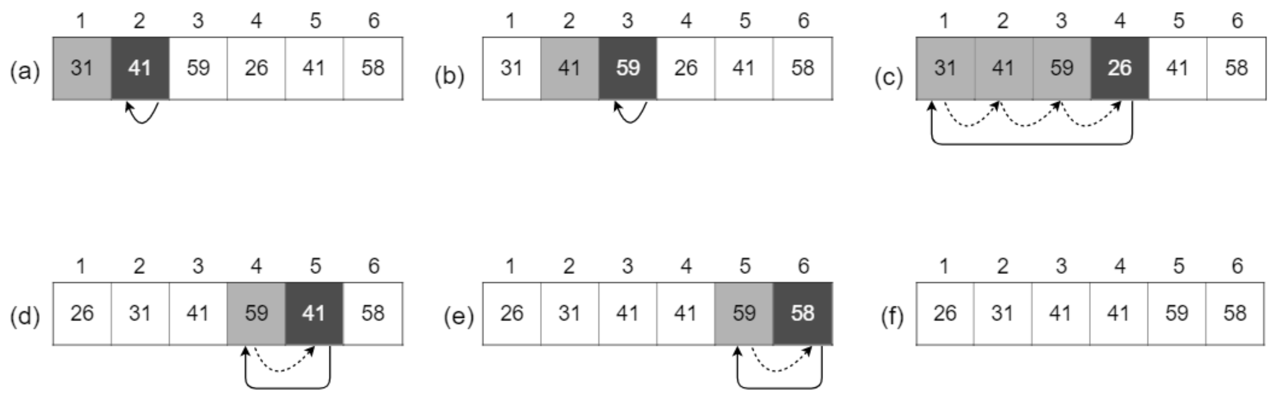
2 Getting Started

2.1 Insertion sort

2.1-1

> Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.



The operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles.

(a)-(e) are iterations of the for loop of lines 1-8.

In each iteration, the black rectangle holds the key taken from $A[i]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Dotted arrows show array values moved one position to the right in line 6. and solid arrows indicate where the key moves to in line 8.

(f) is the final sorted array.

The changes of array A during traversal:

$$\begin{aligned}
A &= \langle 31, 41, 59, 26, 41, 58 \rangle \\
A &= \langle 31, 41, 59, 26, 41, 58 \rangle \\
A &= \langle 31, 41, 59, 26, 41, 58 \rangle \\
A &= \langle 26, 31, 41, 59, 41, 58 \rangle \\
A &= \langle 26, 31, 41, 41, 59, 58 \rangle \\
A &= \langle 26, 31, 41, 41, 58, 59 \rangle
\end{aligned}$$

2.1-2

> Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of nondecreasing order.

```
INSERTION-SORT(A)
    for j = 2 to A.length
        key = A[j]
        i = j - 1
        while i > 0 and A[i] < key
            A[i + 1] = A[i]
            i = i - 1
        A[i + 1] = key
```

2.1-3

> Consider the ***searching problem***:
>
> **Input**: A sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a value v.
>
> **Output:** An index i such that $v = A[i]$ or the special value NIL if v does not appear in A.
>
> Write pseudocode for ***linear search***, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

```
LINEAR-SEARCH(A, v)
    for i = 1 to A.length
       if A[i] == v
            return i
    return NIL
```

Loop invariant: At the start of each iteration of the **for** loop, the subarray $A[1..i-1]$ consists of elements that are different than v.

Initialization: Before the first loop iteration ($i = 1$), the subarray is the empty array, so the proof is trivial.

Maintenance: During each loop iteration, we compare v with $A[i]$. If they are the same, we return i, which is the correct result. Otherwise, we continue to the next iteration of the loop. At the end of each loop iteration, we know the subarray $A[1..i]$ does not contain v, so the loop invariant holds true. Incrementing i for the next iteration of the **for** loop then preserves the loop invariant.

