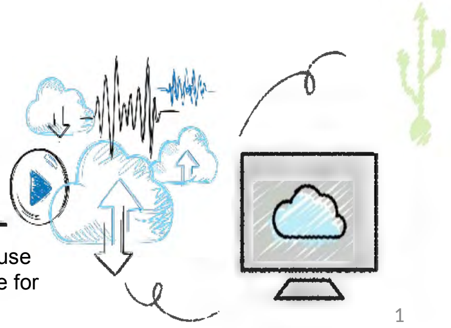
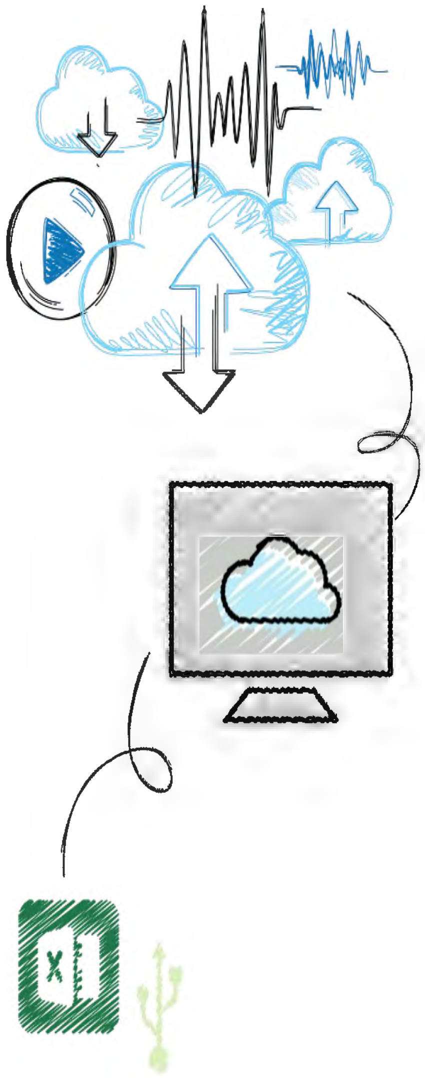
Fundamentals of Python: First Programs Second Edition

Chapter 11

Searching, Sorting, and Complexity Analysis



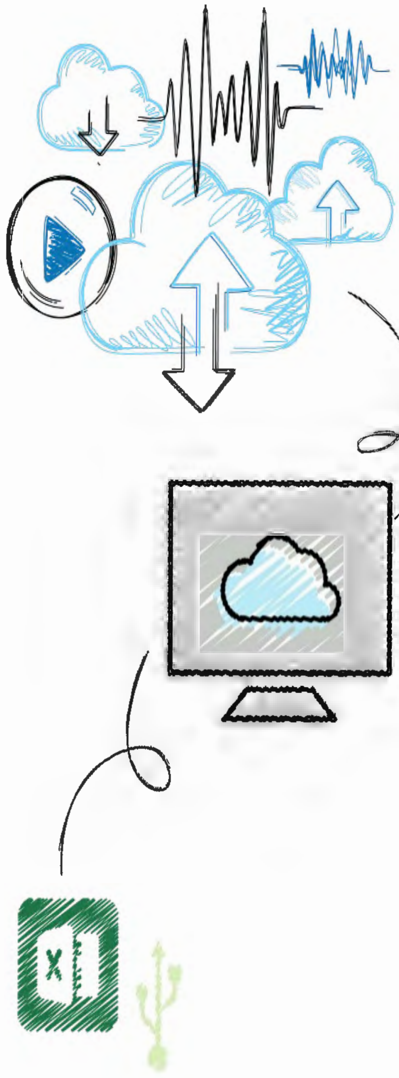
CENGAGE



Objectives (1 of 2)

11.1 Measure the performance of an algorithm by obtaining running times and instruction counts with different data sets

11.2 Analyze an algorithm’s performance by determining its order of complexity, using big-O notation



Objectives (2 of 2)

11.3 Distinguish the common orders of complexity and the algorithmic patterns that exhibit them

11.4 Distinguish between the improvements obtained by tweaking an algorithm and reducing its order of complexity

11.5 Design, implement, and analyze search and sort algorithms

Measuring the Efficiency of Algorithms

- When choosing algorithms, we often have to settle for a space/time tradeoff
  - An algorithm can be designed to gain faster run times at the cost of using extra space (memory), or the other way around
- Memory is now quite inexpensive for desktop and laptop computers, but not yet for miniature devices

Measuring the Run Time of an Algorithm (1 of 5)

