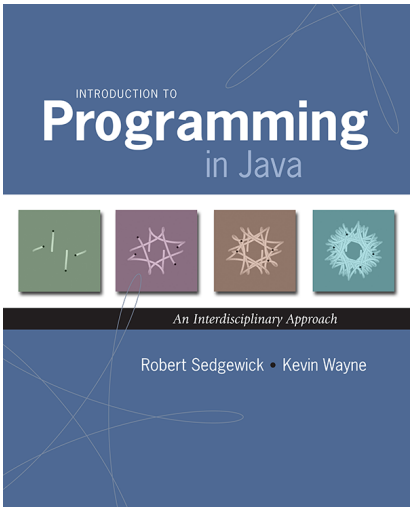
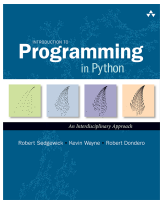


- Related Booksites



Search

4.1 Analysis of Algorithms

This section under construction.

In this section, you will learn to respect a principle whenever you program: *Pay attention to the cost.* To study the cost of running them, we study our programs themselves via the *scientific method*, the commonly accepted body of techniques universally used by scientists to develop knowledge about the natural world. We also apply mathematical analysis to derive concise models of the cost.

Scientific method.

Our approach is the *scientific method*, and it involves the following 5 step approach.

- *Observe* some feature of the natural world.
- *Hypothesize* a model that is consistent with the observations.
- *Predict* events using the hypothesis.
- *Verify* the predictions by making further observations.
- *Validate* by repeating until the hypothesis and observations agree.

One of the key tenets of the scientific method is that the experiments we design must be *reproducible*, so that others can convince themselves of the validity of the hypothesis. In addition, the hypotheses we formulate must be *falsifiable*, so that we can know for sure when a hypothesis is wrong (and thus needs revision).

Observations.

Our first challenge is to make quantitative measurements of the running time of our programs. Although measuring the exact running time of our program is difficult, usually we are happy with approximate estimates. There are a number of tools available to help us make quantitative measurements of the running time of our programs. Perhaps the simplest is a physical stopwatch or the Stopwatch.java data type (from Section 3.2). We can simply run a program on various inputs, measuring the amount of time to process each input.

Our first qualitative observation about most programs is that there is a *problem size* that characterizes the difficulty of the computational task. Normally, the problem size is either the size of the input or the value of a command-line argument. Intuitively, the running time should increase with the problem size, but the question of *how much* it increases naturally arises every time we develop and run a program.

A concrete example.

To illustrate the approach, we start with ThreeSum.java which counts the number of triples in a set of N numbers that sums to 0. What is the relationship between the problem size N and running time for `ThreeSum`?

- *Doubling hypothesis.* For a great many programs, we can quickly formulate a hypothesis for the

following question: *What is the effect on the running time of doubling the size of the input?*

- *Empirical analysis.* One simple way to develop a doubling hypothesis is to double the size of the input and observe the effect on the running time. DoublingTest.java generates a sequence of random input arrays for `ThreeSum`, doubling the array size at each step, and prints the ratio of running times of `ThreeSum.count()` for each input over the previous (which was one-half the size). If you run this program, you will find that the elapsed time increases by about a factor of eight to print each line. This leads immediately to the hypothesis that the running time increases by a factor of eight when the input size doubles. We might also plot the running times, either on a standard plot (left), which clearly shows that the rate of increase of the running time increases with input size, or on a log-log plot. The log-log plot is a straight line with slope 3, clearly suggesting the hypothesis that the running time satisfies a power law of the form cN^3.
- *Mathematical analysis.* The total running time is determined by two primary factors:
  - The cost of executing each statement.
  - The frequency of execution of each statement.

  The former is a property of the system, and the latter is a property of the algorithm. If we know both for all instructions in the program, we can multiply them together and sum for all instructions in the program to get the running time. The primary challenge is to determine the frequency of execution of the statements. Some statements are easy to analyze: for example, the statement that sets `cnt` to 0 in `ThreeSum.count()` is executed only once. Others require higher-level reasoning: for example, the `if` statement in `ThreeSum.count()` is executed precisely N(N-1)(N-2)/6 times (See Exercise 4.1.4).

