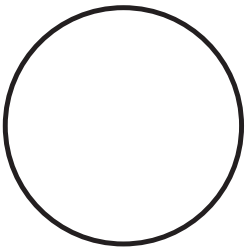
11

COMPUTER SCIENCE ALGORITHMS



While everything we've looked at so far can be called a "randomized algorithm," in computer science, the phrase refers to two broad categories of algorithms—the subject of this chapter.

A *randomized algorithm* employs randomness as part of its operation. The algorithm succeeds in accomplishing its goal, either by producing the correct answer quickly, but sometimes not, or by running rapidly with some probability of returning a false or non-optimal result.

We'll begin by defining the two broad categories of randomized algorithms with examples. Next, we'll learn about estimating the number of animals in a population. Following that, we'll learn how to demonstrate that a number is a prime to any desired level of certainty while avoiding the brute force approach of searching for all possible divisors. We'll end with randomized Quicksort, the textbook example of a randomized algorithm.

Las Vegas and Monte Carlo

Las Vegas and Monte Carlo are locations famously associated with gambling, that is, with games of chance dependent on randomness and probability. However, when computer scientists refer to Las Vegas or Monte Carlo, they are (usually) referring to the two main types of randomized algorithms.

A *Las Vegas algorithm* always returns a correct result for its input in a random amount of time. In other words, how long the algorithm takes to execute isn't deterministic, but the output is correct.

On the other hand, a *Monte Carlo algorithm* offers no assurance that its output is correct, but the run time is deterministic. There is a nonzero probability that the output isn't correct, but for a practical Monte Carlo algorithm, this probability is small. Most algorithms we've encountered, including swarm intelligence and evolutionary algorithms, are Monte Carlo algorithms. Las Vegas algorithms can be transformed into Monte Carlo algorithms by allowing them to exit before locating the correct output.

The first example we'll investigate is a sorting algorithm that is, at our discretion, a Las Vegas or Monte Carlo algorithm. The second is a Monte Carlo algorithm for verifying matrix multiplication.

Permutation Sort

A *permutation* is a possible arrangement of a set of items. If there are n items in the set, there are $n! = n(n-1)(n-2)\ldots 1$ possible permutations. For example, if the set consists of three things, say $A = \{1, 2, 3\}$, then there are six possible permutations:

$$\{1,2,3\},\ \{1,3,2\},\ \{2,1,3\},\ \{2,3,1\},\ \{3,1,2\},\ \{3,2,1\}$$

Notice that one permutation sorts the items from smallest to largest. Therefore, if given a vector of unsorted numbers, we might use a sort algorithm that generates permutations until finding the one that sorts the items. While we can implement this deterministically, we can also use random permutations with the hope that we might stumble across the correct ordering before testing too many candidate permutations. The *permutation sort* algorithm (also known as *bogosort* or *stupid sort*) implements this idea.

Run the file *permutation_sort.py* with no arguments:

```
permutation_sort <items> <limit> [<kind> | <kind> <seed>]

  <items> - number of items in the list
  <limit> - number of passes maximum (0=Las Vegas else Monte Carlo)
  <kind>  - randomness source
  <seed>  - seed value
```

The code generates a random vector of integers in $[0, 99]$ and sorts it by trying up to `limit` random permutations. To score each permutation, the code returns the fraction of pairs of elements that are out of order, where $a > b$ for a at index i and b at index $i + 1$. If the array is sorted, the score is zero.

If `limit` is zero, the algorithm runs until it finds the correct permutation, which depends on the number of possible permutations. As the number of permutations increases ($n!$), the run time rapidly increases if we insist on trying until we succeed. In other words, a `limit` of 0 turns *permutation_sort.py* into a Las Vegas algorithm.

For example, to run *permutation_sort.py* as a Las Vegas algorithm, use:

```
> python3 permutation_sort.py 6 0 minstd 42
sorted: 0  25  44  57  65  96  (268 iterations)
```

The code found the proper order of elements after testing 268 of the possible 6! = 720 permutations. Changing the randomness source from `minstd` to `pcg64` sorts in 59 iterations while `mt19937` uses 20. We set the limit to 0 to make the code run until success, but the number of permutations tested was often far less than the maximum.

If we switch to Monte Carlo:

```
> python3 permutation_sort.py 6 100 minstd 42
partially: 0  57  25  44  65  96  (score = 0.16667, 100 iterations)
```

we get a partially sorted array with a score > 0. Because of the fixed randomness source and seed, we know we need 268 iterations to sort the array:

```
> python3 permutation_sort.py 6 268 minstd 42
sorted: 0  25  44  57  65  96  (268 iterations)
```

Listing 11-1 shows the main loop in *permutation_sort.py*.

```
v = np.array([int(rng.random()*100) for i in range(N)], dtype="uint8")
k = 0
score = Score(v)
while (score != 0) and (k < limit):
    k += 1
    i = np.argsort(rng.random(len(v)))
    s = Score(v[i])
    if (s < score):
        score = s
        v = v[i]
```

Listing 11-1: The main loop for permutation sort main

We create the vector (`v`), along with an initial `score`. The main loop, `while`, runs until the score is zero or the `limit` is exceeded. If Las Vegas, we set `limit` to a huge number to play the odds that we'll find the true ordering long before that many trials.

The body of the `while` loop creates a random ordering of `v` and calculates the score. Whenever it finds a lower score, the code reorders `v` to return at least a partially ordered vector should the limit be reached; however, this is not strictly necessary.

The remainder of the file displays the results or calculates the score (Listing 11-2).

```
def Score(arg):
    n = 0
    for i in range(len(arg)-1):
        if (arg[i] > arg[i+1]):
```

```
            n += 1
    return n / len(arg)
```

Listing 11-2: Scoring a permutation

Let's plot the mean number of permutations tested as a function of the number of items to sort, *permutation_sort_plot.py*, which plots the mean and standard error for ten calls to *permutation_sort.py* for n in [2, 10]. The result is Figure 11-1.

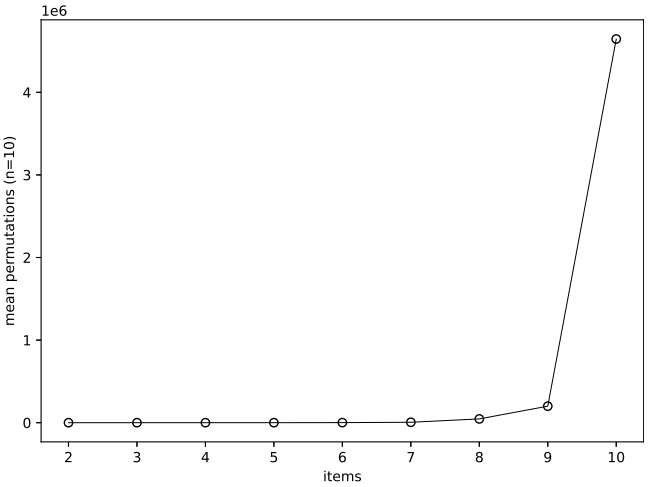


Figure 11-1: The mean number of permutations tested as a function of the number of items (the y-axis is in millions)

Figure 11-1 illustrates *combinatorial explosion*, which is the rapid growth in a problem's runtime or resource use as a function of the size of its input. Permutation sort works decently when sorting lists of up to nine items; any more and the complexity explodes.

We also see this effect in *permutation_sort_plot.py*'s output:

```
 2:  0.127855 +/- 0.002026
 3:  0.128128 +/- 0.001737
 4:  0.129859 +/- 0.002469
 5:  0.131369 +/- 0.002483
 6:  0.136637 +/- 0.003704
 7:  0.172775 +/- 0.008236
 8:  0.534369 +/- 0.081601
 9:  1.987567 +/- 0.488691
10: 44.133984 +/- 10.929158
```

The output shows, as a function of n, the mean ($\pm$ SE) time in seconds to sort a vector of that size. After seven elements, runtimes quickly increase.