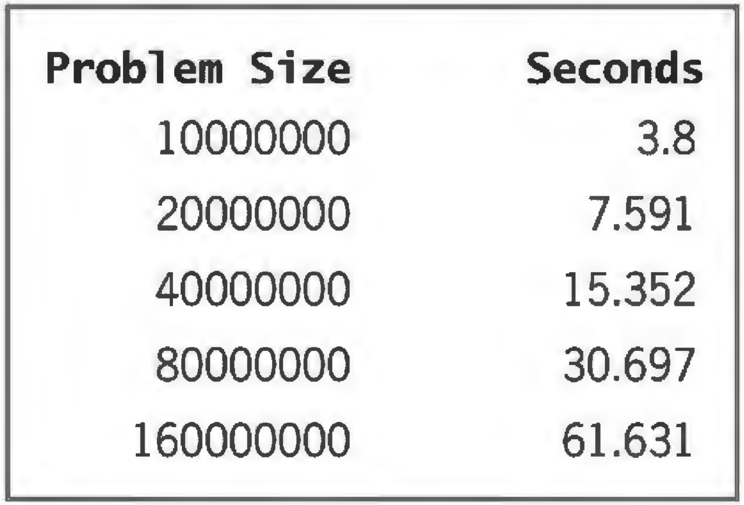
- One way to measure the time cost of an algorithm is to use computer's clock to obtain actual run time
  - **Benchmarking** or **profiling**
- Can use **time()** in **time** module
  - Returns number of seconds that have elapsed between current time on the computer's clock and January 1, 1970

Measuring the Run Time of an Algorithm (2 of 5)

```
"""
File: timing1.py
Prints the running times for problem sizes that double,
using a single loop.
"""
import time

problemSize = 10000000
print("%12s16s" % (“Problem Size”, “Seconds”))
for count in range(5):
    start = time.time()
    # The start of the algorithm
    work = 1
    for x in range(problemSize):
        work += 1
        work −= 1
    # The end of the algorithm
    elapsed = time.time() − start
    print("%12d%16.3f" % (problemSize, elapsed))
    problemSize *= 2
```

Measuring the Run Time of an Algorithm (3 of 5)

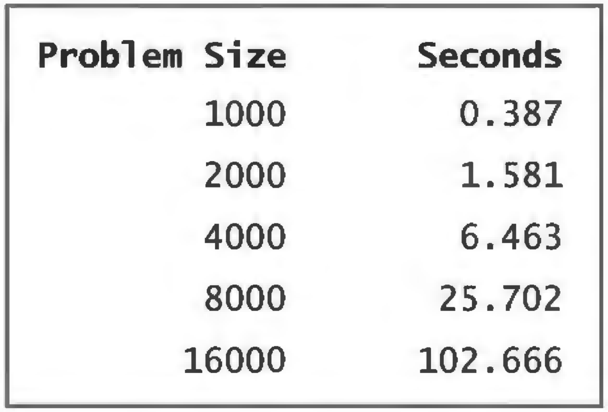


```
Problem Size      Seconds
    10000000          3.8
    20000000        7.591
    40000000       15.352
    80000000       30.697
   160000000       61.631
```

Figure 11-1 The output of the tester program

Measuring the Run Time of an Algorithm (4 of 5)

```
for j in range(problemSize):
    for k in range(problemSize):
        work += 1
        work −= 1
```



```
Problem Size        Seconds
        1000          0.387
        2000          1.581
        4000          6.463
        8000         25.702
       16000        102.666
```

Figure 11-2 The output of the second tester program with a nested loop and initial problem size of 1000

Measuring the Run Time of an Algorithm (5 of 5)

- This method permits accurate predictions of the running times of many algorithms
- Problems:
  - Different hardware platforms have different processing speeds, so the running times of an algorithm differ from machine to machine
    - Running time varies with OS and programming language too
  - It is impractical to determine the running time for some algorithms with very large data sets

Counting Instructions (1 of 5)

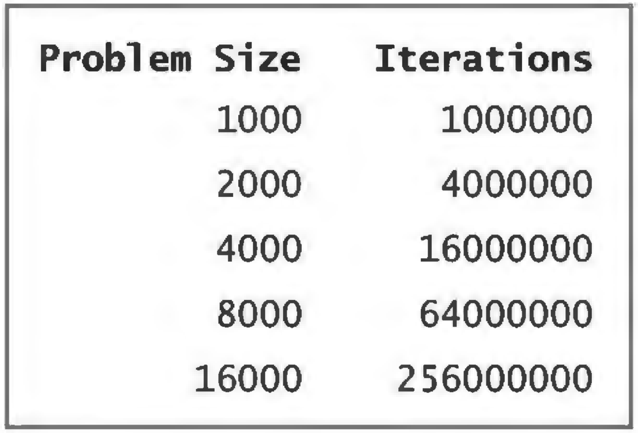
- Another technique is to count the instructions executed with different problem sizes
  - We count the instructions in the high-level code in which the algorithm is written, not instructions in the executable machine language program
- Distinguish between:
  - Instructions that execute the same number of times regardless of problem size
    - For now, we ignore instructions in this class
  - Instructions whose execution count varies with problem size

