Chapter 12

Algorithm Analysis

Goals

- Analyze algorithms
- Understand some classic searching and sorting algorithms
- Distinguish runtime order: O(1), O(n), O(n log n), and $O(n^2)$

12.1 Algorithm Analysis

This chapter introduces a way to investigate the efficiency of algorithms. Examples include searching for an element in an array and sorting elements in an array. The ability to determine the efficiency of algorithms allows programmers to better compare them. This helps when choosing a more efficient algorithm when implementing data structures.

An **algorithm** is a set of instructions that can be executed in a finite amount of time to perform some task. Several properties may be considered to determine if one algorithm is better than another. These include the amount of memory needed, ease of implementation, robustness (the ability to properly handle exceptional events), and the relative efficiency of the runtime.

The characteristics of algorithms discussed in this chapter relate to the number of operations required to complete an algorithm. A tool will be introduced for measuring anticipated runtimes to allow comparisons. Since there is usually more than one algorithm to choose from, these tools help programmers answers the question: “Which algorithm can accomplish the task more efficiently?”

Computer scientists often focus on problems related to the efficiency of an algorithm: Does the algorithm accomplish the task fast enough? What happens when the number of elements in the collection grows from one thousand to one million? Is there an algorithm that works better for storing a collection that is searched frequently? There may be two different algorithms that accomplish the same task, but all other things being equal, one algorithm may take much longer than another when implemented and run on a computer.

Runtimes may be reported in terms of actual time to run on a particular computer. For example, SortAlgorithmOne may require 2.3 seconds to sort 2000 elements while SortAlgorithmTwo requires 5.7 seconds. However, this time comparison does not ensure that SortAlgorithmOne is better than SortAlgorithmTwo. There could be a good implementation of one algorithm and a poor implementation of the other. Or, one computer might have a special hardware feature that SortAlgorithmOne takes advantage of, and without this feature SortAlgorithmOne would not be faster than SortAlgorithmTwo. Thus the goal is to compare algorithms, not programs. By comparing the actual running times of SortAlgorithmOne and SortAlgorithmTwo, programs are being considered—not their algorithms. Nonetheless, it can prove useful to observe the behavior of algorithms by comparing actual runtimes — the amount of time required to perform some operation on a computer. The same tasks accomplished by different algorithms can be shown to differ dramatically, even on very fast computers. Determining how long an algorithm takes to complete is known as algorithm analysis.

Generally, the larger the size of the problem, the longer it takes the algorithm to complete. For example, searching through 100,000 elements requires more operations than searching through 1,000 elements. In the following discussion, the variable **n** will be used to suggest the "*n*umber of things".

We can study algorithms and draw conclusions about how the implementation of the algorithm will behave. For example, there are many sorting algorithms that require roughly n^2 operations to arrange a list into its natural order. Other algorithms can accomplish the same task in $n * \log_2 n$ operations. There can be a large difference in the number of operations needed to complete these two different algorithms when n gets very large.

Some algorithms don't grow with n. For example, if a method performs a few additions and assignment operations, the time required to perform these operations does not change when n increases. These instructions are said to run in *constant time*. The number of operations can be described as a constant function f(n) = k, where k is a constant.

Most algorithms do not run in constant time. Often there will be a loop that executes more operations in relation to the size of the data variable such as searching for an element in a collection, for example. The more elements there are to locate, the longer it can take.

Computer scientists use different notations to characterize the runtime of an algorithm. The three major notations O(n), Ω(n), and Θ(n) are pronounced “big-O”, “big-Omega”, and “big-Theta”, respectively. The big-O measurement represents the upper bound on the runtime of an algorithm; the algorithm will never run slower than the specified time. Big-Omega is symmetric to big-O. It is a lower

bound on the running time of an algorithm; the algorithm will never run faster than the specified time. Big-Theta is the tightest bound that can be established for the runtime of an algorithm. It occurs when the big-O and Omega running times are the same, therefore it is known that the algorithm will never run faster or slower then the time specified. This textbook will introduce and use only big-O.

