Dana Vrajitoru · William Knight

Practical Analysis of Algorithms

Springer

Dana Vrajitoru
William Knight
Indiana University
South Bend, IN
USA

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-319-09887-6 ISBN 978-3-319-09888-3 (eBook)
DOI 10.1007/978-3-319-09888-3

Library of Congress Control Number: 2014946382

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Contents

1 Introduction 1
1.1 Measuring Execution Time 2
1.2 Classes of Algorithms 3
1.3 Order of Complexity 5
1.4 Book Organization 5

2 Mathematical Preliminaries 9
2.1 Floors and Ceilings 9
2.1.1 Further Examples 15
2.1.2 Exercises 17
2.2 Logarithms 19
2.2.1 Further Examples 24
2.2.2 Exercises 26
2.3 Sums and Summation Notation 29
2.3.1 Further Examples 40
2.3.2 Exercises 46
2.4 Factorials 52
2.4.1 Further Examples 55
2.4.2 Exercises 55
2.5 Remainders in Integer Division 56
2.5.1 Implementation of the Mod Operator in C and C++ . . . 60
2.5.2 Exercises 62

3 Fundamental Notations in Analysis of Algorithms 63
3.1 Little-Oh Notation 63
3.1.1 Further Examples 70
3.1.2 Exercises 71
3.2 Big-Oh, Big-Omega, and Big-Theta Notations 72
3.2.1 Further Examples 80
3.2.2 Exercises 84
3.3 Asymptotic Functions 89
3.3.1 Intuitive Analogy 93
3.3.2 Exercises 93

4 Recurrence Relations 95
4.1 Fundamental Definitions 95
4.1.1 Further Examples 99
4.1.2 Exercises 103
4.2 Linear Homogeneous Recurrence Relations with Constant Coefficients 107
4.2.1 Further Examples 113
4.2.2 Exercises 116
4.3 Linear Non-homogeneous Recurrence Relations 117
4.3.1 Further Examples 127
4.3.2 Exercises 136
4.4 Recurrence Relations Involving $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ 137
4.4.1 Special Linear Equations Involving $F(\lfloor n/2 \rfloor)$, $F(\lceil n/2 \rceil)$, etc. 146
4.4.2 Further Examples 146
4.4.3 Exercises 152
4.5 Other Recurrence Relations 153
4.5.1 A Table of Transformations 156
4.5.2 Growth Rates for Solutions of Special Linear Equations Involving $F(\lfloor n/2 \rfloor)$, $F(\lceil n/2 \rceil)$, etc. 160
4.5.3 Further Examples 161
4.5.4 Exercises 166

5 Deterministic Analysis of Algorithms 169
5.1 Loop Analysis 169
5.1.1 Exercises 193
5.2 Euclid's Algorithm for the Greatest Common Divisor 204
5.2.1 Proof of the Underlying Theorem 210
5.2.2 Exercises 211
5.3 Binary Trees 211
5.3.1 Exercises 216
5.4 Traversals of Binary Trees 219
5.4.1 Exercises 225
5.5 Binary Search Trees 227
5.5.1 Exercises 238
5.6 Priority Queues and Binary Heaps 243
5.6.1 Exercises 251
5.7 Divide and Conquer: Merge Sorts 252
5.7.1 Exercises 259
5.8 Quicksort 266
5.8.1 Exercises 274

5.9 Comparison Based Sorting Algorithms 275
5.9.1 Exercises. 277
5.10 Pattern Matching Algorithms 280
5.10.1 Exercises. 293

6 Algorithms and Probabilities. 295
6.1 Simulation and Random Number Generators 295
6.1.1 Exercises. 298
6.2 Randomizing Arrays and Files 299
6.2.1 Exercises. 301
6.3 Choosing a Random Sample from a Sequential File 307
6.3.1 Exercises. 317
6.4 Simulating Observations of a Discrete Random Variable 319
6.4.1 Exercises. 324
6.5 Randomized Algorithms 325
6.5.1 Monte Carlo Algorithms 325
6.5.2 Las Vegas Algorithms. 327
6.5.3 Exercises. 328
6.6 Expected Behavior of Algorithms. 328
6.6.1 Exercises. 333
6.7 Applications of Expected Value Theorems to Analysis of Algorithms 338
6.7.1 Exercises. 340
6.8 Applications of Conditional Expected Value to Analysis of Algorithms 343
6.8.1 Exercises. 356

7 Finite Graph Algorithms 361
7.1 General Definitions. 361
7.1.1 Exercises. 368
7.2 Computer Representations of Finite Graphs 369
7.2.1 A Graph Class in C++ 372
7.2.2 Exercises. 374
7.3 Breadth-First Search and Depth-First Search 377
7.3.1 Breadth-First Search 377
7.3.2 Depth-First Search 380
7.3.3 Complexity of the Breadth-First and Depth-First 383
7.3.4 Exercises. 384
7.4 Numbering the Connected Components of an Undirected Graph. 385
7.4.1 Exercises. 386
7.5 Shortest Path Problems 386
7.5.1 Exercises. 395

7.6 Minimum Spanning Trees 397
7.6.1 Kruskal's Minimum Spanning Tree Algorithm (1956) 404
7.6.2 Prim's Algorithm (1957) 408
7.6.3 Exercises 411
7.7 Topological Sorting of Digraphs (Task Sequencing) 413
7.7.1 Exercises 418
7.8 Maximum Flow in Weighted Digraphs 419
7.8.1 Exercises 432
7.9 Tours 433
7.9.1 Exercises 440
7.10 Hamiltonian Paths and Circuits 441
7.10.1 Exercises 448
7.11 The Traveling Salesman Problem 450
7.11.1 Exercises 452

8 Appendix: Probability Theory 453

References 463

Index 465

6 Algorithms and Probabilities

In the preceding chapter we computed the minimum and maximum amounts of time and space that various algorithms require when they are run on a computer. In this chapter we will be concerned primarily with the "average" amounts of time that various algorithms require. We will also study some algorithms that use randomized procedures to achieve certain goals. Examples of such algorithms are "simulating random events", "randomizing the order of objects in a list", and "selecting a random sample from a population".

Throughout this chapter it is assumed that the reader has familiarity with the fundamental definitions and theorems of discrete probability theory. For readers who need to refresh their memories of that subject, there is an Appendix Chapter that summarizes the material needed. It defines the notions of statistical experiment, sample space, event, probability of an event, conditional probability, random variable, probability density function (p.d.f.), cumulative density function(c.d.f.), expected value of a random variable, and conditional expected value. The necessary theorems about these concepts are stated (but not proved) so that they can be cited in this algorithms chapter.

6.1 Simulation and Random Number Generators

Many computer applications require the **simulation** (i.e., computer modeling) of real-world systems for which at least part of the input is random in some way. For example, in a real-world queuing system such as a machine repair shop or a collection of grocery check-out lines, the arrivals of "customers" at the system occur unpredictably, but with a known probability distribution, and similarly the amount of time required to service one customer is variable in a way that depends on chance. Computer programs can be written that behave like queuing systems, and this makes it possible to study such systems and predict their behavior before they are actually built and put into service.

D. Vrajitoru and W. Knight, *Practical Analysis of Algorithms*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-09888-3_6

Computer simulation of systems that have unpredictable inputs requires the use of **random number generators**. These are functions that, when called repeatedly, return a stream of numbers that appear to be randomly chosen from some range or interval. In reality, of course, the numbers returned by such a function are *pseudo-random* numbers since they are actually generated by specific recurrence relations that would allow us to calculate each value from one or more preceding values. It is nevertheless customary to refer to such numbers as random numbers, dropping the modifier "pseudo-", which means "fake".

Throughout this chapter we assume when convenient that we have access to a random number generator function in C/C++ with the prototype

```
long randint(long first, long last);
```

that returns random integers in the range from `first` to `last` inclusive. There are $(last - first + 1)$ integers in this range, and each will have probability $\frac{1}{last - first + 1}$ of being returned by the function.

We also assume that we have access to a random number generator function in C/C++ with the prototype

```
double randouble(void);
```

that returns random real numbers uniformly distributed in the range [0.0, 1.0). Exercise 6.1.8 discusses some implementation details for these two functions.

The following examples illustrate how we can use a random number generator to simulate various statistical experiments (see Definition 3.3.1 in the Appendix). In each case, the solution given is just one of many possibilities.

Example 6.1.1 Suppose that a program we are writing needs to simulate the statistical experiment of making a single toss of a biased coin that turns up heads only 45 % of the time; that is, the probability of obtaining heads should be 45/100 = 9/20. How can we use the `randint` function to help us simulate such a toss? The answer is easy: we place in our program an instruction of the form

```
r = randint(1, 20);
```

and observe that `r` has probability 9/20 of being between 1 and 9 inclusive. Thus we can have our program report that heads occurs if the `r` value that's returned is in the range 1–9, and that tails occurs if the `r` value is in the range 10–20.

Example 6.1.2 Suppose we want our program to simulate the statistical experiment of choosing an object at random from an array `a[first..last]` of objects of any kind. We place in our program an instruction of the form

```
r = randint(first, last);
```

and then choose `a[r]` from the array.

Example 6.1.3 Suppose we want to simulate the statistical experiment of throwing a biased die for which an ace (one spot) is twice as likely to turn up as any other face. We place in our program an instruction of the form

$$\texttt{r} = \texttt{randint}(0, 6);$$

and then if $r = 0$ or 1 is returned, the program can report that an ace has turned up, while if $r > 1$, the program can report that the face with r spots has turned up.

Example 6.1.4 Suppose we want to simulate the statistical experiment of drawing a card from a shuffled deck. A first step would be to have the program create an array `deck[0..51]` containing representations of the 52 cards in a standard playing deck. Then the program could execute the instruction

$$\texttt{r} = \texttt{randint}(0, 51)$$

and report `deck[r]` to be the card drawn. Even though the array `deck[0..51]` is in some fixed order, by making a random choice from the array we get the effect of drawing from a shuffled deck.

Second Option: This alternative does not require us to set up any array whatsoever. Instead, we make two calls to `randint` as follows:

$$\texttt{rank} = \texttt{randint}(1, 13) \quad \text{and} \quad \texttt{suit} = \texttt{randint}(1, 4).$$

Then a `rank` of 1 means an ace, and so on up to 11 which means jack, 12 which means queen, and 13 which means king. Similarly, if `suit` is 1 we call it a club, 2 is a diamond, 3 is a heart and 4 is a spade.

Third Option: We can be slightly more efficient by making only one call to `randint`, followed by a little arithmetic:

$$\texttt{r} = \texttt{randint}(0, 51),$$

$$\texttt{suit} = \texttt{r}/13 + 1 \quad \text{and} \quad \texttt{rank} = \texttt{r}\%13 + 1,$$

with the same interpretation of the numbers as in the Second Option. Calls to `randint` are likely to be significantly more time consuming than simple arithmetic operations.

Example 6.1.5 Suppose we want to simulate the statistical experiment of throwing a pair of fair dice and reporting the sum of the faces that turn up. Since the sum can be any integer between 2 and 12 inclusive, a naive programmer might place the instruction

$$\texttt{t} = \texttt{randint}(2, 12);$$

in the program and have the program report `t` to be the sum of the spots on the two faces. The reason this is wrong is that when actually rolling a pair of dice the sums of the spots are not equally likely: a sum of 7 is six times as likely as a sum of 2. The call `randint(2,12)` gives all sums from 2 to 12 an equal probability of occurring.

A correct way to simulate the throw is to make two calls to the random number generator, say

$$\texttt{r} = \texttt{randint}(1, 6); \qquad \text{and} \qquad \texttt{s} = \texttt{randint}(1, 6);$$

and return the sum of these two integers as the sum of spots on the faces of the two dice.

Example 6.1.6 Suppose we want to simulate the statistical experiment of tossing a fair coin 5 times. The most straightforward way to do this is to make five successive calls of the form `randint(0,1);` and report each 0 as heads and each 1 as tails.

6.1.1 Exercises

6.1.7 What is the probability that the C++ instruction

```
int rem = randint(1,12) % 5;
```

will assign to `rem` the value 1? The value 4?

6.1.8 (a) Consider the situation where the language only provides a function `rand()` returning a long unsigned integer in the range `[0,RAND_MAX]` where the constant `RAND_MAX` is provided by the same library as the function `rand`. How would you implement the function `randouble()`?

(b) Given a C++ function `rand()` returning a random long unsigned integer, consider the following implementation of the function `randint()`:

```
long randint(long first, long last)
{ return first + rand() % (last- first + 1); }
```

Suppose (for purposes of illustration) that `rand()` returns integers in the absurdly small range $0 \ldots 15$. Calculate the probability that the call `randint (1,9)` will return the value 4. Calculate the probability that the call `randint (1,9)` will return the value 8. Explain why the code given above for `randint` is biased.

(c) Starting from a uniform `randouble()` function, consider the following implementation of the function `randint()`:

```
long randint(long first, long last)
{ return first + int(floor(randouble() * (last - first + 1))); }
```

Show that this function is unbiased.

6.1.9 The solution to the simulation problem posed in Example 6.1.5 involves two calls to `randint()`. Describe a way to simulate the throw of a pair of fair dice using just one call to the random number generator followed by some computer arithmetic.

The outcome of your program must be the two numbers appearing on the pair of dice and not their sum. [Hint: see Example 6.1.4 where two random numbers rank and suit are generated with one call to `randint()` followed by some arithmetic.]

6.1.10 The solution to the simulation problem posed in Example 6.1.6 involves five calls to `randint()`. Describe a way to simulate five tosses of a fair coin using just one call to the random number generator followed by some computer computations. [Hint: think about the binary representation of non-negative integers.]

6.1.11 How could we simulate the experiment of drawing a marble at random from an urn that contains 12 identical red marbles, 19 identical white marbles, and 5 identical blue marbles? The result of the simulation should be one of the three possible colors.

6.2 Randomizing Arrays and Files

Suppose we want to write a C++ program that simulates the playing of various card games. We might conceivably make the definitions and declarations shown in Fig. 6.1.

Now suppose we want to write the code for the "`shuffle`" function. This function should simulate a *fair* shuffle of the deck, by which we mean that the shuffling process should give every one of the 52! possible arrangements of the deck an equal likelihood of occurring. To say it differently, a "fair shuffle" is an algorithm for putting the 52-card array in random order without giving any arrangement a higher chance than others. Figures 6.2 and 6.3 show two candidate algorithms for simulating a fair shuffle. Which (if either) of these algorithms produces a fair shuffle?

Each of these algorithms causes a subscript `k` to move across the deck array, and at each step a randomly chosen card is swapped with `deck[k]`. The two algorithms differ in the way the randomly chosen card is selected: in Algorithm 1, it can be any of the 52 cards in the deck; in Algorithm 2 the choice is restricted to cards in the "not-yet-randomized" portion of the deck.

Most people are surprised to learn that Algorithm 1 is biased, i.e., does not produce a fair shuffle, while Algorithm 2 is unbiased. It will be left as Exercise 6.2.1 to prove that Algorithm 1 does not give a fair shuffle.

Let's now prove that Algorithm 2 is unbiased. Here is what we must prove: given any arrangement $(c_1, c_2, c_3, \dots, c_{52})$ of the 52 cards in the deck, the probability that Algorithm 2 will produce that particular arrangement is $\frac{1}{52!}$. Let A_k denote the event that card c_k ends up in cell `deck[k]`. Then we want to prove that

$$P(A_1 \cap A_2 \cap A_3 \cap \cdots \cap A_{52}) = \frac{1}{52!}.$$

Using the Chain Rule for conditional probability (Theorem 8.0.14), we can compute as follows: $P(A_1 \cap A_2 \cap A_3 \cap \cdots \cap A_{52}) = P(A_1) \times P(A_2|A_1) \times$

```
struct card_type
{
  int rank;
  int suit;
};

class deck_type
{
 public:

  enum suit_type {CLUB  = 1, DIAMOND = 2, HEART = 3, SPADE = 4};

  enum rank_type {TWO   =  2, THREE =  3, FOUR =  4, FIVE =  5, SIX  =  6,
                  SEVEN =  7, EIGHT =  8, NINE =  9, TEN  = 10, JACK = 11,
                  QUEEN = 12, KING  = 13, ACE  = 14 };

  deck_type()        // Constructor
  {
    int i = 1, rank, suit;      // i will run along the deck array
    for (rank = TWO; rank <= ACE; ++rank)
      for (suit = CLUB; suit <= SPADE; ++suit)
      {
        deck[i].suit = suit;
        deck[i].rank = rank;
        ++i;
       }
  }

  void shuffle();     // Randomize the order of cards in a deck

 private:

  card_type deck[53]; // This allows subscripts to go from 1 to 52
};
```

Fig. 6.1 A class representing a deck of cards

Shuffling Algorithm 1:

```
int k, r;
for (k = 1; k <= 52; ++k)
{
  r = randint(1, 52); // Generate a random integer in the range  1... 52.
  swap(deck[r], deck[k]);
}
```

Fig. 6.2 The first shuffling algorithm

Shuffling Algorithm 2:

```
int k, r;
for (k = 1; k <= 51; ++k)
{
  r = randint(k, 52);  // Generate a random integer in the range  k...52.
  swap(deck[r], deck[k]);
}
```

Fig. 6.3 The second shuffling algorithm

$P(A_3|A_1 \cap A_2) \ldots \times P(A_{52}|A_1 \cap A_2 \cap \cdots \cap A_{51}) = \frac{1}{52}\,\frac{1}{51}\,\frac{1}{50} \cdots \frac{1}{1}$. It is quite obvious why $P(A_1) = 1/52$: when `k` is 1 , the call `randint(k,52)` gives each of the subscripts from 1 to 52 an equal probability of being chosen, and c_1 must be in one of those 52 cells. The reason that $P(A_2|A_1) = 1/51$ is that if A_1 does occur, then c_1 has definitely been swapped into cell `deck[1]`, and so c_2 must be in one of the remaining 51 cells. Similarly, $P(A_3|A_1 \cap A_2) = 1/50$ because if A_1 and A_2 have indeed occurred, then c_1 and c_2 have been swapped into cells `deck[1]` and `deck[2]`, and so c_3 must be in one of the remaining 50 cells etc.

Shuffling Algorithm 2 can be generalized to apply to any array `a [first.. last]`. You are asked to construct this generalization in Exercise 6.2.1. Thus from now on we can assume we are in possession of an algorithm for randomizing the order of the objects in a given array of length n, i.e., an algorithm for rearranging the objects in such a way that each of the $n!$ possible arrangements has probability $\frac{1}{n!}$ of occurring.

The problem of randomizing the order of a collection of objects also arises with files. Suppose we are given a sequential file on a disk or tape and asked to randomize the order of the objects in the file. (A "sequential" file is a file whose objects can be accessed only in the order in which they are arranged in the file; there is no "random access" to the objects as there would be if the objects were in an array.) This file randomization is easy to do if our program is able to allocate an array large enough to hold all the objects in the file. In this case we can simply read all the objects into the array, randomize the array, and then write the objects in their new order back to disk or tape. The problem becomes more difficult if the file is so large that its contents will not all fit into an array in the computer's memory. This problem is considered in Exercise 6.2.8.

6.2.1 Exercises

6.2.1 On p. 301 it is asserted that Shuffling Algorithm 1 on that page is biased, i.e., does not produce a fair shuffle of the deck. This exercise is designed to demonstrate that fact when we reduce the deck to just 3 cards.

(a) Suppose we have 3 cards, call them a, b, and c. Suppose they are placed in an array called `deck` so that `deck[1] = a` , `deck[2] = b`, and `deck[3] = c` initially. Now suppose we apply Algorithm 1, but with the number 3 in place of the number 52. Draw a tree diagram (see Example 8.0.15) showing all the arrangements that can be produced at every step along the way. Show that some final arrangements of a, b, and c have higher probability of occurring than others. (If the shuffle were fair, each of the 3! possible final arrangements would have probability $\frac{1}{3!}$ of occurring.)