Counting Instructions (2 of 5)

```
"""
File: counting.py
Prints the number of iterations for problem sizes
that double, using a nested loop.
"""
problemSize = 1000
print("%12s%15s" % ("Problem Size", "Iterations"))
for count in range(5):
    number = 0
    # The start of the algorithm
    work = 1
    for j in range(problemSize):
        for k in range(problemSize):
            number += 1
            work += 1
            work -= 1
    # The end of the algorithm
    print("%12d%15d" % (problemSize, number))
    problemSize *= 2
```

CENGAGE

Counting Instructions (3 of 5)



```
Problem Size     Iterations
        1000        1000000
        2000        4000000
        4000       16000000
        8000       64000000
       16000      256000000
```

Figure 11-3 The output of a tester program that counts iterations

Counting Instructions (4 of 5)

```
"""
File: countfib.py
Prints the number of calls of a recursive Fibonacci
function with problem sizes that double.
"""
from counter import Counter
def fib(n, counter = None):
   """Count the number of calls of the Fibonacci function."""
   if counter: counter.increment()
   if n < 3:
        return 1
   else:
        return fib(n – 1, counter) + fib(n - 2, counter)
problemSize = 2
print("%12s%15s" % ("Problem Size", "Calls"))
for count in range(5):
   counter = Counter()
   # The start of the algorithm
   fib(problemSize, counter)
   # The end of the algorithm
   print("%12d%15s" % (problemSize, counter))
   problemSize  *= 2
```

Counting Instructions (5 of 5)

```
Problem Size              Calls
           2                  1
           4                  5
           8                 41
          16               1973
          32            4356617
```

Figure 11-4 The output of a tester program that runs the Fibonacci function

Complexity Analysis

- Complexity Analysis
  - Reading the algorithm and using pencil and paper to work out some simple algorithms

Orders of Complexity (1 of 4)

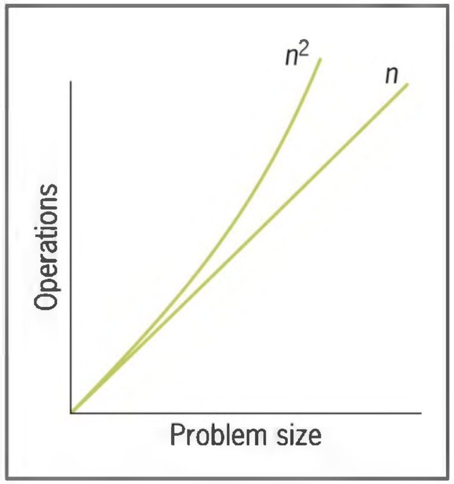


Figure 11-5 A graph of the amounts of work done in the tester programs

- The performances of these algorithms differ by what we call an **order of complexity**
  - The performance of the first algorithm is **linear** in that its work grows in direct proportion to the size of the problem
  - The behavior of the second algorithm is **quadratic** in that its work grows as a function of the square of the problem size

Orders of Complexity (2 of 4)

- An algorithm has **constant** performance if it requires the same number of operations for any problem size
  - List indexing is a good example of a constant-time algorithm
- Another order of complexity that is better than linear but worse than constant is called **logarithmic**
  - The amount of work of a logarithmic algorithm is proportional to the $\log_2$ of the problem size
- The work of a **polynomial time algorithm** grows at a rate of n^k Where k is a constant greater than 1
- An order of complexity that is worse than polynomial is called **exponential**
  - An example rate of growth of this order is 2^n

Orders of Complexity (3 of 4)

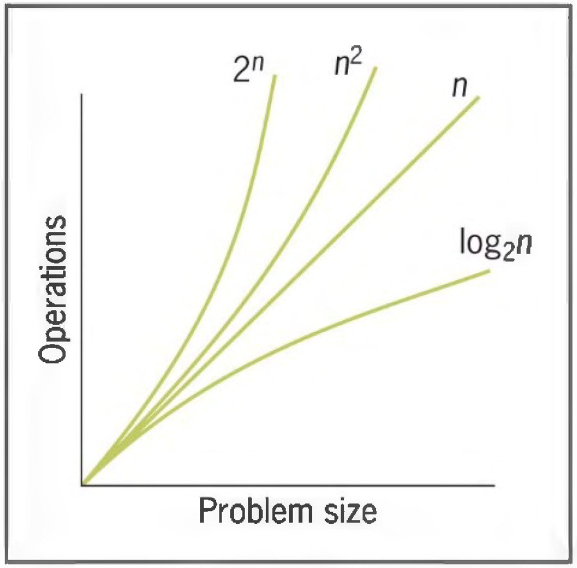


Figure 11-6 A graph of some sample orders of complexity

Orders of Complexity (4 of 4)

