Appendix:

The Cost of Computing: Big-O and Recursion

Big-O Notation

In the previous section we have seen a chart comparing execution times for various sorting algorithms. The chart, however, did not show any absolute numbers to tell us the actual time it took, say, to sort an array of 10,000 double numbers using the insertion sort algorithm. That is because the actual time depends, among other things, on the processor of the machine the program is running on, not only on the algorithm used. For example, the worst possible sorting algorithm executing on a Pentium II 450 MHz chip may finish a *lot* quicker in some situations than the best possible sorting algorithm running on an old Intel 386 chip. Even when two algorithms execute on the same machine, other processes that are running on the system can considerably skew the results one way or another.

Therefore, execution time is not really an adequate measurement of the "goodness" of an algorithm. We need a more mathematical way to determine how good an algorithm is, and which of two algorithms is better. In this section, we will introduce the "big-O" notation, which does provide just that: a mathematical description that allows you to rate an algorithm.

The Math behind Big-O

There is a little background mathematics required to work with this concept, but it is not much, and you should quickly understand the major concepts involved. The most important concept you may need to review is that of the limit of a function.

Definition: Limit of a Function

We say that a function `f` *converges to a limit* `L` *as* `x` *approaches a number* `c` *if* `f(x)` *gets closer and closer to* `L` *if* `x` *gets closer and closer to* `c`*, and we use the notation* $\lim_{x \to c} f(x) = L$

Similarly, a function `f` *converges to a limit* `L` *as* `x` *approaches positive (negative) infinity if* `f(x)` *gets closer and closer to* `L` *as* `x` *is positive (negative) and gets bigger (smaller) and bigger (smaller). We use the notation* $\lim_{x \to \infty} f(x) = L$ *(* $\lim_{x \to -\infty} f(x) = L$ *).*

Note that functions need not have a limit (in which case we say that the limit as `x` *approaches* `c` *does not exist).*

Please review this concept of a limit from your Calculus math texts, where you should also find plenty of examples. We will apply this concept to define "big-O" notation:

Definition: Big-O Notation and Order of Growth

If f and g are two functions, then we say that "g is big-O of f", or g = O(f), or "g is of the order of f", if:

- $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$

where c is a number different from zero. Note that if g = O(f) then f = O(g), i.e. the big-O notation works both ways.
We also say that if a function f is big-O of g, or f = O(g), then f and g have the same growth rate, i.e. they increase at comparable speeds for larger and larger input values.

This definition may not look like it means anything in "real" terms, but after working with it a little, and forgetting any "math phobia" that may still linger, the meaning of this definition will become clear. Here are a few examples to give you a taste:

Example:

Consider the following functions, and determine which one is big-O of which other one.

a) $f(x) = x^2 + 2x + 7$

b) $g(x) = \log(x)$

c) $h(x) = x - 7x^2$

d) $k(x) = xe^x$

e) $l(x) = \log(x^2)$

f) $m(x) = e^x$

g) $n(x) = x(x^3 - x^2 - x)$

h) $p(x) = e^x + x^2$

i) $q(x) = 3xe^x + x^2 \log(x)$

j) $r(x) = 3x^4 + 4x^3$

Before we start working out all kinds of limits, let's group these functions according to categories:

Polynomials:

$f(x) = x^2 + 2x + 7$, $h(x) = x - 7x^2$, $n(x) = x(x^3 - x^2 - x)$, $r(x) = 3x^4 + 4x^3$

Exponentials:

$k(x) = xe^x$, $m(x) = e^x$, $p(x) = e^x + x^2$, $q(x) = 3xe^x + x^2 \log(x)$

Logarithms:

$g(x) = \log(x)$, $l(x) = \log(x^2)$, $q(x) = 3xe^x + x^2 \log(x)$

We have listed q twice, because it's not clear into which category it falls. Next, let's use some common sense and some background math knowledge:

The definition of big-O involves growth rates for large values. What we - hopefully - recall from calculus is:

- For polynomials, the degree, or highest power, determines how fast they grow. The higher the degree, the faster they grow
- Exponential functions grow faster than any polynomial or fractional power (i.e. roots)
- Logarithm functions grow slower than any polynomial or fractional power (i.e. roots)

Armed with this background knowledge, we make the following conjectures:

- $f(x) = x^2 + 2x + 7$ is a polynomial of degree 2, and so is $h(x) = x - 7x^2$ - maybe f = O(h)

- $r(x) = 3x^4 + 4x^3$ and $n(x) = x\left(x^3 - x^2 - x\right)$ are polynomials of degree 4 - maybe r = O(n)
- $m(x) = e^x$ is a simple exponential function, and $p(x) = e^x + x^2$ has a simple exponential function which grows much faster than, in particular, the x-square term - maybe m = O(p)
- $k(x) = xe^x$ and $q(x) = 3xe^x + x^2\log(x)$ both involve xe^x, as well as terms that grow so slow that we can maybe neglect them - maybe k = O(q)
- $g(x) = \log(x)$ and $l(x) = \log\left(x^2\right)$ are both logarithm functions, but it is not clear what the square term does - but maybe g = O(l)

Now we should try to prove that these conjectures are correct by trying to find the appropriate limits:

- $\lim\limits_{x\to\infty} \dfrac{x^2 + 2x + 7}{x - 7x^2} = \lim\limits_{x\to\infty} \dfrac{x^2\left(1 + 2/x + 7/x^2\right)}{x^2\left(1/x^2 - 7\right)} = \lim\limits_{x\to\infty} \dfrac{1 + 2/x + 7/x^2}{1/x^2 - 7} = \dfrac{1+0+0}{0-7} = -\dfrac{1}{7}$ which is a non-zero number, so that indeed f = O(h)
- $\lim\limits_{x\to\infty} \dfrac{3x^4 + 4x^3}{x\left(x^3 - x^2 - x\right)} = \lim\limits_{x\to\infty} \dfrac{3x^4 + 4x^3}{x^4 - x^3 - x^2} = \lim\limits_{x\to\infty} \dfrac{x^4(3 + 4/x)}{x^4\left(1 - 1/x - 1/x^2\right)} = \dfrac{3+0}{1-0-0} = 3$ which is a non-zero number, so that indeed r = O(n)
- $\lim\limits_{x\to\infty} \dfrac{e^x}{e^x + x^2} = \lim\limits_{x\to\infty} \dfrac{e^x}{e^x\left(1 + x^2/e^x\right)} = \lim\limits_{x\to\infty} \dfrac{1}{1 + x^2/e^x} = \dfrac{1}{1+0} = 1$ because as x gets larger and larger, x^2/e^x gets smaller and smaller because the exponential function grows faster than the square function. Hence, p = O(m) or, which is the same, m = O(p)
- $\lim\limits_{x\to\infty} \dfrac{xe^x}{3xe^x + x^2\log(x)} = \lim\limits_{x\to\infty} \dfrac{xe^x}{xe^x\left(3 + \dfrac{x^2\log(x)}{xe^x}\right)} = \lim\limits_{x\to\infty} \dfrac{1}{3+0} = \dfrac{1}{3}$, so indeed k = O(q)
- $\lim\limits_{x\to\infty} \dfrac{\log(x)}{\log(x^2)} = \lim\limits_{x\to\infty} \dfrac{\log(x)}{2\log(x)} = \dfrac{1}{2}$ because of the basic rules of logarithms, so that indeed g = O(l)

■

You may or may not remember the exact mathematical skills necessary to compute the above limits. As one easy solution, of course, you could use a mathematical algebra package such as Maple or Mathematica to work out these limits. Fortunately, however, the above math is complete overkill for most problems involving big-O to evaluate algorithms. The next example, in fact, is much more appropriate for our future discussions.