Tilde notation.

We use *tilde notation* to develop simpler approximate expressions. First, we work with the *leading term* of mathematical expressions by using a mathematical device known as the tilde notation. We write ~ f(N) to represent any quantity that, when divided by f(N), approaches 1 as N grows. We also write g(N) ~ f(N) to indicate that g(N) / f(N) approaches 1 as N grows. With this notation, we can ignore complicated parts of an expression that represent small values. For example, the `if` statement in `ThreeSum.count()` is executed ~ N^3 / 6 times because N(N-1)(N-2) / 6 = N^3/6 -N^2/2 + N/3, which, when divided by N^3/6, approaches 1 as N grows.

We focus on the instructions that are executed most frequently, sometimes referred to as the *inner loop* of the program. In this program it is reasonable to assume that the time devoted to the instructions outside the inner loop is relatively insignificant.

Order of growth.

The key point in analyzing the running time of a program is this: for a great many programs, the running time satisfies the relationship T(N) ~ c f(N) where c is a constant and f(N) a function known as the *order of growth* of the running time. For typical programs, f(N) is a function such as log N, N, N log N, N^2, or N^3.

The order of growth of the running time of `ThreeSum` is N^3. The value of the constant `c` depends both on the cost of executing instructions and on details of the frequency analysis, but we normally do not need to work out the value. Knowing the order of growth typically leads immediately to a doubling hypothesis. In the case of `ThreeSum`, knowing that the order of growth is N^3 tells us to expect the running time to increase by a factor of eight when we double the size of the problem because

```
T(2N) /T(N) -> c (2N)^3 / c (N)^3 = 8
```

Order of growth classifications.

We use just a few structural primitives (statements, conditionals, loops, and method calls) to build Java programs, so very often the order of growth of our programs is one of just a few functions of the problem size, summarized in the table below.

Complexity	Description	Examples
1	*Constant* algorithm does not depend on the input size. Execute one instruction a fixed number of times	Arithmetic operations (+, -, *, /, %) Comparison operators (<, >, ==, !=) Variable declaration Assignment statement Invoking a method or function
log N	*Logarithmic* algorithm gets slightly slower as N grows. Whenever N doubles, the running time increases by a constant.	Bits in binary representation of N Binary search Insert, delete into heap or BST
N	*Linear* algorithm is optimal if you need to process N inputs. Whenever N doubles, then so does the running time.	Iterate over N elements Allocate array of size N Concatenate two string of length N
N log N	*Linearithmic* algorithm scales to huge problems. Whenever N doubles, the running time more (but not much more) than doubles.	Quicksort Mergesort FFT
N^2	*Quadratic* algorithm practical for use only on relatively small problems. Whenever N doubles, the running time increases fourfold.	All pairs of N elements Allocate N-by-N array
N^3	*Cubic* algorithm is practical for use on only small problems. Whenever N doubles, the running time increases eightfold.	All triples of N elements N-by-N matrix multiplication
2^N	*Exponential* algorithm is not usually appropriate for practical use. Whenever N doubles, the running time squares!	Number of N-bit integers All subsets of N elements Discs moved in Towers of Hanoi

N!	*Factorial* algorithm is worse than exponential. Whenever N increases by 1, the running time increases by a factor of N	All permutations of N elements

Estimating memory usage.

To pay attention to the cost, you need to be aware of memory usage. You probably are aware of limits on memory usage on your computer (even more so than for time) because you probably have paid extra money to get more memory. Memory usage is well-defined for Java on your computer (every value will require precisely the same amount of memory each time that you run your program), but Java is implemented on a very wide range of computational devices, and memory consumption is implementation-dependent. For primitive types, it is not difficult to estimate memory usage: We can count up the number of variables and weight them by the number of bytes according to their type.

type	bytes
`boolean`	1
`byte`	1
`char`	2
`int`	4
`float`	4
`long`	8
`double`	8

type	bytes
`byte[]`	16 + N
`boolean[]`	16 + N
`char[]`	16 + 2N
`int[]`	16 + 4N
`double[]`	16 + 8N
`int[][]`	4N^2 + 20N + 16
`double[][]`	8N^2 + 20N + 16

type	bytes
object reference	4
`String`	40 + 2N
`Charge`	32
`Charge[]`	36N + 16
`Complex`	24
`Color`	12