When using notation like big-O, the concern is the *rate of growth* of the function instead of the precise number of operations. When the size of the problem is small, such as a collection with a small size, the differences between algorithms with different runtimes will not matter. The differences grow substantially when the size grows substantially.

Consider an algorithm that has a cost of $n^2 + 80n + 500$ statements and expressions. The upper bound on the running time is $O(n^2)$ because the larger growth rate function dominates the rest of the terms. The same is true for coefficients and constants. For very small values of n, the coefficient 80 and the constant 500 will have a greater impact on the running time. However, as the size grows, their impact decreases and the highest order takes over. The following table shows the growth rate of all three terms as the size, indicated by n, increases.

Function growth and weight of terms as a percentage of all terms as n increases

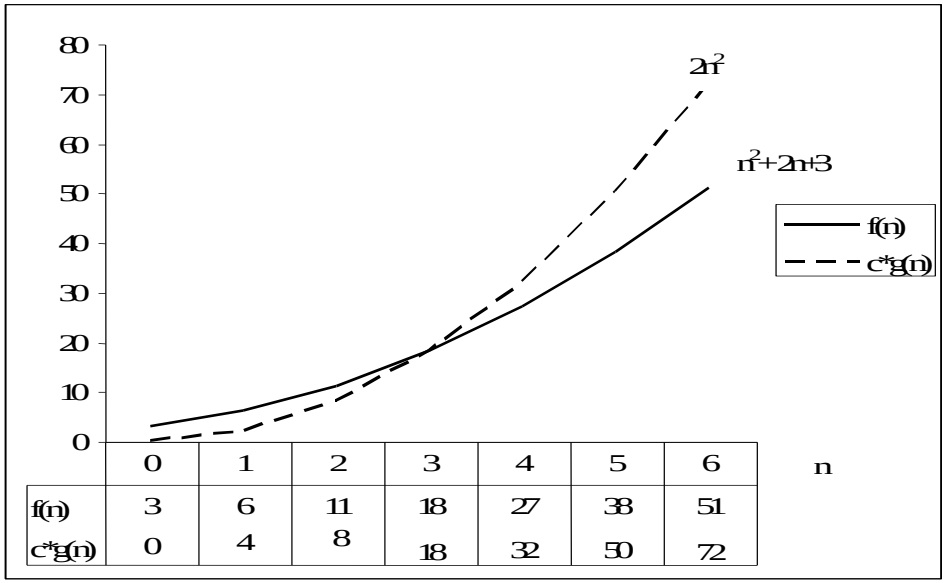
n	f(n)	n^2	80n	500
10	1,400	100 (7%)	800 (57%)	500 (36%)
100	18,500	10,000 (54%)	8,000 (43%)	500 (3%)
1000	1,0805,000	1,000,000 (93%)	80,000 (7%)	500 (0%)
10000	100,800,500	100,000,000 (99%)	800,000 (1%)	500 (0%)

This example shows that the constant 500 has 0% impact (rounded) on the running time as n increases. The weight of this constant term shrinks to near 0%. The term 80n has some impact, but certainly not as much as the term n^2, which raises n to the 2nd power. Asymptotic notation is a measure of runtime complexity when n is large. Big-O ignores constants, coefficients, and lower growth terms.

12.2 Big-O Definition

The big-O notation for algorithm analysis has been introduced with a few examples, but now let's define it a little further. We say that f(n) is O(g(n)) if and only if there exist two positive constants c and N such that $f(n) <= c*g(n)$ for all $n > N$. We say that g(n) is an asymptotic upper bound for f(n). As an example, consider this graph where $f(n) = n^2 + 2n + 3$ and $g(n) = c*n^2$

Show that $f(n) = n^2 + 2n + 3$ is $O(n^2)$



	0	1	2	3	4	5	6	n
f(n)	3	6	11	18	27	38	51	
c*g(n)	0	4	8	18	32	50	72	

To fulfill the definition of big-O, we only find constants c and N at the point in the graph where c*g(n) is greater than f(n). In this example, this occurs when c is picked to be 2.0 and N is 4. The above graph shows that if n < N, the function g is at or below the graph of f. In this example, when n ranges from 0 through 2, g(n) < f(n). c*g(n) is equal to f(n) when c is 2 and n is 3 ($2*3^2 == 18$ as does $3^2+2*3+3$). And for all n >= 4, f(n) <= c*g(n). Since g(n) is larger than f(n) when c is 2.0 and N >= 4, it can be said that f(n) is O(g(n)). More specifically, f(n) is $O(n^2)$.