Termination: The loop terminates when $i > A.\text{length} = n$. Since i increases by 1, we must have $i = n + 1$ at that time. Substituting $n + 1$, for i in the wording of the loop invariant, we have that the subarray $A[1..n]$ consists of elements that are different than v. Thus, we return NIL. Observing that $A[1..n]$, we conclude that the entire array does not have any element equal to v. Hence the algorithm is correct.

2.1-4

> Consider the problem of adding two n-bit binary integers, stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in an $(n + 1)$-element array C. State the problem formally and write pseudocode for adding the two integers.

Input: An array of booleans $A = \langle a_1, a_2, \ldots, a_n \rangle$ and an array of booleans $B = \langle b_1, b_2, \ldots, b_n \rangle$, each representing an integer stored in binary format (each digit is a number, either 0 or 1, **least-significant digit first**) and each of length n.

Output: An array $C = \langle c_1, c_2, \ldots, c_{n+1} \rangle$ such that $C' = A' + B'$ where A', B' and C' are the integers, represented by A, B and C.

```
ADD-BINARY(A, B)
    carry = 0
    for i = 1 to A.length
        sum = A[i] + B[i] + carry
        C[i] = sum % 2  // remainder
        carry = sum / 2 // quotient
    C[A.length + 1] = carry
    return C
```

2.2 Analyzing algorithms

2.2-1

> Express the function $n^3/1000 - 100n^2 - 100n + 3n$ in terms of Θ-notation.

$\Theta(n^3)$.

2.2-2

> Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A, and exchange it with $A[2]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as ***selection sort***. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in Θ-notation.

- Pseudocode:

  ```
  n = A.length
  for i = 1 to n - 1
      minIndex = i
      for j = i + 1 to n
          if A[j] < A[minIndex]
              minIndex = j
      swap(A[i], A[minIndex])
  ```

- Loop invariant:

  At the start of the loop in line 1, the subarray $A[1..i-1]$ consists of the smallest $i-1$ elements in array A with sorted order.

- Why does it need to run for only the first $n-1$ elements, rather than for all n elements?

  After $n-1$ iterations, the subarray $A[1..n-1]$ consists of the smallest $i-1$ elements in array A with sorted order. Therefore, $A[n]$ is already the largest element.

- Running time: $\Theta(n^2)$.

2.2-3

> Consider linear search again (see Exercise 2.1-3). How many elements of the in- put sequence 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? What are the average-case and worst-case running times of linear search in Θ-notation? Justify your answers.

If the element is present in the sequence, half of the elements are likely to be checked before it is found in the average case. In the worst case, all of them will be checked. That is, $n/2$ checks for the average case and n for the worst case. Both of them are $\Theta(n)$.

2.2-4

> How can we modify almost any algorithm to have a good best-case running time?

You can modify any algorithm to have a best case time complexity by adding a special case. If the input matches this special case, return the pre-computed answer.

2.3 Designing algorithms

2.3-1

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

$$[3] \quad [41] \quad [52] \quad [26] \quad [38] \quad [57] \quad [9] \quad [49]$$

$$\downarrow$$

$$[3|41] \quad [26|52] \quad [38|57] \quad [9|49]$$

$$\downarrow$$

$$[3|26|41|52] \quad [9|38|49|57]$$

$$\downarrow$$

$$[3|9|26|38|41|49|52|57]$$

2.3-2

> Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A.

```
MERGE(A, p, q, r)
    n1 = q - p + 1
    n2 = r - q
    let L[1..n1] and R[1..n2] be new arrays
    for i = 1 to n1
        L[i] = A[p + i - 1]
    for j = 1 to n2
        R[j] = A[q + j]
    i = 1
    j = 1
    for k = p to r
        if i > n1
            A[k] = R[j]
            j = j + 1
        else if j > n2
            A[k] = L[i]
            i = i + 1
        else if L[i] ≤ R[j]
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            j = j + 1
```

2.3-3

> Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence
>
> $$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{for } k > 1 \end{cases}$$
>
> is $T(n) = n \lg n$.

- Base case

  For $n = 2^1$, $T(n) = 2 \lg 2 = 2$.

