Chapter 3

# Analysis of Algorithms

**Estimating the efficiency of a method (so we can evaluate a method, or compare two methods).**

**Independent of computer used; also independent of Java restrictions such as maximum int value (approximately 2 billion).**

**Execution-Time Requirements**

**≈ Number of statements executed in a trace of the method, given as a function of *n*, the problem size.**

**For example, you might read in an integer and generate that many prime numbers.**

**Then *n* would represent the value read in.**

**You will either be given n explicitly, or n will be clear from the context.**

**Given a problem of size *n*, a method's *worstTime(n)* is the maximum number of statements executed in a trace of the method.**

**Example: Assume** a [0 … n –1] **OF int.**

```
for (int i = 0; i < n - 1; i++)
    if (a [i] > a [i + 1])
        System.out.println (i);
```

**What is worstTime(n)?**

**The worstTime(*n*) IS 1 + n + (n – 1) + (n – 1) + (n – 1)**

**= 4n - 2**

**Similarly, a method's *averageTime(n)* is the average number of statements executed in a trace of the method.**

**Example:**

```
for (int i = 0; i < n - 1; i++)
    if (a [i] > a [i + 1])
        System.out.println (i);
```

**What is averageTime(n)?**

**The averageTime(n) iss 3.5n – 1.5.**

**Maximum and average are over all possible traces of the method, for all possible field, parameter, and input values.**

**We want an upper bound estimate of worstTime(*n*) and averageTime(*n*) to get an idea of how bad the time can be.**

**Definition of Big-O:**

**Let $g$ be a function that has non-negative integer arguments and returns a non-negative value for all arguments.**

**We define $O(g)$, the order of g,**

**To be the set of functions $f$ such that for some pair of non-negative constants $C$ and $K$,**

$$f(n) <= C\ g(n) \quad \text{for all } n >= K$$

**We say that $f$ is O($g$).**

**"$f$ is O of $g$"**

**If f is O(g), f is eventually less than or equal to some constant times g. So g can be viewed as an upper bound for f.**

**Notation: Suppose g is such that**

**g($n$) = $n^2$, for $n$ = 0, 1, 2, …**

**we write O($n^2$) instead of O(g).**

**Example:**

**f($n$) = ($n^2$ + 3)($n$ – 5) + 20, for $n$ = 0, 1, 2, …**

**show that f is O($n^3$).**

**f($n$) = $n^3 - 5n^2 + 3n + 5$**

**Basic idea: Show that each term is <= some constant times $n^3$**

**$n^3 \le 1\ n^3$, for all $n \ge 0$**

**$-5\ n^2 \le 5\ n^3$ for all $n \ge 0$**

**$3n \le 3\ n^3$ for all $n \ge 0$**

**$5 \le 5\ n^3$ for all $n \ge 1$**

**Adding up the left-hand sides and the right-hand sides:**

$n^3 - 5\,n^2 + 3n + 5 <= 14\,n^3$ **for all** $n >= 1$

**In other words, for** $C = 14$ **and** $K = 1$**,**

**f(**$n$**) <=** $C\,n^3$**, for all** $n >= K$

**that is, f is O(**$n^3$**).**

**The above f is also O(**$n^4$**), O(**$n^5$**), …**

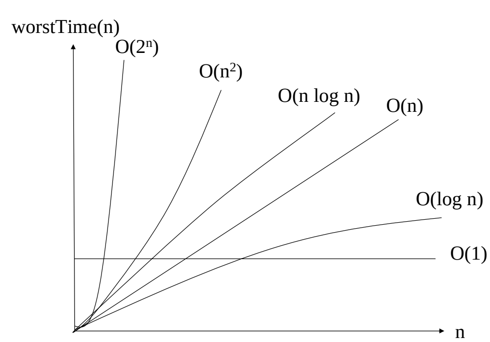
**Note that**

$O(n^3) = O(n^3 - 5) = O(4n^3 + 3) \subset O(n^4)$

**Often, the upper bounds will be from the following sequence of orders:**

**O(1), O(log** $n$**), O(**$n$**), O(**$n$ **log** $n$**), O(**$n^2$**)**

## Growth Rates



**In the following examples, determine an upper bound, in Big-O notation, worstTime(*n*).**

**Example 1.**

```
for (int j = 0; j < 10000; j++)
    System.out.println (j);
```

**worstTime(*n*) is O(1)**

**Because the number of loop iterations is independent of any *n*.**

**Example 2.**

```
for (int j = 0; j < n; j++)
        System.out.println (j);
```

**The number of statements executed is 3*n* + 2, so worstTime(*n*) IS O(*n*).**

**In fact, we could arrive at the O(*n*) estimate without counting the number of statements executed. Because O(3*n* + 2) = O(7*n* – 4) = O(12n + 83) = O(*n*), all we need to count is the number of loop iterations!**

