**Chapter 3**

**The Efficiency of Algorithms**

**A Guide to this Instructor’s Manual:**

We have designed this Instructor’s Manual to supplement and enhance your teaching experience through classroom activities and a cohesive chapter summary.

This document is organized chronologically, using the same headings that you see in the textbook. Under the headings you will find: lecture notes that summarize the section, Teaching Tips, Class Discussion Topics, and Additional Projects and Resources. Pay special attention to teaching tips and activities geared towards quizzing your students and enhancing their critical thinking skills.

In addition to this Instructor’s Manual, our Instructor’s Resources also contain PowerPoint Presentations, Test Banks, and other supplements to aid in your teaching experience.

|  |
| --- |
| **At a Glance** |

**Instructor’s Manual Table of Contents**

* Overview
* Learning Objectives
* Teaching Tips and Quick Quizzes
* Class Discussion Topics
* Additional Projects
* Additional Resources
* Key Terms
* Solutions to End-of-Chapter Exercises

|  |
| --- |
| Lecture Notes |

**Overview**

Chapter 3 introduces the kinds of attributes algorithms have and the methods by which computer scientists evaluate and compare algorithms. Among the algorithm attributes, the most important are time efficiency and space efficiency. The chapter introduces the concept of orders of magnitude and examines a variety of algorithms to determine the order of magnitude of each algorithm’s time efficiency. Some examples show the use of efficiency analysis, both time and space, to compare different algorithms for the same problem. It discusses the time/space tradeoff and the existence of intractable problems with no known efficient solutions.

# **Learning Objectives**

* Describe algorithm attributes and why they are important
* Explain the purpose of efficiency analysis and apply it to new algorithms to determine the order of magnitude of their time efficiencies
* Describe, illustrate, and use the algorithms from the chapter, including: sequential and binary search, selection sort, data cleanup algorithms, pattern-matching
* Explain which orders of magnitude grow faster or slower than others
* Describe what a “suspected intractable” problem is, giving one or more examples, and the purpose of approximation algorithms that partially solve them

# **Teaching Tips**

**3.1 Introduction**

1. Are some algorithms better than others? Remind students that they’ve seen multiple algorithms that solve the same problem in different ways. Explain that we need a way to compare algorithms against each other, and assess the quality of individual algorithms. Use the metaphor from the book: when purchasing a car, we take into account ease of handling, style, and fuel efficiency. Explain the parallels to algorithms: ease of understanding, elegance, and time/space efficiency.

**3.2 Attributes of Algorithms**

1. Correctness is a critical expectation. Keep in mind two components of correctness: is the problem specified correctly, and does the algorithm produce the correct result for every possible input. Use an algorithm from the previous chapter to illustrate issues with determining correctness. Explain that just like with cars, an error in an important algorithm could be fatal.
2. Before going into details, summarize the attributes that this section talks about: correctness, ease of understanding, elegance, and efficiency. Continue with the car metaphor as much as possible.
3. Algorithms developed by computer scientists are not merely of academic interest. They are intended to be used in the world. Computer programs run on real computers to solve problems of interest to real people, such as the word processor used to produce both the textbook and this instructor manual.
4. Program maintenance occurs when a program is modified to add new features or correct bugs. Ease of understanding, clarity, “ease of handling” are all terms to express the idea that the easier an algorithm is to understand, the easier it will be to perform maintenance on it.
5. Algorithms that are elegant use a clever or nonobvious approach and have a simple structure (sometimes deceptively so). Give examples of elegant algorithms, including Gauss’s summing example from the book and also others. Beginning students find this concept hard to pin down.
6. Note that these attributes are sometimes at odds with each other: efficiency may reduce ease of understanding, and ease of understanding opposes elegance. Sometimes they reinforce each other: ease of understanding can make determining correctness easier.

|  |  |
| --- | --- |
| ***Teaching Tip*** | Give students the following in-class activity to help them understand issues of correctness, ease of understanding, and elegance: Divide students into teams, and ask the teams to design an algorithm for a given problem. After each team writes down its algorithm, swap the algorithms to new teams. The new team is responsible for reading the algorithm and assessing it for the three attributes mentioned above. Each new team presents its algorithm to the class along with its assessment. |

1. Introduce the term **efficiency**. Distinguish between space efficiency and time efficiency. Talk about the confounding factors, such as machine speed, size of input, and particular input values, that make simply timing an algorithm infeasible.
2. Introduce the term **benchmarking**, and talk about its uses: for comparing different computers on the same inputs and for comparing different inputs to the same algorithm on a single computer.