- *Primitive types.* For example, since the Java `int` data type is the set of integer values between -2,147,483,648 and 2,147,483,647, a grand total of 2^32 different values, it is reasonable to expect implementations to use 32 bits to represent `int` values.
- *Objects.* To determine the memory consumption of an object, we add the amount of memory used by each instance variable to the overhead associated with each object, typically 8 bytes. For example, a Charge.java object uses 32 bytes (8 bytes of overhead, plus 8 bytes for each of its three `double` instance variables). A reference to an object typically uses 4 bytes of memory. When a data type contains a reference to an object, we have to account separately for the 4 bytes for the reference and the 8 bytes overhead for each object, *plus* the memory needed for the object's instance variables.
- *Arrays.* Arrays in Java are implemented as objects, typically with two instance variables (a pointer to the memory location of the first array element and the length). For primitive types, an array of N elements uses 16 bytes of header information, plus N times the number of bytes needed to store an element.
- *Two-dimensional arrays.* A two-dimensional array in Java is an array of arrays. For example, the two-dimensional array in Markov.java uses 16 bytes (overhead for the array of arrays) plus 4N bytes (references to the row arrays) plus N times 16 bytes (overhead from the row arrays) plus N times N times 8 bytes (for the N double values in each of the N rows) for a grand total of 8N^2 +

20N + 16 ~ 8N^2 bytes.

- *Strings.* A `String` uses a total of 40 + 2N bytes: object overhead (8 bytes), a reference to a character array (4 bytes), three `int` values (4 bytes each), plus a character array of size N (16 + 2N bytes). Note that when working with substrings, two strings may share the same underlying character array.

Typically the JVM allocates memory in 8 byte blocks so a string of size 1, 2, 3, or 4 would consume the same amount of memory (48 bytes). `Float` and `Double` each use 16 bytes, as would a user-defined data type just containing a single `double` instance variable.

Perspective.

Good performance is important. An impossibly slow program is almost as useless as an incorrect one. In particular, it is always wise to have some idea of which code constitutes the inner loop of your programs. Perhaps the most common mistake made in programming is to pay too much attention to performance characteristics. Your first priority is to make your code clear and correct. Modifying a program for the sole purpose of speeding it up is best left for experts. Indeed, doing so is often counterproductive, as it tends to create code that is complicated and difficult to understand. C. A. R. Hoare (a leading proponent of writing clear and correct code) once summarized this idea by saying that "premature optimization is the root of all evil," to which D. Knuth added the qualifier "(or at least most of it) in programming."

Perhaps the second most common mistake made in developing an algorithm is to ignore performance characteristics. Users of a surprising number of computer systems lose substantial time waiting for simple quadratic algorithms to finish solving a problem, even though linear or linearithmic algorithms are available that are only slightly more complicated and could therefore solve the problem in a fraction of the time. When we are dealing with huge problem sizes, we often have no choice but to seek better algorithms.

Improving a program to make it clearer, more efficient, and elegant should be your goal every time that you work on it. If you pay attention to the cost all the way through the development of a program, you will reap the benefits every time you use it.

Q + A

Q. How do I find out how long it takes to add or multiply two `double` values on my computer?

A. Run some experiments! The program TimePrimitives.java tests the execution time of various arithmetic operations on primitive types. On our system division is slower than addition and multiplication and trigonometric operations are substantially slower than arithmetic operations. This technique measures the actual elapsed time as would be observed on a wall clock. If your system is not running many other applications, this can produce accurate results. Also, the JIT compiler needs to get warmed up, so we disregard the first bunch of output.

Q. Q. How much time do functions such as `Math.sqrt()`, `Math.log()`, and `Math.sin()` take?

A. Run some experiments! Stopwatch.java makes it easy to write programs to answer questions of this sort for yourself, and you will be able to use your computer much more effectively if you get in the habit of doing so.