**Example 3:**

```
for (int j = 8; j < n – 3; j++)
{
    System.out.println (j * j);
    if (n / 2 > j)
        System.out.println (j * n);
} // for
```

**The number of statements executed is … who cares?**

**The number of loop iterations is n – 11, so worstTime($n$) is O($n$).**

**And the constant 11 is disregarded in the Big-O estimate, so all that matters is O(number of loop iterations).**

**Example 4.**

```
for (int j = 0; j < n; j++)
   for (int k = 0; k < n; k++)
        System.out.println (j + " " + k);
```

**Hint: Calculate**

**O(number of inner-loop iterations).**

**# of inner-loop iterations = $n^2$**

**so worstTime($n$) is O($n^2$).**

**Example 5:**

```
while (n > 1)
    n = n / 2;
```

**Starting at $n$, how many times can I divide by 2 until $n$ = 1?**

**Simple case: $n$ a power of 2**

**For example, $n$ = 32**

**32 / 2 / 2 / 2 / 2 / 2 = 1**

**So when $n$ = 32, the number of times to divide $n$ by 2 (to get to 1) is 5.**

**$\log_2 32 = 5$**

**If $n$ is a power of 2, the number of times to divide $n$ by 2 until $n$ = 1 is**

**$\log_2 n$**

**If $n$ is not necessarily a power of 2, some divisions, such as 17 / 2, will reduce $n$ by slightly more than half, so the number of halvings will be slightly less than $\log_2 n$.**

**Specifically, for any positive integer $n$, the number of divisions by 2 to get from $n$ to 1 is floor($\log_2 n$) – see example A2.2.**

**Where floor($x$) returns the largest integer <= $x$.**

**1000**
**500**
**250**
**125**
**62**
**31**
**15**
**7**
**3**
**1**

**There are 9 divisions required; floor($\log_2 1000$) = 9**

**The number of iterations of**

```
while (n > 1)
    n = n / 2;
```

**is floor($\log_2 n$).**

```
while (n > 1)
    n = n / 2;
```

**The worstTime($n$) is O(log $n$), and This is the smallest upper bound.**

**The Splitting Rule:**

**The number of halvings to get from $n$ to 1 is floor($\log_2 n$).**

**The Splitting Rule is the basis for most estimates that are O(log $n$).**

**By the base-conversion formula in Appendix 2,**

**O($\log_2 n$) = O(ln $n$) = O ($\log_3$n) = O($\log_{10} n$) = …**

**Example 6.**

```
while (n > 1)
    n = n / 3;
```

The worstTime($n$) is floor($\log_3 n$).

so worstTime($n$) is O(log $n$).

**Exercise: Determine a Big-O estimate of worstTime(n) for the following fragment:**

```
while (n > 5)
{
    System.out.println (n);
    n = n / 12;
} // while
```

**Example 7.**

```
for (int j = 0; j < n; j++)
    System.out.println (j * j);
while (n > 1)
    n /= 2;   // same as n = n / 2;
```

**The worstTime($n$) is O($n$).**

**In general, if worstTime($n$) is O(g) for one part of a method, and O(h) for the rest of the method, worstTime($n$) is O(g + h) for the entire method.**

**Note that O(n + log n) = O(n).**

**Example 8.**

```
for (int j = 0; j < n; j++)
{
    int temp = n;
    while (temp > 1)
        temp = temp / 2;
} // for
```

**The worstTime($n$) is O($n$ log $n$).**

**Example 9.**

```
for (int i = 0; i * i < n; i++)
    System.out.println (i);
```

**The worstTime($n$) is O($n^{1/2}$).**

**Exercise: Provide a Big-O estimate of worstTime(n):**

a.
```
for (int i = 0; Math.sqrt (i) < n; i++)
    System.out.println (i);
```

b.
```
for (int i = 0; i < n; i++)
    System.out.println (i);
while (n > 0)
{
    n /= 2;
    System.out.println (n);
} // while
```

c.
```
int k = 1;

for (int i = 0; i < n; i++)
    k = k * 2;
for (int j = 0; j < k; j++)
    System.out.println (j);
```

**Big-O notation provides an upper bound for a function. Sometimes, as in Chapter 11, we will be interested in a lower bound. Big-Omega notation provides a lower bound.**