Combinatorial explosion is the bane of many otherwise useful algorithms, often reaching a point where many lifetimes of the universe are insufficient to find the correct output.

Permutation sort is tied closely to the factorial, which is why we're getting these results:

2! = 2	5! = 120	8! = 40,320
3! = 6	6! = 720	9! = 362,880
4! = 24	7! = 5,040	10! = 3,628,800

The factorial grows at a tremendous rate. If we want to sort 20 items, we have:

$$20! = 2{,}432{,}902{,}008{,}176{,}640{,}000$$

permutations to check. At 1 millisecond per permutation, we'd need over 77 million years of computing time to check them all. This doesn't mean permutation sort couldn't, by pure chance, sort 20 items in less than a second, but the probability is exceedingly low. This is the paradox of randomized algorithms.

Matrix Multiplication

I have three matrices, $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$. We'll use bold, uppercase letters to represent matrices. Does $\boldsymbol{AB} = \boldsymbol{C}$? How can we know?

Introducing Matrix Multiplication

First, let's ensure we're on the same page regarding matrix multiplication. A *matrix* is a 2D array of numbers, so the matrices here might be:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix}, \quad \boldsymbol{C} = \begin{bmatrix} 5 & 6 \\ 11 & 12 \end{bmatrix}$$

These are 2×2 matrices with two rows and two columns. If the number of rows equals the number of columns, we're working with *square matrices*. However, the number of rows and columns need not match; for example, we might have a matrix of 3×5 or $1{,}000 \times 13$. The latter case is typical in machine learning, where rows represent observations and columns represent features associated with those observations. An $n \times 1$ matrix is a *column vector*, while a $1 \times n$ matrix is a *row vector*.

Asking whether $\boldsymbol{AB} = \boldsymbol{C}$ implies that we know how to find $\boldsymbol{AB}$. In NumPy, to multiply two 2D arrays, we multiply each corresponding element (Listing 11-3).

```
>>> A = np.array([[1,2],[3,4]])
>>> B = np.array([[1,0],[2,3]])
>>> A*B
array([[ 1,  0],
       [ 6, 12]])
```

Listing 11-3: Multiplying element-wise in NumPy

Unfortunately, multiplying matrices is more involved. We begin by checking that the number of columns of the first matrix equals the number of rows of

the second. If not, then we can't multiply the matrices. Therefore, to multiply an $n \times m$ matrix by a $u \times v$ matrix requires $m = u$. If this is true, we can multiply to produce an $n \times v$ result. The square matrices in this section automatically satisfy this requirement.

The matrix multiplication process requires multiplying each column of the second matrix by the rows of the first matrix, where the elements of the column multiply the corresponding elements of the row. We then sum these products to produce a single output value. For example, in symbols, multiplying two 2×2 matrices returns a new 2×2 matrix:

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{bmatrix}$$

We're indexing matrices from 0, as we would NumPy arrays. However, many math books index from 1 so that the first element of the first row of matrix $\boldsymbol{A}$ is denoted a_{11}, not a_{00}.

Mathematically, if $\boldsymbol{A}$ is an $n \times m$ matrix and $\boldsymbol{B}$ is an $m \times p$ matrix, then the elements of $\boldsymbol{C} = \boldsymbol{AB}$, an $n \times p$ matrix, are:

$$c_{ij} = \sum_{k=0}^{m-1} a_{ik}b_{kj}, \quad i = 0, 1, 2, \ldots, n-1; \quad j = 0, 1, 2, \ldots, p-1 \tag{11.1}$$

Remember that matrix multiplication does not commute; in general, $\boldsymbol{AB} \neq \boldsymbol{BA}$. The sum over k in Equation 11.1 illustrates why: the single index accesses by row for $\boldsymbol{A}$ and by column for $\boldsymbol{B}$ so that swapping the order of $\boldsymbol{A}$ and $\boldsymbol{B}$ means different elements of the matrices are mixed.

The sum in Equation 11.1 uses index variable k with two more implied sums over all values of i and j to fill in the output matrix, $\boldsymbol{C}$. These observations point toward an implementation: matrix multiplication becomes a triple loop indexing elements of 2D arrays.

Listing 11-4 translates the loops of Equation 11.1 into code.

```
def mmult(A,B):
    n,m = A.shape
    p = B.shape[1]
    C = np.zeros((n,p), dtype=A.dtype)
    for i in range(n):
        for j in range(p):
            for k in range(m):
                C[i,j] += A[i,k]*B[k,j]
    return C
```

Listing 11-4: Naive matrix multiplication

We'll use this implementation even though NumPy supports matrix multiplication natively in several ways, for example, via the `@` operator. To understand why, we'll learn how computer scientists measure algorithm performance.

Introducing Big O Notation

Computer scientists characterize the resource use of an algorithm by comparing the algorithm's performance as input size increases to a similar function that captures how the algorithm's resource use changes as the input grows. Here, resource refers to either memory or time. For example, an $\mathcal{O}(n)$ algorithm linearly increases the memory used as the size of the input, n, increases. A linear function can be written as $y = mx + b$ for input x, but in big O notation, we ignore multiplicative and constant factors so that $y = x$ is functionally the same as x gets very large.

The matrix multiplication code in Listing 11-4 contains a triply nested loop. If the input matrices are square ($n \times n$), then `I` = `J` = `K` = n. Each loop runs n times, making the innermost loop run n times for every increment of the next outer loop, which must run n times to increment the outermost loop. Therefore, the total number of operations required to multiply two $n \times n$ matrices scales as n^3. The time needed to create the output matrix, `C`, and evaluate the first two lines of the function doesn't alter the essential character of the function—namely, that it takes n^3 passes through the three loops.

A computer scientist would therefore write that Listing 11-4 is an $\mathcal{O}(n^3)$ algorithm and an "n cubed" implementation. In general, we want algorithms that scale as $\mathcal{O}(n)$ or better. As n increases, the work required by the algorithm scales linearly or, better still, sublinearly like $\mathcal{O}(\log n)$ or $\mathcal{O}(n \log n)$. In other words, a plot of the work as a function of n is a straight line. Ideally, we want $\mathcal{O}(1)$ algorithms that run in constant time regardless of the size of their input, but that isn't always possible. An algorithm that runs in $\mathcal{O}(n^2)$ is often tolerable, but $\mathcal{O}(n^3)$ is suitable only for small values of n.

Note that $\mathcal{O}(n)$, $\mathcal{O}(n^2)$, and $\mathcal{O}(n^3)$ are all powers of n. Such algorithms are known as *polynomial time* algorithms. We never want algorithms that run in *superpolynomial time*, with a run time (or resource use) such that no polynomial tracks it. For example, an algorithm running in $\mathcal{O}(2^n)$ time is an *exponential time* algorithm, and its resource use grows dramatically with the size of the input at a rate no polynomial can match. Worse still is the permutation sort we experimented with previously; it's an $\mathcal{O}(n!)$ algorithm that runs in *factorial time*. To see how factorial time is worse than exponential time, make a plot comparing 2^n and $n!$ for $n = [1, 8]$.

Matrix multiplication as in Listing 11-4 is $\mathcal{O}(n^3)$. Our goal is to quickly check whether $\boldsymbol{AB} = \boldsymbol{C}$ when given three matrices. We first multiply $\boldsymbol{A}$ and $\boldsymbol{B}$ and then check whether each element of the result matches the corresponding element in $\boldsymbol{C}$. The multiplication is $\mathcal{O}(n^3)$ and the check runs in $\mathcal{O}(n^2)$ time because we have to examine each element. As the cube grows much faster than the square, the overall naive algorithm runs in essentially $\mathcal{O}(n^3)$ time. Let's see if we can do better.