Q. How much time do string operations take?

A. Run some experiments! (Have you gotten the message yet?) The standard implementation is written to allow the methods `length()`, `charAt()`, and `substring()` to run in constant time. Methods such as `toLowerCase()` and `replace()` are linear ear in the string size. The methods `compareTo()`, and `startsWith()` take time proportional to the number of characters needed to resolve the answer (constant in the best case and linear in the worst case), but `indexOf()` can be slow. String concatenation takes time proportional to the total number of characters in the result.

Q. Why does allocating an array of size N take time proportional to N?

A. In Java, all array elements are automatically initialized to default values (0, false, or null). In principle, this could be a constant time operation if the system would defer initialization of each element until just before the program accesses that element for the first time, but most Java implementations go through the whole array to initialize each value.

Q. How do I find out how much memory is available for my Java programs?

A. Since Java will tell you when it runs out of memory, it is not difficult to run some experiments. For example, if you use PrimeSieve.java by typing

```
% java PrimeSieve 100000000
5761455

% java PrimeSieve 1000000000
Exception in thread "main"
java.lang.OutOfMemoryError: Java heap space
```

then you can figure that you have enough room for an array of 100 million boolean values but not for an array of 1 billion boolean values. You can increase the amount of memory allotted to Java with command-line flags.

```
% java PrimeSieve -Xmx1100m 1000000000
50847534
```

The 1100MB is the amount of memory you are requesting from the system.

Q. What does it mean when someone says that the running time of an algorithm is O(N log N)?

A. That is an example of a notation known as *big-Oh* notation. We write f(N) is O(g(N)) if there exists a constant c such that f (N) <= cg (N) for all N. We also say that the running time of an algorithm is O(g(N)) if the running time is O(g(N)) for all possible inputs. This notation is widely used by theoretical computer scientists to prove theorems about algorithms, so you are sure to see it if you take a course in algorithms and data structures. It provides a worst-case performance guarantee.

Q. So can I use the fact that the running time of an algorithm is O(N log N) or O(N^2) to predict performance?

A. No, because the actual running time might be much less. Perhaps there is some input for which the running time is proportional to the given function, but perhaps the that input is not found among those expected in practice. Mathematically, big-Oh notation is less precise than the tilde notation we use: if f (N) ~ g (N), then f(N) is O(g(N)), but not necessarily vice versa. Consequently, big-Oh notation cannot

be used to predict performance. For example, knowledge that the running time of one algorithm is O(N log N) and the running time of another algorithm is O(N^2) does not tell you which will be faster when you run implementations of them. Generally, hypotheses that use big-Oh notation are not useful in the context of the scientific method because they are not falsifiable.

Q. Is the loop `for (int i = N-1; i >= 0; i--)` more efficient than `for (int i = 0; i < N; i++)`?

A. Some programmers think so (because it simplifies the loop continuation expression), but in many cases it is actually less efficient. Don't do it unless you have a good reason for doing so.

Q. Any automated tools for profiling a program?

A. If you execute with the `-Xprof` option, you will obtain all kinds of information.

```
% java -Xprof TwoSum < input5000.txt
Flat profile of 3.18 secs (163 total ticks): main

  Interpreted + native   Method
  0.6%     0  +     1    sun.misc.URLClassPath$JarLoader.getJarFile
  0.6%     0  +     1    sun.nio.cs.StreamEncoder$CharsetSE.writeBytes
  0.6%     0  +     1    sun.misc.Resource.getBytes
  0.6%     0  +     1    java.util.jar.JarFile.initializeVerifier
  0.6%     0  +     1    sun.nio.cs.UTF_8.newDecoder
  0.6%     1  +     0    java.lang.String.toLowerCase
  3.7%     1  +     5    Total interpreted

     Compiled + native   Method
 88.3%   144  +     0    TwoSum.main
  1.2%     2  +     0    StdIn.readString
  0.6%     1  +     0    java.lang.String.charAt
  0.6%     1  +     0    java.io.BufferedInputStream.read
  0.6%     1  +     0    java.lang.StringBuffer.length
  0.6%     1  +     0    java.lang.Integer.parseInt
 92.0%   150  +     0    Total compiled
```

For our purposes, the most important piece of information is the number of seconds listed in the "flat profile." In this case, the profiler says our program took 3.18 seconds. Running it a second times may yield an answer of 3.28 or 3.16 since the measurement is not perfectly accurate.