n	Logarithmic ($\log_2 n$)	Linear (n)	Quadratic (n^2)	Exponential (2^n)
100	7	100	10,000	Off the charts
1,000	10	1000	1,000,000	Off the charts
1,000,000	20	1,000,000	1,000,000,000,000	Really off the charts

Big-O Notation

- The amount of work in an algorithm typically is the sum of several terms in a polynomial
  - We focus on one term as **dominant**
- As **n** becomes large, the dominant term becomes so large that the amount of work represented by the other terms can be ignored
  - **Asymptotic analysis**
- **Big-O notation:** used to express the efficiency or computational complexity of an algorithm

The Role of the Constant of Proportionality

- The **constant of proportionality** involves the terms and coefficients that are usually ignored during big-O analysis
- Determine the constant of proportionality for the first algorithm discussed in this chapter. Here is the code:

```
work = 1
for x in range(problemSize):
    work += 1
    work −= 1
```

Measuring the Memory Used by an Algorithm

- A complete analysis of the resources used by an algorithm includes the amount of memory required
- We focus on rates of potential growth
- Some algorithms require the same amount of memory to solve any problem
- Other algorithms require more memory as the problem size gets larger

Search Algorithms

- We now present several algorithms that can be used for searching and sorting lists
  - We first discuss the design of an algorithm,
  - We then show its implementation as a Python function, and,
  - Finally, we provide an analysis of the algorithm’s computational complexity
- To keep things simple, each function processes a list of integers

Search for a Minimum

- Python’s **min** function returns the minimum or smallest item in a list
- Alternative version:

```
def ourMin(lyst):
"""Returns the position of the minimum item."""
    minpos = 0
    current = 1
    while current < len(lyst):
        if lyst[current] < lyst[minpos]:
           minpos = current
        current += 1
return minpos
```

n – 1 comparisons for a list of size **n**

- O(**n**)

Sequential Search of a List

- Python's **in** operator is implemented as a method named **__contains__** in the **list** class
  - Uses a **sequential search** or a **linear search**
- Python code for a linear search function:

```
def sequentialSearch(target, lyst):
    """Returns the position of the target item if found,
    or −1 otherwise."""
    position = 0
    while position < len(lyst):
        if target == lyst[position]:
            return position
        position += 1
    return −1
```

Best-Case, Worst-Case, and Average-Case Performance

- Analysis of a linear search considers three cases:
  - In the worst case, the target item is at the end of the list or not in the list at all
    - O(**n**)
  - In the best case, the algorithm finds the target at the first position, after making one iteration
    - O(1)
  - Average case: add number of iterations required to find target at each possible position; divide sum by **n**
    - O(**n**)

Binary Search of a List (1 of 2)

- A linear search is necessary for data that are not arranged in any particular order
- When searching sorted data, use a binary search

```
def binarySearch(target, lyst):
    """"Returns the position of the target item if found,
    or −1 otherwise."""
    left = 0
    right = len(lyst) – 1
    while left <= right:
            midpoint = (left + right) // 2
            if target == lyst[midpoint]:
                    return midpoint
            elif target < lyst[midpoint]:
                    right = midpoint – 1 # Search to left
            else:
                    left = midpoint + 1 # Search to right
    return −1
```

Binary Search of a List (2 of 2)