The g(n) part of these charts could be any of the following common big-O expressions that represent the upper bound for the runtime of algorithms:

Big-O expressions and commonly used names

O(1)	*constant (an increase in the amount of data (n) has no effect)*
O(log n)	*logarithmic (operations increase once each time n doubles)*
O(n)	*linear*
O(n log n)	*n log n*
O(n^2)	*quadratic*
O(n^3)	*cubic*
O(2^n)	*exponential*

Properties of Big-O

When analyzing algorithms using big-O, there are a few properties that will help to determine the upper bound of the running time of algorithms.

Property 1, coefficients: If f(n) is x * g(n) then f(n) is O(g(n))
This allows the coefficient (x) to be dropped.
Example:
f(n) = 100*g(n)
then f(n) is O(n)

Property 2, sum: If $f_1(n)$ is O(g(n)) and $f_2(n)$ is O(g(n)) then $f_1(n) + f_2(n)$ is O(g(n))
This property is useful when an algorithm contains several loops of the same order.
Example:
$f_1(n)$ is O(n)
$f_2(n)$ is O(n)
then $f_1(n) + f_2(n)$ is O(n) + O(n), which is O(n)

Property 3, sum: If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$. This property works because we are only concerned with the term of highest growth rate.

Example:

$f_1(n)$ is $O(n^2)$
$f_2(n)$ is $O(n)$
so $f_1(n) + f_2(n) = n^2 + n$, which is $O(n^2)$

Property 4, multiply: If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n) * f_2(n)$ is $O(g_1(n) * g_2(n))$. This property is useful for analyzing segments of an algorithm with nested loops.

Example:

$f_1(n)$ is $O(n^2)$
$f_2(n)$ is $O(n)$
then $f_1(n) * f_2(n)$ is $O(n^2) * O(n)$, which is $O(n^3)$

12.3 Counting Operations

We now consider one technique for analyzing the runtime of algorithms—approximating the number of operations that would execute with algorithms written in Java. This is the *cost* of the code. Let the cost be defined as the total number of operations that would execute in the worst case. The operations we will measure may be assignment statements, messages, and logical expression evaluations, all with a cost of 1. This is very general and does not account for the differences in the number of machine instructions that actually execute. The cost of each line of code is shown in comments. This analysis, although not very exact, is precise enough for this illustration. In the following code, the first three statements are assignments with a cost of 1 each.

Example 1

```
int n = 1000;                    // 1 instruction
int operations = 0;              // 1
int sum = 0;                     // 1
for (int j = 1; j <= n; j++) {   // 1 + (n+1) +  n
  operations++;                  // n
  sum += j;                      // n
}
```

The loop has a logical expression $j \leq n$ that evaluates $n + 1$ times. (The last time it is false.) The increment j++ executes n times. And both statements in the body of the loop execute n times. Therefore the total number of operations $f(n) = 1 + 1 + 1 + 1 + (n+1) + n + n + n = 4n + 5$. To have a runtime O(n), we must find a real constant c and an integer constant N such that $4n + 5 \leq cN$ for all $N > n$. There are an infinite set of values to choose from, for example $c = 6$ and $N = 3$, thus $17 \leq 18$. This is also true for all $N > 3$, such as when $N = 4$ ($21 \leq 24$) and when $N = 5$ ($25 < 30$). A simpler way to determine runtimes is to drop the lower order term (the constant 5) and the coefficient 4.

