Analyzing the Efficiency of Algorithms

We can calculate an approximation to the value e by summing up a series of terms. The more terms we add together, the better our approximation will be:

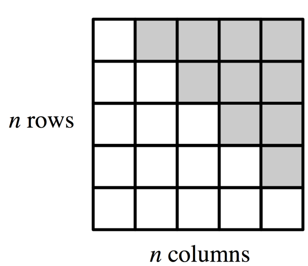
$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots$$

```java
// e1.java

public static void e1(int n) { // n is the number of terms to add up
    double sum = 0.0;
    for (int i = 0; i < n; i++) {
        sum = sum + 1.0 / factorial(i);
    }
    System.out.printf("e is approximately %s\n", sum);
}

public static int factorial(int k) {
    int product = 1;
    for (int i = 1; i <= k; i++) {
        product = product * i;
    }
    return product;
}
```

What is the total number of *multiplication operations* performed by the function `e1` when it is called with an input value of n? There are no multiplication operations within the body of `e1` itself. However, notice that `e1` calls the helper function `factorial` repeatedly, in a for-loop.

Each time the `factorial` function is called with an input value of k, it performs exactly k multiplications. Since `e1` calls `factorial(0)`, `factorial(1)`, `factorial(2)`, etc., up to `factorial(n-1)`, it performs a total of $0+1+2+3+\ldots+n-1$ multiplications when adding up n terms of the series.

We can express $0+1+2+3+\ldots+n-1$ equivalently as the formula $\frac{1}{2}n^2 - \frac{1}{2}n$



$$\begin{aligned}
&\text{Number of white squares} && = 1+2+\ldots+n-1+n \\
&\text{Number of gray squares} && = 1+2+\ldots+n-1 \\
&\text{Total number of squares} && = 2 \times \big(1+2+\ldots+n-1\big) + n \\
& && = n^2 \\
&\text{Solving for } \big(1+2+\ldots+n-1\big) && = (n^2-n)/2 \\
& && = \tfrac{1}{2}n^2 - \tfrac{1}{2}n
\end{aligned}$$

So `e1` performs $\frac{1}{2}n^2 - \frac{1}{2}n$ multiplications in all, when called with an input value of n.

Here is a table of the number of multiplications performed for various values of n:

n	$\frac{1}{2}n^2 - \frac{1}{2}n$
1	0
2	1
3	3
4	6
5	10
10	45
20	190
30	435
40	780
50	1225
60	1770

We could use other measures of running time, such as the number of additions, the number of comparisons, or the total number of arithmetic operations:

`factorial`(k):

Number of multiplications	$= k$
Number of additions	$= k$
Number of comparisons	$= k + 1$

`e1`(n):

$$
\begin{aligned}
&\text{Number of multiplications} && = \tfrac{1}{2}n^2 - \tfrac{1}{2}n \\
&\text{Number of additions} && = \big(0 + 1 + 2 + 3 + \ldots + n{-}1\big) + 2n \;=\; \tfrac{1}{2}n^2 + \tfrac{3}{2}n \\
&\text{Number of divisions} && = n \\
&\text{Number of comparisons} && = (n{+}1) + \big[(0{+}1) + (1{+}1) + (2{+}1) + \ldots + (n{-}1){+}1\big] \\
& && = (n{+}1) + \big[1 + 2 + 3 + \ldots + n\big] \\
& && = (n{+}1) + \big[1 + 2 + 3 + \ldots + n{-}1\big] + n \\
& && = (n{+}1) + \big[\tfrac{1}{2}n^2 - \tfrac{1}{2}n\big] + n \\
& && = \tfrac{1}{2}n^2 + \tfrac{3}{2}n + 1 \\
&\text{Total operations performed} && = \#\text{multiplications} + \#\text{additions} + \#\text{divisions} + \#\text{comparisons} \\
& && = \big(\tfrac{1}{2}n^2 - \tfrac{1}{2}n\big) + \big(\tfrac{1}{2}n^2 + \tfrac{3}{2}n\big) + n + \big(\tfrac{1}{2}n^2 + \tfrac{3}{2}n + 1\big) \\
& && = \tfrac{3}{2}n^2 + \tfrac{7}{2}n + 1
\end{aligned}
$$

Notice that whichever measure we use, we still end up with an n^2 term. We say that the *running time* of `e1` is $T(n) = \frac{3}{2}n^2 + \frac{7}{2}n + 1$, and its *time complexity* is "order n squared", or $O(n^2)$.