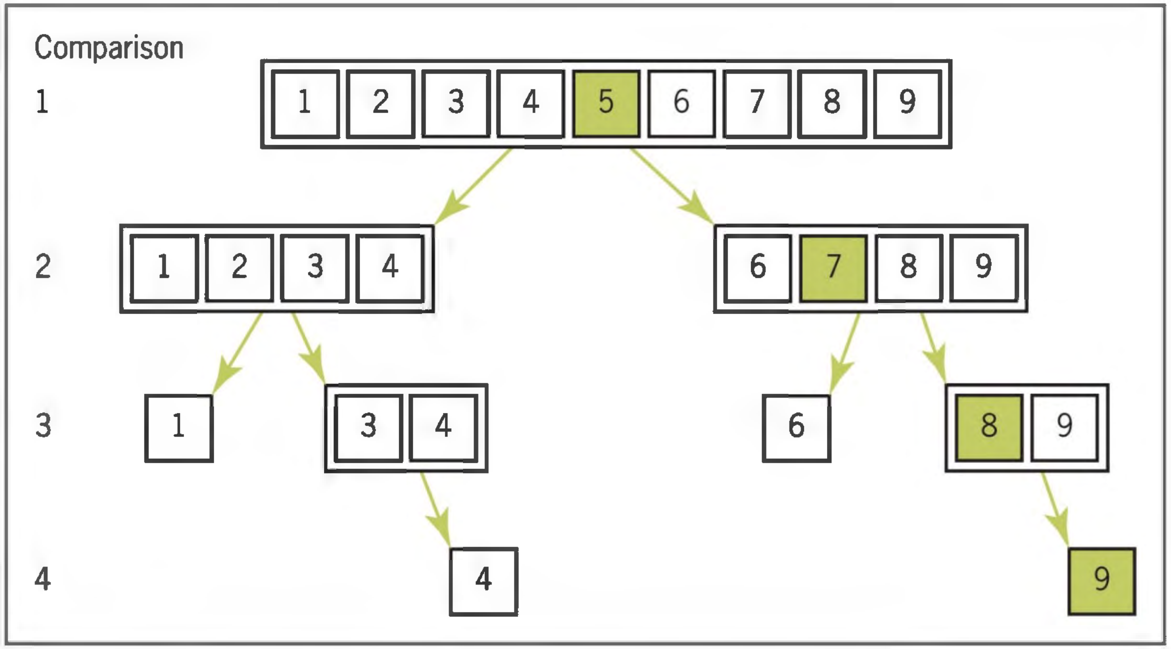


Figure 11-7 The items of a list visited during a binary search for 10

Basic Sort Algorithms

- The sort functions that we develop here operate on a list of integers and uses a **swap** function to exchange the positions of two items in the list

```
def swap(lyst, i, j):
    """Exchanges the items at positions i and j."""
    # You could say lyst[i], lyst[j] = lyst[j], lyst[i]
    # but the following code shows what is really going on
    temp = lyst[i]
    lyst[i] = lyst[j]
    lyst[j] = temp
```

Selection Sort (1 of 2)

- Perhaps the simplest strategy is to search the entire list for the position of the smallest item
    - If that position does not equal the first position, the algorithm swaps the items at those positions

Unsorted List	After 1st Pass	After 2nd Pass	After 3rd Pass	After 4th Pass
5	1*	1	1	1
3	3	2*	2	2
1	5*	5	3*	3
2	2	3*	5*	4*
4	4	4	4	5*

Selection Sort (2 of 2)

```
def selectionSort(lyst):
"""Sorts the items in lyst in ascending order."""
    i = 0
    while i < len(lyst) − 1: # Do n – 1 searches
        minIndex = i        #for the smallest item
        j = i + 1
       while j < len(lyst): # Start a search
            if lyst[j] < lyst[minIndex]:
                   minIndex = j
            j += 1
       if minIndex != i: # Swap if necessary
            swap(lyst, minIndex, i)
        i += 1
```

Bubble Sort (1 of 2)

- Starts at beginning of list and compares pairs of data items as it moves down to the end
  - When items in pair are out of order, swap them

Unsorted List	After 1st comparison	After 2nd comparison	After 3rd comparison	After 4th comparison
5	4*	4	4	4
4	5*	2*	2	2
2	2	5*	1*	1
1	1	1	5*	3*
3	3	3	3	5*

Bubble Sort (2 of 2)

- The Python function for a bubble sort:

```
def bubbleSort(lyst):
    """Sorts the items in lyst in ascending order."""
    n = len(lyst)
    while n > 1: # Do n − 1 bubbles
        i = 1 # Start each bubble
        while i < n:
            if lyst[i] < lyst[i − 1]: # Exchange if needed
                swap(lyst, i, i − 1)
            i += 1
        n −= 1
```

Insertion Sort (1 of 2)

- Code for the **InsertionSort** function:

```
def insertionSort(lyst):
    """Sorts the items in lyst in ascending order."""
    i = 1
    while i < len(lyst):
        itemToInsert = lyst[i]
        j = i – 1
        while j >= 0:
            if itemToInsert < lyst[j]:
                lyst[j + 1] = lyst[j]
                j –= 1
            else:
                break
        lyst[j + 1] = itemToInsert
        i += 1
```

Insertion Sort (2 of 2)

- The more items in the list that are in order, the better insertion sort gets until, in the best case of a sorted list, the sort's behavior is linear
- In the average case, insertion sort is still quadratic