(b) Now consider the case of a deck of size 52. Explain why it is impossible for all arrangements of the deck to have equal probability of being produced by

Shuffling Algorithm 1. Do this by proving that it is impossible for *any* arrangement to have probability $\frac{1}{52!}$ of being produced. [Hint: imagine drawing the same kind of tree diagram as in part (a), except that such a diagram would be HUGE.] How many leaves would that tree diagram have? What would be the probability of each of the paths from root to leaf in the tree?

(c) Some students note that there is a difference between the numbers of times the body of the loop in Algorithm 1 and the body of the loop in Algorithm 2 are executed. In Algorithm 1 the body of the loop is executed 52 times, while in Algorithm 2 the body of the loop is executed only 51 times. If we change the code for Algorithm 1 so that its loop stops at 51, will this make Algorithm 1 fair? Justify your answer.

6.2.2 Write a C++ function template that randomizes the order of the objects in a subarray `a[first...last]`, where the objects can be of any data type. The prototype for the template should be as follows:

```
template <class otype>
void randomize (otype a[], int first, int last);
```

6.2.3 Here is an answer that has sometimes been given for Exercise 6.2.2:

```
template <class otype>
void randomize (otype a[], int first, int last)
{
  int k, r, s;
  for (k = first; k <= last; ++k)
  {
    r = randint(first, last);
    s = randint(first, last);
    swap(a[r], a[s]);
  }
}
```

(a) Show that if the length of the array is 2 (i.e., if `last` is `first + 1`), then this algorithm gives each of the two possible arrangements equal probability of occurring.

(b) Show that if the length of the array is 3 (i.e., if `last` is `first + 2`), then the algorithm is biased, i.e., it does *not* give each of the $3! = 6$ possible arrangements equal probability of occurring. You may solve this problem either by drawing a tree diagram or by using a counting and divisibility argument.

(c) Show that for all arrays of length $n \geq 3$, the randomization algorithm above is biased.

Shuffling Algorithm 3:

```
int j, k;
for (k = 1; k <= 51; ++k)
{
  j = randint(k, 52);     // Produce a random integer i in the range  k...52.
  aux[k] = deck[j];       // Copy deck[j] into the auxiliary array.
  deck[j] = deck[k];      // Overwrite  deck[j]  location with the first card of the
}                         // subarray  deck[k...52], and now consider the deck
                          // to be reduced to the subarray  deck[k+1...52].
for (k = 1; k <= 51; ++k) // Copy the cards in the aux array back to deck.
  deck[k] = aux[k];
```

Fig. 6.4 The third shuffling algorithm

Shuffling Algorithm 4:

```
int j, k;
for (k = 1; k <= 52; ++k)
{
  j = randint(k+1, k+51);  // Produce a random integer in the range k+1... k+51.
  if (j > 52)              // If the  j  value is beyond the end of the deck,
    j = (j % 52);          // "wrap around" to the beginning of the deck.
  swap (deck[k], deck[j]);
}
```

Fig. 6.5 The fourth shuffling algorithm

6.2.4 Consider the algorithm in Fig. 6.4 for the "`shuffle`" function of the "`deck_type`" class. It uses an auxiliary array `aux[1..51]`.

(a) Show that it produces a fair shuffle.
(b) Compare the number of object moves (assignment statements that copy a card from a cell to a cell) in Algorithm 3 above with the number of moves in Algorithm 2 in Fig. 6.3, p. 300. Note that a swap requires 3 object moves.
(c) Compare the number of calls to `randint` in Algorithm 3 with the number in Algorithm 2.
(d) Compare the space requirements of Algorithm 3 and Algorithm 2.
(e) Both Algorithm 3 and Algorithm 2 can be generalized in a trivial way to randomize arrays of length n instead of length 52. Express the running time for each of these generalized algorithms in big-theta notation. Assume that a swap requires $\Theta(1)$ time and $\Theta(1)$ space.

6.2.5 Algorithm 1 in Fig. 6.2, p. 300, allows a cell to be swapped with itself. Is it this property that causes it to be a biased algorithm? So consider the algorithm in Fig. 6.5 for the "`shuffle`" function of the "`deck_type`" class. The idea of this modified algorithm is that for each cell in the array "`deck`", the algorithm randomly picks a *different* cell and swaps the contents of those two distinct cells.

Shuffling Algorithm 5:
```
int r, k;
for (k = 2; k <= 52; ++k)
{
  r = randint(1, k);       // Produce a random integer in the range 1...k.
  swap(deck[k], deck[r]);
}
```

Fig. 6.6 The fifth shuffling algorithm

As an example of how this works, suppose the loop index k reaches the value 5. We want an integer j *different from* 5 to be selected at random, and then we'll swap the cards at locations k and j. The body of the loop sets j to a randomly chosen integer from among the 51 integers in the set 6, 7, 8, . . . , 52, 53, 54, 55, 56. The last four numbers in the set are outside the array limits, so if j has one of those values, we take its remainder after division by 52, which produces one of the number 1, 2, 3, 4. Thus a cell number between 1 and 52 is produced, but it is *not* 5.

(a) Determine whether this algorithm produces a fair shuffle on a deck of 3 cards. That is, replace the number 52 everywhere in Algorithm 4 with the number 3, and replace 51 with 2. Does the algorithm operate so as to give each possible final arrangement an equal probability of occurring?
(b) Is Algorithm 4 in Fig. 6.5 fair on a deck of 52 cards?

6.2.6 Consider the Shuffling Algorithm 5 in Fig. 6.6. Note that it differs from Algorithm 2 in that the index of the object swapped with the object at index k is less than or equal to k instead of greater than or equal to k. Prove that this algorithm also produces a fair shuffling.

6.2.7 Consider the C++ algorithm in Fig. 6.7 that randomizes the order of the nodes of a NULL-terminated, singly linked list, following the idea in Algorithm 5 in Fig. 6.6. Analyze the best and worst case running times for your algorithm for lists of length n assuming that the swap requires $\Theta(1)$ time and $\Theta(1)$ space.

6.2.8 Suppose we have a sequential file F so large that we cannot allocate an array large enough to hold all the objects in the file. Suppose, however, that we can create an array that holds at least half of the objects in F. How can we randomize the order of the objects in F? One way is to read the first half of F into our array, randomize the order of the objects in the array (see Exercise 6.2.2), write those objects out to a temporary file G, read the second half of F into the array, randomize, and write out those objects to a file H. Then we can open both G and H simultaneously and combine them into one output file, call it K, by a kind of "randomized merge" of G and H. Figure 6.8 shows a "naive" algorithm for implementing the randomized merge.

```
// This is a function that randomizes a NULL-terminated linked list of nodes.
// The algorithm follows the same idea as Shuffling Algorithm 5.

void randomize_linked_list (node_ptr list)
{
  node_ptr p, q;
  long k, r, i;

  if (! list || !list->next)    // If the given list is empty or of length 1,
    return;                     // there is no work to be done.
  else
  {
    p = list->next;             // we don't need to swap the first node
    k = 1;                      // index of the node p

    while (p)
    {
      r = randint(0, k);        // index of the node to swap p with
      q = list;                 // move q to the node to swap p with
      for (i=0; i<r; i++)       // move q forward r steps
        q = q->next;
      swap(p->datum, q->datum); // swap the data in the nodes

      p = p->next;              // then advance p
    }
  }
}
```

Fig. 6.7 A list shuffling algorithm

```
void randomized_merge(istream &G, istream &H, ostream &K)
//Assumes that G and H are open for reading, K is open for writing
{
  while ("G and H are not empty")
  {
    int r = randint(1, 2); // "Toss a fair coin" to choose a file.
    if (r == 1)
      "read the next object from G and write it out to K";
    else  // r must be 2
      "read the next object from H and write it out to K";
  }

  if ("H is empty")
    "copy the remaining objects from G to K";
  else
    "copy the remaining objects from H to K";
}
```

Fig. 6.8 Merges two files in a randomized fashion

(a) Use the special case where the number of objects in the file F is $n = 4$ to show that this merge algorithm will bias the randomization process. Begin by assuming that the objects in F are a, b, c, d in that order. Then G will get a, b in one of two orders, and H will get c, d in one of two orders. Draw a tree diagram that branches initially to the four possibilities for G and H at the instant at which the merge algorithm above begins its work. Let further branches (based on the "coin tossing") show the possibilities for G, H, and K at each step. Calculate the probabilities of the various arrangements in K at the end of the process. Show that some arrangements are more likely than others.

(b) Consider the tree diagram in the solution for part (a). Suppose we change the probabilities on the "merging branches" so that at each step along the way the probability of choosing from each of the two intermediate files (G or H) is proportional to the number of unread objects remaining in that file. For example, when both G and H have the same number of unread objects remaining, the probability of choosing from G should be 1/2 and the same for H, but when G has only 1 unread object and H has 2, then the probability of choosing from G should be 1/3 and the probability of choosing from H should be 2/3. Show that in this case the randomization process is unbiased.

(c) Modify the pseudo-code in Fig. 6.8 so that at each step during the merge process the probability of choosing from each of the two intermediate files (G or H) is proportional to the number of unread objects remaining in that file. This can be shown to produce an unbiased randomization process. Use the following function prototype in which g and h are the initial sizes of (number of objects in) the files G and H.

```
//Assumes that G and H are open for reading, K is open for writing
void randomized_merge(istream &G, istream &H, ostream &K, int g, int h);
```

(d) Give a file randomization algorithm for taking care of the case where the file F is so large that *three* intermediate randomized files must be created and then merged. Your algorithm can *call* the function you wrote for part (c) if that is convenient.

6.2.9 (a) Suppose an unbiased randomization algorithm is applied to an array of length n (so that every possible arrangement of the objects in the array has equal probability $1/n!$ of occurring). Prove that each object in the original array has equal probability of being placed in any cell of the array when the algorithm is applied.

(b) Let's call that property the "any-object-can-go-anywhere" property. We have just noted that *if* a randomization algorithm is unbiased, *then* it must have the "any-object-can-go-anywhere" property. Some people assume that the converse must also be true: if a proposed randomization algorithm can be shown to have the "any-object-can-go-anywhere" property, then it must be unbiased. Consider, however, the algorithm in Fig. 6.9.

```
template <class otype>
void randomize (otype a[], int first, int last)
{
  int n = last - first + 1;     // Number of cells in a[first..last]
  otype temp[n];                // Need an extra storage array.
  int  r = randint(0, n-1);     // Generate a "shift distance."
  int  j, k;                    // j and k will run along the temp array
  for (j=0, k = first + r; k <= last; ++k)
    temp[j++] = a[k];           // Copy back part of "a" to front of "temp"
  for (k = first; k <= first + r - 1; ++k)
    temp[j++] = a[k];           // Copy front part of "a" to back of "temp"
  // Copy all of "temp" back to a[first..last]
  for (j = 0, int k = first;  k <= last; ++j, ++k)
    a[k] = temp[j++];
}
```

Fig. 6.9 A function randomizing an array

To illustrate the algorithm with an example, suppose array `a[first..last]` initially looks like

4	5	6	7	8
a	*b*	*c*	*d*	*e*

If the random number generator returns $r = 3$, then the algorithm will copy the objects in cells 7 and 8 into the first two cells of `temp` and the objects in cells 4, 5, and 6 into the last three cells of `temp` to produce this arrangement:

	0	1	2	3	4
`temp:`	*d*	*e*	*a*	*b*	*c*

Then all five cells would be copied back to the original array. The effect is to produce a “3-cell right shift with wrap-around”. As you can easily see, object *a* is equally likely to end up in any of the five cells of temp, and the same is true of the other four objects in the array. Thus in this example, and in general, the algorithm given above has the “any-object-can-go-anywhere” property. Explain why even so, the algorithm is badly biased.

6.3 Choosing a Random Sample from a Sequential File

Suppose we are given a large set of size N and asked to choose a random sample of size K from the set, i.e., a subset of size $K < N$. To be considered a random sample, the subset must be chosen by a process that gives every subset of size K an equal chance of being chosen. Since there are $\binom{N}{K}$ different subsets of size K in a set of size N , each of these subsets must be given the probability $1/\binom{N}{K}$ of being selected.

Why might someone want to choose a random sample of specified size from a large set, which we call a "population"? The answer generally is that a scientist or a commercial firm wants to obtain data about certain characteristics of the individuals or objects that make up the population, but the population is so large that it is impractical to study every member of the population. In such a case, an alternative is to choose as large a random sample as one can afford from the whole population, study the individuals of that sample, and then generalize the results to the entire population, using appropriate statistical techniques of estimation. Another situation, however, would be where a research team wants to select *randomly* half of a population (e.g., of cancer patients) on whom to try out a new and promising treatment.

Let's imagine that a scientist or marketing firm has come to us with a large file containing data records (perhaps names, addresses, and phone numbers) for all N individuals in the population to be studied. The data records in the file can be accessed sequentially only; that is, we do not have random access into the file. We are asked to write a program to select a random sample of K data records from the file and place them in another file where they will serve as the list of individuals to be studied as members of the sample. What is an efficient algorithm for choosing a truly random sample?

The first algorithm we'll examine assumes that the program is given the number N of data records in the file available as an input even before the file is read. That is, the program is told how many records the file contains before it begins reading the file. The algorithm chooses K distinct integers in the range from 1 to N and then reads the file, discarding all data records except those whose "number" was chosen. For example, if the file size is $N = 10$ (an unrealistically small size) and the required sample size is $K = 3$, and if the algorithm chooses the integers 2, 5, and 8, then the first 8 data records of the file will be read and the 2nd, 5th, and 8th records will be written to another file and will constitute the desired sample.

The algorithm we'll examine chooses the K integers by setting up a Boolean array with cells numbered from 0 to N and then generating random integers in the range from 1 to N, marking them in the Boolean array until K distinct integers have been marked. (Cell 0 is ignored in all this.) Code for the algorithm is given in Fig. 6.10. In the documentation for the algorithm we have used the phrase "data object" instead of the phrase "data record" (the word "object" is more general than "record" in computer science).

We have used the familiar file insertion « and extraction » operators from C++. The first question we should ask about Sample Select Algorithm 1 is whether it gives a valid random sample. That is, is the selection process unbiased, or do some subsets of size K have higher probability of being generated than do other subsets? The answer is that the process is unbiased.

Sample Select Algorithm 1

```
// This is a function template for selecting a random sample of K data objects
// from an open sequential file F known to contain N data objects.  The selected
// objects are written out to an open file G.  It is assumed that the stream
// insertion and extraction operators << and >> are overloaded for the otype
// class.  Error checking code has been omitted to avoid clutter.

template <class otype>
void select_sample (istream & F, int K, int N, ostream & G)
{
  bool in_sample[N+1];
  int i, r, count = 0;
  otype temp;

  for ( i = 1; i <= N; ++i )   // Initialize the in_sample array.
    in_sample[i] = false;

  while ( count < K )
  {
    r = randint(1, N);       // Randomly select the "file index" of an object
                             // in F. (The file index of the first object in F
    if ( ! in_sample[r] )    // is 1, the second object has file index 2, etc.
    {
      in_sample[r] = true;
      ++count;
    }
  }

  // When execution reaches this point, exactly K of the cells in the array
  // in_sample are marked true, and count has the value K.

  for (i = 1; count > 0; ++i)
  {
    F >> temp;               // Read the object with file index i from file F.
    if ( in_sample[i] )      // Was the i-th object in F selected for the sample?
    {
       G << temp;            // If so, write the i-th object into file G.
       --count;
    }
  }
}
```

Fig. 6.10 First algorithm for selecting a random sample from a file

Theorem 6.3.1 *Let K and N be positive integers, with $K \leq N$. Then for each subset R of size K of the set $\{1, 2, 3, \ldots, N\}$, Algorithm 1 has probability $1/\binom{N}{K}$ of choosing exactly the integers in the set R. Thus Algorithm 1 gives each subset of size K of the file F equal probability of being chosen.*

Proof Specify any particular subset R of size K of the set $\{1, \ldots, N\}$. Let A_i denote the event that the i-th number chosen by Algorithm 1 will be in R. We want to prove that

$$P(A_1 \cap A_2 \cap A_3 \cap \cdots \cap A_K) = 1/\binom{N}{K}.$$

We will do this by using the Chain Rule (Theorem 8.0.14): we will prove that

$$P(A_1)P(A_2|A_1)P(A_3|A_1 \cap A_2) \ldots P(A_K|A_1 \cap A_2 \cap A_3 \cap \cdots \cap A_{K-1}) = 1/\binom{N}{K}.$$

When the loop with control "`while (count <K)`" begins, the file index generated by the first call to `randint(1,N)` has probability K/N of being a member of our specified subset R. That is, $P(A_1) = \frac{K}{N}$.

The second time through the same loop, the call to `randint(1, N)` may produce the same file index that was returned by the first call, in which case that integer is ignored and another call is made, and if necessary another, and so on, until an integer different from the first is returned. What is the probability that this second integer will belong to R, given that the first chosen integer was in R? By Theorem 8.0.19, $P(A_2|A_1) = \frac{K-1}{N-1}$. Similarly, $P(A_3|A_1 \cap A_2) = \frac{K-2}{N-2}$. Etc. For the last number chosen, the probability that it will belong to R, given that the first $K-1$ integers chosen belong to R is $P(A_K|A_1 \cap A_2 \cap A_3 \cap \cdots \cap A_{K-1}) = \frac{K-(K-1)}{N-(K-1)} = \frac{1}{N-K+1}$. Multiplying all these probabilities together gives the desired probability:

$$P(A_1)\, P(A_2|A_1)\, P(A_3|A_1 \cap A_2) \ldots\, P(A_K|A_1 \cap A_2 \cap A_3 \cap \cdots \cap A_{K-1})$$
$$= \frac{K!}{N(N-1)(N-2)\ldots(N-K+1)} = \frac{K!\,(N-K)!}{N!} = \frac{1}{\binom{N}{K}}. \qquad \blacksquare$$

Algorithm 1 stops when it has read the r_K-th object of the input file, where r_K is the largest of the K random numbers marked by the algorithm. Usually r_K is smaller than, but close to, N, so the algorithm generally reads most but not all of its input file.

One disadvantage of Sample Select Algorithm 1 is that if N is large it will be necessary to create a large Boolean array. In some program execution environments, this may even be impossible.

A quick solution to this issue is to create an integer array of size K to hold the selected indexes (the value of `r` in Fig. 6.10). Since K is usually much smaller than N, this typically uses far less memory. This introduces a new difficulty: when generating a new file index to be selected, we need to check if it has been previously selected, to avoid repetition. While this was a trivial matter in the case of the Boolean array, with an index array, if we keep the selected indexes in a random order, we could have to check all of them for repetition for each new one that we introduce.

Thus, it is more efficient to keep the generated file indexes in order, and use binary search to check for repetition. The process of inserting a new element in a sorted array can be shown, however, to require $\Theta(K)$ operations on the average, so the algorithm has an expected time complexity that is $\Theta(K^2)$ in this version for the part

Sample Select Algorithm 2

```
// This is a function template for selecting a random sample of K data objects
// from an open sequential file F known to contain N data objects.  The selected
// objects are written out to an open file G.  It is assumed that the stream
// insertion and extraction operators << and >> are overloaded for the otype
// class.  This version uses less memory than Sample Select Algorithm 1.

template <class otype>
void select_sample_sort(istream & F, int K, int N, ostream & G)
{
  int sample[K+1];
  int pos, r, count = 0;
  bool found;

  while (count < K)
  {
    r = randint(1, N);

    binary_search(sample, r, 1, count, found, pos);

    if (!found)
    {
      // The following function call shifts each of the objects in the
      // subarray sample[pos, count] one cell to the right, opening up
      // a space into which the index r is then inserted.
      insert(sample, pos, count, r);
      ++count;
      // At this point, sample[1, count] contains the randomly chosen
      // array indexes, arranged in strictly increasing order.
    }
  }

  int i, j;
  for (i = 1, j = 1; i <= K; ++i)
  {
    for ( ; j <= sample[i]; ++j)
      F >> temp;                 // Read an object from input file F.
    G << temp;                   // Write out the object to file G if the
  }                              // the object's index is in sample[1,K].
}
```

Fig. 6.11 The second algorithm for selecting a random sample from a file

of selecting the K sample indexes. The process of reading the objects from the file F and writing them into the file G still has $\Theta(N)$ complexity on the average, just like for Algorithm 1. Sample Select Algorithm 2 in Fig. 6.11 presents an implementation of this idea, where the `binary_search` function is the one from Fig. 5.3 on p. 176. In terms of selected sample, though, the two algorithms are equivalent. This means that when given the exact same sequence of random numbers, they will produce the exact same sample from the file. Even with this optimization, a significant amount of memory is required for this algorithm to run.