**Quick Quiz 1**

1. Name two of the attributes of an algorithm discussed in this section.   
   Answer: Correctness, ease of understanding, elegance, and efficiency.
2. Timing an algorithm on different inputs on the same computer is one example of which of the following?

(a) Correctness testing

(b) Program maintenance

(c) Benchmarking

(d) Algorithm swizzling

Answer: (c)

1. (True or false) An algorithm that is elegant is always easy to understand.

Answer: False

1. Choosing between one that uses fewer resources than another to arrive at the same result is an example of \_\_\_\_\_\_\_.

Answer: efficiency

**3.3 Measuring Efficiency**

1. Introduce the term **analysis of algorithms** as the study of the efficiency of algorithms. Point out that there is a special term for this because of its importance. This section will introduce the concepts related to algorithm analysis through a series of examples.
2. Introduce the term **sequential search algorithm**, and remind students that they have come across this algorithm before, in Chapter 2.
3. The key idea for this example is the notion of the central unit of work: finding the operations that are most important for the task the algorithm is solving. For sequential search, the essential operation is the comparison of the *NUMBER* to each number in the list. Discuss why this is so.
4. Explain clearly what best case and worst case mean. It is important to make clear that the best case does not refer to the smallest input size but always is relative to some unspecified but large input size. Best case for sequential search is one comparison; worst case is n comparisons.
5. Average case depends on how often different cases recur. For sequential search, if all numbers are equally likely to be searched, then the average balances out at roughly n/2 comparisons. Point out extremely large groups, like the 350,000,000 listed telephone numbers, will take a very long time to complete even at 50,000 comparisons per second. While not seemingly time efficient, a sequential search algorithm is the best option for searching an unordered list.
6. Introduce the term **order of magnitude n** and its notation Θ(n). The key to understanding this group is to note that all linear functions have a fixed rate of change: the change from one value of n to the next is always constant.
7. Introduce the two terms **searching** and **sorting**, and explain that these are two very common tasks for computers. Mention common applications, including web searching, where results are sorted by relevance.
8. Introduce the term **selection sort algorithm**, and describe its approach: divide the list into a sorted and unsorted section, and repeatedly find the largest value in the unsorted section, and move it to its proper location. A physical example is useful when introducing sorting: playing cards, books, and students sorted by birthday are all great approaches.

|  |  |
| --- | --- |
| ***Teaching Tip*** | Refer students to the following link for a nice animation of sorting algorithms: <http://math.hws.edu/TMCM/java/xSortLab/> |

1. Selection sort has a hidden unit of work: the comparisons done inside the “find largest” step are crucial. Discuss carefully how the amount of work done for each instance of finding the largest changes, gradually diminishing. Discuss the X, Y, T memory needs of selection sort algorithms.
2. Introduce the term **order of magnitude n2**, and its notation Θ(n2). The most important concept here is the fact that eventually every function that has order of magnitude n2 will have greater values than any function that has order of magnitude n. Use the walking versus driving analogy.

**Quick Quiz 2**

1. The problem of putting a list of items in order is called \_\_\_\_\_\_\_\_\_\_\_\_\_.  
   Answer: sorting
2. (True or false) Functions with order of magnitude n will eventually grow faster than a function with order of magnitude n2, for some big enough value of n.

Answer: False

1. (True or false) Sequential search has order of magnitude n in the worst case and the average case.

Answer: True

1. To determine the time efficiency of an algorithm, we must first determine:
2. the number of lines in the algorithm.
3. the central unit of work in the algorithm.
4. whether or not a while loop appears in the algorithm.
5. whether or not the algorithm is named properly.

Answer: (b)

1. Software that served a small number of data points can still be used when that number has increased significantly (1,000 active Facebook users vs. more than one billion).

Answer: False

**3.4 Analysis of Algorithms**

1. Tie this section together by explaining that it illustrates algorithm analysis in three ways. It shows how to use analysis to compare three different algorithms for a new problem (data cleanup), it uses analysis to show a better solution to a known problem (searching), and it uses analysis to study the performance of a known algorithm (pattern matching).
2. Describe the new problem: the *data cleanup problem*, the task of removing zeros from a collection of age data.

|  |  |
| --- | --- |
| ***Teaching Tip*** | Begin this section by giving students an example of this problem and asking them to solve it and then discussing the different approaches they took to the problem. Note that different media will tend to support different algorithms: pen and paper versus on the board versus separate cards for individual data items. |