Example 2

A sequence of statements that does not grow with n is O(1) (constant time). For example, the following algorithm (implemented as Java code) that swaps two array elements has the same runtime for any sized array. $f(n) = 3$, which is O(1).

```
private void swap(String[] array, int left, int right)  {
  String temp = array[left];       // 1
  array[left] = array[right];      // 1
  array [right] = temp;            // 1
}
```

For a runtime O(1), we must find a real constant c and an integer constant N such that $f(n) = 3 \leq cN$. For example, when $c = 2$ and $N = 3$ we get $3 \leq 6$.

Example 3

The following code has a total cost of $6n + 3$, which after dropping the coefficient 6 and the constant 3, is O(n).

```
// Print @ for each n
for (int i = 0; i < 2 * n; i++)   // 1 + (2n+1) + 2n
  System.out.print("@");          // 2n+1
```

To have a runtime O(n), we must find a real constant c and an integer constant N such that $f(n) = 2n+1 \leq cN$. For example, $c = 4$ and $N = 3$ ($7 \leq 12$).

Example 4

Algorithms under analysis typically have one or more loops. Instead of considering the comparisons and increments in the loop added to the number of times each instruction inside the body of the loop executes, we can simply consider how often the loop repeats. A few assignments before or after the loop amount to a small constant that can be dropped. The following loop, which sums all array elements and finds the largest, has a total cost of about $5n + 1$. The runtime once again, after dropping the coefficient 5 and the constant 1, is O(n).

```
double sum = 0.0;              // 1
double largest = a[0];         // 1
for (int i = 1; i < n; i++) { // 1 + n + (n-1)
  sum += a[i];                 // n-1
  if (a[i] > largest)          // n-1
    largest = a[i];            // n-1, worst case: a[i] > largest always
  }
```

Example 5

In this next example, two loops execute some operation n times. The total runtime could be described as O(n) + O(n). However, a property of big O is that the sum of the same orders of magnitude is in fact that order of magnitude (see big-O properties below). So the big-O runtime of this algorithm is O(n) even though there are two individual `for` loops that are O(n).

```
// f(n) = 3n + 5 which is O(n)
// Initialize n array elements to random integers from 0 to n-1
int n = 10;                                              // 1
int[] a = new int[n];                                    // 1
java.util.Random generator = new java.util.Random();     // 1
for (int i = 0; i < n; i++)                              // 2n + 2
  a[i] = generator.nextInt(n);                           // n

// f(n) = 7n + 3 which is O(n)
// Rearrange array so all odd integers in the lower indexes
int indexToPlaceNextOdd = 0;             // 1
for (int j = 0; j < n; j++) {            // 2n + 2
  if (a[j] % 2 == 1) {                   // n: worst case always  odd
    // Swap the current element into
    // the sub array of odd integers
    swap(a, j, indexToPlaceNextOdd);     // n
    indexToPlaceNextOdd++;               // n
  }
}
```

To reinforce that O(n) + O(n) is still O(n), all code above can be counted as f(n)= 10n + 8, which is O(n). To have a runtime O(n), use c = 12 and N = 4 where $10n+8 \leq cN$, or $48 \leq 48$.

Example 6

The runtime of nested loops can be expressed as the product of the loop iterations. For example, the following inner loop executes (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 times, which is n/2 operations. The outer loop executes the inner loop n-1 times. The inner loop executes (n-1)*(n/2) twice, which is $n^2 - n$ operations. Add the others to get $f(n) = 3n^2+4n-2$. After dropping the coefficient from n^2 and the lower order terms 4n and –2, the runtime is $O(n^2)$.

```
// Rearrange arrays so integers are sorted in ascending order
for (int top = 0; top < n - 1; top++) {         // 2n + 1
  int smallestIndex = top;                      //  n - 1
  for (int index = top; index < n; index++) {   // (n-1)*(2n)
    if (a[index] < a[smallestIndex])            // (n-1)*(n/2)
      smallestIndex = index;                    // (n-1)*(n/2) at worst
  }
  // Swap smallest to the top index
  swap(a, top, smallestIndex);                  // 3n
}
```

To have a runtime $O(n^2)$, use c = 4 and N = 4 where $3n^2 + 4n - 2 \leq cN$, or $62 \leq 64$.