Another disadvantage of Sample Select Algorithms 1 and 2 is that in the loop that selects K distinct integers, the random number generator may repeatedly "hit" file indexes that have been previously selected, which is a waste of time. This is not a serious drawback, however, if K is small relative to N, which is usually the case, because in this case the probability of duplicate "hits" is quite small. Later we will be able to calculate the average number of calls that will have to be made to the random number generator.

For these reasons it is important to know that there are algorithms that can operate "on-line" to select the objects of the random sample while objects are being read from the input file. The idea is to read the objects from the file one by one and use a random process to decide, as each is read, whether to retain it in the random sample. How is this decision made?

By Theorem 8.0.11, if a random sample of size k is selected from a population of size N, then each member of the population has the probability $\dfrac{K}{N}$ of being in the sample. If we want to read objects one by one from an input file of known size N and choose a random sample of size K as we go, then each object should have probability K/N of being selected. Thus, we give the first object read that probability of being selected. If it is selected, then we need a random sample of size $K-1$ from the remaining $N-1$ objects, so we give the second object the probability $(K-1)/(N-1)$ of being selected. However, if the first object is not selected, then we still need a random sample of size K from the remaining $N-1$ objects, so we give the second object the probability $K/(N-1)$ of being selected. At each remaining stage in the process, if M objects remain to be read from the input file, and if we still need J more objects for the sample, then we give the next object the probability J/M of being selected. Code for this algorithm is given in Fig. 6.12. In the code, we do not use the variables J and M. Instead, we vary N and K so that at every stage they tell us the number of objects remaining unread in the file and the number of objects that are still needed for the random sample.

Although it is possible that Sample Select Algorithm 3 will end up reading the entire file, the probability of that happening is quite small if K is much less than N. The question of what fraction of the file will be read, on the average, will be explored later.

Does Algorithm 3 give a valid random sample? That is, is the algorithm unbiased? Let's look at a particular numerical case. Suppose the file to be read is known in advance to contain $N = 12$ objects. Suppose we want a random sample of size $K = 3$. What is the probability that the 2nd, 5th, and 8th objects will be selected? (It should be $1/\dbinom{12}{3}$). The tree diagram in Fig. 6.13 will help us calculate this probability.

In the tree diagram in Fig. 6.13, only a part of the entire tree is shown. Enough of the tree is shown to allow us to calculate easily the probability that the 2nd, 5th, and 8th objects will be selected.

Let A_i denote the event that the i-th object in the file gets selected. Then we want to compute $P(A_1^c \cap A_2 \cap A_3^c \cap A_4^c \cap A_5 \cap A_6^c \cap A_7^c \cap A_8) = P(A_1^c) \cdot P(A_2|A_1^c) \cdot P(A_3^c|A_1^c \cap A_2) \cdot P(A_4^c|A_1^c \cap A_2 \cap A_3^c) \ldots P(A_8|A_1^c \cap \cdots \cap A_7^c) = \frac{9}{12}\frac{3}{11}\frac{8}{10}\frac{7}{9}\frac{2}{8}\frac{6}{7}\frac{5}{6}\frac{1}{5} = \frac{9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 3!}{12 \cdot 11 \cdot 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5} = \frac{9!\,3!}{12!} = \frac{1}{\binom{12}{3}}$ as desired.

Sample Select Algorithm 3

```
// This is a function template for selecting a random sample of K data objects
// from an open sequential file F known to contain N data objects.  The selected
// objects are written out to an open file G.  It is assumed that the stream
// insertion and extraction operators << and >> are overloaded for the otype
// class.  Error checking code has been omitted to avoid clutter.
// The value parameters N and K will be modified inside the function.
// At all times N will tell us how many objects remain unread in file F,
// and K will tell how many objects are still needed for the sample.

template <class otype>
void select_sample (ifstream & F, int K, int N, ofstream & G)
{
  otype temp;
  int r;

  while (K > 0) // while we still need more objects for our sample
  {
    F >> temp;           // Read the next object from input file F.
    r = randint(1, N);
    if (r <= K)          // Give the object probability K/N of being chosen.
    {
      G << temp;         // Write the chosen object to file G.
      --K;               // Decrease the count of objects needed for the sample.
    }
    --N; // Decrease the count of objects remaining unread in input file F.
  }
}
```

Fig. 6.12 The third algorithm for selecting a random sample from a file

The example given above does not constitute a *proof* that Algorithm 3 works correctly in all cases, but anyone who studies the example should be able to see why the algorithm is a correct one. Writing out all the details is tedious, however.

Algorithms 1, 2, and 3 require that the number N of objects in the input file be known before they start work. In many cases, that would mean that the entire input file would have to be read and counted before the `select_sample` function was called to read through it a second time. It is an amazing fact that it is possible to do the selection "on line" *without even knowing the size N of the file in advance!* All the algorithm needs to know is the desired sample size K. Here is a description of this wizard algorithm, known as *reservoir sampling* [11].

Begin by reading in the first K objects from the input file `F` and placing them in an array `a`, which we'll call our "sample array". (If it turns out that there are fewer than K objects in the file, report that it is impossible to take a sample of size K.) Consider these K objects to be the "initial sample".

Now read the next object, call it F_{K+1}, from file `F`. (If there is no next object, the initial sample is the only possible sample of size K.) Now we have $K + 1$ objects, which is too many. We must reject one of these $K + 1$ objects, and each must have an equal chance of being rejected. Thus we give each object of the initial sample and also the object F_{K+1} the probability $1/(K + 1)$ of being rejected. If F_{K+1} is not

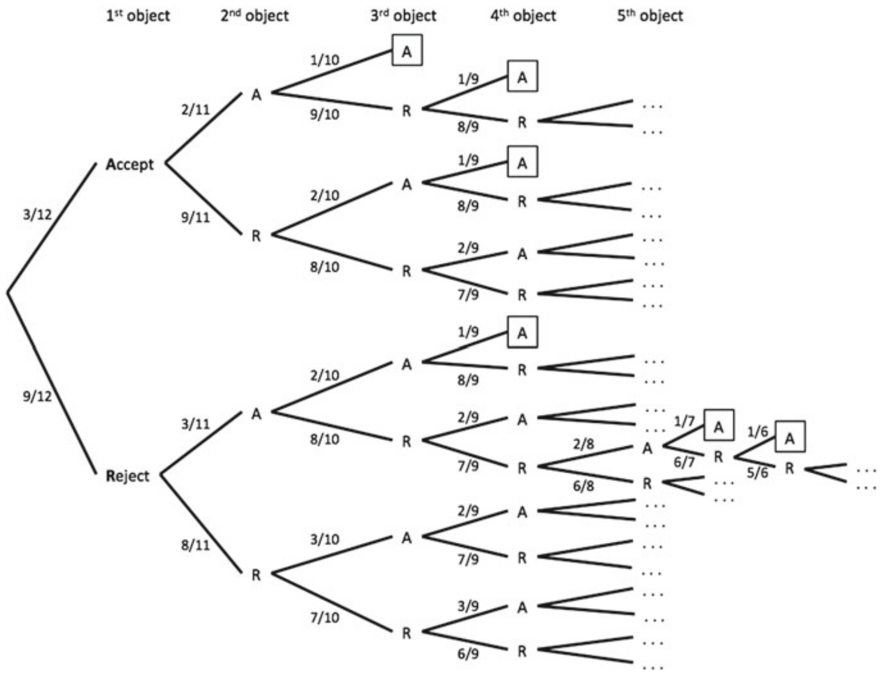

Fig. 6.13 Partial decision tree for the Algorithm 3

being rejected, then we overwrite the rejected object with F_{K+1}. The objects now in the sample array have not been rejected, and they form the new "current sample".

Note that each of the first $K + 1$ objects has been given probability $K/(K + 1)$ of being in the current sample. This is a special case of the more general case: if it turns out that there are N objects in the file, and if we are to choose a random sample of size K, then by Theorem 8.0.11, each object in the file must have a priori probability K/N of ending up in the sample. By the phrase "a priori probability", we mean a non-conditional probability, i.e., a probability, before the experiment has begun, of a particular event occurring.)

If file `F` is not completely read, then read the next object, call it F_{K+2}, from `F`. Now we must decide whether to keep F_{K+2} as part of our sample. By the preceding paragraph, we know that if the file turns out to have $K + 2$ objects (i.e., if N turns out to be $K + 2$), then we should give F_{K+2} the probability $K/(K + 2)$ of being chosen for the sample, and if it is chosen we will need to select one of the previously chosen K objects to be ejected from the sample. Each of those K objects should have an equal probability of being ejected, so we must give each one the probability $\frac{1}{K} \cdot \frac{K}{K+2}$ of being ejected, i.e., 1 chance in $K + 2$. Using this idea, select an object to reject, and if it is not F_{K+2}, then overwrite the object being rejected with F_{K+2}.

Sample Select Algorithm 4

```
// This is a function template for selecting a random sample of K data objects
// from an open sequential file F containing an unknown number of data objects.
// The selected objects are placed in an array a[1..K] that's dynamically
// allocated by the function.  The function returns false if there are fewer
// than K objects in F; otherwise it returns true. It is assumed that the
// stream extraction operator >> is overloaded on the class otype.

template <class otype>
bool select_sample (int K, istream & F, otype & a[])
{
  int M = 0;              // M counts the objects that have been read from F.
  a = new otype[K+1];     // Create an array in which to build the sample.

  while (M < K)           // Try to read K objects initially from F into a[1..K].
  {
    ++M;
    F >> a[M];
    if (F.eof())          // If the end of file F has been reached, exit from the
      return false;       // function because F contained fewer than K objects.
  }

  // If execution reaches this point, the array a[1..K] holds the first K
  // objects read from F, and M has the value K.

  F >> a[0];              // Try to read the (M+1)-st object into spare cell a[0].
  while (! F.eof())       // This loop will halt when an attempt to read an object
  {                       // fails because the end of file F has been reached.
    r = randint(1, M+1);
    if (r <= K)           // If r <= K we reject a[r] by overwriting it with a[0].
      a[r] = a[0];
                          // (If r > K we consider a[0] to be rejected.)
    ++M;                  // Increase the count of objects already read from F.
    F >> a[0];            // Try to read the next object (if any) from F.
  }

  return true;            // The function succeeded in selecting a sample.
}
```

Fig. 6.14 The fourth algorithm for selecting a random sample, this time from a file of unknown size

In general, suppose we have read M objects from the file and have constructed a current sample in which each object had a priori probability K/M of being in the current sample. If the file has not been completely read, then read the next object, call it F_{M+1}. This object must be given probability $K/(M+1)$ of being chosen for the sample, which is to say that with probability $K/(M+1)$ some object of the current sample must be ejected. Giving each an equal probability of being ejected means that each object of the current sample must have probability $1/(M+1)$ of being rejected. The algorithm is given in Fig. 6.14.

In those cases where $N \geq K$, does Sample Select Algorithm 4 give a valid random sample? That is, is the algorithm unbiased? Let's look at a specific example to give us some insight into the process. Suppose the size N of the file F is 12 objects, and suppose $K = 3$. That is, the algorithm is required to choose a random sample of size 3 from the file without knowing until reaches the end of the file that there are 12 objects in the file. Let's compute the probability that the set of file objects $\{F_2, F_5, F_8\}$ will be chosen.

The algorithm begins by reading file objects F_1, F_2, and F_3 into cells 1, 2, and 3 of the array `a[0..3]`. At this exer we can define the following events:

- let E_4 denote the event "F_2 will be in the sample after F_4 is read from the file and decided on"
- let E_5 denote the event "F_2 and F_5 will be in the sample after F_5 is read from the file and decided on"
- let E_6 denote "F_2 and F_5 will be in the sample after F_6 is read from the file and decided on"
- let E_7 denote "F_2 and F_5 will be in the sample after F_7 is read from the file and decided on"
- let E_8 denote "F_2, F_5, F_8 will be in the sample after F_8 is read from the file and decided on"
- let E_i denote "F_2, F_5, F_8 will be in the sample after F_i is read from the file", $i = 9, 10, 11, 12$.

We want to prove that $P(E_4 \cap E_5 \cap E_6 \cap \cdots \cap E_{12}) = 1/\binom{12}{3}$, which we can do by using the Chain Rule. We'll prove that

$$P(E_4)\ P(E_5|E_4)\ P(E_6|E_5 \cap E_4) \ldots P(E_{12}|E_4 \cap \cdots \cap E_{11}) = 1/\binom{12}{3}.$$

When F_4 is read from file F it is placed in array cell `a[0]`. Then a random number r is generated in the range 1–4 to determine whether F_4 will be selected or discarded. If r turns out to be ≥ 1 then we will overwrite `a[r]` with `a[0]`, thereby putting F_4 into the sample and ejecting F_r ; but if r is 0 we will do nothing, which will leave the sample unchanged. Thus the probability that F_2 will remain in the sample is 3/4. That is, $P(E_4) = 3/4$.

Now suppose E_4 occurs: F_2 remains in the sample when F_4 is read. The algorithm will go on to read F_5 into `a[0]` and then generate a random number r in the range 0...4 to determine whether F_5 will be selected or discarded. We now want to compute $P(E_5|E_4)$, which is the probability that if F_2 remains in the sample just before F_5 is read, then F_5 will be added to the sample. This occurs when r is either 1 or 3, which tells us that $P(E_5|E_4) = 2/5$.

Now suppose E_4 and E_5 have occurred. The algorithm will read F_6 into `a[0]` and generate r in the range 1...6 to determine whether F_6 will be selected or discarded. The event E_6 will occur if and only if $r \geq 4$ or r is the subscript of the cell containing neither F_2 nor F_5, so $P(E_6|E_4 \cap E_5) = 4/6$.

Similar reasoning yields the following values:
$P(E_7|E_4 \cap E_5 \cap E_6) = 5/7$
$P(E_8|E_4 \cap E_5 \cap E_6 \cap E_7) = 1/8$
$P(E_9|E_4 \cap E_5 \cap E_6 \cap E_7 \cap E_8) = 6/9$

$P(E_{10}|E_4 \cap E_5 \cap E_6 \cap E_7 \cap E_8 \cap E_9) = 7/10$
$P(E_{11}|E_4 \cap E_5 \cap E_6 \cap E_7 \cap E_8 \cap E_9 \cap E_{10}) = 8/11$
$P(E_{12}|E_4 \cap E_5 \cap E_6 \cap E_7 \cap E_8 \cap E_9 \cap E_{10} \cap E_{11}) = 9/12$
Multiplying all these values together gives

$$P(E_4)\ P(E_5|E_4)\ P(E_6|E_5 \cap E_4) \dots P(E_{12}|E_4 \cap \cdots \cap E_{11})$$
$$= \frac{3 \cdot 2 \cdot 4 \cdot 5 \cdot 1 \cdot 6 \cdot 7 \cdot 8 \cdot 9}{4 \cdot 5 \cdot 6 \cdot 7 \cdot 8 \cdot 9 \cdot 10 \cdot 11 \cdot 12} = \frac{9!\,3!}{12!} = \frac{1}{\binom{12}{3}}.$$

Again, the calculations we have done in the example above do not constitute a proof of the correctness of Algorithm 4, but the ideas for the proof can be extracted from the example.

6.3.1 Exercises

6.3.2 Compare the four sample select algorithms in terms of the minimum and maximum number of objects they need to read from the file F, and the minimum and maximum number of times the function `randint` is called.

6.3.3 Analyze the minimum and maximum execution times for Algorithms 1, 2, 3, and 4 for selecting a random sample. Express your answers in terms of K and/or N.

6.3.4 Suppose we have a sequential file containing $N = 5$ objects a, b, c, d, and e in that order. Suppose we want to select a random sample of size $K = 3$. There are $\binom{5}{3}$ different possible samples.

(a) List all the various possible samples.
(b) Draw the decision tree for Algorithm 3 in this case, and label its branches with appropriate probabilities. Verify that each of the possible samples you listed in part (a) has probability $1/\binom{5}{3}$ of being chosen by Algorithm 3.
(c) Draw the decision tree for Algorithm 4 in this case, and label its branches with appropriate probabilities. (Each node in the tree should represent a "sample so far". If K were equal to 2, for example, initially the "sample so far" would consist of objects a and b. After another object is read and processed, there would be three possible "samples so far": $\{a, b\}$, $\{a, c\}$, $\{b, c\}$ etc.) Verify that each of the possible samples you listed in part (a) has probability $1/\binom{5}{3}$ of being chosen by Algorithm 4.

6.3.5 Suppose we have a sequential file containing $N = 4$ objects a, b, c, d in that order. Suppose we want to select a random sample of size $K = 2$. There are $\binom{4}{2}$ different possible samples.

(a) List all the various possible samples.

(b) Suppose Algorithm 1 is used to choose the random sample. What is the probability that only two calls will be made to the random number generator? What is the probability that exactly three calls will be made? What is the probability that exactly four calls will be made? That exactly n calls will be made?
(c) Draw the decision tree for Algorithm 3 in this case, and label its branches with appropriate probabilities. Verify that each of the possible samples you listed in part (a) has probability $1/\binom{4}{2}$ of being chosen by Algorithm 3.
(d) Draw the decision tree for Algorithm 4 in this case, and label its branches with appropriate probabilities. (Each node in the tree should represent a "sample so far". Initially, for example, the "sample so far" consists of objects a and b. After another object is read and processed, there are three possible "samples so far": $\{a, b\}$, $\{a, c\}$, $\{b, c\}$ etc.) Verify that each of the possible samples you listed in part (a) has probability $1/\binom{4}{2}$ of being chosen by Algorithm 3.

6.3.6 Suppose a random sample of size 2 is to be selected from a population of 10 objects using Algorithm 3. Show using a (partial) tree diagram how to compute the probability that the 2nd and 7th objects in the file will be the ones selected for the sample. Is the probability what it ought to be? Explain.

6.3.7 Suppose that the data objects for a population from which a random sample is drawn are held in an array instead of a sequential file. This means that any algorithm for choosing the random sample will have "random access" into the set of data objects. Devise an algorithm similar to Algorithm 1 for choosing the random sample, but without using an auxiliary Boolean array and without the drawback of having some random numbers "wasted" by duplicate "hits". Draw a decision tree for your algorithm when the population size is 5 and the sample size is 2, and use the tree to determine whether your algorithm is fair or biased in this special case.

6.3.8 Suppose a sequential file of size N is known to be in random order. Suppose we wish to select a random sample of size K from the file, where $K \leq N$. Prove that if we simply take the first K objects from the file, then we have a random sample.

6.3.9 It is not hard to see that in the first half of Algorithm 1, the repeated calls to the random number generator eventually produce a random sample of K integers from the set $\{1, 2, 3, \ldots, N\}$. (If you are skeptical of that idea, imagine that the file containing the population consists of the integers $1, 2, 3, \ldots, N$ in that order. Then the integers r_i chosen by the random number generator will select "themselves" out of the population file.) Thus we know that each of the $\binom{N}{K}$ possible K-sized subsets of the set $\{1, 2, 3, \ldots, N\}$ are equally likely to be chosen during execution of this algorithm.

(a) What is the probability that the largest integer chosen in this way will be K?
(b) What is the probability that the largest integer chosen in this way will be $K + 1$?
(c) What is the probability that the largest integer chosen in this way will be $K + 2$?

(d) What is the probability that the largest integer chosen in this way will be N?
(e) Let A_x denote the event that the largest integer chosen by Algorithm 1 will be x, where x is an integer in the range $K \le x \le N$. Compute $P(A_x)$ for all such integers x. If your formula for $P(A_x)$ is correct, you should be able to show that the sum $\sum_{x=K}^{N} P(A_x)$ is equal to 1. Use Exercise 2.3.51, p. 52, to help you show that this sum is indeed equal to 1.

