n, x)
1  p = 0
2  for i = 0 to n
3      pow = 1
4      for t = 1 to i
5          pow = pow * x
6      p = p + A[i] * pow
7  return p
```

The inner power loop costs $\Theta(i)$; summing gives $\sum_{i=0}^{n} \Theta(i) = \Theta(n^2)$. Thus Horner's $\Theta(n)$ is asymptotically faster.
(c) Correctness (loop invariant). At the start of each iteration with index i ($n \geq i \geq -1$),

$$p \;=\; \sum_{k=0}^{n-i-1} a_{i+1+k}\, x^k.$$

Initialization ($i = n$): The sum is empty, so $p = 0$ matches line 1.
Maintenance: Update $p \leftarrow a_i + x\,p$ transforms

$$p \;=\; a_i + x \sum_{k=0}^{n-i-1} a_{i+1+k} x^k \;=\; \sum_{k=0}^{n-i} a_{i+k} x^k,$$

which is the invariant with $i \leftarrow i - 1$.
Termination ($i = -1$): We obtain $p = \sum_{k=0}^{n} a_{0+k} x^k = P(x)$, as required. □

Exercise 31. ***2-4 Inversions***
Let $A[1:n]$ contain n distinct numbers. If $i < j$ and $A[i] > A[j]$, then (i, j) is an inversion.

(a) List the five inversions of $\langle 2, 3, 8, 6, 1 \rangle$.

(b) Which array on $\{1, \ldots, n\}$ has the most inversions? How many?

(c) Relate Insertion-Sort*'s running time to the number of inversions.*

(d) Give a $\Theta(n \lg n)$ algorithm to count inversions.

Proof. Solution.

(a) Inversions: $(1, 5), (2, 5), (3, 4), (3, 5), (4, 5)$.

(b) The reverse-sorted array $\langle n, n-1, \ldots, 1 \rangle$ maximizes inversions: each pair (i, j) with $i < j$ is inverted, giving $\binom{n}{2} = n(n-1)/2$.

(c) In Insertion-Sort, each swap of adjacent out-of-order elements reduces the inversion count by 1. Hence the number of element moves (and thus time beyond scanning) is $\Theta(\#\text{ inversions})$. Total running time is $\Theta(n + \#\text{ inversions})$, which is $\Theta(n^2)$ in the worst case.

(d) Counting in $\Theta(n \lg n)$. Modify merge sort: while merging two sorted halves L and R, whenever an element from R is copied before an element from L, it contributes $|L_{\text{remaining}}|$ inversions. Summing these counts over $O(\lg n)$ merge levels yields total time $\Theta(n \lg n)$ and the exact number of inversions. □