Now consider an alternative way of computing e:

```java
// e2.java

public static void e2(int n) { // n is the number of terms to add up
    double sum = 0.0;
    double denom = 1.0;
    for (int i = 1; i <= n; i++) {
        sum = sum + 1.0 / denom;
        denom = denom * i;
    }
    System.out.printf("e is approximately %s\n", sum);
}
```

$$
\begin{aligned}
\text{Number of multiplications} &= n \\
\text{Number of additions} &= 2n \\
\text{Number of divisions} &= n \\
\text{Number of comparisons} &= n+1 \\
\text{Total operations performed} &= \#\text{multiplications} + \#\text{additions} + \#\text{divisions} + \#\text{comparisons} \\
&= n + 2n + n + (n+1) \\
&= 5n + 1
\end{aligned}
$$

We say that the running time of `e2` is $T(n) = 5n+1$, and its time complexity is "order n", or $O(n)$.

This table compares the running times of `e2` and `e1` for various values of n:

n	$5n+1$	$\frac{3}{2}n^2 + \frac{7}{2}n + 1$
0	1	1
10	51	186
20	101	671
30	151	1456
40	201	2541
50	251	3926
60	301	5611
70	351	7596
80	401	9881
90	451	12466
100	501	15351

Some examples

Big-O notation	Description	Examples
$O(1)$	"constant time"	1 6 342 5 trillion
$O(n)$	"linear time"	n $1000n$ $5n+1$ $40n + 5$ trillion
$O(n^2)$	"quadratic time"	n^2 $\frac{1}{100}n^2$ $7n^2 + 3n + 24$
$O(n^3)$	"cubic time"	$100n^3 + 700n^2 + 1000$
$O(\log n)$	"logarithmic time"	binary search, fast exponentiation
$O(2^n)$	"exponential time"	recursive fibonacci, lookahead in games

Standard Exponentiation Algorithm

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \times b^{n-1} & \text{if } n > 0 \end{cases}$$

```java
public static double power(double b, long n) {
    if (n == 0) {
        return 1;
    } else {
        return b * power(b, n - 1);
    }
}
```

Fast Exponentiation Algorithm

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{n}{2}})^2 & \text{if } n > 0 \text{ and } n \text{ is even} \\ b \times b^{n-1} & \text{if } n > 0 \text{ and } n \text{ is odd} \end{cases}$$

```java
public static double fastpower(double b, long n) {
    if (n == 0) {
        return 1;
    } else if (isEven(n)) {
        return square(fastpower(b, n / 2));
    } else {
        return b * fastpower(b, n - 1);
    }
}
```

Best case example: b^{32}

n	result
32	$(b^{16})^2$
16	$((b^8)^2)^2$
8	$(((b^4)^2)^2)^2$
4	$((((b^2)^2)^2)^2)^2$
2	$(((((b^1)^2)^2)^2)^2)^2$
1	$(((((b \times b^0)^2)^2)^2)^2)^2$
0	$(((((b \times 1)^2)^2)^2)^2)^2$