Introducing Freivalds' Algorithm

In 1977, Latvian computer scientist Rūsiņš Freivalds invented a randomized algorithm that correctly answers the question of whether $\boldsymbol{AB} = \boldsymbol{C}$ with high probability, yet runs in $\mathcal{O}(n^2)$ time.

For the following, we'll assume that $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are $n \times n$ matrices. The algorithm works for non-square matrices, but this restriction makes things easier to follow.

The algorithm itself is straightforward:

1. Pick a random n-element vector, $\boldsymbol{r} = \{0, 1\}^n$, that is, a random vector of 0s and 1s.
2. Calculate $\boldsymbol{D} = \boldsymbol{A}(\boldsymbol{Br}) - \boldsymbol{Cr}$. (Note: the parentheses matter.)
3. If all elements of $\boldsymbol{D}$ are zero, claim "yes," $\boldsymbol{AB} = \boldsymbol{C}$; otherwise, claim "no."

At first glance, Freivalds' algorithm doesn't look like it will help. However, recall how matrix multiplication works. The expression $\boldsymbol{Br}$ is multiplying an $n \times n$ matrix by a $n \times 1$ vector, which returns an $n \times 1$ vector. The next multiplication by $\boldsymbol{A}$ returns another $n \times 1$ vector. Likewise, $\boldsymbol{Cr}$ is also an $n \times 1$ vector. At no point is a full $n \times n$ matrix multiplication happening. As n grows, the savings in the number of calculations grows all the faster. Freivalds' algorithm runs in $\mathcal{O}(n^2)$ time—a considerable improvement over the $\mathcal{O}(n^3)$ runtime of the naive algorithm.

Multiplying $\boldsymbol{B}$ by $\boldsymbol{r}$ is equivalent to selecting a random subset of $\boldsymbol{B}$'s columns and adding the value of those columns across the rows. For example:

$$\begin{bmatrix} 1 & 3 & 4 & 0 \\ 2 & 4 & 3 & 1 \\ 0 & 0 & 1 & 2 \\ 2 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1+4 \\ 2+3 \\ 0+1 \\ 2+1 \end{bmatrix} = \begin{bmatrix} 5 \\ 5 \\ 1 \\ 3 \end{bmatrix}$$

The algorithm is betting that examining random elements of the three matrices will, if they are equal, result in $\boldsymbol{D}$ being a vector of all zeros more often than $\boldsymbol{D}$ being all zeros by chance. An analysis of the probabilities involved, which we won't cover, demonstrates that the probability of $\boldsymbol{D}$ being all zeros given $\boldsymbol{AB} \neq \boldsymbol{C}$ is less than or equal to $1/2$.

If the probability of one calculation involving a randomly selected $\boldsymbol{r}$ returning the wrong answer is at most $1/2$, then two random vectors (if we run the algorithm twice) have a probability of returning the wrong answer of at most $(1/2)(1/2) = 1/4$. Here the wrong answer is an output of "yes" when in fact $\boldsymbol{AB} \neq \boldsymbol{C}$.

Each application of the algorithm is independent of any previous application. For independent events, like the flip of a fair coin, probabilities multiply, so k runs of Freivalds' algorithm implies that the probability of a false "yes" result is $1/2^k$ or less. In other words, we can be as confident of the result as we like by running the algorithm multiple times.

The algorithm always returns "yes" when $\boldsymbol{AB} = \boldsymbol{C}$, meaning it is *one-sided*—an error in the output happens only if $\boldsymbol{AB} \neq \boldsymbol{C}$. In a *two-sided* error, the algorithm could be wrong in either case, with some probability.

Testing Freivalds' Algorithm

Let's give the algorithm a try using *freivalds.py*, which generates 1,000 random triplets of $n \times n$ matrices, with n given on the command line. In all cases, $\boldsymbol{AB} \neq \boldsymbol{C}$, so we report failures as a fraction of 1,000.

Run *freivalds.py* like so:

```
freivalds <N> <mode> <reps> [<kind> | <kind> <seed>]

  <N>     - matrix size (always square)
  <mode>  - 0=Freivalds', 1=naive
  <reps>  - reps of Freivalds' (ignored for others)
  <kind>  - randomness source
  <seed>  - seed
```

The first argument is the dimensionality of the matrices. The second decides whether to use the naive algorithm that calculates $\boldsymbol{AB} - \boldsymbol{C}$ or Freivalds'. The third is the number of times to repeat the test with random $\boldsymbol{r}$ vectors. We'll use this option shortly to track the error rate. As usual, the other arguments enable any randomness source and a seed to repeat the same sequence of random matrices.

For example:

```
> python3 freivalds.py 3 0 1 mt19937 19937
0.08598161 0.132
```

tells us that testing 1,000 3×3 matrices using Freivalds' algorithm once each took some 0.09 seconds and failed 13.2 percent of the 1,000 tests.

To use the naive algorithm, change the 0 to 1 on the command line:

```
> python3 freivalds.py 3 1 1 mt19937 19937
0.05456829 0.000
```

As expected, there are no failures because the complete calculation always catches when $\boldsymbol{AB} \neq \boldsymbol{C}$. While the naive algorithm seems to run faster than Freivalds', this is an illusion; as n increases, the two diverge.

Failing 13 percent of the time when checking 3×3 matrices isn't too inspiring. Let's repeat the test, but check twice instead of once:

```
> python3 freivalds.py 3 0 2 mt19937 19937
0.14664984 0.016
```

Now we fail only 1.6 percent of the tests at the expense of nearly doubling the running time. Let's try four tests instead of two:

```
> python3 freivalds.py 3 0 4 mt19937 19937
0.26030445 0.000
```

With four tests, Freivalds' algorithm is 1,000 out of 1,000.

Freivalds' algorithm is probabilistic. The likelihood of error decreases quickly as matrix size increases. To see this effect, alter the matrix size while

fixing the repetitions at 1. By the time $n = 11$, the error is generally below 0.1 percent.

It makes sense that the error rate goes down with matrix size. The probability that a random selection of values sum by accident to two equal values (***A***(***Br***) and ***Cr***) should decrease as the number of values summed increases.

Let's explore running time as a function of n. Run *freivalds_plots.py* to produce the graphs in Figure 11-2.

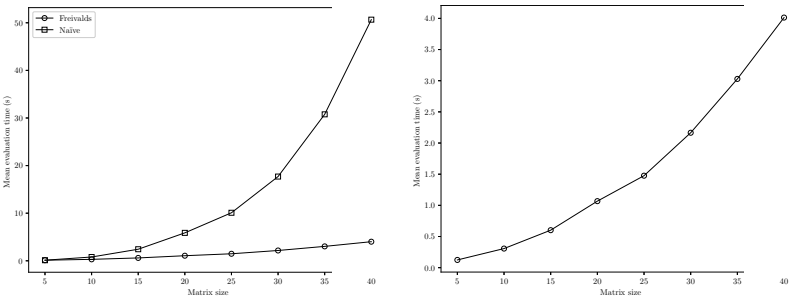


Figure 11-2: Comparing Freivalds' running time to the naive algorithm as a function of matrix size (left), and plotting Freivalds' running time alone to show the $\mathcal{O}(n^2)$ complexity—note the y-axis range (right)

On the left of Figure 11-2, we see the growth in running time between Freivalds' and the naive algorithm as the size of the matrices increases. The naive algorithm, $\mathcal{O}(n^3)$, grows substantially faster than Freivalds' $\mathcal{O}(n^2)$, shown by itself on the right.

The combination of performance gain and decreasing likelihood of error as n increases makes Freivalds' algorithm particularly nice. Yes, it's probabilistic, but in the places where it's most desirable (large n), it's also most likely to be correct.