1. The shuffle-left algorithm removes zeros by shifting all values to the right of a zero one cell to the left. If the values were written on cards, this would correspond to sliding the cards to the left to fill in the empty slot from removing the zero card. Illustrating this algorithm with an example on cards might be easier than writing and erasing an example on the board.
2. The best case for shuffle-left is when no zeros occur; the algorithm just checks each value to see if it is zero for a time efficiency of Θ(n). The worst case occurs when all the values are zero; the algorithm will move n-1 zeros, then n-2 zeros, and so forth. This has a time efficiency of Θ(n2). Note that shuffle-left uses no significant extra space.
3. The copy-over algorithm copies nonzero values to a new list. The best and worst cases are opposite to the shuffle-left algorithm. When the original list contains only zeros, then the copy-over algorithm copies nothing and does just the work required to check each value: Θ(n). It also uses no extra space in this case. When the original list contains no zeros, then the algorithm copies every value. This takes Θ(n) time but also uses an extra n in space. Note that the best and worst cases here are the same order of magnitude.
4. Compare shuffle-left with copy-over for worst-case performance, and note the time/space tradeoff between the two.
5. The converging-pointers algorithm is more complex than the previous two. Illustrate this algorithm using clear markers for the left and right pointers (your fingers, arrows drawn or taped, or two students). This algorithm has best and worst cases of Θ(n), because we avoid moving more than one value per zero.
6. Introduce the term **binary search algorithm**, and explain that it is a different approach to solving the search problem from before. Emphasize the tradeoff between better performance and the constraint that the data must be sorted. Students often attempt to apply this algorithm to unsorted data!
7. Use the metaphor of searching a telephone directory to illustrate the way in which binary search splits the data in half at each point. Show examples of thinking of the list as a binary search tree.

|  |  |
| --- | --- |
| ***Teaching Tip*** | It is incredibly effective to illustrate binary search by bringing in a reasonably small telephone directory or dictionary and physically tearing the book in halves, following the binary search algorithm. |

1. Best case for binary search and any search algorithm is always one comparison. The worst case is log2 n, where n is the number of elements. Spend time explaining logarithms; many students do not remember them well.
2. Introduce the term **order of magnitude lg n**, Θ(lg n), and show students how slowly it grows compared to linear or quadratic growth.
3. The pattern-matching algorithm was discussed in Chapter 2; its analysis is more complex than previous examples. Emphasize that both best and worst case can happen when the pattern does not appear in the text. Best and worst cases are both somewhat artificial; explain that they are the extremes and actual performance should be between.
4. Have students work through small examples of best and worst case in detail, and then generalize to the overall formulas.

**Quick Quiz 3**

1. Name an order of magnitude that grows faster than Θ(n).  
   Answer: Any one of: Θ(n2), Θ(n3), Θ(2n), or similar
2. The binary search algorithm has a worst-case order of \_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Answer: Θ(lg n)

1. (True or false) The best-case performance for the pattern-matching algorithm occurs when the pattern occurs at every position of the text.

Answer: False

1. (True or false) The converging-pointers algorithm to solve the data cleanup problem is most efficient both in time and space.

Answer: True

1. It is more important to track the amount of \_\_\_\_\_\_\_\_\_ than the amount of \_\_\_\_\_\_\_\_\_\_ because the former has a fixed upper bound.

Answer: memory, time

1. (True or false) The binary search algorithm works on data that is both sorted and unsorted.

Answer: False

**3.5 When Things Get Out of Hand**

1. Introduce the term **polynomially bounded**, and explain that the problems seen so far all have such solutions.
2. Describe the Hamiltonian circuit problem, and ask students to solve a small instance of the problem.
3. The algorithm to find a Hamiltonian circuit checks every possible path in the graph, also known as a brute force algorithm. Use concrete examples other than the book’s example to illustrate how we can conceive of that search as traversing a tree of choices. Give examples that have more than two choices per vertex, such as picking your way to a destination using public transportation when there are multiple options available (bus or subway).
4. Using the simple case of a graph with exactly two choices at each node, show how we determine that the number of possible paths is 2n. Introduce the term **exponential algorithm**.
5. Introduce the term **intractable**, and describe some problems that seem intractable, including the Hamiltonian circuit, bin-packing, and solving a game of chess.
6. Introduce the term **approximation algorithms**, which provide partial solutions to intractable problems.