6 multiplications

about $\log_2(n)$ multiplications in the best case

exact # of multiplications: $\log_2(n) + 1$

Worst case example: b^{31}

n	result
31	$b \times b^{30}$
30	$b \times (b^{15})^2$
15	$b \times (b \times b^{14})^2$
14	$b \times (b \times (b^7)^2)^2$
7	$b \times (b \times (b \times b^6)^2)^2$
6	$b \times (b \times (b \times (b^3)^2)^2)^2$
3	$b \times (b \times (b \times (b \times b^2)^2)^2)^2$
2	$b \times (b \times (b \times (b \times (b^1)^2)^2)^2)^2$
1	$b \times (b \times (b \times (b \times (b \times b^0)^2)^2)^2)^2$
0	$b \times (b \times (b \times (b \times (b \times 1)^2)^2)^2)^2$

9 multiplications

about $2\log_2(n)$ multiplications in the worst case

exact # of multiplications: $2\lfloor \log_2(n) \rfloor + 1$

$O(\log n)$ time complexity

Prime Test 1

How to determine if n is prime? Simple approach: check all numbers 2, 3, 4, $\ldots$, $n-1$ to see if any of them is a factor of n (that is, a number that divides into n evenly, with no remainder).

```java
public static boolean primeTest1(long n) {
    if (n < 2) return false;
    for (long i = 2; i < n; i++) {
        if (n % i == 0) return false;
    }
    return true;
}
```

Worst case, when n is prime: $n-2$ loop cycles

$O(n)$ time complexity

Prime Test 2

No factors greater than $\frac{n}{2}$ can exist, so we only need to check up to $\frac{n}{2}$.

$$\begin{aligned}\text{Example: } 24 &= 2 \times 12\\ &= 3 \times 8\\ &= 4 \times 6\\ &= 6 \times 4\\ &= 8 \times 3\\ &= 12 \times 2\end{aligned}$$

```java
public static boolean primeTest2(long n) {
    if (n < 2) return false;
    for (long i = 2; i <= n / 2; i++) {
        if (n % i == 0) return false;
    }
    return true;
}
```

Worst case, when n is prime: $\frac{n}{2} - 1$ loop cycles

$O(n)$ time complexity

Prime Test 3

We really only need to check up to $\sqrt{n}$ because of symmetry.

$$\begin{aligned} \text{Example: } 36 &= 2 \times 18 \\ &= 3 \times 12 \\ &= 4 \times 9 \\ &= 6 \times 6 \\ &= 9 \times 4 && \text{redundant} \\ &= 12 \times 3 && \text{redundant} \\ &= 18 \times 2 && \text{redundant} \end{aligned}$$

$$\text{Example: } 49 = 7 \times 7$$

```java
public static boolean primeTest3(long n) {
    if (n < 2) return false;
    long squareRoot = (long) Math.sqrt(n); // (long) truncates fractional part
    for (long i = 2; i <= squareRoot; i++) {
        if (n % i == 0) return false;
    }
    return true;
}
```

Worst case, when n is prime: $\sqrt{n} - 1$ loop cycles

$O(\sqrt{n})$ time complexity

Prime Test 4

We also don't need to check even numbers greater than 2.

```java
public static boolean primeTest4(long n) {
    if (n < 2) return false;
    if (n == 2) return true;
    if (n % 2 == 0) return false;
    long squareRoot = (long) Math.sqrt(n); // (long) truncates fractional part
    for (long i = 3; i <= squareRoot; i += 2) {
        if (n % i == 0) return false;
    }
    return true;
}
```

`for (long i = 1; i <= squareRoot; i++)`	$\sqrt{n}$ cycles
`for (long i = 1; i <= squareRoot; i += 2)`	$\frac{1}{2}\sqrt{n}$ cycles
`for (long i = 3; i <= squareRoot; i += 2)`	$\frac{1}{2}\sqrt{n} - 1$ cycles

Worst case, when n is prime: $\frac{1}{2}\sqrt{n} - 1$ loop cycles

$O(\sqrt{n})$ time complexity

Prime Test 5

We really only need to check *primes* up to $\sqrt{n}$.

```
public static boolean primeTest5(long n) {
    if (n < 2) return false;
    if (n == 2) return true;
    if (n % 2 == 0) return false;
    long squareRoot = (long) Math.sqrt(n); // (long) truncates fractional part
    for (Long i : primeList) {
        if (i > squareRoot) return true; // only check up to sqrt(n)
        if (n % i == 0) return false;
    }
    return true;
}
```