Before examining *freivalds.py*, I have a confession. We can do matrix multiplication better than $\mathcal{O}(n^3)$, especially for matrices with $n > 100$. Volker Strassen's 1969 matrix multiplication algorithm has a runtime of about $\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.807})$, which is slightly better than the naive algorithm. NumPy, based on the BLAS library, makes use of Strassen's algorithm, which is why we didn't use NumPy in this section. However, $\mathcal{O}(n^2)$ is better than $\mathcal{O}(n^{2.807})$, so Freivalds' algorithm is still useful, even with Strassen matrix multiplication.

GALACTIC ALGORITHMS

There are matrix multiplication algorithms with even better asymptotic behavior than Strassen's algorithm. The current best have complexity $\mathcal{O}(n^{2.373})$ or so. However, these algorithms are, in practice, completely useless. The seeming contradiction has to do with Big O notation, which shows the overall behavior but ignores multiplicative factors and constants. This means that an algorithm

running in $10n^3$ time is the same as one running in $10000n^3 + 10000$ time. Both scale as $\mathcal{O}(n^3)$, but in practice, the first is more likely to be helpful.

Matrix multiplication algorithms that beat Strassen's algorithm in overall complexity, like the *Coppersmith-Winograd* algorithm, have constants so large that the algorithm becomes practical only once n is some number far larger than anything computers can currently handle, if ever.

Such algorithms have been christened *galactic algorithms* by Kenneth W. Regan. We cannot effectively use galactic algorithms in practice even if they are "the best" in terms of asymptotic behavior. While these algorithms are of theoretical importance, they won't show up in our toolkits any time soon.

Exploring the Code

Listing 11-5 contains the code implementing Freivalds' algorithm. The `mmult` function is in Listing 11-4. The `array_equal` function asks whether the absolute maximum of the difference between $\boldsymbol{A}(\boldsymbol{Br})$ and $\boldsymbol{Cr}$ is below `eps`, and returns `True` if so.

```
def array_equal(a,b, eps=1e-7):
    return np.abs(a-b).max() <= eps

k = 0
m = 1000
s = time.time()
for i in range(m):
    A = 100*rng.random(N*N).reshape((N,N))
    B = 100*rng.random(N*N).reshape((N,N))
    C = A@B + 0.1*rng.random(N*N).reshape((N,N))
    if (mode == 0):
        t = True
        for j in range(reps):
            r = (2*rng.random(N)).astype("uint8").reshape((N,1))
            t &= array_equal(mmult(A,mmult(B,r)), mmult(C,r))
    else:
        t = array_equal(mmult(A,B), C)
    k += 1 if t else 0
print("%0.8f %0.3f" % (time.time()-s, k/m))
```

Listing 11-5: Freivalds' algorithm

The outer `for` loop executes 1,000 trials using a randomly selected set of matrices each time. $\boldsymbol{C}$ is such that it never equals $\boldsymbol{AB}$, so every call to `array_equal` should return `False`.

The body of the outer `for` loop either multiplies $\boldsymbol{A}$ and $\boldsymbol{B}$ directly (`mode==1`), or uses Freivalds' algorithm by generating a random binary vector, `r`. Note that `r` is reshaped to be a $n \times 1$ column vector, as required for matrix multiplication.

The inner `for` loop applies Freivalds' repeatedly (`reps`) each time, ANDing the result with `t`. The AND operation means that after `reps` tests with

different `r` vectors each time, so the only way `t` is still true is if all tests give a wrong result. Each test should see `array_equal` return `False` because $\boldsymbol{AB} \neq \boldsymbol{C}$ by design. Once `t` becomes `False`, it remains `False` for all remaining tests, so even one correct output from `array_equal` causes `t` to have the expected value.

If `t` is still `True` after the inner loop, then the trial failed and we increment `k`. After all trials, we print the total run time and the fraction of the 1,000 trials that failed.

Freivalds' algorithm is a Monte Carlo algorithm because it might, with a probability we can minimize, produce a false output and claim $\boldsymbol{AB} = \boldsymbol{C}$ when it isn't true.

Let's turn to a different type of question for the next section: to get an estimate of the number of things in a collection, is it necessary to count them individually?

Counting Animals

Ecologists often want to know how many animals of a particular species live in an area, though counting each one is often impossible. Enter *mark and recapture*, a strategy for estimating population size from a small sample.

In mark and recapture, the ecologist first goes into the field and captures n specimens, which they then mark and release. A short while later, they revisit the field and capture animals again until they get at least one that is marked. If they capture K animals to get k that are marked, they now have everything necessary to estimate the full population size, N. They do this by using ratios.

Initially, the ecologist marked n of the N animals, meaning the fraction of the total population marked is n/N. The recapture phase netted k marked animals out of K. Assuming no births, deaths, or migrations, the two ratios should be approximately equal, so solving for N gives:

$$N_L \approx \frac{nK}{k}$$

This equation results in the *Lincoln-Petersen population estimate*, hence N_L.

A slightly less biased estimate of the population (or so it's claimed) comes from the *Chapman population estimate*:

$$N_C \approx \frac{(n+1)(K+1)}{k+1} - 1$$

Finally, we have a Bayesian approach to mark and recapture:

$$N_B \approx \frac{(n-1)(K-1)}{k-2}$$

This approach requires at least three marked animals in the recapture group to avoid dividing by zero.

Let's compare these three different estimates for the same population size with *mark_recapture.py*:

```
> python3 mark_recapture.py
mark_recapture <pop> <mark> <reps> [<kind> | <kind> <seed>]
  <pop>  - population size (e.g. 1000)
  <mark> - number to mark (< pop)
  <reps> - number of repetitions
  <kind> - randomness source
  <seed> - seed
```

The code simulates marking and recapturing by randomly marking a specified number of animals before recapturing a fraction of the population to count how many are marked. Let's run the code a few times to get a feel for the output. We'll fix the true population size at 1,000 and initially mark 100, or 10 percent. Setting the repetitions to 1 takes a single sampling, which is similar to what an ecologist might do in practice. We get:

```
> python3 mark_recapture.py 1000 100 1 mt19937 11
Lincoln-Petersen population estimate = 1250
Chapman population estimate          = 1132
Bayesian population estimate         = 1633

> python3 mark_recapture.py 1000 100 1 mt19937 12
Lincoln-Petersen population estimate = 666
Chapman population estimate          = 636
Bayesian population estimate         = 753

> python3 mark_recapture.py 1000 100 1 mt19937 13
Lincoln-Petersen population estimate = 833
Chapman population estimate          = 783
Bayesian population estimate         = 980
```

The estimates vary widely from run to run, as we might expect from a randomized algorithm. While the Lincoln-Petersen and Chapman estimates are generally low, the Bayesian estimates are closer to or even exceed the population size.

Using a single repetition is akin to attempting to generalize from a single collected data point, so let's increase the repetitions:

```
> python3 mark_recapture.py 1000 100 25 mt19937 11
Lincoln-Petersen population estimate = 1028.4713 +-  78.4623
Chapman population estimate          =  940.0367 +-  61.6982
Bayesian population estimate         = 1345.2015 +- 166.3078

> python3 mark_recapture.py 1000 100 25 mt19937 12
Lincoln-Petersen population estimate = 1052.0985 +-  61.3198
Chapman population estimate          =  963.4317 +-  49.9620
Bayesian population estimate         = 1345.9986 +- 108.4192

> python3 mark_recapture.py 1000 100 25 mt19937 13
```

```
Lincoln-Petersen population estimate = 1112.8340 +/-  80.5759
Chapman population estimate          = 1009.5146 +/-  63.3742
Bayesian population estimate         = 1485.2546 +/- 169.0492
```

The output now reflects the mean and standard error for 25 repetitions, providing a better idea of how the estimates behave. The Lincoln-Petersen and Chapman results are closer to the actual population size, while the Bayesian estimate is consistently too high. The standard errors are illustrative as well, with the Bayesian standard error being larger than the others, indicating more trial-to-trial variation.