# **Class Discussion Topics**

1. Consider the attributes discussed in Section 3.2. Which attributes are subjective to evaluate, and which could be objectively evaluated? Explain your reasoning.
2. Binary search requires that the data be sorted before it can be applied. If we start with unsorted data and use selection sort to sort it, how many searches must we do using binary search to be more efficient than sequential search on the unsorted data?
3. Compare the three data cleanup algorithms in the book for all the attributes discussed in Section 3.2. Which algorithm do you rate as best for each of the individual attributes (e.g., which algorithm is the most elegant, which is the most maintainable)? Taking all the attributes into account, which algorithm seems best overall?
4. Why isn’t the best-case performance of an algorithm always when the input size is zero or one?

# **Additional Projects**

1. Give students an algorithm that determines whether a list of values has no repetitions in it. Ask them to work together to determine what the basic unit of work in the algorithm is, what the best- and worst-case efficiencies are, and under what circumstances they occur.
2. Evaluate the efficiency of the find maximum algorithm from the book. Does it have a best case that differs from its worst case?
3. Take a sorted list containing the following values: 3, 6, 10, 14, 15, 23, 27, 29, 32, and 44. Find a value to search for that causes sequential search to perform the most comparisons. Find a value to search for that causes binary search to perform the most comparisons. How do the worst cases compare for these two algorithms on this particular list?

# **Additional Resources**

1. A listing of famous intractable problems: <https://sites.google.com/site/dtcsinformation/other-topics/complexity/classic-intractable-problems>
2. CS unplugged video demonstration of binary search: <http://www.youtube.com/watch?v=iDVH3oCTc2c&feature=related>
3. Animation of bin-packing approximation algorithm: <http://www-cg-hci.informatik.uni-oldenburg.de/~da/iva/baer/binpacking/index.html>
4. How to solve supposedly intractable problems: https://www.johndcook.com/blog/2012/02/28/solving-intractable-problems/

**Key Terms**

* **Analysis of algorithms:** The study of the efficiency of algorithms.
* **Approximation algorithm:** An algorithm that doesn’t solve a problem but provides a close approximation to a solution.
* **Benchmarking:** Running a program on many data sets to be sure its performance falls within required limits; timing the same algorithm on two different machines.
* **Binary search algorithm:** An algorithm that searches for a target value in a sorted list by checking at the midpoint and then repeatedly cutting the search section in half.
* **Bubble sort:** [Exercises 11–14] A sorting algorithm that makes multiple passes through the list from front to back, each time exchanging pairs of entries that are out of order.
* **Efficiency:** An algorithm’s careful use of time and space resources.
* **Exponential algorithm:** An algorithm whose work varies as some constant to the power of the input size n.
* **Flops:** A unit of measure of processor speed: floating-point operations per second.
* **Intractable:** A problem for which no polynomially bounded solution exists.
* **Order of magnitude lg n:** The efficiency classification of an algorithm whose work varies as a constant times lg(n).
* **Order of magnitude n:** The efficiency classification of an algorithm whose work varies as a constant times the input size n.
* **Order of magnitude n2:** The efficiency classification of an algorithm whose work varies as a constant times the square of the input size.
* **Polynomially bounded:** An algorithm that does less work than some polynomial expression of the input size n.
* **Searching:** The task of finding a specific value in a list of values or deciding it is not there.
* **Selection sort algorithm:** A sorting algorithm that keeps moving larger items toward the back of the list.
* **Sequential search algorithm:** An algorithm that searches for a target value in a random list by checking each list item in turn.
* **Short sequential search:** [Exercise 22] A variation of the sequential search algorithm that requires a sorted list and stops once it has passed the place where the target could occur.
* **Smart bubble sort:** [Exercise 12] A variation of the bubble sort algorithm that stops when no exchanges occur on a given pass.
* **Sorting:** The task of putting a list of values into numeric or alphabetical order.

# **Solutions to End-of-Chapter Exercises**

1. (a) Group into 25 pairs of the form

2 + 100 = 102

4 + 98 = 102

.

.

.

50 + 52 = 102

so that the sum = 25(102) = 2550.

(b) (*n*/2)(2*n* + 2) = *n*(*n* + 1)

1. (a) Using Gauss's formula for the sum of the numbers from 1 to n, ,

1 + 2 + 3 + … + 12 =  = 78

(b) Applying Gauss's formula for each of the 12 days, the total equals



= 

= 

**3.** (a) The terms of the Fibonacci sequence are:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765

so F(20) = 6765

(b) 6765

(c) Using the definition is probably clearer, because it is not easy to see why the formula is true. However, adding up the 20 terms in the definition to find *F*(20) is somewhat less efficient. For *F*(100), the formula is definitely quicker.