6.3.10 Recall that when an unbiased sampling algorithm is applied to a population of size N to select a sample of size K, every subset of size K must be given equal probability $\frac{1}{\binom{N}{K}}$ of being selected. Theorem 8.0.11 implies that if a sampling algorithm in unbiased, then when it is applied to a population of size N to select a sample of size K, each member of the population will have probability $\frac{K}{N}$ of ending up in the sample. Some people mistakenly assume that the converse must also be true: if a proposed sampling algorithm can be shown to give each member of the population of size N the probability $\frac{K}{N}$ of being selected property for a sample of size K, then it must be unbiased. Consider, however, the algorithm in Fig. 6.15 for choosing a random sample of size $K < N$ from an array `a[0,N-1]`.

To illustrate the algorithm with an example, suppose array `a[0..6]` initially looks like

0	1	2	3	4	5	6
a	b	c	d	e	f	g

Here $N = 7$. Suppose the function above is called with $K = 3$. If the random number generator returns $r = 2$, then the algorithm will select objects c, d, and e to be written to the output file. If, instead, the random number generator returns $r = 5$, then the algorithm will select objects f, g and a to be written to the output file. It is easy to see that in this example, each object will have probability $\frac{3}{7}$ of being selected for the random sample. The same is true in general for the algorithm above. Explain why even so, the algorithm is badly biased when $2 \le K \le N - 2$.

6.4 Simulating Observations of a Discrete Random Variable

Let X denote the sum of the two faces that turn up when a pair of fair dice is thrown. Then X is a random variable, as described in Definition 8.0.26. Suppose we are asked to write a C++ function that returns simulated observations X. The most “natural”

```
template <class otype>
void select (otype a[], int N, int K, file F)
{
  int  r = randint(0, N-1), i;     // Generate a ‘‘starting point"
  for (i = r; i <= N-1 && K > 0; ++i)
  {
    F << a[i] // Write a[i] to the file F.
    --K;
  }
  if (K > 0)   // Wrap around; get remaining objects from front.
    for (i = 0; K > 0; ++i)
    {
      F << a[i]; // Write a[i] to the file F.
      --K;
    }
}
```

Fig. 6.15 A seriously bad sample selection algorithm

```
int sum_of_faces()
{
  int r = randint(1, 6);
  int s = randint(1, 6);
  return  r + s;
}
```

Fig. 6.16 Simulating a pair of fair dice

way to do this is to have the function make two calls to a random number generator to simulate the numbers on the two faces and then return the sum of those two values:

Calls to random number generators with parameters, for example our `randint` function, are somewhat time-consuming because of the amount of arithmetic involved and the fact that they call other generators. For this reason, when writing a function that involves randomness, programmers often try to use algorithms that minimize the number of calls to random number generators. To illustrate how this can sometimes be accomplished, we're going to replace the algorithm for the `sum_of_faces` function in Fig. 6.16 with one that makes just one call instead of two to the `randint` function. The technique uses the values of the c.d.f. of the random variable X. (See Definition 8.0.22 for c.d.f., the cumulative distribution function, for which we have the notation $F_X(x)$.

To see how this is going to work, look at the graph of the c.d.f. in Fig. 6.17. The values of the c.d.f. of X are plotted along the vertical axis. The gap between 0 (c.d.f. of 2) and 1/36 (c.d.f. of 3) is exactly the probability that $X = 2$, the gap between 1/36 (c.d.f. of 3) and 3/36 (c.d.f. of 4) is the probability that $X = 3$, and so on. The gap between 35/36 (c.d.f. of 11) and 1 (c.d.f. of 12) is the probability that $X = 12$. This suggests the following strategy: generate a random real number u in the interval $[0.0, 1.0) = \{t : 0.0 \le t < 1.0\}$. If u lies in the interval $[0.0, 1/36)$, i.e., if $0.0 \le u < 1/36$, then interpret this as the simulated observation "$X = 2$". If u lies in the interval $[1/36, 3/36)$, i.e., if $1/36 \le u < 3/36$, interpret this as "$X = 3$" etc.

To make a C++ function do this, we make it first set up the following static arrays:

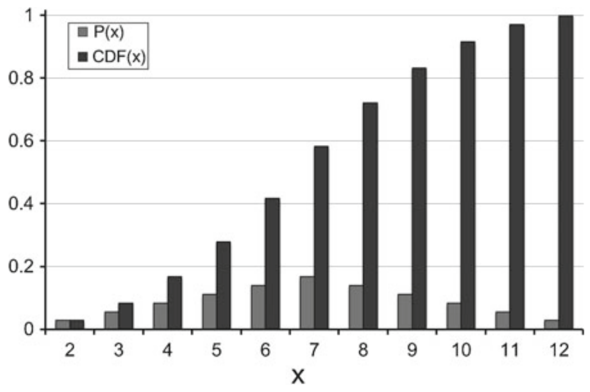

Fig. 6.17 Probability and c.d.f. for the sum of a pair of dice

	0	1	2	3	4	5	6	7	8	9	10
`cdf:`	1/36	3/36	6/36	10/36	15/36	21/36	26/36	30/36	33/36	35/36	36/36
	0.0278	0.0833	0.1667	0.2778	0.4167	0.5833	0.7222	0.8333	0.9167	0.9722	1.0
`X_value:`	2	3	4	5	6	7	8	9	10	11	12

Then whenever the function is called it can generate u in the interval [0.0, 1.0), search the array `cdf` for the left-most cell containing an entry larger than u, look up the corresponding value of X in the second array, and return that value. For example, if the value 0.714 is generated for u, then a linear search of cdf starting at the left end would reveal that 26/36 is the first number in the array larger than 0.714. Using the subscript 6 discovered during the linear search, the corresponding number in the `X_value` array would be found to be 8, and so 8 would be returned as the simulated value of X on that call to the function. Some code to do all this is shown in Figs. 6.18 and 6.19.

We can generalize the ideas in the preceding example to simulate observations of any discrete random variable X with a prescribed set of probabilities. We compute and store the possible values of X in one array and the corresponding values of $F_X(x)$ in a parallel array. Suppose the possible values of X are $x_0 < x_1 < x_2 < x_3 < \cdots < x_{n-1}$. Then the arrays would look like

	0	1	2	3	…	$n-1$
`X_value:`	x_0	x_1	x_2	x_3	…	x_{n-1}
`cdf:`	$F_X(x_0)$	$F_X(x_1)$	$F_X(x_2)$	$F_X(x_3)$	…	1.0

If X can have infinitely many different discrete values, then we must, of course, make the array finite, but make it sufficiently long that all values of $F_X(x_i)$ are very nearly equal to 1.0 when $x_i > x_{n-1}$. Also, it is critical to make sure that the last cell contains the value 1.0.

```
int sum_of_faces()
{
  static bool    called_previously = false;
  static double  cdf[11];
  static int     X_value[11];
         int     i;
  if (!called_previously)      // Must set up all the arrays.
  {
    called_previously = true;  // Make sure this code is executed
                               // only once.
    cdf[0]     = 1.0/36.0;
    X_value[0] = 2;
    int i;
    for (i = 1; i <= 5; ++i)
    {
      cdf[i] = cdf[i-1] + (i+1)/36.0;
      X_value[i] = i + 2;
    }
    for (i = 6; i <= 9; ++i)
    {
      cdf[i] = cdf[i-1] + (11-i)/36.0;
      X_value[i] = i + 2;
    }
    cdf[10] = 1.0;  // Make sure the last cell contains 1.0.
    X_value[10] = 12;
  }
  return select_event(cdf, X_value);
}
```

Fig. 6.18 Simulating the sum of the faces of two dice using the c.d.f.; the function `select_event` is shown in Fig. 6.19

```
// Note that we do not need to know the size of the cdf array because the last
// element will always be equal to 1. The X_value array has the same size.
int select_event(double cdf[], int X_value[])
{
  double u = randouble();  // Generate real number in [0.0,1.0).
  int i = 0;
  while (u >= cdf[i])      // Linear search for first cell containing
    ++i;                   // a c.d.f. value greater than u.
  return X_value[i];
}
```

Fig. 6.19 Function selecting a simulated event based on a pre-calculated c.d.f. array

Once the arrays are constructed, we can generate one simulated observation of X by carrying out the following steps:

(1) Use a random number generator to generate a value u for the uniform random variable U on the interval [0.0, 1.0).
(2) Find the smallest integer i such that $u < F_X(x_i)$.
(3) Return the number x_i as the simulated observation of X.

Example 6.4.1 Consider the experiment of tossing a fair coin until it comes up heads. Let X denote the number of tosses required to produce the first heads outcome. Then X has infinitely many possible values, namely $x_0 = 1, x_1 = 2, x_2 = 3, x_3 = 4, x_4 = 5, \ldots$ Suppose we want to use the computer to simulate observations of the random variable X. An inefficient way to do this would be mimic the coin tossing by repeatedly calling `randint(1,2)` and treating an outcome of 1 as heads and an outcome of 2 as tails. Thus one "run" of the experiment might generate the numbers 2, 2, 2, 1. The first three of these "tosses" are regarded as tails and the fourth is regarded as heads, so the simulated observed value of X for this run would be 4.

The method described in the preceding paragraph is very slow compared with the discrete random variable simulation method outlined in this section of the notes. Using the method outlined, we should set up two parallel arrays:

	0	1	2	3	...	$n-1$
`X_value:`	1	2	3	4	...	n
`cdf:`	$F_X(1)$	$F_X(2)$	$F_X(3)$	$F_X(4)$	...	1.0

Since this random variable X can assume infinitely many possible values, and since the cdf array must be finite, we choose n so large that $F_X(n)$ is extremely close to 1.0, and then we explicitly put 1.0 into cell $n-1$ in place of the true value $F_X(n)$, which means that the simulation will never generate a value greater than n. How large should we take n to be? A good choice might be to take n so large that the difference between $F_X(n)$ and 1.0 is less than one-billionth. How can we find the smallest integer n such that $1.0 - F_X(n) < 10^{-9}$ (one billionth)?

We begin by writing the general formula for $F_X(k), k = 1, 2, 3, 4, \ldots$ The values for the p.d.f. (probability density function; see Definition 8.0.26) are given by $f_X(1) = \frac{1}{2}$, $f_X(2) = \frac{1}{4}$, and in general $f_X(k) = \frac{1}{2^k}$. Thus the c.d.f. is given by

$$\begin{aligned} F_X(k) = P(\text{“}X \le k\text{”}) &= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^k} \\ &= \frac{1}{2} \cdot \left[1 + (1/2) + (1/2)^2 + \cdots + (1/2)^{k-1}\right] \\ &= \frac{1}{2} \cdot \frac{1-(1/2)^k}{1-(1/2)} = 1 - (1/2)^k. \end{aligned}$$

Now we can see that finding the smallest integer n for which $1.0 - F_X(n) < 10^{-9}$ is equivalent to finding the smallest integer n for which $1.0 - (1 - (1/2)^n) < 10^{-9}$, which simplifies to $(1/2)^n < 10^{-9}$. This is equivalent to $n \lg(1/2) < -9 \lg(10)$, which in turn is equivalent to $n > \dfrac{-9 \lg(10)}{\lg(1/2)}$. [Note: the direction of the inequality was reversed because we divided both sides of the preceding inequality by the *negative* number $\lg(1/2)$.] A fairly easy computation shows that $\dfrac{-9 \lg(10)}{\lg(1/2)} \sim 29.89$. The smallest integer n satisfying $n > 29.89$ is $n = 30$. Thus we can create a cdf array containing 30 cells indexed by the integers $0, 1, 2, 3, \ldots, 29$, and in the last cell (cell 29) we would put the value 1.0 instead of the actual value of $F_X(x_{29})$. The parallel arrays would look like this:

	0	1	2	3		...	29
`X_value:`	1	2	3	4		...	30
`cdf:`	0.5	0.75	0.875	0.9375		...	1.0

Suppose we set up the cdf array and then call to a uniform random number generator on the interval [0.0, 1.0). Suppose that call produces the floating exer value 0.802, what simulated value of X would that correspond to? It would be the smallest integer i such that $0.802 <$ `cdf[i]`. Looking at the numbers we calculated in above, we see that the smallest such integer i is 2. Thus a randomly generated floating point value of 0.802 would produce the simulated value $x_2 = 3$ for the random variable X.

In a situation like this, where the values of the random variable are successive integers in some particular range, it is easy to see that we can dispense with the `X_value` array. When our random number generator generates a particular floating point number u, we search the cdf array to find the smallest subscript i such that $u <$ `cdf[i]`, and then we simply return $i + 1$ as our value of the random variable X.

Finally, note that the `cdf` array is an *ordered* array of 30 numbers. In such a case it would be possible to use a binary search instead of a linear search to locate the smallest integer `i` such that $0.802 <$ `cdf[i]`. This requires a slight modification of the usual binary search function in which a specific target value is passed to the search function. We leave the details to the reader.

6.4.1 Exercises

6.4.2 Consider the statistical experiment in which a fair die is thrown repeatedly until one of the faces appears for the second time. An outcome of the experiment is a list of the faces that are thrown. Here are some possible outcomes of this experiment: (3, 6, 2, 6) ; (5, 5) ; (1, 4, 3, 6, 2, 4). Let Y denote the random variable that tells how many throws were necessary to produce a repetition of a face. For the three outcomes listed above, the corresponding values of Y would be 4, 2, and 6. Describe two different algorithms for simulating an observation of Y. The first algorithm should be the "naïve" one that makes repeated calls to a random number generator to represent the separate throws of the die. The second algorithm should be one that makes only one call to a random number generator and uses an array containing the c.d.f. values of Y, as described in this section. Calculate the p.d.f. and c.d.f of Y and answer this question: what value of Y will be produced if the call to `randouble()` returns the value $u = 0.6184$?

6.4.3 Suppose that a family with 5 children has a rule that says that every day the child to do the dishes will be chosen randomly with a probability directly proportionate to the number of hours they have spent playing games on the computer. Suppose that on one particular day we have the following distribution of hours: Joe 2 h, Bob 3.2 h, Sandy 1.5 h, Kim 2.5 h, Sue, 1.8 h.

(a) Calculate the probability function for the child to be chosen to do the dishes.
(b) Compute the c.d.f. from this function.
(c) Suppose that the function `select_event` in Fig. 6.19 is called to select the child to do the dishes, and that the value of the variable `u` is 0.65. Which child will be assigned the chore that day?

6.5 Randomized Algorithms

In this section we will discuss a category of algorithms relying on pseudo-random numbers to find the solution to a problem. The difference between these algorithms and the simulation algorithms that we've seen in the previous sections is that for this category of algorithms the answer is deterministic, while for the simulation algorithms, it is not.

More precisely, for a simulation or array shuffling algorithm, two separate executions of an algorithm on the same input data will likely result in a different outcome, which makes the answer non-deterministic. For randomized algorithms, for a specific problem, there is, at least theoretically, a single correct answer. The probabilistic aspects from one execution of the algorithm to the next are whether this solution is found or not, the precision of the solution, or the time it takes to reach this solution.

There are two major categories of randomized algorithms: Monte Carlo and Las Vegas. Monte Carlo algorithms are always fast, but they give the correct answer only with a given probability. Las Vegas algorithms will always give the correct answer, but they are not guaranteed to always be fast.

6.5.1 Monte Carlo Algorithms

Monte Carlo algorithms are guaranteed to be fast, but they may or may not find the correct solution. What makes them interesting for the users is that the probability that the correct answer is found can be computed and controlled through the parameters of the algorithm. In some cases, the probability of error can be so low, even within efficient execution time, that it can be considered as negligible for practical purposes. Thus, the execution time for these algorithms is deterministic, but the result may be wrong with a small probabilistic error.

The most common examples of Monte Carlo algorithms are for decision problems. These can be described as questions for which the answer is true or false, yes or no. For this type of problem, randomized algorithms can be true-biased or false-biased. If the answer given by a *true-biased* algorithm is *true* or *yes*, then this answer is surely correct. If the answer is false, then the answer given may be the correct answer with

```
// This function tests whether it is likely that n is prime. It does this by
// testing k positive integers a to determine whether they satisfy the
// equation a^(n-1) mod n = 1 of Fermat's Little Theorem. The boolean value
// returned by the function may be incorrect. If k is large, the probability
// of an incorrect answer is extremely small.

bool is_prime(int n, int k)
{
  int i;
  for (i=0; i<k; i++)
  {
    a = randint(2, n-1);
    if (power(a, n-1) % n != 1)
       return false;
  }
  return true;
}
```

Fig. 6.20 Monte Carlo Primality Test algorithm

a given probability. For *false-biased* algorithms, an answer of *false* or *no* will always be correct, while the opposite is correct only with a given probability.

Primality Test. For an example of such an algorithm, Fig. 6.20 shows a randomized algorithm that tests whether a number is prime. Here the function `power` is the common function found in many languages, such as the one provided by the `cmath` library in C++. This algorithm is based on Fermat's Little Theorem.

Theorem 6.5.1 (Fermat 1640). *If the integer p is prime, then for every integer a satisfying $1 \le a < p$,*

$$a^{p-1} mod\ p = 1.$$

In other words, if the value n of the parameter `n` of the algorithm in Fig. 6.20 is a prime number, then any number a between 1 and $n - 1$ raised to the power $n - 1$ will yield the remainder 1 when divided by n. The algorithm selects a number of candidates a for the test, and checks if the property holds for each of them. If we find any such number a for which the property is *not* satisfied, then it is certain that the number n is not a prime. Thus, this algorithm is *false-biased.*

What is important now is to figure out is how accurate a *true* answer is. In other words, if the algorithm returns the value `true`, stating that the number `n` is prime, how much confidence can we have that this is indeed the case? A second theorem related to prime numbers will provide the answer to this question.

Theorem 6.5.2 *If the integer p is not a prime, then at most half of the integers a from 1 to $p - 1$ satisfy the equation stated in* Theorem 6.5.1.

What this second theorem tells us is that if we run the test once ($k = 1$) with the number n not being prime, we have at least 50 % probability that we will find one of the numbers that do not satisfy the property, thus getting the correct answer. This means that in general, if the algorithm returns `true`, then the probability of error is at most 50 %.

Now let's consider the case where we run the test with $k = 2$. Since the two random numbers a are chosen independently of each other, the chances that we find one that satisfies the equation in Theorem 6.5.1 twice if the number n is not prime are reduced to 25 %. We can simply multiply the probabilities of the two independent events to get to this result. Thus, the probability of error in the case of a `true` answer decreases to $1/4$. By generalization, for a given value of the parameter k, the probability of error in the case where the algorithm returns the value `true` is $1/2^k$. Thus, the error can easily be made exponentially low, and even for $k = 10$, it is negligible.

This is a remarkable result that allows us to avoid testing whether the number n is divisible by any number between 2 and its square root, an operation that would be a lot more costly. Even though an efficient deterministic primality test algorithm has been discovered, the Monte Carlo primality test is still the one most used for this purpose in cryptography.

6.5.2 Las Vegas Algorithms

Las Vegas algorithms are guaranteed to give the correct answer in all the cases, but their execution time is probabilistic. They are generally fast, and the probability of an efficient execution time can be computed and in some cases even controlled through the parameters of the algorithm. Thus, even though they are generally expected to be fast, and will always give the correct answer, it can happen (with a low probability) that they take a long time to give the answer.

For an example, consider the case of a Quicksort choosing the pivot for partitioning randomly. This method is more likely to yield the expected complexity of $\Theta(n \lg(n))$ than the standard deterministic version, where n is the size of the array to be sorted. The worst case complexity of $\Theta(n^2)$ can still occur, but its probability is much smaller. Thus, this algorithm will be fast with a high probability.

Another example would be a randomized search for a given target in an array of the form `a[0.. n-1]`, shown in Fig. 6.21. This algorithm chooses an element of the array randomly until either it exhausts all possibilities and returns failure, or until it finds the target. This type of search can be shown to be faster than the linear search on the average in the case where the array contains multiple occurrences of the target.