Unsorted List	After 1st Pass	After 2nd Pass	After 3rd Pass	After 4th Pass
2	2	1*	1	1
5 ←	5 (no insertion)	2	2	2
1	1 ←	5	4*	3*
4	4	4 ←	5	4
3	3	3	3 ←	5

Best-Case, Worst-Case, and Average-Case Performance Revisited

- Thorough analysis of an algorithm’s complexity divides its behavior into three types of cases:
  - Best case
  - Worst case
  - Average case
- There are algorithms whose best-case and average-case performances are similar, but whose performance can degrade to a worst case
- When choosing/developing an algorithm, it is important to be aware of these distinctions

Faster Sorting

n	n Log n	n^2
512	4,608	262,144
1,024	10,240	1,048,576
2,048	22,458	4,194,304
8,192	106,496	67,108,864
16,384	229,376	268,435,456
32,768	491,520	1,073,741,824

Quicksort (1 of 4)

- An outline of the strategy used in the **quicksort** algorithm:
  - 1. Begin by selecting the item at the list’s midpoint. We call this item the pivot. (
  - 2. Partition items in the list so that all items less than the pivot are moved to the left of the pivot, and the rest are moved to its right.
  - 3. Divide and conquer. Reapply the process recursively to the sublists formed by splitting the list at the pivot.
  - 4. The process terminates each time it encounters a sublist with fewer than two items.

CENGAGE

Quicksort (2 of 4)

- **Partitioning**
  - 1. Swap the pivot with the last item in the sublist.
  - 2. Establish a boundary between the items known to be less than the pivot and the rest of the items
  - 3. Starting with the first item in the sublist, scan across the sublist. Every time an item less than the pivot is encountered, swap it with the first item after the boundary and advance the boundary.
  - 4. Finish by swapping the pivot with the first item after the boundary.

Quicksort (3 of 4)

- **Complexity Analysis of Quicksort**

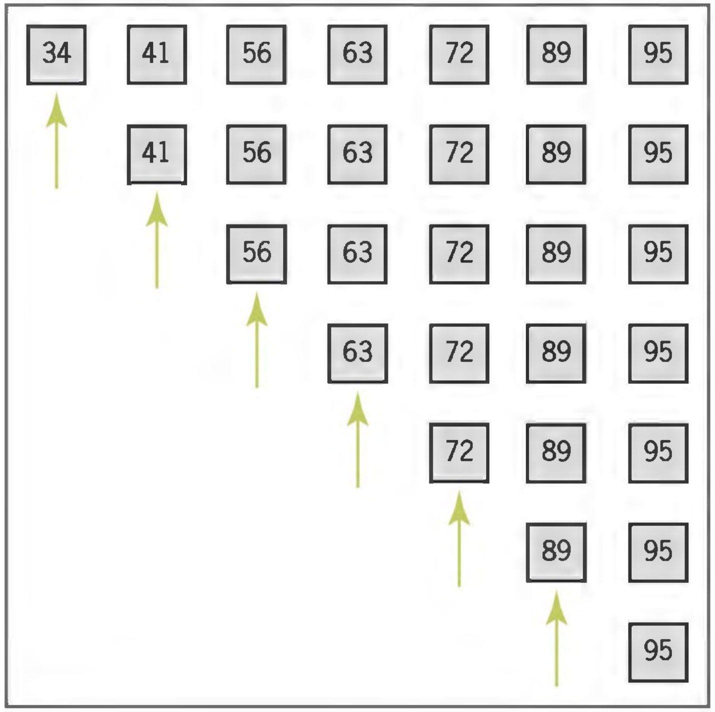


Figure 11-8 A worst-case scenario for quicksort (arrows indicate pivot elements)

Quicksort (4 of 4)

- **Implementation of Quicksort**
  - The quicksort algorithm is most easily coded using a recursive approach

Merge Sort (1 of 5)

- Informal summary of the algorithm
  - Compute the middle position of a list and recursively sort its left and right sublists (divide and conquer).
  - Merge the two sorted sublists back into a single sorted list.
  - Stop the process when sublists can no longer be subdivided.
- Three Python functions collaborate in this top-level design strategy:
  - **mergeSort**—The function called by users.
  - **mergeSortHelper**—A helper function that hides the extra parameters required by recursive calls.
  - **merge**—A function that implements the merging process.