Q. Any peformance tips?

Q. Here is a huge list of Java performance tips.

Exercises

1. Implement the method `printAll()` for ThreeSum.java, which prints all of the triples that sum to zero.
2. Modify ThreeSum.java to take a command-line argument x and find a triple of numbers on standard input whose sum is closest to x.
3. Write a program FourSum.java that takes an integer N from standard input, then reads N long values from standard input, and counts the number of 4-tuples that sum to zero. Use a quadruple loop. What is the order of growth of the running time of your program? Estimate the largest N that your program can handle in an hour. Then, run your program to validate your hypothesis.

4. Prove by induction that the number of distinct pairs of integers between 0 and N-1 is N(N-1)/2, and then prove by induction that the number of distinct triples of integers between 0 N-1 is N(N-1)(N-2)/6.
5. Show by approximating with integrals that the number of distinct triples of integers between 0 and N is about N^3/6.
6. What is the value of x after running the following code fragment?

```
int x = 0;
for (int i = 0; i < N; i++)
   for (int j = i + 1; j < N; j++)
      for (int k = j + 1; k < N; k++)
         x++;
```

   Answer: N choose 3 = N (N-1) (N-2) / 6.
7. Use tilde notation to simplify each of the following formulas, and give the order of growth of each:
   1. N (N-1) (N-2) (N-3) /24
   2. (N-2) (lg N - 2) (lg N + 2)
   3. N (N+1) - N^2
   4. N (N+1) / 2 + N lg N
   5. ln ((N-1) (N-2) (N-3))^2
8. Determine the order of growth of the running time of the input loop of `ThreeSum`:

```
int N = Integer.parseInt(args[0]);
int[] a = new int[N];
for (int i = 0; i < N; i++) {
   a[i] = StdIn.readInt();
}
String s = sb.toString();
```

   Answer: Linear. The bottlenecks are the array initialization and the input loop. Depending on your system and the implementation, the `readInt()` statement might lead to inconsistent timings for small values of N. The cost of an input loop like this might dominate in a linearithmic or even a quadratic program with N that is not too large.
9. Determine whether the following code fragment is linear, quadratic, or cubic (as a function of N).

```
for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; j++) {
      if (i == j) c[i][j] = 1.0;
      else        c[i][j] = 0.0;
   }
}
String s = sb.toString();
```

10. Suppose the running time of an algorithm on inputs of size one thousand, two thousand, three thousand, and four thousand is 5 seconds, 20 seconds, 45 seconds, and 80 seconds, respectively. Estimate how long it will take to solve a problem of size 5,000. Is the algorithm linear, linearithmic, quadratic, cubic, or exponential?
11. Which would you prefer: a quadratic, linearithmic, or linear algorithm?

Answer: While it is tempting to make a quick decision based on the order of growth, it is very easy to be misled by doing so. You need to have some idea of the problem size and of the relative value of the leading coefficients of the running time. For example, suppose that the running times are N 2 seconds, 100 N lg N seconds, and 10000 N seconds. The quadratic algorithm will be fastest for N up to about 1000, and the linear algorithm will never be faster than the linearithmic one (N would have to be greater than 2100, far too large to bother considering).

12. Apply the scientific method to develop and validate a hypothesis about order of growth of the running time of the following code fragment, as a function of the input argument n.

```
public static int f(int n) {
    if (n == 0) return 1;
    return f(n-1) + f(n-1);
}
```

13. Apply the scientific method to develop and validate a hypothesis about order of growth of the running time of the Coupon.collect() as a function of the input argument N. Note: Doubling is not effective for distinguishing between the linear and linearithmic hypotheses - you might try squaring the size of the input.
14. Apply the scientific method to develop and validate a hypothesis about order of growth of the running time of Markov.java as a function of the input parameters T and N.
15. Apply the scientific method to develop and validate a hypothesis about order of growth of the running time of each of the following two code fragments as a function of N.

```
String s = "";
for (int i = 0; i < N; i++) {
   if (StdRandom.bernoulli(0.5)) s += "0";
   else                          s += "1";
}

StringBuilder sb = new StringBuilder();
for (int i = 0; i < N; i++) {
   if (StdRandom.bernoulli(0.5)) sb.append("0");
   else                          sb.append("1");
}
String s = sb.toString();
```

Answer: The first is quadratic; the second is linear.