One of the most popular Las Vegas algorithms is related to randomized hashing functions for hash tables. This algorithm can be shown to be more efficient than binary search trees on the average.

```
bool las_vegas_search(int a[], int n, int target)
{
  int test;
  for (i=0; i<n; i++)
  {
    test = randint(0, n-1);  // Make sure that no repeating indexes are generated.
    if (a[test] == target)
       return true;
  }
  return false;
}
```

Fig. 6.21 Las Vegas Randomized Search algorithm

6.5.3 Exercises

6.5.3 Verify Theorem 6.5.1 for $p = 5$. Find a number a that does not satisfy the equation in Theorem 6.5.1 for $p = 10$. You may use a calculator for this.

6.5.4 Write a Monte Carlo algorithm that searches for a target in an array; it could be similar but not identical to the one in Fig. 6.21. Is this algorithm true-biased or false-biased? Calculate the probability of error of your algorithm as a function of the prescribed number of trials.

6.6 Expected Behavior of Algorithms

In this section we'll investigate the "theoretical average behavior" of a variety of algorithms. Recall the following definition from your study of probability theory.

Definition 6.6.1 Given a random variable X with the possible values $\{a_1, a_2, \ldots, a_n\}$, the *expected value* of X (informally, the *theoretical average* of X) is denoted by $E(X)$ and is defined by

$$E(X) = \sum_{i=1}^{n} a_i \; P(X = a_i),$$

where $P(X = a_i)$ is the probability that X will assume the value a_i.

Example 6.6.2 Suppose a linear search must be made for a target object stored in an unordered array `a[first..last]` using the algorithm in Fig. 6.22 (previously seen in Fig. 5.2 on p. 175). Assume that `otype` is a data type on which the non-equality operator != is defined.

Let n denote the number of cells in the array to be searched; i.e., $n = last - first + 1$.

```
template <class otype>
int location_by_linear_search (const otype a[], const otype &target,
                               int first, int last)
{
  while (first <= last && a[first] != target)
    ++first;     // Go to the next cell if target not found.
  return first;  // first will be last+1 if target is not present.
}
```

Fig. 6.22 Location of a target in an array

(a) Suppose a copy of the target object is present in the array, and suppose it is equally likely to be in any of the n locations. What is the *expected* number of times that the body of the loop will be executed during the search? What is the *expected* search time required by a successful search?

(b) Suppose the target value is not in the array (of course, this is not known in advance of the search). What is the *expected* number of times that the body of the loop will be executed during the unsuccessful search? What is the *expected* search time required by an unsuccessful search?

Solutions.

(a) Let X denote the number of times the body of the loop is executed during a successful search.

Possible values of X :	$x =$	0	1	2	3	...	$n-1$	
Values of the p.d.f.:	$f_X(x) =$	$\frac{1}{n}$	$\frac{1}{n}$	$\frac{1}{n}$	$\frac{1}{n}$	...	$\frac{1}{n}$	

$$\begin{aligned} E(X) &= 0 \cdot \tfrac{1}{n} + 1 \cdot \tfrac{1}{n} + 2 \cdot \tfrac{1}{n} + 3 \cdot \tfrac{1}{n} + \cdots + (n-1)\tfrac{1}{n} \\ &= \tfrac{1}{n}[0 + 1 + 2 + 3 + \cdots + (n-1)] = \tfrac{1}{n}\tfrac{(n-1)n}{2} = \tfrac{n-1}{2}. \end{aligned}$$

On average, we have to search approximately half of the array to find the target when it is present.

The search time, which is a random variable Y, is the sum of two components: the time, call it T_1 for the "function overhead", which is $\Theta(1)$, and the time, call it T_2, required to execute the loop. Since the body of the loop requires $\Theta(1)$ time each time it is executed, and each loop control operation requires $\Theta(1)$ each time it is executed, $T_2 = \Theta(X)$. That is, there exist positive constants c_1 and c_2 such that $c_1 X \le T_2 \le c_2 X$. By the properties of the expected value of a variable, $c_1 E(X) \le E(T_2) \le c_2 E(X)$. Since $E(X) = \Theta(n)$, it follows that $E(T_2) = \Theta(n)$. Then $E(Y) = E(T_1) + E(T_2) = \Theta(n)$.

(b) Let Z denote the number of times the body of the loop is executed during an unsuccessful search. Then Z is a constant random variable, with fixed value n.

```
template <class otype>
void binary_search (const otype a[], const otype & target, int first,
                    int last, bool & found, int & subscript)
{
  int mid;

  found = false;  // Will remain false until target is found.

  while (first <= last && !found)    // The value parameters "first"
  {                                  // and "last" are modified
    mid = (first + last)/2;          // during loop execution.
    if (target < a[mid])
      last = mid - 1;
    else if (a[mid] < target)
      first = mid + 1;
    else // Only remaining logical possibility: a[mid] matches target
      found = true;
  }

  if (found)
    subscript = mid;     // The location of "target".
  else
    subscript = first;   // This is the appropriate subscript to
                         // return if "target" is not present.
}
```

Fig. 6.23 Binary search function

Thus $E(Z) = nP(\text{"}Z = n\text{"}) = n \cdot 1 = n$. By an argument similar to the one in part (a), it follows that the expected search time required by an unsuccessful search is $\Theta(n)$.

Example 6.6.3 Suppose we have an ordered array of 10 objects. Suppose a binary search is to be made in the array to locate an object that's in the array. The binary search algorithm is shown in Fig. 6.23 (which appeared earlier in Fig. 5.3, p. 176).

If the target is in the array and is equally likely to be in any of the 10 locations, what is the expected number of times that the body of the loop will be executed? What is the expected number of `otype` *object comparisons* that will be required to find the target?

Solution. Let X denote the number of times the body of the loop will be executed. Let Y denote the number of object comparisons that will be made. For concreteness, assume that initially `first = 0` and `last = 9`. Then the search will begin with `mid = 4`. If the target is in cell `a[4]`, then this will be discovered the first time the body of the loop is executed; two object comparisons ("`target <a[4]`" and then "`a[4] <target`") will be made, both of which will return false . Thus in this case X will be 1 and Y will be 2. If, instead, it turns out that "`target <a[4]`" is true, then mid will be re-calculated to the value 1. If the target is in cell `a[1]`, then this will be discovered with 1 more execution of the body of the loop, which will make two more object comparisons. In this case X will be 2 and Y will be 3 (one

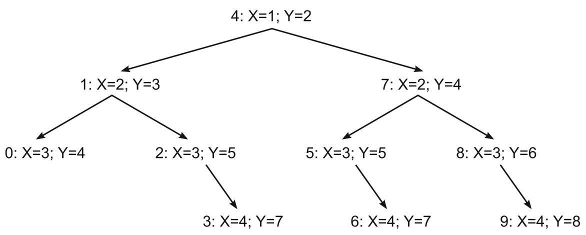

Fig. 6.24 Execution tree for a successful binary search

object comparison when `mid` is 4 and two more when `mid` is 1). And so on. The *logical* search tree in Fig. 6.24 gives the entire story. The label "6:X = 4; Y = 7" on a node means that if the algorithm finds the target in the cell `a[6]`, then the number of times the body of the loop is executed is 4, and the number of object comparisons required to make that discovery is 7.

x (possible values of X)	1	2	3	4
$f_X(x) = P(\text{“}X = x\text{”})$	$\frac{1}{10}$	$\frac{2}{10}$	$\frac{4}{10}$	$\frac{3}{10}$

$$E(X) = 1 \cdot \frac{1}{10} + 2 \cdot \frac{2}{10} + 3 \cdot \frac{4}{10} + 4 \cdot \frac{3}{10} = \frac{29}{10} = 2.9.$$

y (possible values of Y)	2	3	4	5	6	7	8
$f_Y(y) = P(\text{“}Y = y\text{”})$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{2}{10}$	$\frac{2}{10}$	$\frac{1}{10}$	$\frac{2}{10}$	$\frac{1}{10}$

$$E(Y) = 2 \cdot \frac{1}{10} + 3 \cdot \frac{1}{10} + 4 \cdot \frac{2}{10} + 5 \cdot \frac{2}{10} + 6 \cdot \frac{1}{10} + 7 \cdot \frac{2}{10} + 8 \cdot \frac{1}{10} = \frac{51}{10} = 5.1.$$

When we have more powerful techniques for computing probabilities, we will solve this binary search problem for arrays of arbitrary size n, not just $n = 10$.

Example 6.6.4 In the preceding problem, suppose the target is NOT in the array. What is the expected number of times that the body of the loop will be executed? What is the expected number of `otype` object comparisons that will be required to discover that the target is not in the array? Assume that the target object has equal probability of belonging at any "cell boundary" in the array, i.e., between any two objects in the array or at the ends.

Solution. Again, for concreteness, assume that initially `first = 0` and `last = 9`. When the target is not in the array, then the search will have to narrow down the subarray in which it searches until that subarray is of length 0. This will determine which of the 11 "cell boundaries" is the spot where the target belongs. In the logical search tree, this corresponds to going down some path in the tree until an empty subtree is encountered. Let X denote the number of times the body of the loop is executed during the search, and let Y denote the number of object comparisons required. Then we have the logical search tree shown in Fig. 6.25:

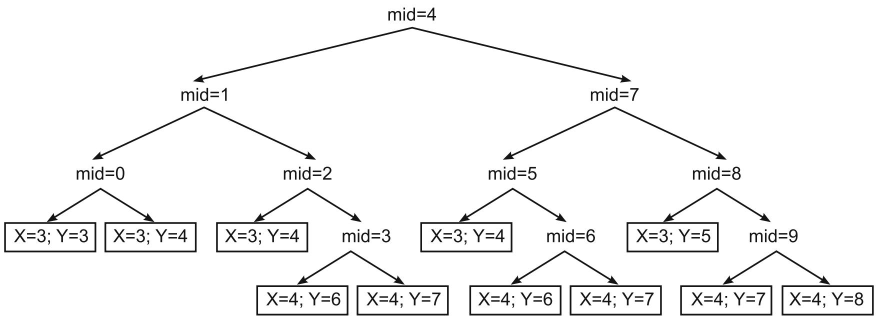

Fig. 6.25 Execution tree for an unsuccessful the binary search

x (possible values of X)	3	4
$f_X(x) = P(\text{“}X = x\text{”})$	$\frac{5}{11}$	$\frac{6}{11}$

$$E(X) = 3 \cdot \frac{5}{11} + 4 \cdot \frac{6}{11} = \frac{39}{11} = 3.5.$$

y (possible values of Y)	3	4	5	6	7	8
$f_Y(y) = P(\text{“}Y = y\text{”})$	$\frac{1}{11}$	$\frac{3}{11}$	$\frac{1}{11}$	$\frac{2}{11}$	$\frac{3}{11}$	$\frac{1}{11}$

$$E(Y) = 3 \cdot \frac{1}{11} + 4 \cdot \frac{3}{11} + 5 \cdot \frac{1}{11} + 6 \cdot \frac{2}{11} + 7 \cdot \frac{3}{11} + 8 \cdot \frac{1}{11} = \frac{61}{11} = 5.5.$$

Example 6.6.5 (This example assumes acquaintance with the concept of a hash table.) The diagram in Fig. 6.26 shows a small hash table in which collision resolution was performed by separate chaining of unordered lists. Each node in a list contains a string that was hashed to that array location. For example, the hash function that was used to create the table hashed the name “Chris” to location 8. (The hash function is not given here because it is not relevant to the questions we want to consider.) A list node was dynamically at location 8 and “Chris” was copied into the node. Later the name “Juan” was hashed to location 8 as well, so a list node was dynamically allocated, the name was copied into the node, and the node was inserted at the front of the existing list.

(a) What is the expected number of string comparisons that will be made during a successful search of the table if each string has equal likelihood of being the target of the search?
(b) What is the expected number of string comparisons that will be made during an unsuccessful search of the table if the target string is equally likely to be hashed to any of the 13 addresses in the table?

Solutions.

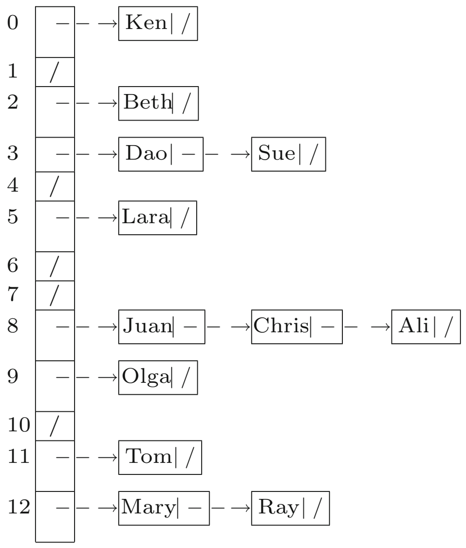

Fig. 6.26 A hashing table with 13 locations: unordered chains

(a) There are 12 strings in the table, so each has probability 1/12 of being the target of the search. Let X denote the number of string comparisons that will be made during a successful search. Then the 8 strings that are in the front nodes of the lists will require only one string comparison, which means that $P(\text{“}X = 1\text{”}) = 8/12$. The 3 strings that are in second position in their lists will require two string comparisons, which means that $P(\text{“}X = 2\text{”}) = 3/12$. Only 1 string (“Ali”) requires three comparisons, so $P(\text{“}X = 3\text{”}) = 1/12$. Thus

$$E(X) = 1 \times \frac{8}{12} + 2 \times \frac{3}{12} + 3 \times \frac{1}{12} = \frac{17}{12} \approx 1.4\,.$$

(b) Let Y denote the number of string comparisons that will be made during an unsuccessful search. If the target string hashes to one of the 5 addresses that has an empty list, then the value of Y will be 0, so $P(\text{“}X = 0\text{”}) = 5/13$. If the target string hashes to one of the 5 addresses that has a list of length 1, then Y will be 1, so $P(\text{“}Y = 1\text{”}) = 5/13$. If the target string hashes to one of the 2 addresses that has a list of length 2, then Y will be 2, so $P(\text{“}Y = 2\text{”}) = 2/13$. Finally, if the target string hashes to the address with a list of length 3, then Y will be 3, so $P(\text{“}Y = 3\text{”}) = 1/13$. Thus

$$E(Y) = 0 \times \frac{5}{13} + 1 \times \frac{1}{13} + 2 \times \frac{2}{13} + 3 \times \frac{1}{13} = \frac{8}{13} \approx 0.62\,.$$

6.6.1 Exercises

6.6.6 Suppose a search of an *ordered* linked list is going to be made for some specified target object using the function below. Assume that the list is `NULL`-terminated and contains n objects. Also assume that the operator $<$ is defined on the `otype` data type.

```
template <class otype>
node_ptr location_in_list (node_ptr front, const otype & target)
{
  while ( front != NULL  &&  front->datum < target )
    front = front->next;
  return front; // The calling function can test this pointer to see
}               // whether it points to a node containing the target.
```

(a) Suppose a copy of the target object is present in the list, and suppose it is equally likely to be in any of the n nodes in the list. What is the expected number of times that the body of the loop will be executed during the search? What is the expected number of `otype` object comparisons that will be required to find the target?

(b) Suppose the target value is not in the list (of course, this is not known in advance of the search). Suppose also that the target is equally likely to belong at any of the "node boundaries" in the list, i.e., between any two of the objects in the list or at the ends. What is the expected number of times that the body of the loop will be executed during the (unsuccessful) search? Keep in mind that the objects are in increasing order in the list. Read the code above with care so that you do not fall into the error of thinking this is just like Example 6.6.2.

(c) (Continuation of part (b).) What is the expected number of `otype` object comparisons that will be required to discover that the target is not in the list? This last question is tricky. Begin by examining all 3 possibilities when the list is of length $n = 2$.

6.6.7 Suppose we have a table maintained as an array or linked list, and suppose the objects in the table obey Zipf's Law, which says that the $k-$th most common word in natural language text seems to occur with a frequency inversely proportional to k. Suppose we arrange the objects in the table in order of decreasing frequency of access, so that the most frequently accessed object is at the front and the least frequently accessed is at the end. Find a simple expression that approximates the expected number of object comparisons (i.e., tests of equality between two objects) that will be made during a successful linear search of the table if there are n objects in the table and n is large. The search algorithm is exactly the one used in Example 6.6.2 (but now the objects are *not* in random order).
Hint: if X denotes the number of comparisons, then the p.d.f of X will involve the n-th harmonic sum $H(n)$.

6.6.8 Suppose that in Example 6.6.3 on p. 330, the order of the two object comparisons is interchanged. That is, suppose the code reads this way:

```
...
 if (a[mid] < target)
```

```
    first = mid + 1;
  else if (target < a[mid])
    last = mid - 1;
  else...
```

Compute the expected number of `otype` object comparisons that will be made during a successful search of an array of 10 objects under the assumption that the target is equally likely to be in any of the 10 cells. Is the answer better or worse than the answer we obtained in Example 6.6.3? Offer a reason why your answer makes sense.

6.6.9 Suppose a binary search using the algorithm in Example 6.6.3, p. 330, is made on an array of 13 `otype` objects arranged in increasing order. For concreteness, suppose that the cells of the array are numbered from 0 to 12.

(a) If a copy of the target is present in the array, what is the *exact* expected number of `otype` object comparisons that will have to be made to find the target? Assume that all 13 objects in the array are equally likely to be the target. (You are more likely to get the right answer if you draw the logical search tree.)
(b) If the target is not present, what is the *exact* expected number of `otype` object comparisons that will have to be made to discover that the target is not there? Assume that the target is equally likely to belong at any one of the 14 cell boundaries of the array.

6.6.10 Repeat Exercises 6.6.9 (a) and (b) using the binary search algorithm given in Fig. 5.15b on p. 201. Compare your answers with the answers in 6.6.9. (Note: there is a difference between the successful and unsuccessful searches here. In the *successful* case, if the search arrives at a point at which `mid` is 12, then it is not logically possible, for `a[mid]` to be less than `target`.)

6.6.11 Figure 6.27 shows an AVL tree, i.e., a height-balanced binary search tree. The integers shown at the nodes are the keys of the data objects stored at those nodes. Now suppose a search is about to be made for an object with a specified "target" key. Code for the search algorithm is shown above the tree.

(a) If the target key is actually present in the tree, and if the target is equally likely to be any one of the keys in the tree, then what is the expected number of key comparisons (i.e., tests of inequalities of the form `p->key <target` and `p->key >target`) that will be required to find the object containing the target key?
(b) If the target key is not in the tree, and if the key is equally likely to belong in any of the empty subtrees of the tree, then what is the expected number of key comparisons that will be required to find that the target is not in the tree? (As an

example, the key 35 is not in the tree; it "belongs" in the empty subtree to the left of 40 because that's where we would place it if we now inserted it.)

```
node_ptr key_location (node_ptr p, int target)
{
  bool found = false;
  while (p && !found)
    if (p->key < target)
      p = p->right;
    else if (p->key > target)
      p = p->left;
    else
      found = true;
  return p;
}
```

6.6.12 Suppose the strings in Example 6.6.5 are placed in a hash table of size 13 using ***linear probing*** instead of separate chaining. (Recall that in linear probing, when a collision occurs during insertion, the insertion algorithm moves down the array, one cell at a time, with wrap-around, until it finds an empty cell into which it can place the object being inserted.) We can tell by looking at Fig. 6.26 exactly which addresses the strings hash to: for example, the string "Lara" hashes to address 5. Suppose the strings are inserted into the table in the following order (the numbers in parentheses show the hash addresses): Ali (8), Lara(5), Ken(0), Olga(9), Sue(3), Chris(8), Ray(12), Tom(11), Juan(8), Mary(12), Beth(2), Dao(3). The result will be the array shown below.

0	1	2	3	4	5	6	7	8	9	10	11	12
Ken	Juan	Mary	Sue	Beth	Lara	Dao		Ali	Olga	Chris	Tom	Ray

(a) What is the expected number of string comparisons that will be made during a successful search of the table if each string has equal likelihood of being the target of the search? (To take one example, suppose the string "Mary" is the target of the search. Then 4 string comparisons will be made during the search for that target: "Ray" in cell 12 (the address to which "Mary" hashes), "Ken" in cell 0, "Juan" in cell 1.)