Merge Sort (2 of 5)

- **Implementing the Merging Process**
  - The merging process uses a temporary list of the same size as the list being sorted
    - We call this list the **copyBuffer**
  - To avoid the overhead of allocating and deallocating the **copyBuffer** each time **merge** is called, the buffer is allocated once in **mergeSort** and passed as an argument to **mergeSortHelper** and **merge**

Merge Sort (3 of 5)

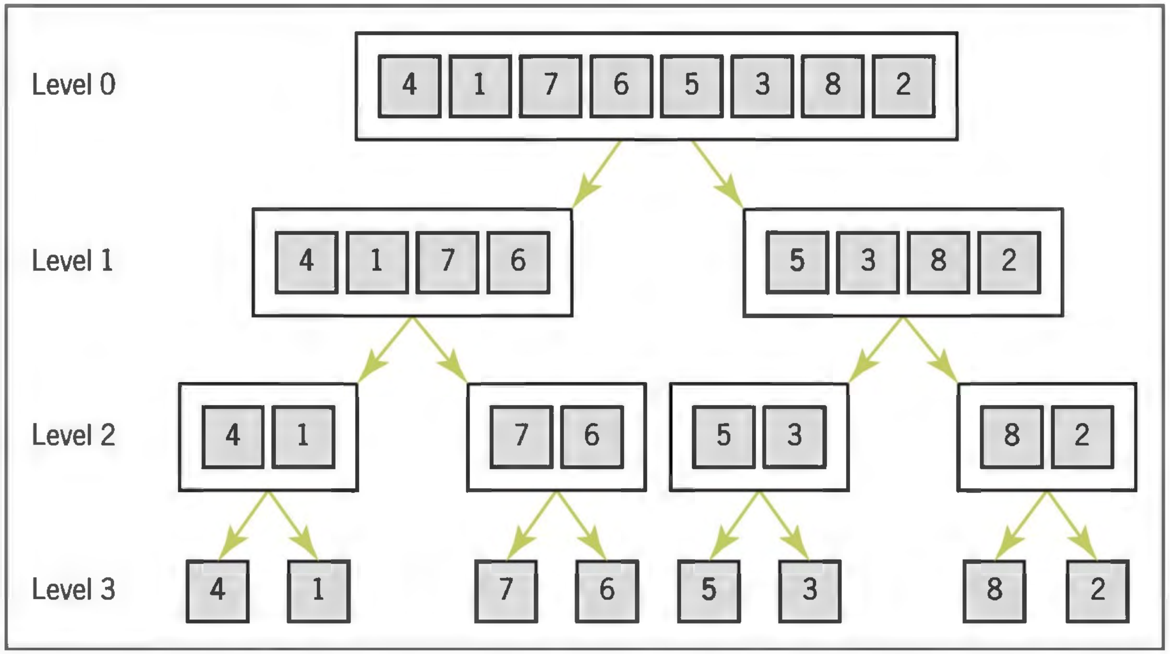


Figure 11-9 Sublists generated during calls of `mergeSortHelper`

Merge Sort (4 of 5)

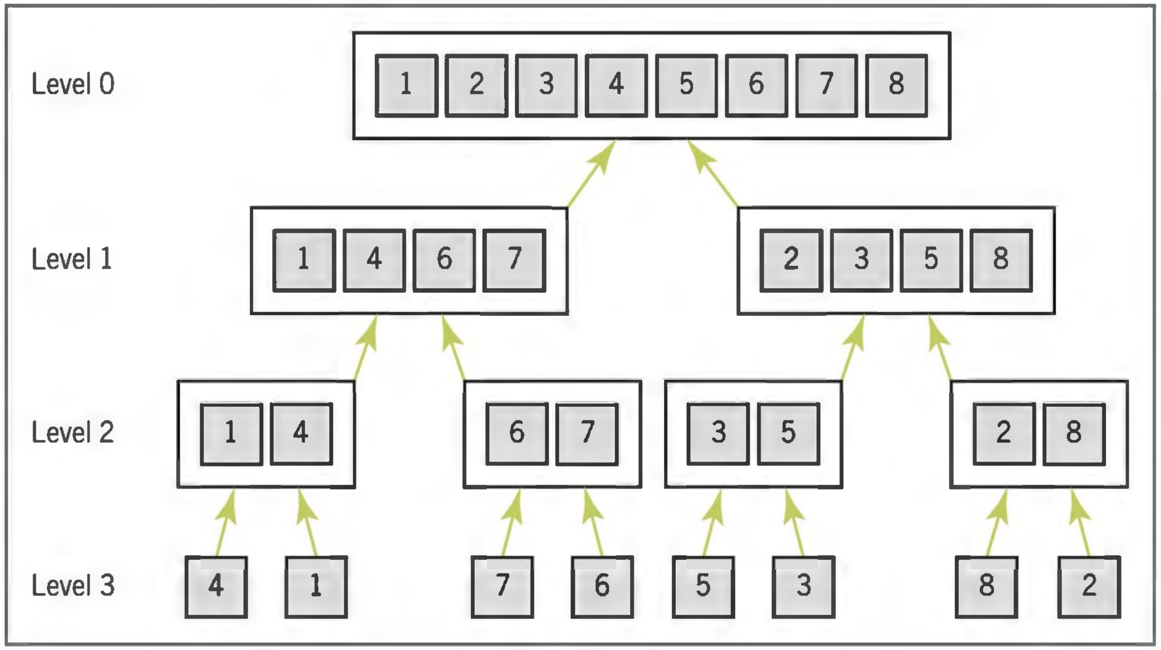


Figure 11-10 Merging the sublists during a merge sort

Merge Sort (5 of 5)