(d) Once cell C1 contains =A1 + B1, which computes F(3),you can drag this formula along line 1 as far as you want so that cell D1 computes the result B1 + C1 = F(4), cell E1 computes the result C1 + D1 = F(5), etc. The value of F(20) is displayed in the 20th cell of row 1, which is cell T1.

(e) Assuming the value of n is stored in cell B6, the formula to compute F(n) is

(SQRT(5)/5)\*POWER((1 +SQRT(5))/2,B6)-(SQRT(5)/5)\*POWER((1 - SQRT(5))/2,B6)

(f) To compute F(50) using the definition, you have to drag the formula to cell AX1. T0 compute F(50) using the definition, you just have to enter 50 in cell B6. Even though the spreadsheet is doing all the work, it still seems quicker to just change the value of n. In either case, F(50) = 12,586,269,025

**4**. (a) 171 + 85 + 43 + 21 + 11 + 5 + 3 + 1 + 1 = 341

(b) There is 1 winner so there must be 341 losers, therefore 341 matches.

(c) The second algorithm is more elegant and efficient. Perhaps the first is clearer, since the second is so clever.

**5**. (a) middle position = 8.

8 comparisons required.

(*n* + 1)/2.

(b) two middle positions = 8 and 9.

8 and 9 comparisons required, average = 8.5.

(*n* + 1)/2.

(c) (*n* + 1)/2.

**6**. 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28.

28/7 = 4.

Yes, the number is slightly higher than *n*/2 = 7/2 = 3.5, and it is exactly (*n* + 1)/2 = 8/2 = 4 (see Exercise 5).

**7.** (a) (32,000,000/2) × (1/12000) = 1333.3 secs = more than 22 minutes

(b) 225 ≈ 33,000,000 so lg (32,000,000) = 25. Therefore, the time required would be

25 × (1/12000) = 0.002 secs

**8.** At 14 pancakes, both formulas give the same result (25 flips). If the number of pancakes is more than 14, the new algorithm is faster, but may give a fractional answer, which does not make physical sense. At 17 pancakes, the original algorithm requires 31 flips and the new formula requires 30 flips.

**9.** 7, 4, 2, 9, 6 (original list)

7, 4, 2, 6, 9

6, 4, 2, 7, 9

2, 4, 6, 7, 9

**10.** When the unsorted section consists of one number, the largest so far is set to that number. There are no other numbers to compare it to so the find-largest algorithm terminates, leaving that number as the largest. This process uses zero comparisons, so adds nothing to the total of the comparisons done, and removing it has no effect.

**11.** (a)

4 8 2 6

4 **2 8** 6

4 2 **6 8**

**2 4** 6 8

(b)

12 3 6 8 2 5 7

**3 12** 6 8 2 5 7

3 **6** **12** 8 2 5 7

3 6 **8 12** 2 5 7

3 6 8 **2 12** 5 7

3 6 8 2 **5 12** 7

3 6 8 2 5  **7**  **12**

3 6 **2 8** 5 7 12

3 6 2 **5 8** 7 12

3 6 2 5 **7** **8** 12

3 **2 6** 5 7 8 12

3 2 **5 6** 7 8 12

**2 3** 5 6 7 8 12

(c)

D B G F A C E H

**B D** G F A C E H

B D **F G** A C E H

B D F **A G** C E H

B D F A **C G** E H

B D F A C **E G H**

B D **A F** C E G H

B D A **C F** E G H

B D A C **E F** G H

B **A D** C E F G H

B A **C D** E F G H

**A B** C D E F G H

For these cases, bubble sort required the same or more exchanges than selection sort.

**12.** Initially, the entire list is unsorted (the marker *U* for the unsorted section is at the end of the list, position *n*). Within the loop at step 3, the marker is decreased (at step 8) once for each pass until the unsorted section of the list is reduced to one element, which means *U* = 1. The inner loop at step 5 is always done as many times as it takes to move C through positions 2, … , U.

|  |  |  |
| --- | --- | --- |
| Value of U | Number of passes through loop at step 5 |  |
| *n* | *n - 1* | (C goes from 2 through *n*) |
| *n* – 1 | *n* – 2 | (C goes from 2 through *n* - 1) |
| *. . .* | *. . .* | . . . |
| 2 | 1 | (C goes from 2 through 2) |

The loop at step 5 is therefore done (*n* – 1) + (*n* – 2) + . . . + 2 + 1 times, which is Ө(*n*2). Each pass through this loop requires one comparison, at step 6.

**13.** Selection sort does one exchange for each position in the list, regardless of what the list looks like. Bubble sort only exchanges pairs that are out of order, so if the list is already sorted, it will do no exchanges.