(b) What is the expected number of string comparisons that will be made during an unsuccessful search of the table if the target string is equally likely to be hashed to any of the 13 addresses?

Note: those who remember how hash tables should be operated will see that this table is badly over-loaded. When a hash table is implemented using linear (or quadratic) probing, the load factor on the table should never be allowed to go above 50 %. Load factors higher than that cause the performance of the table to degrade, as we can see in the calculations above. You should compare your answers here to the numbers we derived in Example 6.6.5.

6.6.13 (For the mathematically gifted.) Suppose we partition an array `a[0...n-1]` of *distinct* objects in random order into two subarrays, say `a[0..m-1]` and `a[m..n-1]`, where $1 \le m < n$. Suppose that each subarray is then sorted using some sorting algorithm (which algorithm is irrelevant here). Next, suppose the objects in these sorted subarrays are merged into a single sorted array `c[0...n-1]` using the following algorithm (slightly modified from Fig. 5.50):

```
template <class otype>
void merge_array(const otype a[], otype c[], int m, int n)
{
  int afirst = 0;     // Make afirst ''point'' to the 1st cell of a[0..m-1] and
  int bfirst = m;     // bfirst ''point'' to the 1st cell of a[m..n-1] and
  int cfirst = 0;     // cfirst ''point'' to the 1st cell of c[0..n-1].
  while (afirst < m  &&  bfirst < n)  // Don't go outside of subarrays.
    if (a[afirst] <= a[bfirst])
      c[cfirst++] = a[afirst++];  // Copy from left subarray into c.
    else
      c[cfirst++] = a[bfirst++];  // Copy from right subarray into c.
  // When the loop above stops, exactly one of two things will be true:
  // afirst will be < m and bfirst will be = n, in which case we must
  // copy the remaining objects from a[afirst..m-1] into c[cfirst..n-1];
  // OR bfirst will be < n and afirst will be = m, in which case we
  // must copy the remaining objects from a[bfirst..n-1] into c[first..n-1].
  while (afirst < m) // This loop or the one following it will be executed.
    c[cfirst++] = a[afirst++];
  while (bfirst < n)
    c[cfirst++] = a[bfirst++];
}
```

(a) Derive a summation formula in the variables m and n for the expected number of object comparisons of the form "`a[afirst] <= a[bfirst]`" that will occur during the first "while" loop above. (Note that we are ignoring subscript comparisons such as `afirst <m`.) A useful observation is that when the first loop above is executed, one of two things must occur: *either* the "while" loop will stop because afirst reaches the value m, which occurs if and only if the largest object in `a[0..m-1]` is smaller than one or more of the objects in `a[m..n-1]`, *or else* the loop will stop because `bfirst` reaches n, which occurs if and only if the largest object in `a[0..m-1]` is larger than all the objects in `a[m..n-1]`.

(b) (HARD) Prove that the summation formula derived in part (a) can be reduced to the closed form

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

(c) Show that if m is $\lfloor (1+n)/2 \rfloor$, then the formula in part (b) lies between $n-2$ and $n-1$ (and is therefore asymptotic to n). Keep in mind that $1 \le m < n$, so $n \ge 2$. (In the Merge Sort in Fig. 5.51 on p. 225 the number

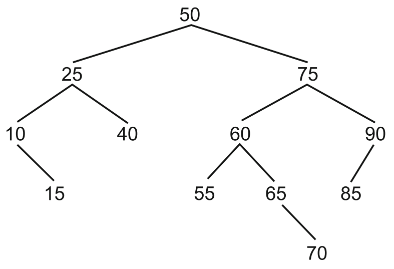

Fig. 6.27 An AVL tree

```
template <class otype>
void simple_bubble_sort (otype a[], int n)
{
  int j, k;
  for (k = n-1; k >= 1; --k)
    for (j = 0; j <= k-1; ++j)
      if (a[j] > a[j+1])          // then these 2 objects are out of order
        swap (a[j], a[j+1]);
}
```

Fig. 6.28 Simple Bubble Sort function

`mid = (first + last)/2` plays the role that m is playing in this exercise. We'll use this exercise later to calculate the expected number of object comparisons made during a run of Merge Sort.)

6.7 Applications of Expected Value Theorems to Analysis of Algorithms

This section contains two examples and several exercises that illustrate how standard theorems about expected values can be put to work to calculate the average (i.e., expected) behavior of some well known elementary algorithms.

Example 6.7.1 In Fig. 5.9, p. 185, we looked at a simple version of Bubble Sort (reproduced in Fig. 6.28) and calculated the minimum and maximum execution times for the algorithm. Now let's look at its average behavior. Assume that all the objects in the array are distinct and are arranged initially in random order. Calculate the expected number of object comparisons (this does not include subscript comparisons) that will be made during execution of this function when it is applied to an array of n objects. Also calculate the expected number of object swaps.

As it turns out, the number of object comparisons is a deterministic function of n: the first time the body of the outer loop (controlled by variable k) is executed, the inner loop variable j will satisfy the condition $j < k-1$ exactly $n-1$ times, so there will be exactly $n - 1$ object comparisons. We call this the "first pass" over the array. On the second pass, the inner loop body will make $n - 2$ object comparisons. Etc. Altogether, the number of object comparisons is $(n-1)+\ldots+2+1 = \frac{(n-1)n}{2}$. If we let X denote the random variable that counts the number of object comparisons, then X is constant and $E(X) = \frac{(n-1)n}{2}$ (see Theorem 8.0.29).

Now let Y denote the number of object swaps that will take place. Then Y is very much dependent on the initial arrangement of the objects in the array. If by chance the objects are in increasing order initially, then $Y = 0$, whereas if the objects are in decreasing order initially, you can verify that $Y = \frac{(n-1)n}{2}$. These are the extreme possible values of Y. Calculating the expected value of Y is not especially easy.

It turns out to be convenient to introduce some auxiliary random variables that are related to Y. For each integer i in the range $0 \le i \le n - 1$ let L_i denote the random variable that counts the number of objects that are *larger* than `a[i]` but that lie to the *left* of `a[i]` in the initial arrangement of the objects in the array. For example, if the array is initially

	0	1	2	3	4	5	6	7	8	9
`a:`	`Ken`	`Nan`	`Ida`	`Lee`	`Pat`	`Cal`	`Gus`	`Ray`	`Joy`	`Moe`

then $L_0 = 0$, $L_1 = 0$, $L_2 = 2$, $L_3 = 1$, $L_4 = 0$, $L_5 = 5$, $L_6 = 5$, $L_7 = 0$, $L_8 = 5$, $L_9 = 3$. Now at some stage in the Bubble Sort, the object Moe will have to be swapped to the left with each of the 3 objects that are larger than it because they need to move to the right of Moe. A similar statement is true about all the other objects in the array: the i-th object will have to be swapped to the left with the L_i objects that are larger but lie to its left. Thus the total number of "left swaps" (which is the same as the total number of swaps) that will be made, which we call Y, can be expressed as $Y = L_0 + L_1 + \cdots + L_{n-1}$. By Theorem 8.0.30 in the Appendix, $E(Y) = 0 + E(L_1) + E(L_2) + \cdots + E(L_{n-1})$. We have thus reduced the problem to calculating $E(L_i)$ for each $i = 1, 2, 3, \ldots, n - 1$.

So now consider, for a fixed integer i in the range $1 \le i \le n - 1$, the random variable L_i. Its possible values are $0, 1, 2, 3, \ldots, i$. The event "$L_i = 0$" occurs if and only if `a[i]` is initially larger than all the objects to its left. The probability of this is $\frac{1}{i+1}$. The event "$L_i = 1$" occurs if and only if `a[i]` is initially the second-largest object in the subarray `a[0..i]`, and the probability of this is also $\frac{1}{i+1}$ (see Theorem 8.0.12). In fact, for each of its possible values, the probability that L_i will assume that value is $\frac{1}{i+1}$, and thus

$$E(L_i) = \frac{1}{i+1}[0 + 1 + 2 + \cdots + i] = \frac{1}{i+1}\frac{i(i+1)}{2} = \frac{i}{2}.$$

Now we can return to $E(Y)$. As shown above, $E(Y) = E(L_1) + \cdots + E(L_{n-1})$,

$$\text{so } E(Y) = \frac{1}{2} + \frac{2}{2} + \cdots + \frac{n-1}{2} = \frac{1}{2}[1 + 2 + \cdots + (n-1)] = \frac{(n-1)n}{4}.$$

Example 6.7.2 In this example we will make use of Theorem 8.0.27.

Review Algorithm 1 on p. 309 for choosing a random sample of size K from a population of size N. The method involves initializing a Boolean array of length N to false everywhere and then using a random number generator to choose integers in the range $1 \ldots N$ until K distinct integers have been chosen. What is the expected number of calls to the random number generator required to produce the sample of size K? The answer should be expressed in terms of K and N.

Let X_2 denote the number of calls, after the first call, necessary to produce a second integer different from the first. Let X_3 denote the number of calls, after the appearance of the second integer, necessary to produce a third integer different from the first two. Etc. to X_K. Let X denote the total number of calls necessary to produce K distinct integers. Then

$X = 1 + X_2 + X_3 + \cdots + X_K$, so $E(X) = 1 + \sum_{i=2}^{K} E(X_i)$. By Theorem 8.0.27,

$$E(X_i) = \frac{1}{\frac{N-(i-1)}{N}} = \frac{N}{N-i+1}, \text{ so}$$

$$E(X) = 1 + \sum_{i=2}^{K} \frac{N}{N-i+1} = 1 + N\left(\frac{1}{N-1} + \frac{1}{N-2} + \cdots + \frac{1}{N-K+1}\right)$$
$$= N\left(\frac{1}{N} + \frac{1}{N-1} + \frac{1}{N-2} + \cdots + \frac{1}{N-K+1}\right)$$

If K is small relative to N, then each of the K fractions in the sum in parentheses is approximately equal to $\frac{1}{N}$, so N times that sum is approximately K, which is what common sense tells us should be the approximate expected value. If K is not small relative to N, then we can note that the sum in parentheses above is the difference between two harmonic sums, provided $K < N$:

$$E(X) = N[H(N) - H(N-K)] \approx N[\ln(N) - \ln(N-K)] = N \ln \frac{N}{N-K},$$

where $H(n)$ denotes the n-th harmonic sum.

6.7.1 Exercises

6.7.3 Suppose we use the following algorithm (taken from Fig. 5.1, p. 170) to find the largest object in an array `a[first..last]`.

```
template <class otype>
int location_of_max (const otype a[], int first, int last)
{
  int max_loc = first;                    // The first cell of the subarray will
                                          // contain the ``largest seen so far''.
```

```
    for (++first; first <= last; ++first) // Start at the second cell.
      if (a[first] > a[max_loc])          // The value paramtr ``first''
        max_loc = first;                  // can be modified since its
                                          // initial value need not be
    return max_loc;                       // preserved.
  }
```

We know that if there are n objects in the array, then exactly $n - 1$ object comparisons will be made. The assignment statement `max_loc = first` inside the loop body may never be executed, or it may be executed $n - 1$ times, or it may be executed some number of times between 0 and $n - 1$. What is the *expected* number of times that it will be executed if all the objects are distinct, and in random order? **Hint:** Express the random variable here as the sum of a bunch of indicator random variables.

6.7.4 Let W denote the number of swaps that will be made during the first pass of the Bubble Sort algorithm on p. 339 when applied to an array of n distinct objects in random order. (By "first pass" we mean the complete execution of the inner loop controlled by the index j during the first execution of the body of the outer loop.)

(a) Compute the expected value of W as a function of n. **Hint:** an object `a[j]` will be swapped to its right if and only if what is true? Use indicator random variables (see Definition 8.0.21 and Theorem 8.0.39).
(b) At the end of the first pass, what is the probability that `a[1]` will be less than `a[2]`?

6.7.5 Below is code for a version of Linear Insertion Sort.

```
template <class otype>
void linear_insertion_sort (otype a[], int n)  // sorts a[1...n]
{
  int i, k;
  for (k = 2; k <= n; ++k)
  {                 // Move a[k] into cell 0 where it can act
     a[0] = a[k];   // as a sentinel and to open a hole in the array.
     // Make a backward linear search, using a subscript i.
     // The value in a[0] prevents  i  from going too far left.
     for (i = k-1; a[i] > a[0]; --i)
       a[i+1] = a[i]; // Slide a[i] to the right by one cell
     // Subscript  i  was decremented after correct position found
     a[i+1] = a[0];    // Move a[0] into its correct position.
  }
}
```

Assume that all n objects in `a[1..n]` are distinct and that they are arranged in random order initially. Calculate the expected number of object comparisons (`a[i] >a[0]`) and the expected number of object moves (assignments of one data object to another).

Hint: Let X_2 denote the number of object comparisons that will be made when k has the fixed value 2; let X_3 denote the number of comparisons that will be made when k has the value 3; and so on up to X_n when k has the value n. Let X denote the total number of object comparisons altogether, so that $X = X_2 + X_3 + \cdots + X_n$. What are the possible values of X_2? The possible values of X_3? What are the possible values of X_k, where $2 \leq k \leq n$? Start by calculating the probability of each of the values of X_2. Of each of the values of X_3. Etc. for each X_k, $2 \leq k \leq n$. Use these to compute $E(X_k)$ for $2 \leq k \leq n$, and then compute $E(X)$, i.e., the expected total number of object comparisons. To deal with object moves, let Y_2 denote the number of object moves when $k = 2$, etc. Then note that Y_k is related in a simple way to X_k.

6.7.6 In Exercise 6.7.4 you computed the expected number of swaps that will be made during the first pass of the Bubble Sort algorithm on Example 6.7.1 when applied to an array of length *n in random order*. The answer should have been $\frac{1}{2} + \frac{2}{3} + \frac{3}{4} + \frac{4}{5} + \ldots + \frac{n-1}{n}$. This can be written as

$$(1 - \frac{1}{2}) + (1 - \frac{1}{3}) + (1 - \frac{1}{4}) + (1 - \frac{1}{5}) + \cdots + (1 - \frac{1}{n})$$
$$= (n-1) - \left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots \frac{1}{n}\right) = n - H(n),$$

where $H(n)$ denotes the n-th harmonic sum. Now suppose we decide to apply this to the analysis of the *complete* Bubble Sort Algorithm when it acts on an array of length n in random order. Let W_n denote the number of swaps that will be made during the first pass over the array of length n; let W_{n-1} denote the number of swaps that will be made during the second pass over the reduced array of length $n - 1$; etc. Then the total number of swaps during the execution of the entire algorithm will be $W_n + W_{n-1} + W_{n-2} + \cdots + W_2$ (the last pass is over an array of length 2). It follows that the expected number of swaps during the execution of the entire algorithm will be $E(W_n) + E(W_{n-1}) + E(W_{n-2}) + \cdots + E(W_2)$. By Exercise 6.7.4, $E(W_n) = n - H(n)$. On the next pass, the array we are working with is of length $n - 1$ so we ought to have $E(W_{n-1}) = (n-1) - H(n-1)$. Similarly, we ought to have $E(W_{n-2}) = (n-2) - H(n-2)$ etc. Putting all these together will give the following formula for the expected number of swaps during the execution of the entire algorithm:

$$n + (n-1) + (n-2) + \cdots + 2 - [H(n) + H(n-1) + H(n-2) + \cdots + H(2)].$$

This analysis of Bubble Sort is simpler than the one used in Example 6.7.1. Note, however, that it gives a different answer! (You can verify in a simple case such as $n = 3$ that the answer here does not match the answer in Example 6.7.1.) As it turns out, there is an error of reasoning in this simpler analysis. Find it.
Hint: it has something to do with the italicized phrase at the beginning of this exercise.

6.7.7 Look back at Algorithm 1, p. 300, for choosing a random sample of size K from a file of size N. A collection of K distinct random integers is chosen, and then the file is read and the file records corresponding to those K integers are selected for the random sample. Generally only a portion of the file needs to be read. More precisely, if L is the largest of the K integers chosen in the first part of the algorithm, then L records must be read from the file, which is to say that the fraction L/N of the file will have to be read. Compute the expected number of records that will be read from the file.
Hint: use Exercise 2.3.51 on p. 52. Also use one of the formulas from Exercise 2.3.51 on p. 52. Be sure to reduce your answer algebraically to as simple a form as possible (a lot of factorials should cancel out).
Note: before doing this problem, you may want to predict the answer based on the following intuitive notion: the K chosen integers should partition the remaining $N - K$ integers into $K + 1$ segments having average size $\dfrac{N - K}{K + 1}$. The largest of the K chosen integers should be located just before the final one of these segments. Thus the largest of the K chosen integers should be what?

6.7.8 Look at Algorithm 3, p. 313, for choosing a random sample. Explain carefully why, for a given pair N and K, the expected number of records that will be read from the file by Algorithm 3 is exactly the same as the expected number that will be read by Algorithm 1 (see Exercise 6.7.7). You may use the fact that Algorithm 3 is known to be unbiased.

6.7.9 Compute the expected number of iterations for the linear search algorithm seen in Fig. 5.2, p. 175, with `first` $= 0$ and `last` $= n - 1$, when the array contains exactly two occurrences of the target placed at random in the array.
Hint: The linear search will find a target at position k in the array if neither target value is present from 0 to $k - 1$ in the array, and one of the two target values is at position k.

6.8 Applications of Conditional Expected Value to Analysis of Algorithms

This section contains several examples and exercises that apply theorems about conditional expected values to analysis of familiar algorithms. See Definition 8.0.34 and Theorems 8.0.35 and 8.0.36 in the Appendix.

Example 6.8.1 Suppose we have a table of 50 objects maintained as an array in random order. Suppose the table is static, and that frequent searches are made in the table using the algorithm in Fig. 6.23 on p. 330. If 80 % of the searches are unsuccessful, then what is the expected number of times the body of the loop will be executed during a single search of the table?

Solution. Let X denote the number of object comparisons that will be made during a search of this table. We are asked to compute $E(X)$. Since the number of comparisons surely depends on whether the target is present or not, it makes sense to introduce the following events: let A_1 denote the event "the target will be in the table" and let A_2 denote the event "the target will not be in the table". Then A_1 and A_2 make up a partition of the sample space of possible targets, so by Theorem 8.0.36 we have

$$E(X) = P(A_1)E(X|A_1) + P(A_2)E(X|A_2).$$

We are told in the problem that $P(A_1) = 0.20$ and $P(A_2) = 0.80$. We can express the results derived in Example 6.6.2 on p. 328 this way: $E(X|A_1) = \dfrac{50-1}{2}$ and $E(X|A_2) = 50$. It follows that

$$E(X) = 0.20 \cdot \frac{49}{2} + 0.80 \cdot 50 = 4.9 + 40 = 44.9$$

Example 6.8.2 **(Analysis of Successful Binary Search)**: Suppose we have a table of n objects maintained as an array `a[first..last]` in sorted order. Suppose we are about to make a binary search of the array using the algorithm given in Example 6.6.3 on p. 330. Assume that target is in the array and is equally likely to be in any of the n cells of the array. What is the expected number of object comparisons that will be made during the search?