Prime Number Theorem: the number of primes $\leq x$ is roughly $\dfrac{x}{\log x}$

$\left(\text{Note: } \dfrac{x}{\log(x) - 1} \text{ is actually a better approximation.}\right)$

Worst case, when n is prime: there are about $\dfrac{\sqrt{n}}{\log \sqrt{n}}$ primes to check

$= \dfrac{\sqrt{n}}{\log(n^{\frac{1}{2}})} = \dfrac{\sqrt{n}}{\frac{1}{2}\log n} = \dfrac{2\sqrt{n}}{\log n}$ loop cycles

$O(\dfrac{\sqrt{n}}{\log n})$ time complexity

Fast Exponentiation Algorithm, Modulo M

$2^{1000} = 10715086071862673209484250490600018105614.....$*(250 digits omitted)*$.....05668069376$

$2^{1000} \bmod 10 = 6$

$2 \times 2 \bmod 10 = 4$
$4 \times 2 \bmod 10 = 8$
$8 \times 2 \bmod 10 = 6$
$6 \times 2 \bmod 10 = 2$
$2 \times 2 \bmod 10 = 4$
$\ldots$
$8 \times 2 \bmod 10 = 6$

```
public static long fastpowerModulo(long b, long n, long M) {
    if (n == 0) {
        return 1;
    } else if (isEven(n)) {
        return square(fastpowerModulo(b, n / 2, M)) % M;
    } else {
        return (b * fastpowerModulo(b, n - 1, M)) % M;
    }
}
```

Takes about $2 \log_2 n$ steps in the worst case.

Prime Test 6: the Fermat Test

Pierre de Fermat's "Little Theorem" (17th century): If N is a prime number, then the relation

$$a^N \bmod N = a$$

holds for *all* numbers from 1 to $N-1$, inclusive. On the other hand, if N is not prime, then *usually most* numbers from 1 to $N-1$ will not satisfy this relation.

Idea: pick a number from 1 to $N-1$ at random and see if the relation holds. If it fails, we know N isn't prime. If it passes, N is *probably* prime, but try a few more spot checks to be more certain.

```
public static boolean primeTest6(long n) {
    int TRIALS = 10;
    if (n < 2) return false;
    for (int i = 1; i <= TRIALS; i++) {
        long a = pickRandom(1, n-1);
        if (fastpowerModulo(a, n, n) != a) return false;
    }
    return true;
}
```

Worst case running time: $10 \times (2 \log n) = 20 \log n$

$O(\log n)$ time complexity, which is *much less* than $O(\sqrt{n})$

Carmichael numbers fool the Fermat test: *all* of the numbers from 1 to $N-1$ pass the $a^N \bmod N$ test, but N is still not prime!

Carmichael numbers are very rare: only 255 of them below 100,000,000

The first few are: 561, 1105, 1729, 2465, 2821, 6601, 8911

> *In testing primality of very large numbers chosen at random, the chance of stumbling upon a value that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a "correct" algorithm. Considering an algorithm to be inadequate for the first reason but not for the second illustrates the difference between mathematics and engineering.*
>
> —Hal Abelson and Gerry Sussman

More sophisticated versions of the Fermat test exist that cannot be fooled (*e.g.*, Miller-Rabin test).

Summary of worst-case running times

Prime test	Running time (loop cycles)	Time complexity
1	$n-2$	$O(n)$
2	$n/2-1$	$O(n)$
3	$\sqrt{n}-1$	$O(\sqrt{n})$
4	$\sqrt{n}/2-1$	$O(\sqrt{n})$
5	$2\sqrt{n}/\log(n)$	$O(\sqrt{n}/\log n)$
6	$20\log(n)$	$O(\log n)$