**14.** (a) 1. Get values for *n* and the *n* list items

2. Set the marker *U* for the unsorted section at the end of the list

Set the value of *Exchanges* to 1

3. While (*Exchanges* > 0) and the unsorted section has more than one element do   
 steps 4 through 11

4.   Set the current element marker *C* at the second element of the list  
 5. Set the value of *Exchanges* to 0

6.   While *C* is has not passed *U* do steps 7-10

7. If the item at position *C* is less than the item to its left then do steps 8 and 9

8. Exchange these two items

9. Add 1 to the value of *Exchanges*

10. Move *C* to the right one position

11. Move *U* left one position

12. Stop

(b) 7, 4, 12, 9, 11 | 7 and 4 are compared (and exchanged)

4, 7, 12, 9, 11 | 7 and 12 are compared

4, 7, 12, 9, 11 | 12 and 9 are compared (and exchanged)

4, 7, 9, 12, 11 | 12 and 11 are compared (and exchanged)

4, 7, 9, 11, 12

End of Pass 1, 4 comparisons (and 3 exchanges) took place

4, 7, 9, 11 | 12 4 and 7 are compared

4, 7, 9, 11 | 12 7 and 9 are compared

4, 7, 9, 11 | 12 9 and 11 are compared

End of Pass 2, 3 comparisons (and 0 exchanges) took place

The algorithm stops; a total of 7 comparisons was done.

(c) An already sorted list is the best case; it requires n – 1 comparisons and 0 exchanges.

(d) In a reverse-sorted list, smart bubble sort acts exactly the same as regular bubble sort.

**15.** A = 8, 12, 19, 34 B = 3, 5, 15, 21 C =

A = 8, 12, 19, 34 B = 5, 15, 21 C = 3

A = 8, 12, 19, 34 B = 15, 21 C = 3, 5

A = 12, 19, 34 B = 15, 21 C = 3, 5, 8

A = 19, 34 B = 15, 21 C = 3, 5, 8, 12

A = 19, 34 B = 21 C = 3, 5, 8, 12, 15

A = 34 B = 21 C = 3, 5, 8, 12, 15, 19

A = 34 B = C = 3, 5, 8, 12, 15, 19, 21

A = B = C = 3, 5, 8, 12, 15, 19, 21, 34

**16.** The original list is 6, 3, 1, 9, with size > 1. Split the list into two halves (step 3):

6, 3 1, 9

Use mergesort on the first half, 6, 3, with size > 1. Split the list into two halves (step 3):

6 3

Each half contains 1 item, so there is nothing to do. This completes steps 4 and 5 for the list 6, 3, so step 6 requires merging A = 6 and B = 3 to get C = 3, 6 (2 comparisons). This completes step 6 for the first half, so mergesort on the first half stops (step 7).

Use mergesort on the second half, 1, 9, with size > 1 Split the list into two halves (step 3):

1 9

Each half contains 1 item, so there is nothing to do. This completes steps 4 and 5 for the list 1, 9, so step 6 requires merging A = 1 and B = 9 to get C = 1, 9 (2 comparisons). This completes step 6 for the second half, so mergesort on the second half stops (step 7).

This completes steps 4 and 5 on the list 6, 3, 1, 9. Now merge A = 3, 6, B = 1, 9 to get C = 1, 3, 6, 9 (4 comparisons).

This completes step 6 on the list 6, 3, 1, 9, so the algorithm stops, with the resulting sorted list 1, 3, 6, 9 that required a total of 8 comparisons.

**17.** (a) lg n

(b) n

(c) Ө(n lg n)

(d) Yes. Exercise 16 required 8 comparisons, which equals 4\*lg 4 = 4\*2 = 8.

**18.** (a) 4, 3, 7

4, 7, 3

3, 4, 7

3, 7, 4

7, 4, 3

7, 3, 4

(b) 3! = (3)(2)(1) = 6. This agrees with the number of permutations from Part a.

(c) The body of the while loop that contains step 5 is always executed *n*! times. In the best case, the first permutation checked is the sorted permutation, so step 10 is executed only once, giving a total of *n*! + 1 work units. In the worst case, the last permutation checked is the sorted permutation, so step 10 is executed *n*! times, giving a total of 2*n*! work units.

(d)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *n* | | | |
| Order | 3 | 5 | 10 | 20 |
| *n*2 | 0.0009 sec | 0.0025 sec | 0.01 sec | 0.04 sec |
| *n*! | 0.0006 sec | 0.012 sec | 6 min | 77 ×103 centuries |

(e) The new algorithm requires *n*! copies of the original list, so it is extremely space-inefficient (as well as time-inefficient).