Example 7

If there are two or more loops, the longest running loop takes precedence. In the following example, the entire algorithm is $O(n^2) + O(n)$. The maximum of these two orders of magnitudes is $O(n^2)$.

```
int operations = 0;                         // 1
int n = 10000;                              // 1
// The following code runs O(n*n)
for (int j = 0; j < n; j++)                 // 2n+2
  for (int k = 0; k < n; k++)               // n*(2n+2)
    operations++;                           // n*(2n+2)

// The following code runs O(n)
for (int i = 0; i < n; i++)                 // 2n+2
  operations++;                             // n
```

Since $f(n) = 4n^2+9n+6 < cn^2$ for c = 6.05 when N = 5, f(n) is $O(n^2)$.

Tightest Upper Bound

Since big-O notation expresses the notion that the algorithm will take no longer to execute than this measurement, it could be said, for example, that sequential search is $O(n^2)$ or even $O(2^n)$. However, the notation is only useful by stating the runtime as a tight upper bound. The tightest upper bound is the lowest order of magnitude that still satisfies the upper bound. Sequential search is more meaningfully characterized as O(n).

Big-O also helps programmers understand how an algorithm behaves as n increases. With a linear algorithm expressed as O(n), as n doubles, the number of operations doubles. As n triples, the number of operations triples. Sequential search through a list of 10,000 elements takes 10,000 operations in the worst case. Searching through twice as many elements requires twice as many operations. The runtime can be predicted to take approximately twice as long to finish on a computer.

Here are a few algorithms with their big-O runtimes.

- Sequential search (shown earlier) is O(n)
- Binary search (shown earlier) is O(log n)
- Many sorting algorithms such as selection sort (shown earlier) are $O(n^2)$
- Some faster sort algorithms are O(n log n) — one of these (Quicksort) will be later
- Matrix multiplication is $O(n^3)$

Self-Check

12-1 Arrange these functions by order of growth from highest to lowest

$100*n^2$ 1000 2^n $10*n$ n^3 $2*n$

12-2 Which term dominates this function when n gets really big, n^2, 10n, or 100?

$n^2 + 10n + 100$

12-3. When n = 500 in the function above, what percentage of the function is the term?

12-4 Express the tightest upper bound of the following loops in big-O notation.

a)
```
int sum = 0;
int n = 100000;
```

b)
```
int sum = 0;
for (int j = 0; j < n; j++)
  for (int k = 0; k < n; k++)
     sum += j * k;
```

c)
```
for (int j = 0; j < n; j++)
  for (int k = 0; k < n; k++)
    for (int l = 0; l < n; l++)
      sum += j * k * l;
```

d)
```
for (int j = 0; j < n; j++)
  sum++;
for (int j = 0; j < n; j++)
  sum--;
```

e)
```
for (int j = 0; j < n; j++)
  sum += j;
```

f)
```
for (int j = 1; j < n; j *= 2)
  sum += j;
```

Search Algorithms with Different Big-Os

A significant amount of computer processing time is spent searching. An application might need to find a specific student in the registrar's database. Another application might need to find the occurrences of the string "data structures" on the Internet. When a collection contains many, many elements, some of the typical operations on data structures—such as searching—may become slow. Some algorithms result in programs that run more quickly while other algorithms noticeably slow down an application.

Sequential Search

This sequential search algorithm begins by comparing the first element in the array.

```
sequentially compare all elements, from index 0 to size-1 {
    if searchID equals ID of the object , return a reference to that object
}
return null because searchID does not match any elements from index 0..size-1
```

If there is no match, the second element is compared, then the third, up until the last element. If the element being sought is found, the search terminates. Because the elements are searched one after another, in sequence, this algorithm is called **sequential** search. However since the worst case is a comparison of all elements and the algorithm is O(n), it is also known as **linear** search.