Try experimenting with different combinations of population size and number of animals initially marked.

Figure 11-3 presents three somewhat crowded graphs.

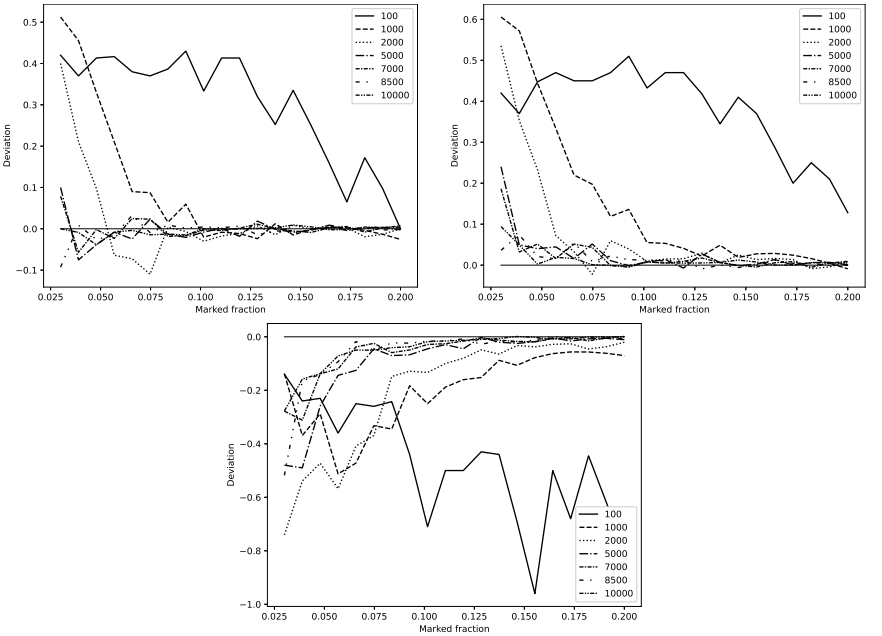


Figure 11-3: The three mark and recapture estimators as a function of the true population size and the fraction of that size initially marked. The plots show the signed deviation from the true population size: Lincoln-Peterson (top left), Chapman (top right), and Bayesian (bottom).

In the top-left graph in Figure 11-3, each of the seven plotted lines represents a different true population size from 100 to 10,000. The *x*-axis indicates the fraction of the true population marked by the ecologist on their first trip to the field. The value plotted is the median signed difference between the Lincoln-Petersen estimate for that combination of population size and fraction initially marked and the true population size. If the curve is above zero, the estimate is too low; below zero, and it's too high. In other words, the graph shows $N_{\text{true}} - N_{\text{est}}$ so that underestimating is a positive difference and overestimating is negative. The remaining two graphs show the

same information for the Chapman (top-right) and Bayesian (bottom) estimators.

For populations above 1,000, the Lincoln-Petersen estimator is generally useful when initially marking more than 10 percent of the population, which may not be feasible in practice. However, for small populations, the estimator requires some 20 percent of the population to be marked to achieve reliability. One might use a simulation to generate a correction function for the Lincoln-Petersen estimator based on the suspected population size and the number of animals initially marked.

The Chapman estimator consistently underestimates the true population size to the point where one questions its utility compared to the Lincoln-Petersen estimate. However, the underestimate is relatively consistent for populations above 1,000, so again, a fudge factor might be derived from a simulation.

The Bayesian estimator's performance is quite different. It consistently overestimates the actual population, converging to the true population value only when the population becomes large and the percent initially marked is also significant (at least 15 percent). In practice, these conditions are unlikely to be met.

Figure 11-3 is the output of *mark_recapture_range.py*, which can be understood by examining the relevant parts of *mark_recapture.py* in Listing 11-6.

```
lincoln = []
chapman = []
bayes = []

for j in range(nreps):
    pop = np.zeros(npop, dtype="uint8")
    idx = np.argsort(rng.random(npop))[:nmark]
    pop[idx] = 1

    K = nmark
    while (True):
        idx = np.argsort(rng.random(npop))[:K]
        k = pop[idx].sum()
        if (k > 2):
            break
        K += 5

    lincoln.append(nmark*K/k)
    chapman.append((nmark+1)*(K+1)/(k+1) - 1)
    bayes.append((nmark-1)*(K-1)/(k-2))
```

Listing 11-6: Simulating mark and recapture estimates

The outer `for` loop over `nreps` handles the trials. For each trial, we create a population (`pop`) vector where `npop` is the population size from the command line. The vector is initially zero as we haven't marked any animals yet.

The next two lines represent the ecologist's first field trip. The `argsort` trick, coupled with keeping only the first `nmark` elements of the sort order, sets `idx` to the indices of `pop` that the ecologist has initially captured and marked (`pop[idx]=1`).

The second code paragraph represents the recapture phase in which the ecologist returns to the field and captures as many animals as were initially marked (`K`). We represent the captured animals by the `K` indices in `idx` as assigned in the inner `while` loop.

Marks are binary, so the sum of the selected elements of `pop` is the number of marked animals, `k`. If `k` is 3 or greater, break out of the `while` loop. Otherwise, increase `K` by five and try again. The final code paragraph calculates the three estimates of the true population size for this trial.

When the outer `for` loop exits, we have three vectors of estimates for the given population size and number initially marked. The remainder of *mark_recapture.py* displays the results. Given the simulation results, my money's on the Lincoln-Petersen estimator.

Let's move on from counting to the mathematically important task of primality testing.

Testing Primality

Prime numbers—integers evenly divisible by only themselves and one—are greatly beloved by number theorists. Primes have fascinated humanity since antiquity, and significant computing power is currently devoted to locating *Mersenne primes* of the form $2^p - 1$, where p is a prime number.

The largest known primes are Mersenne primes. As of this writing, the largest known Mersenne prime, discovered in 2018, is:

$$M_{82589933} = 2^{82,589,933} - 1$$

$M_{82589933}$ is a 24,862,048 digit number. Mersenne primes are sometimes denoted by their number and not their exponent. Therefore, $M_{82589933}$, the 51st Mersenne prime, might be given as M_{51}.

NOTE *To contribute in locating Mersenne primes, visit http://www.mersenne.org and sign up for the Great Internet Prime Search.*

How do we know if n is a prime number? The definition gives us a natural starting point for a primality testing algorithm: if the only numbers that evenly divide n (resulting in no remainder) are 1 and n, then n is a prime.

Let's turn this definition into an algorithm. The brute force approach is to test every number that could be a factor of n. In practice, this means testing every integer up to $\sqrt{n}$ because any factor of n larger than $\sqrt{n}$ will necessarily be multiplied by some number less than $\sqrt{n}$, and will be caught before reaching $\sqrt{n}$.

When contemplating nearly 25 million digit numbers, the amount of work involved increases dramatically. And if n is prime, must we test *every* integer up to $\sqrt{n}$?

The *Miller-Rabin test* is a fast, randomized algorithm to decide whether a positive integer, n, is prime. However, to understand the Miller-Rabin test, we need to know a bit about modular arithmetic.

Modular Arithmetic

We're used to the set of integers, denoted $\mathbb{Z}$ from the German for number. Integers are unbounded and extend infinitely in both directions from zero. If we restrict the range to the set $\{0, 1, 2, 3\}$, we can define arithmetic operations over this set by wrapping around as needed. Adding works as expected if the sum is less than 4: $1 + 1 = 2$ and $2 + 1 = 3$. However, if the sum exceeds 4, we wrap around. For example, $2+3 = 1$ because $2+3 = 5$, and we subtract 4 from 5 to get 1. Another way to view these operations is to apply the modulo operator after each addition to return the remainder after dividing by 4. For example, $5 \bmod 4 = 1$.