- **Complexity Analysis of Merge Sort**
  - The running time of the merge function is dominated by the two for statements
  - Each of which loops (**high** - **low** + 1) times
  - The function's running time is O(**high** - **low**), and all the merges at a single level take O(**n**) time
  - Because mergeSortHelper splits sublists as evenly as possible at each level, the number of levels is O(log **n**)
    - And the running time for this function is O(**n log n**) in all cases.

An Exponential Algorithm: Recursive Fibonacci (1 of 2)

- Code for the Fibonacci function:

```
def fib(n):
"""Returns the nth Fibonacci number."""
if n < 3:
return 1
else:
return fib(n – 1) + fib(n – 2)
```

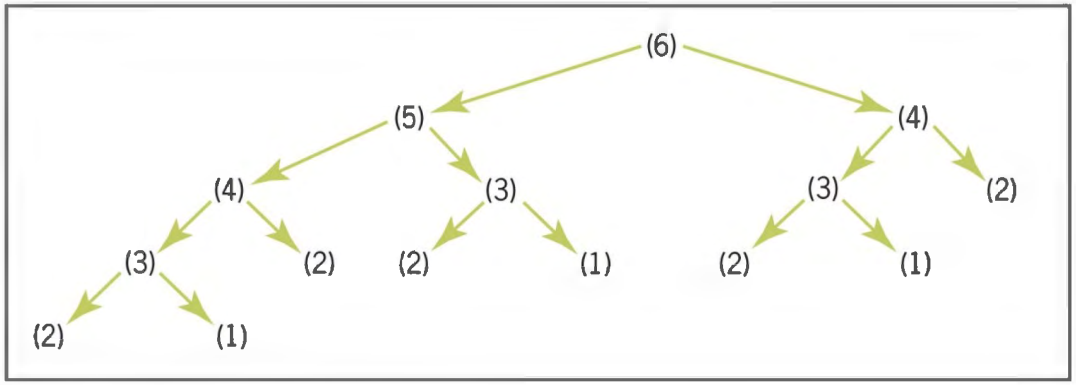


Figure 11-11 A call tree for `fib(6)`

An Exponential Algorithm: Recursive Fibonacci (2 of 2)

- Exponential algorithms are generally impractical to run with any but very small problem sizes
- Recursive functions that are called repeatedly with same arguments can be made more efficient by technique called **memoization**
  - Program maintains a table of the values for each argument used with the function
  - Before the function recursively computes a value for a given argument, it checks the table to see if that argument already has a value

Converting Fibonacci to a Linear Algorithm (1 of 3)

- Pseudocode:

```
Set sum to 1
  Set first to 1
  Set second to 1
  Set count to 3
  While count <= N
  Set sum to first + second
  Set first to second
  Set second to sum
  Increment count
```

Converting Fibonacci to a Linear Algorithm (2 of 3)

```
def fib(n, counter = None):
   """Count the number of iterations in the Fibonacci
   function."""
   theSum = 1
   first = 1
   second = 1
   count = 3
   while count <= n:
        if counter: counter.increment()
        theSum = first + second
        first = second
        second = theSum
        count += 1
   return theSum
```

Converting Fibonacci to a Linear Algorithm (3 of 3)

Problem Size	Iterations
2	0
4	2
8	6
16	14
32	30

Chapter Summary (1 of 2)

- Thread synchronization problems can occur when two or more threads share data
- Each computer on a network has a unique IP address that allows other computers to locate it
- Servers and clients communicate on a network by sending bytes through their socket connections
- A server can handle several clients concurrently by assigning each client request to a separate handler thread

Chapter Summary (2 of 2)

- A class variable is a name for a value that all instances of a class share in common
- Pickling is the process of converting an object to a form that can be saved to permanent file storage
- **try-except** statement is used to catch and handle exceptions
- Most important features of OO programming: encapsulation, inheritance, and polymorphism
  - Encapsulation restricts access to an object’s data to users of the methods of its class
  - Inheritance allows one class to pick up the attributes and behavior of another class for free
  - Polymorphism allows methods in several different classes to have the same headers
- A data model is a set of classes that are responsible for managing the data of a program