- Suppose $n = 2^k$, $T(n) = n \lg n = 2^k \lg 2^k = 2^k k$.

  For $n = 2^{k+1}$,

$$\begin{aligned} T(n) &= 2T(2^{k+1}/2) + 2^{k+1} \\ &= 2T(2^k) + 2^{k+1} \\ &= 2 \cdot 2^k k + 2^{k+1} \\ &= 2^{k+1}(k+1) \\ &= 2^{k+1} \lg 2^{k+1} \\ &= n \lg n. \end{aligned}$$

By P.M.I., $T(n) = n \lg n$, when n is an exact power of 2.

2.3-4

> We can express insertion sort as a recursive procedure as follows. In order to sort $A[1..n]$, we recursively sort $A[1..n-1]$ and then insert $A[n]$ into the sorted array $A[1..n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

It takes $\Theta(n)$ time in the worst case to insert $A[n]$ into the sorted array $A[1..n-1]$. Therefore, the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

The solution of the recurrence is $\Theta(n^2)$.

2.3-5

> Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration. The ***binary search*** algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

- Iterative:

```
ITERATIVE-BINARY-SEARCH(A, v, low, high)
    while low ≤ high
        mid = floor((low + high) / 2)
        if v == A[mid]
            return mid
        else if v > A[mid]
            low = mid + 1
        else high = mid - 1
    return NIL
```

- Recursive:

```
RECURSIVE-BINARY-SEARCH(A, v, low, high)
    if low > high
        return NIL
    mid = floor((low + high) / 2)
    if v == A[mid]
        return mid
    else if v > A[mid]
        return RECURSIVE-BINARY-SEARCH(A, v, mid + 1, high)
    else return RECURSIVE-BINARY-SEARCH(A, v, low, mid - 1)
```

Each time we do the comparison of v with the middle element, the search range continues with range halved.

The recurrence

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

The solution of the recurrence is $T(n) = \Theta(\lg n)$.

2.3-6

> Observe that the **while** loop of lines 5–7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[i..j-1]$. Can we use a binary search (see Exercise 2.3-

> 5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

Each time the **while** loop of lines 5-7 of INSERTION-SORT scans backward through the sorted array $A[1..j-1]$. The loop not only searches for the proper place for $A[j]$, but it also moves each of the array elements that are bigger than $A[j]$ one position to the right (line 6). These movements takes $\Theta(j)$ time, which occurs when all the $j-1$ elements preceding $A[j]$ are larger than $A[j]$. The running time of using binary search to search is $\Theta(\lg j)$, which is still dominated by the running time of moving element $\Theta(j)$.

Therefore, we can't improve the overrall worst-case running time of insertion sort to $\Theta(n \lg n)$.

2.3-7 ⋆

> Describe a $\Theta(n \lg n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

First, sort S, which takes $\Theta(n \lg n)$. Then, for each element s_i in S, $i = 1, \dots, n$, search $A[i+1..n]$ for $s'_i = x - s_i$ by binary search, which takes $\Theta(\lg n)$.

- If s'_i is found, return its position;
- otherwise, continue for next iteration.

The time complexity of the algorithm is $\Theta(n \lg n) + n \cdot \Theta(\lg n) = \Theta(n \lg n)$.

Problem 2-1 Insertion sort on small arrays in merge sort

> Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to ***coarsen*** the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.
>
> **a.** Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.
>
> **b.** Show how to merge the sublists in $\Theta(n \lg(n/k))$ worst-case time.
>
> **c.** Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?
>
> **d.** How should we choose k in practice?

a. The worst-case time to sort a list of length k by insertion sort is $\Theta(k^2)$. Therefore, sorting n/k sublists, each of length k takes $\Theta(k^2 \cdot n/k) = \Theta(nk)$ worst-case time.

b. We have n/k sorted sublists each of length k. To merge these n/k sorted sublists to a single sorted list of length n, we have to take 2 sublists at a time and continue to merge them. This will result in $\lg(n/k)$ steps and we compare n elements in each step. Therefore, the worst-case time to merge the sublists is $\Theta(n \lg(n/k))$.