16. Each of the four Java functions below returns a string of length N whose characters are all x. Determine the order of growth of the running time of each function. Recall that concatenating two strings in Java takes time proportional to the sum of their lengths.

```
public static String method1(int N) {
   if (N == 0) return "";
   String temp = method1(N / 2);
   if (N % 2 == 0) return temp + temp;
   else            return temp + temp + "x";
}

public static String method2(int N) {
```

```
   String s = "";
   for (int i = 0; i < N; i++)
      s = s + "x";
   return s;
}

public static String method3(int N) {
   if (N == 0) return "";
   if (N == 1) return "x";
   return method3(N/2) + method3(N - N/2);
}

public static String method4(int N) {
   char[] temp = new char[N];
   for (int i = 0; i < N; i++)
       temp[i] = 'x';
   return new String(temp);
}
```

Program Repeat.java contains the four functions.

17. The following code fragment (adapted from a Java programming book) creates a random permutation of the integers from 0 to N-1. Determine the order of growth of its running time as a function of N. Compare its order of growth with Shuffle.java from Section 1.4.

```
int[] a = new int[N];
boolean[] taken = new boolean[N];
int count = 0;
while (count < N)
{
   int r = StdRandom.uniform(N);
   if (!taken[r])
   {
       a[r] = count;
       taken[r] = true;
       count++;
   }
}
```

18. What is order of growth of the running time of the following function, which reverses a string s of length N?

```
public static String reverse(String s) {
   int N = s.length();
   String reverse = "";
   for (int i = 0; i < N; i++)
      reverse = s.charAt(i) + reverse;
   return reverse;
}
```

19. What is order of growth of the running time of the following function, which reverses a string s of length N?

```
public static String reverse(String s) {
    int N = s.length();
    if (N <= 1) return s;
    String left = s.substring(0, N/2);
    String right = s.substring(N/2, N);
    return reverse(right) + reverse(left);
}
```

20. Give a linear algorithm for reversing a string.

    Answer:

```
public static String reverse(String s) {
    int N = s.length();
    char[] a = new char[N];
    for (int i = 0; i < N; i++)
       a[i] = s.charAt(N-i-1);
    String reverse = new String(a);
    return reverse;
}
```

21. Write a program `MooresLaw` that takes a command-line argument N and outputs the increase in processor speed over a decade if microprocessors double every N months. How much will processor speed increase over the next decade if speeds double every N = 15 months? 24 months?
22. Using the model in the text, give the memory requirements for each object of the following data types from Chapter 3:
    1. `Stopwatch`
    2. `Turtle`
    3. `Vector`
    4. `Body`
    5. `Universe`
23. Estimate, as a function of the grid size N, the amount of space used by Visualize.java with the vertical percolation detection (Program 2.4.2).

    Extra credit: Answer the same question for the case where the recursive percolation detection method in Program 2.4.5 is used.

24. Estimate the size of the biggest two-dimensional array of int values that your computer can hold, and then try to allocate such an array.
25. Estimate, as a function of the number of documents N and the dimension d, the amount of space used by CompareAll.java.
26. Write a version of PrimeSieve.java that uses a `byte` array instead of a `boolean` array and uses all the bits in each `byte`, to raise the largest value of `N` that it can handle by a factor of 8.
27. The following table gives running times for various programs for various values of N. Fill in the blanks with estimates that you think are reasonable on the basis of the information given.

```
program     1,000      10,000        100,000     1,000,000
   A    .001 seconds .012 seconds   .16 seconds   ? seconds
   B       1 minute    10 minutes   1.7 hours     ? hours
   C       1 second   1.7 minutes   2.8 hours     ? days
```

Give hypotheses for the order of growth of the running time of each program.