Pierre Fermat, a 17th-century French mathematician, realized that if n is a prime number, then:

$$a^{n-1} \equiv 1 \pmod{n},\ 0 < a < n$$

where $\equiv$ means:

$$a^{n-1} \bmod n = 1 \bmod n = 1$$

Great! We have a primality test: pick an integer $0 < a < n$, raise it to the $n - 1$ power, divide by n, and see if the remainder is 1. If it is, then n is a prime number, so the algorithm works and identifies n as a prime. However, some composite numbers also pass the test for many values of a, meaning this alone isn't sufficient to prove n is a prime. If this test fails, then n is definitely not a prime.

The Miller-Rabin test combines Fermat's test with another fact: if n is prime, the following is likely also true:

$$a^{2^r d} \equiv -1 \pmod{n},\ 0 \leq r < s \text{ and } 0 < a < n$$

for some r in $[0, s)$ where $n = 2^s d + 1$ and d is odd. It's likely true because there are sometimes a values satisfying the congruence even if n is composite. We'll discuss these nonwitness numbers shortly.

The first condition, Fermat's test, is straightforward enough, but let's unpack this second condition. We need to express n as $2^s d + 1$ or, equivalently, as $n - 1 = 2^s d$. For suitable choices of s and d, $2^s d$ is another way of writing the exponent in the Fermat condition.

All of the math in $\equiv -1 \pmod{n}$ is modulo n, meaning the numbers are in the set $\{0, 1, 2, \ldots, n - 1\}$, usually denoted as $\mathbb{Z}_n$. We view a negative number as counting backward, so $-1 \equiv n - 1$.

The second condition checks to see if $x^2 \equiv -1 \pmod{n}$ for some x. The Miller-Rabin test uses a sequence of such values of x, looking for one that, when squared modulo n, gives -1 (that is, $n - 1$). The sequence begins with $r = 0$ and d as the exponent. The next check uses the square, which is the

same as $r = 1$:

$$(a^{2^0 d})^2 = a^{2(2^0 d)} = a^{2^{0+1} d} = a^{2^1 d}$$

This is all modulo n. The next squaring returns $r = 2$, and so on.

If any of the sequence of such expressions is equivalent to $n - 1$, then n has a reasonably high probability of being a prime number. Otherwise, n is definitely *not* prime, and a is a *witness* to this fact.

The Miller-Rabin Test

Let's put Miller-Rabin into code, as in Listing 11-7.

```
def MillerRabin(n, rounds=5):
    if (n==2):
        return True
    if (n%2 == 0):
        return False

    s = 0
    d = n-1
    while (d%2 == 0):
        s += 1
        d //= 2

    for k in range(rounds):
        a = int(rng.random()) # [1,n-1]
        x = pow(a,d,n)
        if (x==1) or (x == n-1):
            continue
        b = False
        for j in range(s-1):
            x = pow(x,2,n)
            if (x == n-1):
                b = True
                break
        if (b):
            continue
        return False
    return True
```

Listing 11-7: Miller-Rabin in code

The function `MillerRabin` accepts n and `rounds` with a default value of 5. The first code paragraph captures trivial cases. As half of all numbers are even, testing directly for 2 and evenness saves time.

The second code paragraph locates s and d so that $n = 2^s d + 1$. It's always possible to find an s and d decomposition for any n (positive integer).

For now, we'll focus on the body of the outer `for` loop in the third paragraph, which implements a pass through the Miller-Rabin test for a randomly selected a and initial x value, $a^d \pmod{n}$.

The built-in Python function, `pow`, computes exponents and accepts an optional third argument so that `pow(a,d,n)` efficiently implements $a^d \pmod{n}$.

The following `if` checks for 1 or −1. If that's the case, the Fermat test has passed, so this pass through the outer `for` loop is over.

Otherwise, the inner `for` loop initiates the sequence of successive squarings of $x = a^d$ while looking for one equivalent to −1. If such a squaring is found, the inner loop breaks and the outer loop cycles; otherwise, n is composite and `MillerRabin` returns `False`.

When all rounds (the loop over `k`) are complete, and every test supports the notion that n is a prime, `MillerRabin` returns `True`.

The outer `for` loop applies the Miller-Rabin test repeated for randomly selected a values. Since an a value demonstrating n to be a composite number is a witness number, an a value that leads to a claim of prime when n is not prime is a *nonwitness* number. It is never the case that all possible a values for a given n are nonwitness numbers, so repeated applications of the outer loop body minimize the probability that a nonprime input will return `True`.

You'll find `MillerRabin` in the file *miller_rabin.py*. It expects a number to test, the number of rounds (a values to try), and the randomness source:

```
> python3 miller_rabin.py
miller_rabin <n> <rounds> [<kind> | <kind> <seed>]
  <n>      - number to test
  <rounds> - number of rounds
  <kind>   - randomness source
  <seed>   - seed
```

For example:

```
> python3 miller_rabin.py 73939133 1
73939133 is probably prime
> python3 miller_rabin.py 73939134 1
73939134 is composite
```

The output must be correct for these cases as 73,939,133 is a prime, and the closest two primes can be to each other is 2 away:

```
> python3 miller_rabin.py 8675309 1
8675309 is probably prime
> python3 miller_rabin.py 8675311 1
8675311 is probably prime
```

8,675,309 and 8,675,311 are twin primes, so the test is correct.

Miller-Rabin always labels a prime a prime. Let's explore when Miller-Rabin fails.

Nonwitness Numbers

As mentioned, a witness number, a, testifies to the fact that n isn't prime. Also, there are composite numbers for which the Miller-Rabin test fails if it selects a nonwitness number as a.

We'll force the Miller-Rabin algorithm to fail, probabilistically, using a composite number with a known set of nonwitness numbers to see if we can detect the expected number of failures.

Our target is $n = 65$. As a multiple of 5, 65 is composite. There are 64 potential witness numbers, from 1 through 64. Of these potential witness numbers, it's known that 8, 18, 47, 57, and 64 are nonwitness numbers. If the Miller-Rabin test runs for one round and selects a nonwitness number for a, it fails and claims that 65 is prime.

Because there are five nonwitness numbers out of 64 possible, the Miller-Rabin test for a single round should fail about $5/64 = 7.8$ percent of the time. I checked this by running *miller_rabin.py* 1,000 times and counting the number of times the output indicated a prime, which it did precisely 78 times, implying a failure rate of 78/1000 = 7.8 percent.

At worst, the Miller-Rabin single-round failure probability for arbitrary n is $1/4$. Since each round is independent of the previous, running the test for k rounds means the worst possible failure probability is $(1/4)^k = 4^{-k}$. However, for most n values, the actual failure probability is far less than this.

Let's stick with 65. Knowing that its single-round failure rate is about 7.8 percent, running two rounds should fail $(5/64)^2 \approx 0.61$ percent of the time. Running *miller_rabin.py* 20,000 times produced 131 failures, giving a failure rate of $131/20000 = 0.655$ percent. Three rounds puts the failure rate at about 0.05 percent. We can achieve any desired precision by setting k high enough.

Miller-Rabin Performance

Let's compare Miller-Rabin's runtime performance to the brute force approach implemented in *brute_primes.py*. The code in *prime_tests.py* runs both Miller-Rabin and the brute force algorithm for the largest 1, 2, 3, . . . , 15-digit prime numbers. Recall, the brute force algorithm runs the longest when the input is a prime.