c. The modified algorithm has time complexity as standard merge sort when $\Theta(nk + n \lg(n/k)) = \Theta(n \lg n)$. Assume $k = \Theta(\lg n)$,

$$\begin{aligned}
\Theta(nk + n \lg(n/k)) &= \Theta(nk + n \lg n - n \lg k) \\
&= \Theta(n \lg n + n \lg n - n \lg(\lg n)) \\
&= \Theta(2n \lg n - n \lg(\lg n)) \\
&= \Theta(n \lg n).
\end{aligned}$$

d. Choose k be the largest length of sublist on which insertion sort is faster than merge sort.

Problem 2-2 Correctness of bubblesort

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

```
BUBBLESORT(A)
    for i = 1 to A.length - 1
        for j = A.length downto i + 1
            if A[j] < A[j - 1]
                exchange A[j] with A[j - 1]
```

a. Let A' denote the output of $\text{BUBBLESORT}(A)$ To prove that BUBBLESORT is correct, we need to prove that it terminates and that

$$A'[1] \le A'[2] \le \cdots \le A'[n], \tag{2.3}$$

where $n = A.\text{length}$. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

The next two parts will prove inequality (2.3).

b. State precisely a loop invariant for the **for** loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

c. Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the **for** loop in lines 1–4 that will allow you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.

d. What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

a. A' consists of the elements in A but in sorted order.

b. Loop invariant: At the start of each iteration of the **for** loop of lines 2-4, the subarray $A[j..n]$ consists of the elements originally in $A[j..n]$ before entering the loop but possibly in a different order and the first element $A[j]$ is the smallest among them.

Initialization: Initially the subarray contains only the last element $A[n]$, which is trivially the smallest element of the subarray.

Maintenance: In every step we compare $A[j]$ with $A[j-1]$ and make $A[j-1]$ the smallest among them. After the iteration, the length of the subarray increases by one and the first element is the smallest of the subarray.

Termination: The loop terminates when $j = i$. According to the statement of loop invariant, $A[i]$ is the smallest among $A[i..n]$ and $A[i..n]$ consists of the elements originally in $A[i..n]$ before entering the loop.

c. Loop invariant: At the start of each iteration of the **for** loop of lines 1-4, the subarray $A[1..i-1]$ consists of the $i-1$ smallest elements in $A[1..n]$ in sorted order. $A[i..n]$ consists of the $n-i+1$ remaining elements in $A[1..n]$.

Initialization: Initially the subarray $A[1..i-1]$ is empty and trivially this is the smallest element of the subarray.

Maintenance: From part (b), after the execution of the inner loop, $A[i]$ will be the smallest element of the subarray $A[i..n]$. And in the beginning of the outer loop, $A[1..i-1]$ consists of elements that are smaller than the elements of $A[i..n]$, in sorted order. So, after the execution of the outer loop, subarray $A[1..i]$ will consists of elements that are smaller than the elements of $A[i+1..n]$, in sorted order.

Termination: The loop terminates when $i = A.\text{length}$. At that point the array $A[1..n]$ will consists of all elements in sorted order.

d. The ith iteration of the **for** loop of lines 1-4 will cause $n-i$ iterations of the **for** loop of lines 2-4, each with constant time execution, so the worst-case running time of bubble sort is $\Theta(n^2)$ which is same as the worst-case running time of insertion sort.

Problem 2-3 Correctness of Horner's rule

The following code fragment implements Horner's rule for evaluating a polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n)\cdots)), \end{aligned}$$

given the coefficients $a_0, a_1, \ldots, a_n$ and a value of x:

```
y = 0
for i = n downto 0
    y = a[i] + x * y
```

a. In terms of Θ-notation, what is the running time of this code fragment for Horner's rule?

b. Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule

c. Consider the following loop invariant: At the start of each iteration of the **for** loop of lines 2-3,

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k.$$

Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to show that, at termination, $y = \sum_{k=0}^{n} a_k x^k$.

d. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$.

a. $\Theta(n)$.

b.

```
NAIVE-HORNER()
    y = 0
    for k = 0 to n
        temp = 1
        for i = 1 to k
            temp = temp * x
        y = y + a[k] * temp
```

The running time is $\Theta(n^2)$, because of the nested loop. It is obviously slower.

c. Initialization: It is pretty trivial, since the summation has no terms which implies $y = 0$.