Creative Exercises

1. **Closest pair.** Design a quadratic algorithm that finds the pair of integers that are closest to each other. (In the next section you will be asked to find a linearithmic algorithm.)
2. **Sum furthest from zero.** Design an algorithm that finds the pair of integers whose sum is furthest from zero. Can you discover an algorithm that linear running time?
3. **The "beck" exploit.** In the Apache 1.2 web server, there is a function called `no2slash` whose purpose is to collapse multiple `'/'`s. For example `/d1///d2////d3/test.html` becomes `/d1/d2/d3/test.html`. The original algorithm was to repeatedly search for a `'/'` and copy the remainder of the string over.

```
void no2slash(char *name) {
   int x, y;
   for(x = 0; name[x]; )
      if(x && (name[x-1] == '/') && (name[x] == '/'))
         for(y = x+1; name[y-1]; y++)
            name[y-1] = name[y];
      else x++;
}
```

Unfortunately, it's running time is quadratic in the number of `'/'`s in the input. By sending multiple simultaneous requests with large numbers of `'/'`s, you can deluge a server and starve other processes for CPU time, thereby creating a denial of service attack. Fix the version of `no2slash` so that it runs in linear time and does not allow for the above attack.

```
offset = 0
for i = 2 to n do
   if a[i-1] = '/' and a[i] = '/' then
      offset = offset + 1
   else
      a[i-offset] = a[i]
end for
```

4. **Young tableaux.** Suppose you have in memory an N-by-N grid of integers `a` such that `a[i][j] < a[i+1][j]` and `a[i][j] < a[i][j+1]` for all `i` and `j` like the table below.

```
 5  23  54  67  89
 6  69  73  74  90
10  71  83  84  91
60  73  84  86  92
99  91  92  93  94
```

Devise an O(N) time algorithm to determine whether or not a given integer x is in a Young tableaux.

Answer: Start at the upper right corner. If element = x, done. If element > x, go left. Otherwise go down. If you reach bottom left corner, then it's not in table. O(N) since can go left at most N times

and down at most N times.

5. **3-D searching.** Repeat the previous question, but assume the grid is N-by-N-by-N and `a[i][j][k] < a[i+1][j][k]`, `a[i][j][k] < a[i][j+1][k]`, and `a[i][j][k] < a[i][j][k+1]` for all `i`, `j`,and `k`. Devise an algorithm that can determine whether or not a given integer x is in the 3-d table in time better than N^2 log N. *Hint*: treat the problem as N independent Young table queries.
6. **Moore's Law.** This problem investigates the ramifications of exponential growth. Moore's Law (named after Intel co-founder Gordon Moore) states that microprocessor power doubles every 18 months. See this article which argues against this conventional wisdom. Write a program MooresLaw.java that takes an integer parameter N and outputs the increase in processor speed over a decade if microprocessors double every N months.
    1. How much will processor speed increase over the next decade in speeds double every N = 18 months?
    2. The true value may be closer to speeds doubling every two years. Repeat (a) with N = 24.

    Subset sum. Write a program Exponential.java that takes a command line integer N, reads in N long integer from standard input, and finds the *subset* whose sum is closest to 0. Give the order of growth of your algorithm.

7. **String reversal.** Given an array of N elements, give a linear time algorithm to reverse its elements. Use at most a constant amount of extra space (array indices and array values).
8. **String rotation.** Given an array of N elements, give a linear time algorithm to rotate the string k positions. That is, if the array contains a_1, a_2, ..., a_N, the rotated array is a_k, a_{k+1}, ..., a_N, a_1, ..., a_{k-1}. Use at most a constant amount of extra space (array indices and array values). This operation is a primitive in some programming languages like APL. Also, arises in the implementation of a text editor (Kernighan and Plauger). *Hint*: reverse three sub-arrays as in the previous exercise.
9. **Finding a duplicated integer.** Given an array of n integers from 1 to n with one integer repeated twice and one missing. Find the missing integer using O(n) time and O(1) extra space. No overflow allowed.
10. **Finding a duplicated integer.** Given a read-only array of n integers, where each integer from 1 to n-1 occurs once and one occurs twice, design an O(n) time algorithm to find the duplicated integer. Use only O(1) space.
11. **Finding a duplicated integer.** Given a read-only array of n integers between 1 and n-1, design an O(n) time algorithm to find a duplicated integer. Use only O(1) space. *Hint*: equivalent to finding a loop in a singly linked structure.
12. **Finding the missing integer.** (Bentley's Programming Pearls.) Given a list of 1 million 20-bit integers, given an efficient algorithm for finding a 20-bit integer that is not on the list
    1. given as much random access memory as you like
    2. given as many tapes as you like (but no random access) and a few dozen words of memory
    3. (Blum) given only a few dozen words of memory