**Let *g* be a function that has non-negative integer arguments and returns a non-negative value for all arguments. We define $\Omega(g)$ to be the set of functions *f* such that for some pair of non-negative constants *C* and *K*,**

$$f(n) >= C\,g(n) \quad \text{for all } n >= K.$$

```
for (int j = 0; j < n; j++)
    System.out.println (j);
```

**worstTime(*n*) is O(n), O(n log n), O($n^2$), …**

**worstTime(*n*) is $\Omega$(n), $\Omega$(log n), $\Omega$(1)**

**Big-Theta provides both a lower and an upper bound.**

**We say that f is $\Theta$(g) if f is O(g) and f is $\Omega$(g).**

```
for (int j = 0; j < n; j++)
        System.out.println (j);
```

**worstTime(n) is $\Theta$(n).**

**If worstTime(*n*) is $\Theta$ (1), we will say**

**"worstTime(*n*) is constant."**

**If worstTime(*n*) is _____, we will say "worstTime(*n*) is ______."**

**$\Theta$(1) … constant**

**$\Theta$(log *n*) … logarithmic in *n***<br>
**$\Theta$(*n*) … linear in *n***<br>
**$\Theta$(*n* log *n*) … linear logarithmic in *n***<br>
**$\Theta(n^2)$ … quadratic in *n***

An *Exponential-Time* method is one whose worstTime(n) is $\Omega(x^n)$ for some real number x > 1.0.

We then say that worstTime(*n*) is exponential in *n*.

**For example,**

```
int k = 1;

for (int i = 0; i < n; i++)
    k = k * 2;
for (int j = 0; j < k; j++)
    System.out.println (j);
```

**Then k = $2^n$, so worstTime(n) is exponential in n.**

**Sometimes we provide Big-O but not Big-Theta (or plain English). For example,**

```
/**
 * The specified array a has been sorted into ascending
 * order. The worstTime(n) is O(n * n) and
 * averageTime(n) is O(n log n).
 *
 * @param a – the array to be sorted.
 *
 */
public static sort (int[ ] a)
```

**An alternate implementation might do better: worstTime(*n*) might be O(*n* log *n*).**

**The original implementation is very fast, on average, in execution speed.**

**Method-Estimate Conventions:**

1. **If the calling object is a collection of elements, *n* = number of elements in the collection.**
2. **If no estimate of worstTime(*n*) given, worstTime(*n*) is constant.**
3. **If no estimate of averageTime(*n*) given, O(averageTime(*n*)) = O(worstTime(*n*)).**

## Run-Time Analysis

**To estimate a method's run time, the System class has**

```
/**
 *  Returns the number of milliseconds from
 *  January 1, 1970 till now.
 *
 *  @return the number of milliseconds from
 *  January 1, 1970 till now.
 *
 */
public static long currentTimeMillis( )
```

**Here is the skeleton of a timing program:**

```
long startTime,
       finishTime,
       elapsedTime;

startTime = System.currentTimeMillis( );

// Perform the task:
…


// Calculate the elapsed time:
finishTime = System.currentTimeMillis( );
elapsedTime = finishTime - startTime;
```

**In multiprogramming environments, such as Windows, elapsed time is a very crude estimate of run time.**

**To see the current processes in Windows, CTRL-ALT-DEL.**

# Randomness

**Given a collection of numbers, a number is selected *randomly* if each number has an equal chance of being selected. A number so selected is called a *random number*.**

**The method** nextInt (**int** n) **in the** Random **class returns a "Random"** **int in the range from 0 to n – 1.**

**The value returned is not rally random: If you look at the method definition, you can calculate the return value.**

**The value calculated by** nextInt **depends on the seed.** seed **is a long variable in the** Random **class**

```
Random random = new Random (100);
        // initializes seed to 100

Random random = new Random( );
        // initializes seed to System.currentTimeMillis( )
```

**The current value of** seed **determines the next value of** seed**, and this is used in calculating the value returned by** nextInt (**int** n).

```
Random r = new Random (100);
for (int i = 0; i < 10; i++)
   System.out.print (r.nextInt (4) + " ");
```

**Each time this segment is run in a particular computing environment, the output will be the same, for example:**

**2 2 0 2 2 0 3 1 2 2**

**Why would we want the same sequence every time? We can compare different methods with the same sequence of random values.**

**In general, repeatability is a hallmark of the scientific method.**

**Exercise: Write the code to print out how long it takes to generate the random integer 11111 if the initial seed is 100.**

**Hint: while** (…);

**Recall the timer skeleton:**

```
long startTime,
       finishTime,
       elapsedTime;

startTime = System.currentTimeMillis( );

// Perform the task:
…


// Calculate the elapsed time:
finishTime = System.currentTimeMillis( );
elapsedTime = finishTime - startTime;
```