Solution. Let X_n denote the number of object comparisons that will be made during a successful binary search of an ordered array of n objects. Let A_1 denote the event that `target <a[mid]` will be true on the first probe of the search ("probe" here means an examination of the contents of `a[mid]`; a probe will involve either one or two object comparisons). Let A_2 denote the event that `a[mid] <target` will be true on the first probe of the search. Let A_3 denote the event that `a[mid] == target` will be true on the first probe of the search. Using the Fig. 2.1 on p. 14, we can see that

$$P(A_1) = \frac{\lfloor (n-1)/2 \rfloor}{n}, \quad P(A_2) = \frac{\lfloor n/2 \rfloor}{n}, \quad P(A_3) = \frac{1}{n} \quad \text{for all } n \geq 1.$$

We can write X_n in the form

$$X_n = 1 + I_{A_2 \cup A_3} + Z_n,$$

where Z_n denotes the number of object comparisons that will be made in `a[first.. last]` after the first probe (i.e., after the first time through the loop body). The expression $I_{A_2 \cup A_3}$ denotes the indicator random variable for the event $A_2 \cup A_3$. Its value is 1 if A_2 or A_3 occurs (i.e., two object comparisons are made on the first time through the loop), while the value is 0 if A_1, the complement of $A_2 \cup A_3$, occurs (i.e., 1 object comparison is made). Then for all $n \geq 1$ we have

$$\begin{aligned}
E(X_n) &= 1 + P(A_2 \cup A_3) + E(Z_n) \\
&= 1 + P(A_2 \cup A_3) + P(A_1)E(Z_n|A_1) \\
&\quad + P(A_2)E(Z_n|A_2) + P(A_3)E(Z_n|A_3) \\
&= 1 + \frac{\lfloor n/2 \rfloor + 1}{n} + \frac{\lfloor (n-1)/2 \rfloor}{n} E(X_{\lfloor (n-1)/2 \rfloor}) + \frac{\lfloor n/2 \rfloor}{n} E(X_{\lfloor n/2 \rfloor}) + 0
\end{aligned}$$

provided we interpret $E(X_0)$ as denoting 0 (when $n = 1$, $E(Z_n|A_1) = E(Z_n|A_2) = 0$, and when $n = 2$, $E(Z_n|A_1) = 0$). In the equation above, multiply on both sides by n to get

$$nE(X_n) = n + (\lfloor n/2 \rfloor + 1) + \lfloor (n-1)/2 \rfloor E(X_{\lfloor (n-1)/2 \rfloor}) \\ + \lfloor n/2 \rfloor E(X_{\lfloor n/2 \rfloor}) \quad \text{for all } n \geq 1.$$

Let $F(n)$ denote the product $nE(X_n)$. Then $F(0) = 0$, and the equation above takes the form

$$F(n) = n + \lfloor n/2 \rfloor + 1 + F(\lfloor (n-1)/2 \rfloor) + F(\lfloor n/2 \rfloor) \quad \text{for all } n \geq 1.$$

This is a recurrence relation we can solve. It requires a slight transformation, however, along the lines shown in Table 4.4 on p. 157. We note that $\lfloor n/2 \rfloor = \lceil (n-1)/2 \rceil$, so the equation can be rewritten as

$$F(n) = n + \lfloor n/2 \rfloor + 1 + F(\lfloor (n-1)/2 \rfloor) + F(\lceil (n-1)/2 \rceil) \quad \text{for all } n \geq 1.$$

Now we apply the transformation $F(n) = G(n+1)$ to obtain (again see Table 4.4, p. 157)

$$G(n) = (n-1) + \left\lfloor \frac{n-1}{2} \right\rfloor + 1 + G(\lfloor n/2 \rfloor) + G(\lceil n/2 \rceil) \quad \text{for all } n \geq 2,$$

and $0 = F(0) = G(1)$. Using the fact that $\lfloor (n-1)/2 \rfloor = \lceil n/2 \rceil - 1$, we can now write

$$G(1) = 0, \qquad G(n) = G(\lfloor n/2 \rfloor) + G(\lceil n/2 \rceil) + n - 1 + \lceil n/2 \rceil \quad \text{for all } n \geq 2.$$

By Table 4.1 on p. 147,

$$G(n) = \alpha\, n + \phi_1(n) - (-1) + \phi_2(n), \text{ where } \phi_1(n) \sim n\lg(n) \text{ and } \phi_2(n) \sim \frac{1}{2} n \lg(n).$$

By Theorem 3.3.8 (e) on p. 90 and Theorem 3.3.6 on p. 90, $G(n) \sim \frac{3}{2} n \lg(n)$. Now we return to $F(n) = G(n+1)$, which gives $F(n) \sim \frac{3}{2}(n+1)\lg(n+1)$, or more simply, $F(n) \sim \frac{3}{2} n \lg(n)$. Finally, we recall that $F(n) = nE(X_n)$, so $E(X_n) = \frac{1}{n} F(n)$, which gives us our final answer:

$$\boxed{E(X_n) \sim \frac{3}{2} \lg(n)}$$

Example 6.8.3 **(Successful Search in Hash Table with Separate Chaining)**: Suppose we are going to construct a hash table named T with size M using separate chaining for collision resolution. This means that T will be an array with cells numbered $0, 1, 2, \ldots, M-1$, and each cell `T[i]` will contain a pointer to a (possibly empty) linked list. We'll assume that when an object is inserted into the hash table, it is simply placed at the front of the list at the cell to which its key hashes. The hash function will be assumed to be a good one: any given key value is equally likely to be hashed to any of the M locations. Figure 6.29 shows the code for placing a copy of an object x in the table T.

Figure 6.30 shows the code for searching the table for an object having a specified key value k.

How well can the hash table be expected to perform under a load of size n, that is, after n objects have been inserted? To find out, we'll count the expected number of key comparisons as a measure of the expected time required to perform one operation on the table. In this example we'll look at the case of a successful search for a specified key k. The exercises will look at the case of an unsuccessful search and of an insertion operation.

Suppose then that a key value k is specified and (although we don't know it before we make the search) there is an object with that key in the table T. Then that object is equally likely to be anywhere in the hash table. Let X denote the number of times that the expression "`temp->info.key == k`" will be evaluated during the successful search. Then X could theoretically take any value from 1 to n because the chain containing the target key could contain as many as n nodes, and the target key could be anywhere in that chain. Thus we know the best and worst cases possible for X. We now calculate $E(X)$. Our approach will use conditional expected values based on the possible lengths of the chain containing k. Let A_λ denote the event that the chain containing k will be of length λ for $\lambda = 1, 2, 3, \ldots, n$. Then by Theorem 8.0.31 and a calculation similar to that of Example 6.6.2 on p. 328,

$$E(X) = \sum_{\lambda=1}^{n} P(A_\lambda)E(X|A_\lambda) = \sum_{\lambda=1}^{n} P(A_\lambda)\frac{\lambda+1}{2}.$$

Let $h(k)$ denote the address to which the key k hashes. The event A_λ occurs when $\lambda-1$ of the other $n-1$ keys hash to $h(k)$. These $n-1$ individual hashing actions can be viewed as independent binary trials (each key either does or does not hash to $h(k)$) with probability $1/M$ of success on each trial, so $P(A_\lambda) = b(\lambda-1; n-1, 1/M)$

```
bool insert (otype &x, nodeptr T[]) // Returns true if insertion is successful.
{                            // Returns 0 if T already contains an
  int i = hash(x.key);       // object whose key matches x.key
  nodeptr temp = T[i];       // Create a temporary hash node pointer.
  while (temp != NULL)       // This works even if T is empty.
    if (temp->info.key == x.key)
      return false;          // RETURN from function. Insert failed.
    else
      temp = temp->next;
  temp = new node;           // Allocate.
  temp->info = x;            // Copy object x into node.
  temp->next = T[i];         // Place the new node at the front
  T[i] = temp;               // of the chain at cell T[i].
  return true;               // Indicate that the insertion operation was successful.
}
```

Fig. 6.29 Inserting an object in a hash table

```
otype* search(nodeptr T[], key_type k)// Returns pointer to the unique
{                             // object in T with key k if such an
  int i = hash(k);            // object exists. Else returns NULL.
  nodeptr temp = T[i];        // Create a temporary hash node pointer.
  while (temp != NULL)
    if (temp->info.key == k)
      return &(temp->info); // EXIT from function if search succeeds.
    else
      temp = temp->next;
  return NULL;                // Search was unsuccessful. Return NULL.
}
```

Fig. 6.30 Searching for an object in a hash table

(see Definition 8.0.34 and Theorem 8.0.35). Thus

$$E(X) = \sum_{\lambda=1}^{n} \frac{\lambda+1}{2} b\left(\lambda-1; n-1, \frac{1}{M}\right) \text{[Shift index of summation by letting}$$
$$x = \lambda - 1, \lambda = x+1.]$$
$$= \sum_{x=0}^{n-1} \frac{x+2}{2} b\left(x; n-1, \frac{1}{M}\right)$$
$$= \frac{1}{2}\sum_{x=0}^{n-1} x\, b\left(x; n-1, \frac{1}{M}\right) + \sum_{x=0}^{n-1} b\left(x; n-1, \frac{1}{M}\right)$$

where $b(j; k, p)$ denotes the probability of j successes in k independent trials of a binary random variable, with probability p of success on each trial. By Definition 8.0.34 and Theorem 8.0.36 the equation above simplifies to

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

Let's write L for the "load index" of the table, which is defined by $L = \frac{n}{M}$. Then if you are planning to create a separate chaining hash table with load index L, you can expect that successful searches will require

$$\textbf{\textit{roughly}} \quad \frac{L}{2} + 1$$

key comparisons on the average. Note that if each cell of the table had a chain of the same length as all the other cells, then the length of each chain would be exactly L.

The point of view we have adopted in the foregoing discussion is that we are doing our calculations before the table is constructed. We are trying to predict how the "average" table will perform. After the table is actually constructed, its particular performance may be better or worse than the predicted, a priori expected value. Thus, for example, if by some stroke of good luck, the table we construct has all its chains of exactly equal length L, then in such a table the expected number of key comparisons during a successful search will be $\frac{L+1}{2}$. If, instead, by some dreadful misfortune, the table we construct has only a single chain (all objects happen by chance to hash to the same cell), then in such a table the expected number of key comparisons during a successful search will be $\frac{n+1}{2}$.

Example 6.8.4 **(Probabilistic Analysis of Quicksort)**: When we analyzed the simple Quicksort algorithm given in Fig. 5.63, we looked only at the best and worst cases. We found that a worst case occurs when the objects to be sorted are already in correct order or in reverse order from the desired order. In these cases the running time for Quicksort is $\Theta(n^2)$. We did not study the question of how Quicksort can be expected to perform when given an array of objects in random order. An intuitive consideration of this case can be found in [3].

Refer back to the Quicksort function on p. 269. The largest part of the work in the function is done during the loops where two objects are compared each time the loop condition is tested. Object swaps are also performed, but the number of swaps is less than the number of object comparisons. So let's compute the expected number of object comparisons that will be made when this function is called with an array of n distinct objects. To do this, we must introduce an appropriate random variable. We'll use X_n to denote the number of object comparisons that will be made when the Quicksort function is called with an array of n objects in random order. Clearly X_n is a random variable. It can be as small as $n\lg(n)$ (approximately) or as large as $n^2/2$ (approximately). Also note that $X_0 = 0$ and $X_1 = 0$ (these fall under the base case test at the beginning of the function). What we want to compute is $E(X_n)$ for all $n \geq 0$.

Since the Quicksort function is recursive, it is natural to seek a recurrence relation involving $E(X_n)$ when $n \geq 2$. The idea is to decompose X_n into the number of comparisons that are made during the non-recursive part of the function body plus the number of comparisons that are made during the two recursive calls. Let Y_n denote the number of comparisons that will be made during the recursive call on the part of the array to the left of the final position of the partitioning object, and let Z_n denote the number of comparisons that will be made during the recursive call on the right part of the array. Then $X_n =$ (number of object comparisons during the non-recursive part)$+Y_n + Z_n$ for all $n \geq 2$.

Every object in the array except the partitioning object is compared at least once with the partition object that's at the left end of the array. This means there are at least $n-1$ object comparisons. Are there any more comparisons? Yes. When the loop subscripts i and j stop moving, it is because they have "crossed over each other" (this is guaranteed by the fact that the objects in the array are all different from each other), so the loop controlled by the condition `i <j` ends by having two objects compared for a second time against the partitioning object. Thus the total number of object comparisons made during the loops is exactly $(n-1)+2 = n+1$. (If you are not certain about this, take a very small example such as an array of length 5 and look at all the various possibilities for where i and j might stop.)

Now we are able to write our equation for X_n this way:

$$X_n = n + 1 + Y_n + Z_n \quad \text{for all } n \geq 2.$$

It follows that

$$E(X_n) = n + 1 + E(Y_n) + E(Z_n) \text{ for all } n \geq 2.$$

The problem we encounter at this point is that Y_n and Z_n will depend on how the array `a[first..last]` "splits" into two subarrays to be sorted recursively. We need to account for all the different possibilities, which are

- the subarray on the left is of length 0 and the subarray on the right is of length $n - 1$;
- the subarray on the left is of length 1 and the subarray on the right is of length $n - 2$; etc.
- the subarray on the left is of length $n - 1$ and the subarray on the right is of length 0.

Let A_k denote the event that the subarray on the left is of length k, where k is an integer in the range $0 \leq k \leq n - 1$. Then by Theorem 8.0.39,

$$E(Y_n) = P(A_0)E(Y_n|A_0) + P(A_1)E(Y_n|A_1) + \cdots + P(A_{n-1})E(Y_n|A_{n-1}) \quad \text{for all } n \geq 2.$$

The event A_0 occurs if and only if the partitioning object in cell `a[first]` is the smallest object in the array. Since the objects in the array are in random order, $P(A_0) = 1/n$. Similarly, the event A_1 occurs if and only if the partitioning object is the next-to-smallest object in the array, so $P(A_1) = 1/n$. And so on. Thus

$$E(Y_n) = \frac{1}{n}[E(Y_n|A_0) + E(Y_n|A_1) + \cdots + E(Y_n|A_{n-1})] \quad \text{for all } n \geq 2. \tag{6.1}$$

Now note that if either A_0 or A_1 occurs, then the recursion on the left subarray encounters a base case, so the recursive call on the left returns with 0 object comparisons. It follows that $E(Y_n|A_0) = 0 = E(Y_n|A_1)$. For $k \geq 2$, if A_k occurs, then the recursion on the left subarray is a Quicksort on an array of k distinct objects in random order (you need to think a little to see that the left array really will be in random order). Thus $E(Y_n|A_k) = E(X_k)$ when $k \geq 2$. Thus

$$E(Y_n) = \frac{1}{n}[0 + 0 + E(X_2) + E(X_3) + \cdots + E(X_{n-1})] \quad \text{for all } n \geq 2.$$

In the special case where $n = 2$, it will be understood that this sum is 0 (to see this, look at Eq. (6.1)).

Now that we have an expression for $E(Y_n)$, let's go back and look at $E(Z_n)$, which is the expected number of object comparisons during the recursion on the right subarray. By arguments entirely symmetric with those for Y_n, we can derive exactly the same formula for $E(Z_n)$ that we have derived for $E(Y_n)$, although the terms in the sum will appear in the opposite order. Thus $E(Z_n) = E(Y_n)$. Putting together all the equations we have derived, we find that

$$E(X_n) = n + 1 + 2E(Y_n) = n + 1 + \frac{2}{n}[E(X_2) + E(X_3) + \cdots + E(X_n - 1)] \quad \text{for all } n \geq 2.$$

This is a recurrence relation in the quantity $E(X_n)$. To make it look more familiar, write $F(n)$ for $E(X_n)$. Also note that $F(0) = E(X_0) = 0$ and $F(1) = E(X_1) = 0$. Thus we have this problem:

$$F(0) = 0, \quad F(1) = 0, \quad F(n) = n + 1 + \frac{2}{n}[F(2) + F(3) + \cdots + F(n-1)] \quad \text{for all } n \geq 2. \tag{6.2}$$

To try to solve it we might begin by multiplying the recurrence relation through by n:

$$nF(n) = n^2 + n + 2[F(2) + F(3) + \cdots + F(n-1)] \quad \text{for all } n \geq 2. \tag{6.3}$$

To get rid of all those low order terms we'll use a little trick. Replace n by $n - 1$ throughout:

$$(n-1)F(n-1) = (n-1)^2+(n-1)+2[F(2)+F(3)+\cdots+F(n-2)] \quad \text{for all } n \geq 3. \tag{6.4}$$

Subtracting the left and right sides of (6.4) from the sides of (6.3) gets rid of many terms on the right:

$$nF(n)-(n-1)F(n-1) = n^2-(n-1)^2+n-(n-1)+2F(n-1) \quad \text{for all } n \geq 3.$$

Rearrangement of terms and some algebraic simplification transforms this into the equation

$$nF(n) = (n+1)F(n-1) + 2n \quad \text{for all } n \geq 3.$$

What are the initial values for this problem? We know that $F(0) = F(1) = 0$, but for the problem above we need $F(2)$. We can get this from Eq. (6.2). We have $F(2) = 2 + 1 + \frac{2}{2}[0] = 3$.

So we are trying to solve

$$F(2) = 3, \quad nF(n) = (n+1)F(n-1) + 2n \quad \text{for all } n \geq 3.$$

Dividing through by both n and $n + 1$ gives this problem:

$$F(2) = 3, \quad \frac{F(n)}{n+1} = \frac{F(n-1)}{n} + \frac{2}{n+1} \quad \text{for all } n \geq 3.$$

Now it is easy to spot a pattern for substitution: let $G(n) = \frac{F(n)}{n+1}$. This transforms the problem into

$$G(2) = \frac{3}{3} = 1, \quad G(n) = G(n-1) + \frac{2}{n+1} \quad \text{for all } n \geq 3.$$

We can solve this by the technique we used in Example 4.5.1 on p. 153.

$G(3) = G(2) + \frac{2}{3+1} = 1 + \frac{2}{4}$

$G(4) = G(3) + \frac{2}{4+1} = 1 + \frac{2}{4} + \frac{2}{5}$

$G(5) = G(4) + \frac{2}{5+1} = 1 + \frac{2}{4} + \frac{2}{5} + \frac{2}{6}$

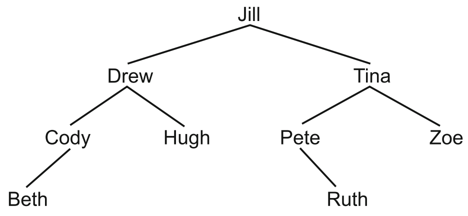

Fig. 6.31 A random binary search tree

etc.
In general,

$$G(n) = 1 + \frac{2}{4} + \frac{2}{5} + \frac{2}{6} + \cdots + \frac{2}{n+1} = 1 + 2\left(\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \cdots + \frac{1}{n+1}\right)$$

for all $n \geq 3$.

The sum in parenthesis in the formula above is the harmonic sum $\mathrm{H}(n)$ defined on p. 33 missing the first 3 terms. Recalling the definition of $G(n)$ in terms of $F(n)$, we can write

$$\frac{F(n)}{n+1} = 1 + 2\left[\mathrm{H}(n+1) - 1 - \frac{1}{2} - \frac{1}{3}\right] \quad \text{for all } n \geq 3.$$

This yields

$$F(n) = 2(n+1)\mathrm{H}(n+1) - \frac{8}{3}(n+1) \quad \text{for all } n \geq 3.$$

Now recall that $E(X_n) = F(n)$, so we have shown that $E(X_n) \sim 2n\ln(n)$. If we want to put this in terms of the logarithm base 2, we can get

$$E(X_n) \sim 2\,n\frac{\lg(2)}{\lg(e)} \approx 1.39\,n\lg(n).$$

Example 6.8.5 **(Random Binary Search Trees)**: Suppose we are given a file containing n objects, each having some key (distinct from all the other keys in the file). Suppose further that we write a program that reads the file and creates a binary search tree by repeatedly inserting each newly read object into the binary search tree, with no re-balancing of the AVL kind. (Cf. pp. 227–236.) Then the resulting tree may have a variety of shapes, and its height can be any number from $\lfloor \lg(n) \rfloor$ to $n-1$. If the objects in the file are in random order, then we'll call a binary search tree built by this method a random binary search tree.

Now suppose we want to calculate the number of nodes that will be examined during a successful search for a specified key in a random binary search tree with n nodes. This number is a random variable, and its value might be anything from 1 (if the specified key happens to be located at the root) to $h+1$, where h is the height of the tree. For example, consider the random search tree generated by a file containing names in this order: Jill, Tina, Pete, Drew, Ruth, Cody, Zoe, Hugh, Beth. The resulting tree is shown in Fig. 6.31.