The binary search algorithm accomplishes the same task as sequential search. Binary search is more efficient. One of its preconditions is that the array must be sorted. Half of the elements can be eliminated from the search every time a comparison is made. This is summarized in the following algorithm:

Algorithm: **Binary Search, use with sorted collections that can be indexed**

```
while the element is not found and it still may be in the array {
    Determine the position of the element in the middle of the array
    If array[middle] equals the search string
        return the index
    If the element in the middle is not the one being searched for:
        remove the half of the sorted array that cannot contain the element
}
```

Each time the search element is compared to one array element, the binary search effectively eliminates half the remaining array elements from the search. This makes binary search O(n log n)

When n is small, the binary search algorithm does not see a gain in terms of speed. However when n gets large, the difference in the time required to search for an element can make the difference between selling the software and having it unmarketable. Consider how many comparisons are necessary when n grows by powers of two. Each doubling of n would require potentially twice as many loop iterations for sequential search. However, the same doubling of n would only require potentially one more comparison for binary search.

Maximum number of comparisons for two different search algorithms

Power of 2	n	Sequential Search	Binary Search
2^2	4	4	2
2^4	16	16	4
2^8	128	128	8
2^{12}	4,096	4,096	12
2^{24}	16,777,216	16,777,216	24

As n gets very large, sequential search has to do a lot more work. The numbers above represent the maximum number of iterations necessary to search for an element. The difference between 24 comparisons and almost 17 million comparisons is quite dramatic, even on a fast computer. Let us analyze the binary search algorithm by asking, "How fast is Binary Search?"

The best case is when the element being searched for is in the middle—one iteration of the loop. The upper bound occurs when the element being searched for is not in the array. Each time through the loop, the "live" portion of the array is narrowed to half the previous size. The number of elements to consider each time through the loop begins with n elements (the size of the collection) and proceeds like this: n/2, n/4, n/8, ... 1. Each term in this series represents one comparison (one loop iteration). So the question is "How long does it take to get to 1?" This will be the number of times through the loop. Another way to look at this is to begin to count at 1 and double this count until the number *k* is greater than or equal to n.

1, 2, 4, 8, 16, ... , $k >= n$ or $2^0, 2^1, 2^2, 2^3, 2^4, \ldots, 2^c >= n$

The length of this series is c+1. The number of loop iterations can be stated as "2 to what power *c* is greater than or equal to n?" .

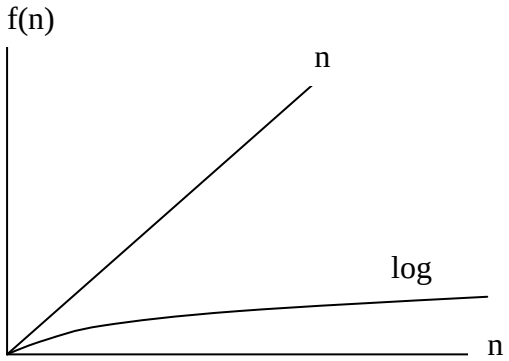
if n is 2, *c* is 1
if n is 4, *c* is 2
if n is 5, *c* is 3
if n is 100, *c* is 7
if n is 1024, *c* is 10
if n is 16,777,216, *c* is 24

In general, as the number of elements to search (n) doubles, binary search requires only one more iteration to effectively remove half of the array elements from the search. The growth of this function is said to be **logarithmic**. Binary search is O(log n). The base of the logarithm (2) is not written, for two reasons:

- The difference between $\log_2 n$ and $\log_3 n$ is a constant factor and constants are not a concern.
- The convention is to use base 2 logarithms.

The following graph illustrates the difference between linear search, which is O(n), and binary search, which takes at most $\log_2 n$ comparisons.

Comparing O(n) to O(log n)



To further illustrate, consider the following experiment: using the same array of objects, search for every element in that array. Do this using both linear search and binary search. This experiment searches for every single list element. There is one O(n) loop that calls the binary search method with an O(log n) loop. Therefore, the time to search for every element using binary search indicates an algorithm that is O(n log n).