**19.** We want to find the value of *n* where the number of instructions executed by the two algorithms is equal. Above that point, algorithm B will be faster. So we must solve the equation:

0.003*n*2 = 243*n*

to get *n* = 81,000

**20.** (a) For each of rows 1 through *n* do the following

For each of columns 1 through *n* do the following

Print the entry in this row and column

n2 print statements

(b) For each of the districts 1 through *n* do the following

For each of rows 1 through *n* do the following

For each of columns 1 through *n* do the following

Print the entry in this row and column

*n*3 print statements.

(c) Ө(*n3­*)

**21**. legit = 6.

3 2 6 7 5 1 1 1 1 1

8 + 8 + 5 + 5 = 26 copies.

**22**. legit = 6.

3 1 5 2 6 7 0 0 5 1

3 copies.

**23.** Once item *n* has been copied one cell left, it need not be copied again. Similarly, once item *n* – 1 has been copied one cell left, it need not be copied again. The value of legit shows how many cells from the right end have been copied; step 11 of the algorithm can be changed to “while right is less than or equal to legit do steps 12 and 13." 8 + 7 + 3 + 2 = 20 copies

**24.** 30 copies = 6(6 – 1).

**25.** (a) John, Elsa

(b) John, Lee, Snyder, Tracy

(c) John, Elsa, JoAnn

**26.** 14, 22, 31, 43

**27.**

Step 1: Get values for the target to be searched for, *n*, and the *n* list items

Step 2: Set the value of *i* to 1, set the value of *Found* to NO, set the value of *Greater* to NO

Step 3: While (*Found* = NO) and (i ≤ n) and (*Greater* = NO) do steps 4 through 8

Step 4: If target is equal to the *i*th element then do steps 5 and 6

Step 5: Set the value of *Found* to YES

Step 6: Print the message "Item found at location" i

Step 7: If element *i* is greater than target, set the value of *Greater* to YES

Step 8: Add 1 to the value of *i*

Step 9: If Found = NO, then print the message 'Sorry, the target is not in this list'

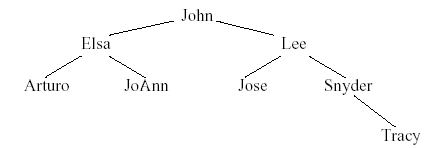
Step 10: Stop

**28.**  (a) *n* (for example, if target is equal to last element in list, must check all *n* elements)

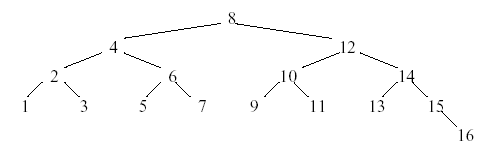
(b) *n*/2 – element will be found just as in regular sequential search

(c) Yes – for elements that are not in the list but are less than some of the list elements, this algorithm will terminate sooner than regular sequential search. (But remember that the list must already be sorted.)

**29.** Worst case = 4; Tracy.

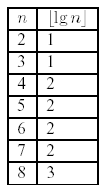


**30.** Worst case = 5 for the value at position16.

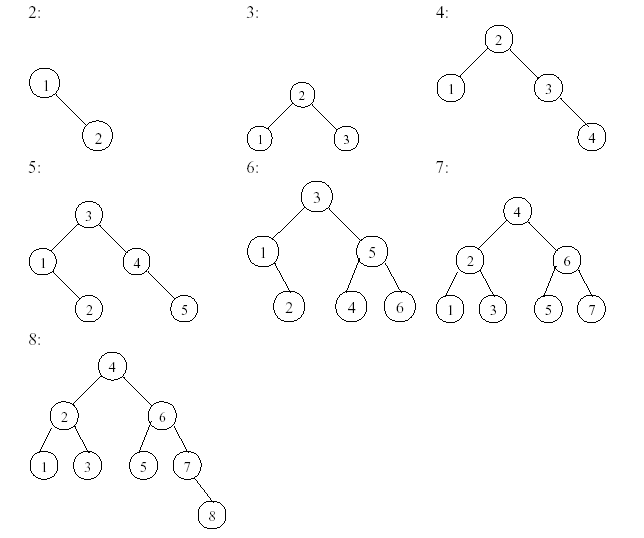


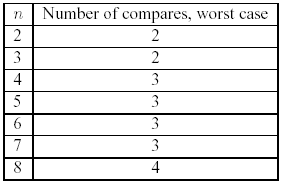
**31.** (a) 1, 2, 8, -5

(b)