The largest single-digit prime is 7, while the largest 15-digit prime is 999,999,999,999,989. Figure 11-4 plots the mean of five runs of *miller_rabin.py* and *brute_primes.py* for each prime to show how the runtime changed as the inputs grew.

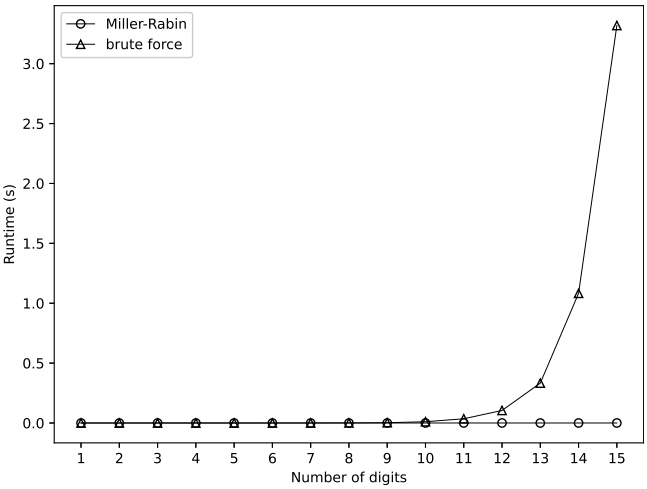


Figure 11-4: Comparing Miller-Rabin to the brute force primality test

The runtime complexity of the brute force algorithm is $\mathcal{O}(\sqrt{n})$ while that of Miller-Rabin is $\mathcal{O}(\log^3 n)$. The brute force algorithm quickly becomes unmanageable, even though it's sublinear, because $\sqrt{n} = n^{1/2}$ and $1/2 < 1$.

Miller-Rabin is a Monte Carlo algorithm because it claims n is prime when there's a nonzero probability that it isn't. If n truly is a prime, Miller-Rabin always correctly labels it, but it also calls some composite numbers prime regardless of the number of rounds. Therefore, Miller-Rabin's false positive rate is nonzero, but its false negative rate is identically zero. In practice, however, increasing the number of rounds can make the false positive rate as low as desired.

We have one more randomized algorithm to contemplate.

Working with Quicksort

Quicksort was developed by British computer scientist Tony Hoare in 1959 and is probably still the most widely used sorting algorithm. It's NumPy's default, for example.

If you take an undergraduate course in algorithms, you'll almost assuredly run across Quicksort, as it's easy to implement and understand, even if it's recursive. While most courses focus on characterizing its runtime complexity, we'll discuss the algorithm at a high level instead, and then run experiments on two versions: the standard nonrandom version and a randomized version.

Quicksort is a recursive, *divide-and-conquer* algorithm, meaning it calls itself on smaller and smaller versions of the problem until it encounters a base condition, at which point the implementation pieces everything back together to produce a sorted output.

The algorithm is as follows:

1. If the input array is empty or has only one element, return it.

2. Pick a *pivot* element, either the first in the array or at random.
3. Separate the array into three subsets: those elements less than the pivot, those equal to the pivot, and those greater than the pivot.
4. Return the concatenation of Quicksort called on the lower elements, the elements matching the pivot, and Quicksort called on the higher elements.

Step 1 is the base condition. If the array is empty or contains a single element, it's sorted. Step 2 picks a pivot value, an array element we use in Step 3 to split the array into three parts: those less than, equal to, and greater than the pivot. Step 2 is where randomness comes into play. Nonrandom Quicksort always picks a specific element of the array, as it's already assumed to be in random order. Randomized Quicksort, however, selects its pivot element at random. We'll experiment with the subtle difference between nonrandom and random Quicksort.

Step 4 is the recursive part. The array is sorted if we merge the sorted lower partition with the same partition followed by the sorted higher partition. We sort the lower and higher partitions by using the sorting routine, that is, by calling Quicksort again. Each call on a portion of the array will, we assume, work with a smaller number of elements until we have single elements, the base condition of Step 1.

Naive sorting methods, like bubble sort or gnome sort, run in $\mathcal{O}(n^2)$ time where n is the number of elements to sort. As we've learned, $\mathcal{O}(n^2)$ algorithms are acceptable for small n values, but quickly become unmanageable as n grows. Quicksort's average runtime complexity is $\mathcal{O}(n \log n)$, which grows at a much slower rate. This is why Quicksort is still widely used over 50 years after its introduction.

While Quicksort's *average* complexity is $\mathcal{O}(n \log n)$, if the array passed to Quicksort is already mostly or completely sorted, the complexity becomes $\mathcal{O}(n^2)$, which is no better than bubble sort. This happens if the array is in order or reverse order. Let's find out whether randomized Quicksort can help us here.

Running Quicksort in Python

The file *Quicksort.py* implements Quicksort twice. The first implementation uses a random pivot (`QuicksortRandom`), and the second implementation always uses the first element of the array as the pivot (`Quicksort`). The functions are in Listing 11-8.

```
def QuicksortRandom(arr):
    if (len(arr) < 2):
        return arr
    pivot = arr[np.random.randint(0, len(arr))]
    low  = arr[np.where(arr < pivot)]
    same = arr[np.where(arr == pivot)]
    high = arr[np.where(arr > pivot)]
    return np.hstack((QuicksortRandom(low), same, QuicksortRandom(high)))
```

```
def Quicksort (arr):
    if (len(arr) < 2):
        return arr
    pivot = arr[0]
    low  = arr[np.where(arr < pivot)]
    same = arr[np.where(arr == pivot)]
    high = arr[np.where(arr > pivot)]
    return np.hstack((Quicksort(low), same, Quicksort(high)))
```

Listing 11-8: Randomized and nonrandom Quicksort

For this example, we use NumPy instead of our `RE` class because it's already loaded, which minimizes the overhead when calling `QuicksortRandom`. The implementations differ only in how they assign `pivot`.

Both implementations follow the Quicksort algorithm step-by-step. First, we check for the base condition where `arr` is already sorted. We then split into `low`, `same`, and `high` based on the selected `pivot`. Finally, NumPy's `hstack` function concatenates the vectors returned by the recursive calls to `Quicksort`.

A high-performance implementation wouldn't call `where` three times, as each makes a full pass over `arr`, but we're interested only in relative performance differences as the input size changes.

Experimenting with Quicksort

The file *quicksort_tests.py* generates two graphs. The first, on the left in Figure 11-5, compares randomized Quicksort and nonrandom Quicksort as the input array size increases. In all cases, the input arrays are in random order. In other words, the left side of Figure 11-5 represents the average case runtime. The points plotted are the mean over five runs. The dashed line represents $y = n \log n$.

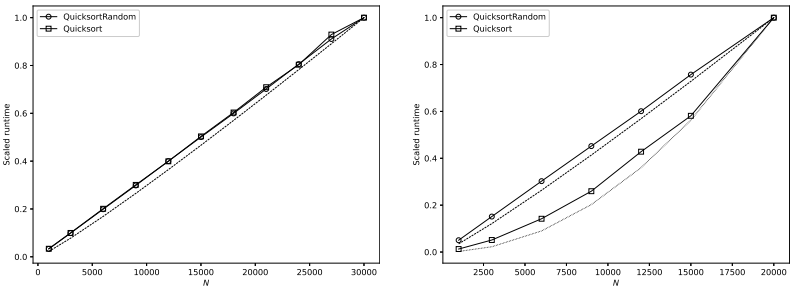


Figure 11-5: Randomized and nonrandom Quicksort on random inputs (left), and the same algorithms on pathological inputs (right)

The rightmost graph in Figure 11-5 shows the runtime for the case with already sorted input. This situation forces deterministic Quicksort into becoming an $\mathcal{O}(n^2)$ algorithm, which is why it tracks the curved plot, $y = n^2$.