Maintenance: By using the loop invariant, in the end of the i-th iteration, we have

$$\begin{aligned} y &= a_i + x \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k \\ &= a_i x^0 + \sum_{k=0}^{n-i-1} a_{k+i+1} x^{k+1} \\ &= a_i x^0 + \sum_{k=1}^{n-i} a_{k+i} x^k \\ &= \sum_{k=0}^{n-i} a_{k+i} x^k. \end{aligned}$$

Termination: The loop terminates at $i = -1$. If we substitute,

$$y = \sum_{k=0}^{n-i-1} a_{k+i+1} x^k = \sum_{k=0}^{n} a_k x^k.$$

d. The invariant of the loop is a sum that equals a polynomial with the given coefficients.

Problem 2-4 Inversions

> Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A.
>
> **a.** List the five inversions in the array $\langle 2, 3, 8, 6, 1 \rangle$.
>
> **b.** What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?
>
> **c.** What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.
>
> **d.** Give an algorithm that determines the number of inversions in any permutation of n elements in $\Theta(n \lg n)$ worst-case time. (Hint: Modify merge sort).

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

b. The array $\langle n, n-1, \ldots, 1 \rangle$ has the most inversions $(n-1) + (n-2) + \cdots + 1 = n(n-1)/2$.

c. The running time of insertion sort is a constant times the number of inversions. Let $I(i)$ denote the number of $j < i$ such that $A[j] > A[i]$. Then $\sum_{i=1}^{n} I(i)$ equals the number of inversions in A.

Now consider the **while** loop on lines 5-7 of the insertion sort algorithm. The loop will execute once for each element of A which has index less than j is larger than $A[j]$. Thus, it will execute $I(j)$ times. We reach this **while** loop once for each iteration of the **for** loop, so the number of constant time steps of insertion sort is $\sum_{j=1}^{n} I(j)$ which is exactly the inversion number of A.

d. We'll call our algorithm COUNT-INVERSIONS for modified merge sort. In addition to sorting A, it will also keep track of the number of inversions.

$\text{COUNT-INVERSIONS}(A, p, r)$ sorts $A[p..r]$ and returns the number of inversions in the elements of $A[p..r]$, so $left$ and $right$ track the number of inversions of the form (i, j) where i and j are both in the same half of A.

$\text{MERGE-INVERSIONS}(A, p, q, r)$ returns the number of inversions of the form (i, j) where i is in the first half of the array and j is in the second half. Summing these up gives the total number of inversions in A. The runtime of the modified algorithm is $\Theta(n \lg n)$, which is same as merge sort since we only add an additional constant-time operation to some of the iterations in some of the loops.

```
COUNT-INVERSIONS(A, p, r)
    if p < r
        q = floor((p + r) / 2)
        left = COUNT-INVERSIONS(A, p, q)
        right = COUNT-INVERSIONS(A, q + 1, r)
        inversions = MERGE-INVERSIONS(A, p, q, r) + left + right
        return inversions
```

```
MERGE-INVERSIONS(A, p, q, r)
    n1 = q - p + 1
    n2 = r - q
    let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
    for i = 1 to n1
        L[i] = A[p + i - 1]
    for j = 1 to n2
        R[j] = A[q + j]
    L[n1 + 1] = ∞
    R[n2 + 1] = ∞
    i = 1
    j = 1
    inversions = 0
    for k = p to r
        if L[i] ≤ R[j]
            A[k] = L[i]
            i = i + 1
        else
            inversions = inversions + n1 - i + 1
            A[k] = R[j]
            j = j + 1
    return inversions
```