    Solutions. (a) allocate a boolean array of 2^{20} bits. (b) copy all numbers starting with 0 to one tape and 1 to the other. Choose leading bit of missing number to be whichever tape has fewer entries (breaking ties arbitrarily). Recur on the smaller half. (c) Pick twenty 20-bit integers at random and search list sequentially to see if they're missing. There are 48,576 missing integers, so you'll have at least a $(1 - (1 - 48576/2^{20})^{20}) > 0.5$ chance of getting one. If you get unlucky, repeat.

13. **Pattern matching.** Given an N-by-N array of black (1) and white (0) pixels, find the largest contiguous subarray that consists of entirely black pixels. In the example below there is a 6-by-2 subarray.

```
1 0 1 1 1 0 0 0
0 0 0 1 0 1 0 0
0 0 1 1 1 0 0 0
0 0 1 1 1 0 1 0
0 0 1 1 1 1 1 1
0 1 0 1 1 1 1 0
0 1 0 1 1 1 1 0
0 0 0 1 1 1 1 0
```

14. **Factorial.** Compute 1000000! as fast as you can. You may use the `BigInteger` class.
15. **Longest increasing subsequence.** Given a sequence of N 64-bit integers, find the longest strictly increasing subsequence. For example, if the input is 56, 23, 33, 22, 34, 78, 11, 35, 44, then the longest increasing subsequence is 23, 33, 34, 35, 44. Your algorithm should run in N log N time.
16. **Maximum sum.** Given a sequence of N 64-bit integers, find a sequence of at most U consecutive integers that has the highest sum.
17. **Maximum average.** Given a sequence of N 64-bit integers, find a sequence of at least L consecutive integers that has the highest average. O(N^2) not too hard; O(N) possible but much harder. O(NL) follows since there must exist an optimal interval between L and 2L. Hint: can compute average from i to j in O(1) time with O(N) preprocessing by precomputing prefix[i] = a[0] + ... + a[i] for each i. Then the average of the interval from i to j is (prefix[j] - prefix[i]) / (j - i + 1).
18. **Knuth's parking problem.** N cars arrive at an initially empty lot with N parking space, arranged in a one-way circle. Car i uniformly chooses a spot at random, independent of previous cars. If spot j is taken, try j+1, j+2, and so on. Write a program to estimate how many tries are made (total displacement) as a function of N.

    Answer. N^3/2.

19. **Amdahl's law.** Limits how much you can speed up a computation by improving one of its two constituent parts.
20. **Sieve of Eratosthenes.** Estimate running time of sieve of Eratoshthenes for finding all primes less than or equal to N as a function of N.

    Answer: in theory, the answer is proportional to N log log N. Follows from Mertens' theorem in number theory.

Web Exercises

1. Suppose the running time of an algorithm on inputs of size 1,000, 2,000, 3,000, and 4,000 is 5 seconds, 20 seconds, 45 seconds, 80 seconds, and 125 seconds, respectively. Estimate how long it will take to solve a problem of size 5,000. Is the algorithm have linear, linearithmic, quadratic, cubic, or exponential?
2. Write a program OneSum.java that takes a command-line argument N, reads in N integers from standard input, and finds the value that is closest to 0. How many instructions are executed in the data processing loop?
3. Write a program TwoSum.java that takes a command-line argument N, reads in N integers from standard input, and finds the pair of values whose sum is closest to 0. How many instructions are executed in the data processing loop?
4. Analyze the following code fragment mathematically and determine whether the running time is linear, quadratic, or cubic as a function of N.

```
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         c[i][j] += a[i][k] * b[k][j];
```

5. The following function returns a random string of length N. How long does it take?

```
public static String random(int N) {
    if (N == 0) return "";
    int r = (int) (26 * Math.random());  // between 0 and 25
    char c = 'a' + r;                    // between 'a' and 'z'
    return random(N/2) + c + random(N - N/2 - 1);
}
```

Last modified on August 05, 2011.