Randomized Quicksort, on the other hand, is unaffected by the order of the input and runs as before.

Correctly interpreting Figure 11-5 requires an explanation. Asymptotic runtime performance of algorithms ignores multiplicative factors and constants as they don't alter the overall form of the function as n increases. The randomized Quicksort function takes slightly longer to run than the nonrandom Quicksort because of the extra step of selecting a random index into the array. Therefore, plotting both runtimes together would make it somewhat difficult to see that the overall functional form is the same between `QuicksortRandom` and `Quicksort`. Moreover, plotting $y = n \log n$ follows an entirely different scale in terms of y-axis values, but again, the form of the function is the same. Therefore, to plot all three together, Figure 11-5 divides each y value by the maximum y value to map the output to $[0, 1]$ regardless of the actual range. This clarifies that randomized Quicksort and nonrandom Quicksort are scaling in the same way and are following $\mathcal{O}(n \log n)$—all curves lie essentially on top of each other.

Now reconsider the right side of Figure 11-5 showing the case where the input array is already sorted. Again, the dashed line shows $y = n \log n$, and randomized Quicksort still follows that form. However, nonrandom Quicksort, which selects the first element of the array as its pivot, does not. Instead, it follows the dotted line, $y = n^2$, meaning the pathological input case alters nonrandom Quicksort, turning it into an $\mathcal{O}(n^2)$ algorithm.

Randomized Quicksort is a Las Vegas algorithm because it always returns the proper output—a sorted array. While the randomness involved doesn't make the implementation easier, it protects against a pathological case that's more frequent in practice than we might initially suspect. Therefore, I recommend always using randomized Quicksort.

To understand why nonrandom Quicksort behaves so poorly with sorted input, consider what happens during a pass when the pivot is the smallest value in the array; for example, when picking the first element as the pivot and the input array is already sorted. When this happens, the low vector is empty and, ignoring duplicates of the pivot, all the remaining values in the array end up in the high vector. This happens every time the function recurses, turning the recursion into a list of function calls n deep. Add the $\mathcal{O}(n)$ pass through the array on each recursive call (implicit in our implementation via `where`), and we arrive at an $\mathcal{O}(n^2)$ algorithm, which is no better than a bubble sort.

Selecting a random pivot at each level ensures that this situation won't happen in the long run, as it would amount to a string of n rolls of an n-sided fair die each landing on 1—an increasingly unlikely event as n grows.

Exercises

Consider the following possibilities to further explore randomized algorithms:

- Write a Las Vegas algorithm to locate positive integers, a, b, and c, that satisfy $a^2 + b^2 = c^2$. Your code will be a Las Vegas algorithm

because there are an infinite number of solutions, namely, all the Pythagorean triples.

- Can you write a successful Las Vegas program to find positive integers a, b, and c such that $a^n + b^n = c^n$ for some $n > 2$? If not, how about a Monte Carlo algorithm? What might your stopping criteria be? I recommend searching for "Fermat's last theorem."
- Extend the permutation sort runtime plot for $n = 11$, 12, or even 13. How long do you have to wait?
- Make a plot of the mean number of trials of Freivalds' algorithm to get a failure case as a function of n, the size of the square matrices.
- The file *test_mmult.py* generates output suitable for *curves.py* from Chapter 4. Use that output and *curves.py* to generate fits. Is the fit exponent what you expect for the naive algorithm? What about NumPy, knowing that it uses Strassen's algorithm?
- I have a bag full of marbles. I want to estimate how many are in the bag. Therefore, I pick one randomly, mark it, and put it back in the bag. I then pick another marble randomly, mark it, and put it back in the bag. I repeat this process, counting the number of marbles selected, until I pick a marble I've already marked. If the number of marbles picked and marked is k, then the number of marbles in the bag is approximately:

$$N \approx \left\lfloor \frac{2k^2}{\pi} \right\rceil$$

where the combination of floor ($\lfloor$) on the left and ceiling ($\rceil$) on the right means "round to the nearest integer." I encountered this algorithm via a brief description of the process, but the description had no derivation for the formula and no references. Nonetheless, it sort of works. After experimenting some, I realized that the estimate is better if the formula is tweaked slightly to become:

$$N \approx \left\lfloor 0.8 \left(\frac{2k^2}{\pi} \right) \right\rceil$$

Implement this algorithm and explore how well it works on average. Then examine *count.py*, which runs the algorithm for many iterations, averages the results, and produces plots. For example:

```
> python3 count.py 1_000_000_000 40 pcg64 6502
N = 1023827699, iterations 41414, total 1656576
```

estimates slightly more than 1 billion marbles in the bag. The correct number is exactly 1 billion. It uses 40 iterations of the algorithm for a total of 1.7 million marbles marked. Figure 11-6 is the resulting plot, *count_plot.png*, which shows each of the 40 estimates, the true value (solid line), and overall estimate (dashed):

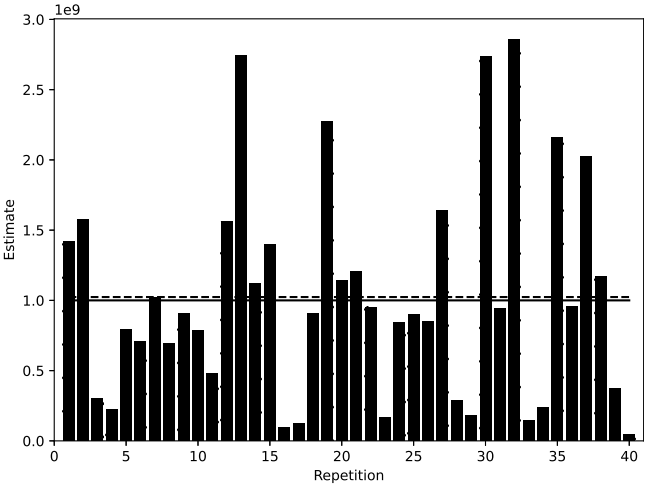


Figure 11-6: 40 estimates of marbles

Please contact me if you know a reference for this algorithm or how to derive the estimate formula.

- Can you think of a "fudge factor" for the Lincoln-Petersen population estimate for the case where the population is believed to be small?
- How does the runtime performance of nonrandom Quicksort vary as the array becomes more disordered? In other words, fix the array size (n) but change the degree of disorder in the array. For example, begin with a sorted array, then swap two elements, then three, and so on. Is the transition from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ linear with the number of elements swapped? Or does it seem more rapid?

Summary

In this chapter, we explored randomized algorithms, differentiating between Las Vegas and Monte Carlo. The former always produce correct output, eventually, while the latter may produce incorrect output. We considered permutation sort and Freivalds' algorithm for testing matrix multiplication. We learned that we can turn permutation sort from a Las Vegas algorithm into a Monte Carlo algorithm by imposing a limit on the number of candidate permutations considered. In general, we can transform Las Vegas algorithms into Monte Carlo algorithms, but not vice versa.

We then discussed the mark and recapture algorithm that ecologists use to estimate animal populations. We estimate the number of animals in a population by marking a known number and then recapturing animals and looking at the number marked. With sufficient numbers, the ratio of marked animals to animals recaptured should match the ratio of animals originally marked to the population size. We explored three estimators associated with this process and saw how they behave.

The Miller-Rabin algorithm quickly decides whether a positive integer is a prime. However, as a randomized algorithm, there's a certain probability that it'll falsely claim a composite number is prime. We learned how to decrease the likelihood of a false positive by repeated applications.

We concluded the chapter by comparing nonrandom and randomized Quicksort implementations. Randomized Quicksort adds little to the runtime while protecting against pathological inputs that are already (or mostly) sorted.

In our final chapter, we'll consider randomness as it relates to sampling from probability distributions.