Searching for every element in the array (1.3 gig processor, 512 meg RAM):

Binary Search for every element O(n log n)					Sequential search for every element O(n^2)		
n	Binary Search #operations	average operations per search	total time in seconds		Sequential Search #operations	average operations per search	total time in seconds
100	588	5.9	0.00		5,050	50.5	0.0
1,000	9,102	9.1	0.01		500,500	500.5	0.3
10,000	124,750	12.4	0.30		5,0005,000	5,000.5	4.7
100,000	1,581,170	15.8	2.40		5,000,050,000	50,000.5	1,168.6

The time for sequential search also reflects a search for every element in the list. An O(n) loop calls a method that in turn has a loop that executes operations as follows (searching for the first element requires 1 comparison, searching for the second element requires 2 comparisons, and searching for the last two elements requires n-1 and n operations).

1 + 2 + 3 +, ... + n-2 + n-1 + n

This sequence adds up to the sum of the first n integers: n(n+1)/2. So when n is 100, 100(100+1)/2 = 5050 operations are required. The specific number of operations after removing the coefficient 1/2 is n*(n+1). Sequentially searching for every element in a list of size n is $O(n^2)$. Notice the large difference when n is 100,000: 1,168 seconds for the $O(n^2)$ algorithm compared to 4.5 seconds for the O(n log n) operation.

One advantage of sequential search is that it does not require the elements to be in sorted order. Binary search does have this precondition. This should be considered when deciding which searching algorithm to use in a program. If the program is rarely going to search for an item, then the overhead associated with sorting before calling binary search may be too costly. However, if the program is mainly going to be used for searching, then the time expended sorting the list may be made up with repeated searches.

12.5 Example Logarithm Functions

Here are some other applications that help demonstrate how fast things grow if doubled and how quickly something shrinks if repeatedly halved.

1. Guess a Number between 1 and 100

Consider a daily number game that asks you to guess some number in the range of 1 through 1000. You could guess 1, then 2, then 3, all the way up to 1000. You are likely to guess this number in 500 tries, which grows in a linear fashion. Guessing this way for a number from 1 to 10,000 would likely require 10 times as many tries. However, consider what happens if you are told your guess is either too high, too low, or just right

Try the middle (500), you could be right. If it is too high, guess a number that is near the middle of 1..499 (250). If your initial guess was too low, check near middle of 501..1000 (750). You should find the answer in $2^c >= 1000$ tries. Here, c is 10. Using the base 2 logarithm, here is the maximum number of tries needed to guess a number in a growing range.

from 1..250, a maximum of $2^c >= 250$, c == 8
from 1..500, a maximum of $2^c >= 500$, c == 9
from 1..1000, a maximum of $2^c >= 1000$, c == 10

2. Layers of Paper to Reach the Moon

Consider how quickly things grow when doubled. Assuming that an extremely large piece of paper can be cut in half, layered, and cut in half again as often as you wish, how many times do you need to cut and layer until paper thickness reaches the moon? Here are some givens:

1. paper is 0.002 inches thick
2. distance to moon is 240,000 miles
3. 240,000 * 5,280 feet per mile * 12 inches per foot = 152,060,000,000 inches to the moon

3. Storing Integers in Binary Notation

Consider the number of bits required to store a binary number. One bit represents two unique integer values: 0 and 1. Two bits can represent the four integer values 0 through 3 as 00, 01, 10, and 11. Three bits can store the eight unique integer values 0 through 7 as 000, 001, 010, ... 111. Each time one more bit is used twice as many unique values become possible.

4. Payoff for Inventing Chess

It is rumored that the inventor of chess asked the grateful emperor to be paid as follows: 1 grain of rice on the first square, 2 grains of rice on the next, and double the grains on each successive square. The emperor was impressed until later that month he realized that the 2^{64} grains of rice on the 64th square would be enough rice to cover the earth's surface.

Answers to Self-Check Questions

12-1 order of growth, highest to lowest

-1 2^n (2 to the nth power)
-2 n^3
-3 $100*n^2$
-4 $10*n$
-5 $2*n$
-6 1000

12-2 n^2 dominates the function

12-3 percentage of the function

n^2 = 98%

10n = 1.96%

100 = 0.0392%

12-4 tightest upper bounds

-a O(1)

-b $O(n^2)$ On the order of n squared

-c $O(n^3)$

-d O(n)

-e O(n)

-f O(log n)