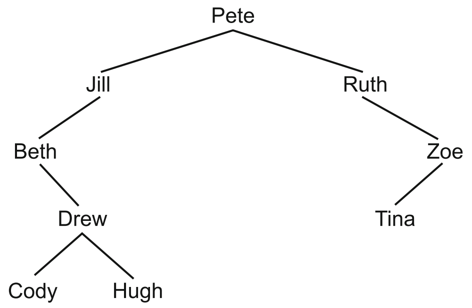

Fig. 6.32 Another random binary search tree

1 node will be examined if the specified key is Jill;
2 nodes will be examined if the specified key is Drew or Tina;
3 nodes will be examined if the specified key is Cody, Hugh, Pete, or Zoe;
4 nodes will be examined if the specified key is Beth or Ruth;

If X denotes the number of nodes that will be examined during a successful search for a specified key in this particular tree, and if each of the 9 keys is equally likely to be specified, then

$$E(X) = \frac{1}{9}[1 \cdot 1 + 2 \cdot 2 + 3 \cdot 4 + 4 \cdot 2] = \frac{25}{9} = 2\frac{7}{9}.$$

Of course, the tree might have turned out differently if the names in the file had been in a different order (although this is not necessarily true). Thus, for example, if the names in the file had been in the order Pete, Jill, Ruth, Zoe, Beth, Drew, Cody, Tina, Hugh then the tree would have looked like the one in Fig. 6.32.

If we let Y denote the number of nodes that will be examined in a successful search for a specified key in this particular tree (Fig. 6.32) and if each of the 9 keys is equally likely to be specified, then

$$E(Y) = \frac{1}{9}[1 \cdot 1 + 2 \cdot 2 + 3 \cdot 2 + 4 \cdot 2 + 5 \cdot 2] = \frac{29}{9} = 3\frac{2}{9}.$$

As we can see from these examples, the expected number of nodes that will be examined during a successful search for a specified key in a random binary search tree depends on the order of the objects in the file from which the tree will be built.

In each of the two cases we have just examined, we considered the tree to be already in existence, and we calculated the expected number of nodes that will be examined in a successful search. The underlying "statistical experiment" then consists of being given a randomly selected key from the set of keys in the tree and asked to make a search for that key in the tree. Now let's back up one step and consider the statistical experiment in which

(1) we will be given a file of n keyed records in random order;

(2) we will build the search tree by reading the file and inserting the records one by one, without rebalancing;
(3) we will then be given a randomly selected key from the set of keys in the tree and asked to make a search for that key in the tree. Each of the n keys will be equally likely to be the target key.

Let X_n denote the number of nodes that will have to be examined during the search. Let's calculate the expected value of X_n.

The difference between this problem and the two problems we looked at involving Figs. 6.31 and 6.32 is that we must now take into account (i.e., must "average over") all the different possible orderings of the records in the file and (therefore) all the different trees that could be built.

Since a search in a binary search tree is somewhat similar to a binary search in an ordered array, it is reasonable to approach this problem along those lines: we can look separately at the case where the search goes into the left subtree, the case where the search goes into the right subtree, and the case where the search stops at the root because the target key is there. Let A_1 denote the event that the search will go into the left subtree, A_2 the event that it will go to the right, and A_3 the event that the search will stop at the root. Also note that X_n can be written as $1 + Y_n$, where Y_n is the number of nodes that will be examined after the root node has been examined. Then since A_1, A_2, and A_3 form a partition of the sample space of X_n, it follows that

$$E(X_n) = 1 + E(Y_n) = 1 + P(A_1)E(Y_n|A_1) + P(A_2)E(Y_n|A_2) + P(A_3)E(Y_n|A_3).$$

Note, however, that $P(A_1)$ is not easy to compute directly because it seems to depend on how the tree is shaped. The first record in the file will determine what goes into the left subtree and what goes into the right (all keys smaller than the first will go left, all larger right). (This should remind you of what happens in Quicksort, where the first object in the array partitions the rest of the array.) Thus it will be easier to work this out the numbers above if we introduce an additional collection of events that partition the sample space in a different way: let B_0 denote the event that the left subtree will be empty; this is exactly the same as saying that the smallest key in the random file is in the first record of the file. Let B_1 denote the event that the left subtree will contain only one node; this is exactly the same as saying that the next-to-smallest key in the file comes first. More generally, for $i = 0, 1, 2, \ldots, n-1$ let B_i denote the event that the left subtree will contain exactly i nodes. Then we can think of the sample space of X_n as partitioned as shown in Fig. 6.33.

Now we can decompose $E(X_n) = 1 + E(Y_n)$ using all the intersections of the B_i's and A_j's:

$$E(X_n) = 1 + \sum_{i=0}^{n-1}[P(B_i \cap A_1)E(Y_n|B_i \cap A_1) + P(B_i \cap A_2)E(Y_n|B_i \cap A_2) + 0],$$

where the 0 at the end of the sum comes from the obvious fact that $E(Y_n|B_i \cap A_3) = 0$ for all i.

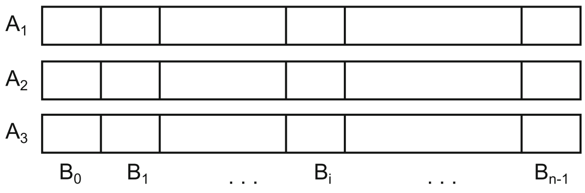

Fig. 6.33 Event partitioning

Now we can use the Chain Rule for Conditional Probability to compute $P(B_i \cap A_1)$:

$$P(B_i \cap A_1) = P(B_i)P(A_1|B_i) = \frac{1}{n} \cdot \frac{i}{n} = \frac{i}{n^2} \quad \text{for all } i,$$

because if there are i nodes in the left subtree of the search tree, then the probability is $\frac{i}{n}$ that one of those nodes will contain the target key. Similarly,

$$P(B_i \cap A_2) = P(B_i)P(A_2|B_i) = \frac{1}{n} \cdot \frac{n-1-i}{n} = \frac{n-1-i}{n^2} \quad \text{for all } i.$$

Also note that $E(Y_n|B_i \cap A_1) = E(X_i)$ and $E(Y_n|B_i \cap A_2) = E(X_{n-1-i})$, so the expression for $E(X_n)$ above now takes the following form:

$$E(X_n) = 1 + \left[\frac{0}{n^2}E(X_0) + \frac{n-1}{n^2}E(X_{n-1}) + \frac{1}{n^2}E(X_1) + \frac{n-2}{n^2}E(X_{n-2}) \right.$$
$$\left. + \frac{2}{n^2}E(X_2) + \frac{n-3}{n^2}E(X_{n-3}) + \cdots + \frac{n-1}{n^2}E(X_{n-1}) + \frac{0}{n^2}E(X_0)\right]$$

Note that every term of the form $\frac{i}{n^2}E(X_i)$ appears twice in this sum, so it reduces to

$$E(X_n) = 1 + \frac{2}{n^2}[1E(X_1) + 2E(X_2) + 3E(X_3) + \cdots + (n-1)E(X_{n-1})].$$

This formula is valid for all $n \geq 2$. Note also that $E(X_1) = 1$, and thus from the sum above,

$$E(X_2) = 1 + \frac{2}{4}E(X_1) = \frac{3}{2}.$$

To simplify the recurrence relation above, we first multiply through by n^2 to get

$$n^2E(X_n) = n^2 + 2[1E(X_1) + 2E(X_2) + 3E(X_3) + \cdots + (n-1)E(X_{n-1}] \quad \text{for all } n \geq 2. \tag{6.5}$$

Then we replace n by $n-1$ everywhere to obtain the relation

$$(n-1)^2E(X_{n-1}) = (n-1)^2 + 2[1E(X_1) + 2E(X_2) + 3E(X_3) + \cdots + (n-2)E(X_{n-2})] \tag{6.6}$$

for all $n \geq 3$. Subtracting the parts of Eq. (6.6) from parts of (6.5) gives

$$n^2E(X_n) - (n-1)^2E(X_{n-1}) = 2n - 1 + 2(n-1)E(X_{n-1}) \quad \text{for all } n \geq 3.$$

Moving $E(X_{n-1})$ to the right and combining terms gives us the rather simpler recurrence relation

$$n^2 E(X_n) = 2n - 1 + (n^2 - 1)E(X_{n-1}) \quad \text{for all } n \geq 3.$$

Here we can see the pattern $nE(X_n)$ and $(n-1)E(X_n - 1)$, but with extra nuisance factors. Dividing by $n(n+1)$ however gives the relation

$$\frac{nE(X_n)}{n+1} = \frac{2n-1}{n(n+1)} + \frac{(n-1)E(X_{n-1})}{n} \quad \text{for all n } \geq 3.$$

If we let the left side be denoted by $F(n)$ we now have

$$F(n) = \frac{2n-1}{n(n+1)} + F(n-1) \quad \text{for all } n \geq 3. \tag{6.7}$$

The initial values of $F(n)$ are $F(1) = \frac{1 \cdot 1}{1+1} = \frac{1}{2}$ and $F(2) = \frac{2 \cdot 3/2}{2+1} = 1$.

While this recurrence problem can be solved exactly, at this point we may be content to estimate $F(n)$. Note that $\frac{2n-1}{n(n+1)} < \frac{2n}{n(n+1)} = \frac{2}{n+1}$, so we have the recurrence inequality

$$F(1) = \frac{1}{2}, \quad F(2) = 1, \quad F(n) < \frac{2}{n+1} + F(n-1) \quad \text{for all } n \geq 3.$$

Writing out the first few instances of this inequality gives us

$F(3) < \frac{2}{4} + 1$

$F(4) < \frac{2}{5} + \frac{2}{4} + 1$

$F(5) < \frac{2}{6} + \frac{2}{5} + \frac{2}{4} + 1$ etc.

Clearly we have

$$F(n) < 2\mathrm{H}(n+1) - \left(\frac{2}{3} + \frac{2}{1}\right) = 2\mathrm{H}(n+1) - \frac{8}{3} \quad \text{for all } n \geq 3.$$

This shows that $F(n) = O(\lg(n))$.

We've now found an upper bound for $F(n)$. Can we get a lower bound as well? Returning to relation (6.7) we see that $\frac{2n-1}{n(n+1)} > \frac{n}{n(n+1)}$ for all positive integers n, and so

$$F(1) = \frac{1}{2}, \quad F(2) = 1, \quad F(n) > \frac{1}{n+1} + F(n-1) \quad \text{for all } n \geq 3.$$

Writing out the first few instances of the inequality will convince you that

$$F(n) > \mathrm{H}(n+2) - \left(\frac{2}{4} + \frac{2}{3} + \frac{2}{2}\right) = \mathrm{H}(n+2) - \frac{22}{3} \quad \text{for all } n \geq 3.$$

Combining this with the upper bound we derived for $F(n)$ gives us

$$2\,\mathrm{H}(n+2) - \frac{22}{3} \leq F(n) < 2\,\mathrm{H}(n+1) - \frac{8}{3}$$

It is easy to show that both $\mathrm{H}(n+1)$ and $\mathrm{H}(n+2)$ are asymptotic to $\mathrm{H}(n)$, which is asymptotic to $\ln(n)$. Thus $F(n) \sim 2\ \ln(n)$.

Now let's return to $E(X_n)$. From the equation $F(n) = \dfrac{nE(X_n)}{n+1}$ we get $E(X_n) = \dfrac{n+1}{n}F(n)$. Since $\dfrac{n+1}{n} \to 1$ as $n \to \infty$, we can conclude that

$$E(X_n) \sim 2\ \ln(n) \approx 1.39\ \ln(n).$$

A warning is in order here: files in the "real world" often arrive for processing with their objects already partially sorted. Just because you know nothing of the way in which a file of keyed objects was constructed is not a good reason to assume that the objects are in random key order. When the objects from a partially sorted file are put into a binary search tree, the result is almost always a collection of long chains of nodes. As a result, the expected search times may not remain logarithmic in the number of nodes. If you must build a binary search tree from a file about which you know nothing, you would be wise to "shuffle" its objects into random order before starting the tree building process, or else use re-balancing as the tree is constructed, as for example with AVL trees.

6.8.1 Exercises

6.8.6 In Example 6.8.2, p. 344, we calculated an asymptotic formula for the expected number of object comparisons that will be made during a successful binary search of an ordered array of n objects. Now do a similar calculation for the expected number of object comparisons that will be made during an unsuccessful binary search of an ordered array of n objects. Use the same binary search algorithm as in Example 6.8.2. Assume that the target object has equal probability of belonging at any "cell boundary" in the array, i.e., between any two objects in the array or at the ends.

6.8.7 Suppose we use the binary search algorithm given in Fig. 5.15b, p. 201 to make a search of an ordered array of n objects.

(a) Calculate the expected number of object comparisons that will be made during a search when the target of the search is not in the array (which, of course, is not discovered until the search ends). Assume that the target object has equal probability of belonging at any "cell boundary" in the array, i.e., between any two objects in the array or at the ends (as in Exercise 6.8.6 above).
(b) Calculate the exact number of object comparisons that will be made if the target is larger than every object of the array. (This is not an "expected value" problem; it can be solved with a recurrence relation.)
(c) Let X_n denote the number of object comparisons that will be made during a search when the target of the search is in the array (which will be discovered only when the search loop terminates). Assume that the target object is equally

likely to be in any of the n cells. Using the answer for part (b) above, write a recurrence relation whose solution would provide a formula for $E(X_n)$. You are not required to solve the recurrence relation, but see if you can get some idea about the asymptotic behavior of $E(X_n)$.

6.8.8 Suppose we are planning to create a separate chaining hash table with load index L. The insertion and search algorithms will be as given in Example 6.8.3 on p. 346, so the chains are unordered. What is the expected number of object comparisons that will be made in such a table during an unsuccessful search for a given key value k?

6.8.9 Suppose we are planning to create a separate chaining hash table of size M with *ordered* linked lists. The table will be initialized so that each chain consists of a single "dummy" hash node containing a "plus infinity" key value, i.e., a key value so large that no actual key can exceed it. The initialization, insertion, and search algorithms are as shown in Fig. 6.34. Suppose we plan to place n objects in the table.

(a) What is the expected number of key comparisons that will be required during a successful search of the table for a given key value k?
(b) What is the expected number of key comparisons that will be required during an unsuccessful search? Assume that the hash function will be a good one. You may use the answer for Exercise 6.6.6 on p. 334 if you have solved it.

6.8.10 Suppose we plan to use the Sample Select Algorithm 3 in Fig. 6.12 on p. 313 to choose a random sample of size K from a file known to contain N records, where $K \leq N$. Let $X_{N,K}$ denote the number of objects that will have to be read from the file to obtain the sample. Then $X_{N,K}$ is a random variable whose smallest possible value is K (the case where the first K records happen by chance to be chosen) and whose largest possible value is N (the case where the final, i.e., K-th, record is not chosen until the last record of the file is reached. Let's calculate $E(X_{N,K})$ in terms of N and K. In the totally trivial case where $K = 0$, we will not read any records from the file, so we can say that $E(X_{N,0}) = 0$ for all positive integers N. Also note that in the special case where $K = N$, we will have to read every record in the file, so we can say that $E(X_{N,N}) = N$ for all positive integers N. Now consider the case where $0 < K < N$. We will certainly have to read the first record from the file, so let's introduce the random variable $Y_{N,K}$ to stand for the number of records that will have to be read after the first one. Then $X_{N,K} = 1 + Y_{N,K}$.

(a) Use Theorem 8.0.38 on p. 461 to derive a recurrence relation for $E(X_{N,K})$ in terms of $E(X_{N-1,K})$ and $E(X_{N-1,K-1})$, valid for all integers N and K satisfying $0 < K < N$.
Hint: express $E(Y_{N,K})$ in terms of events that say whether the first record read from the file is kept or rejected.

```
// Initializes all the elements of the hash table with an "unused" flag +infinity.
void initialize (list_node_ptr T[])
{
  int i;
  for (i = 0; i < M; ++i)
  {
    T[i] = new list_node;             // Dynamically allocate a node.
    T[i]->info.key = "plus infinity"; // Genuine code will require us to
  }                                   // know the value of +infinity.
}

// Returns true if insertion is successful.
// Returns false if T already contains an object whose key matches x.key.
bool insert (otype &x, list_node_ptr T[])
{
  int i = hash(x.key);
 list_ node_ptr temp = T[i];          // Create a temporary hash node pointer.
  while (temp->info.key < x.key)
    temp = temp->next;                // Search the linked list. Loop must halt.
  if (temp->info.key == x.key)
    return false;                     // EXIT from function. Insertion failed.
  else
  {                                   // Insert copy of x into ordered chain.
    list_node_ptr p = new list_node;
    p->info = temp->info;             // Copy the bigger object into new node.
    temp->info = x;                   // Copy x into old node; overwrite bigger
    p->next = temp->next;             // Splice new node into the list following
    temp->next = p;                   // the copy of x.
  }
  return true;                        // Indicate that the insertion operation
}                                     //  was successful.

// Returns pointer to the unique object in T with key k if such an object exists.
// Otherwise it returns NULL.
otype* search(list_node_ptr T[], key_type k)
{
  int i = hash(k);
  list_node_ptr temp = T[i];          // Create a temporary hash node pointer.
  while (temp->info.key < k)
    temp = temp->next;                // Search the linked list. Loop must halt.
  if (temp->info.key == k)            // Decide what to return.
    return &(temp->info);
  else
    return NULL;
}
```

Fig. 6.34 Functions for a hash table with separate chaining using ordered linked lists

(b) Let's try to guess the solution for the recurrence relation derived in part (a). We already have the initial values $E(X_{N,0}) = 0$ and $E(X_{N,N}) = N$. On average, the K records to be chosen should be spread out "evenly" among the $N - K$ records that will not be chosen. This means that the $N - K$ non-chosen records will, on average, be split into $K + 1$ equal parts, each of size $\dfrac{N-K}{K+1}$. One of

```
// Searches for a target key in a binary search tree. Returns a pointer to the
// node containing it if found, NULL otherwise.

template <class otype, class key_type>
tree_node_ptr key_location (tree_node_ptr p, const key_type * target)
{
  while (p)
   if (key(p->datum) < target)       // go into right subtree
     p = p->right;
   else if (key(p->datum) > target)  // go into left subtree
     p = p->left;
   else
     return p;                       // target key has been found

  return NULL;                       // could not find target key
}
```

Fig. 6.35 Search in a binary search tree

these parts will precede the first chosen record, one will follow the last (K-th) chosen record, and the other parts will be in between successive chosen records. Since we will have to read to the last chosen record, this will, on average, leave just $\dfrac{N-K}{K+1}$ records unread (the ones that follow the last chosen record). Thus we can expect that $N - \dfrac{N-K}{K+1} = \dfrac{NK+N-(N-K)}{K+1} = \dfrac{(N+1)K}{K+1}$ records will be read. That is, our conjecture is that $E(X_{N,K}) = \dfrac{(N+1)K}{K+1}$. Verify that this formula satisfies all the initial conditions we have stated and also satisfies the recurrence relation derived in part (a).

Note: This problem was solved in a different way earlier in this chapter, in Exercises 6.7.7 and 6.7.8.

6.8.11 Consider the experiment in which a file of n distinct keys in random order is formed into a binary search tree without rebalancing.

(a) Find an asymptotic expression for the expected number of nodes that will have to be examined in an unsuccessful search for a specified key. Assume that the target key is equally likely to lie between any two keys in the file or at either "end" of the ordered list of keys. (There are $n + 1$ such locations where it might belong, each with equal probability.)
(b) Find an asymptotic expression for the expected number of key comparisons that will have to be examined in a successful search for a specified key if the search algorithm in Fig. 6.35 is used. Note the order in which the comparisons are made. Assume that all n keys are equally likely to be the target of the search.

6.8.12 If we want to solve the recurrence relation (6.7) on p. 355 exactly, then it is helpful to decompose the fraction into two separate ones:

$$F(1) = \frac{1}{2}, \quad F(2) = 1, \quad F(n) = \frac{2}{n+1} - \frac{1}{n(n+1)} + F(n-1) \quad \text{for all } n \geq 3.$$

(a) By computing the values of sums of the form

$$\frac{1}{1 \cdot 2}$$
$$\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3}$$
$$\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 4}$$

obtain (by guesswork) a closed form for the general sum

$$\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 4} + \cdots + \frac{1}{n(n+1)}.$$

The proof of this closed form uses mathematical induction.

(b) Now solve for $F(n)$ exactly.