(c)





(d) 1 + ⎣lg n⎦

**32.**  Average = (1 × 1 + 2 × 2 + 4 × 3) / 7 = 17 / 7 = 2.43. The worst case is 3, so this is somewhat less.

**33.** For sequential search, the worst case is *n* = 100,000, so for *p* searches, *p* × 100,000 comparisons are required. Using an Θ(*n*2) sorting algorithm would require 100,000 × 100,000 comparisons. Binary search would require 1 + ⎣lg n⎦ = 1 + ⎣lg 100,000⎦ = 17 comparisons for each search, so altogether there would be 100,000 × 100,000 + 17*p* comparisons. The first expression is larger than the second for *p* > 100,017.

**34.** (a) The worst, as before, is if the pattern almost occurs everywhere in the text.

Text: AAAAAAAAA Pattern: AAAB

Number of comparisons: *m*(*n – m* + 1)

(b) The best case is if the pattern occurs as the first m characters of the text.

Text: ABCDEFGHI Pattern: ABC

Number of comparisons: *m*

**35.** (a) 6

(b) Each node in the graph has 4 choices for the next node in a path, so there are 46 = 4096 paths.

(c) 4096 paths × 0.0001 seconds per path = 0.4096 sec

(d) 412 paths × 0.0001 seconds per path equals about 28 minutes.

**36.**  (a) The first graph has several Euler paths, for example, C-A-B-C-D-B

The second graph has no Euler path.

(b) (i) has 0 odd nodes, so an Euler path exists, for example A-C-B-D-A-E-B-F-A

(ii) has 4 odd nodes, so no Euler path exists

(iii) has 2 odd nodes, so an Euler path exists, for example C-A-E-B-F-A-D-C-B-D

(c) Θ(n2)

(d) No, a polynomial algorithm exists

**37.** Solve the equation

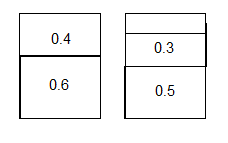
100*n*2 = 0.01(2*n*)

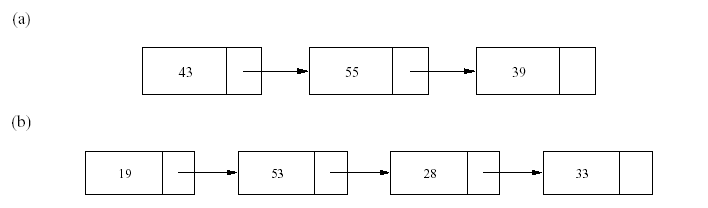
to get *n* = 23

**38.** (a) 5 × 10 = 50

(b) 52 × 10 = 25 × 10 = 250

**39**.



**40.** 

(c) Locate the first item. Follow the pointers through the list; if the next list item ever has the value 0, then change the pointer of the current list item to point to where the zero item is pointing (removing the zero item). Stop at the end of the list.

There are at most *n* – 1 "follow pointer" pointer operations and at most *n* – 2 "change pointer" operations, hence this is an Ө(*n*) algorithm.

**41**. (a) 2, 1, 2, 1, 2, 1, 2, 1

(b) 4, 2, 1, 4, 2, 1, 4, 2, 1, 4, 2, 1, 4, 2, 1, 4, 2, 1, 4, 2, 1, 4, 2, 1

(c) Θ(n lg n)

(d) 16 × 4 = 64

**42**. (a) n - 1 (*largest so far* must be compared with all elements in the list after item 1)

(b) Θ(n2) to sort + 0 comparisons required to get the second largest value = Θ(n2)

(c) (n - 1) + (n - 2) = Θ(n)

**Discussion of Challenge Work**

1. (a) There are 37 syllables in each verse that are not dependent on which verse it is (from the first two and the last lines of each verse). So clearly after *n* verses, there are 37*n* syllables from these lines. The remaining part of each verse is about each animal’s particular noise. The lines for each animal take 22 syllables, and there are   
1 + 2 + . . . + *n* copies of those lines of the song overall (one copy for the first verse, two for the second verse, and so on). So that means there are 22(1 + 2 + . . . + *n*) additional syllables. But 1 + 2 + . . . + *n* is just *n*(*n* + 1) / 2, so the total number of syllables is 22*n*(*n* + 1)/2 + 37*n.*

(b) This means that the song is Ө(*n*2) since *n*(*n* + 1)/2 generalizes to *n*2.

2. Students should research the simplex method in more detail than this question asks in order to fully understand it.