Example:

What order of growth do the following mathematical functions have for large values?

a) $f(n) = \dfrac{n(n+1)}{2} + \dfrac{n}{7}$

b) $h(n) = 4\left(2^n\right) + n^2$

c) $k(n) = \log(n) + n$

d) $p(n) = 2\log\left(n^{3/2}\right) + 7$

e) $q(n) = n\log(n) + \log(n^2)$

f) $r(n) = 1 + 2 + 3 + \ldots + (n-1) + n$

Here the key is to understand the question. From now on, when we say "what order of growth" does a function have for large values, we are really asking for the simplest possible function that is big-O of the given one.

For example, the function $f(n) = \frac{n(n+1)}{2} + \frac{n}{7}$ is a polynomial of degree 2. Therefore, it has the same growth order of the function $g(n) = n^2$. Therefore, f = O(g), or, even shorter: f = O(n^2).

Next, the function h(n) is an exponential function (because 2^n is a particular exponential function) and a square function added together. Since an exponential function grows faster than any polynomial, that part of the function wins in the race for infinity. Therefore, h = O(2^n)

For the next function $k(n) = \log(n) + n$ we need to consider a sum of a logarithm and a simple n. Since the logarithm function grows slower than any polynomial, it will grow, in particular, slower than n. Therefore, we can neglect the log(n) term so that k = O(n)

Next, $p(n) = 2\log\left(n^{3/2}\right) + 7$ can be rewritten, using the laws of logarithms, as $p(n) = 2\frac{3}{2}\log(n) + 7 = 3\log(n) + 7$. The constants 3 and 7 do not depend on n, therefore they can be neglected. Hence, p = O(log(n))

Now let's consider $q(n) = n\log(n) + \log(n^2)$. We can again rewrite this function as $q(n) = n\log(n) + \log(n^2) = n\log(n) + 2\log(n) = (n+2)\log(n)$. Just as before, the constant 2 can be neglected, so that our function q = O(n log(n))

Finally, we have our last function $r(n) = 1 + 2 + 3 + \ldots + (n-1) + n$. This function is adding all integer numbers from 1 to n. We have seen in chapter 1 that the sum of the integers from 1 to *n* is equal to $\frac{n(n+1)}{2}$. Hence, we can rewrite our function as $r(n) = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$. But then it is clear that r = O(n^2)

In the table below, we have summarized the growth order of these functions again:

Function	Order of Growth
$f(n) = \frac{n(n+1)}{2} + \frac{n}{7}$	$O\left(n^2\right)$
$h(n) = 4\left(2^n\right) + n^2$	$O\left(2^n\right)$
$k(n) = \log(n) + n$	$O(n)$
$p(n) = 2\log\left(n^{3/2}\right) + 7$	$O(\log(n))$
$q(n) = n\log(n) + \log(n^2)$	$O(n\log(n))$
$r(n) = 1 + 2 + 3 + \ldots + (n-1) + n$	$O\left(n^2\right)$

Table: Selected Functions and their Growth Order

Big-O applied to Java Methods

The above example illustrates the essence of the big-O notation: if you drop everything that is not essential for large input values from a function, you will get the "growth order" of that function. The big-O definition merely clarifies mathematically what can be safely dropped to determine the growth order of a function. The key to quickly determine the order of growth of a function *is to understand which terms* of particular function *can be safely dropped* without an impact on the growth behavior.

Definition: Rule of Thumb to find Big-O Growth Order

For our purposes big-O notation is used to specify the order of growth of a function in terms that are as simple as possible:

- *for a polynomial, only the highest power counts*
- *if an exponential function is involved, all terms that do not include an exponential function can be dropped*
- *logarithm functions grow slower than any power, including fractional powers and roots*
- *terms that are added can be analyzed separately, and the one with the highest growth order counts*

Common growth orders of computer algorithms are, in decreasing order of speed of growth:

- $O(2^n)$, $O(n^3)$, $O(n^2)$, $O(n\log(n))$, $O(n)$ $O(\sqrt{n})$, $O(\log(n))$, $O(1)$

Below are some graphs illustrating these typical growth rate functions. The graphs have been split into two pictures. To understand how they would fit together, the line f(n) = n is displayed in both pictures. On the left, every function grows faster than f(n) = n, on the right every graph grows slower than f(n) = n.

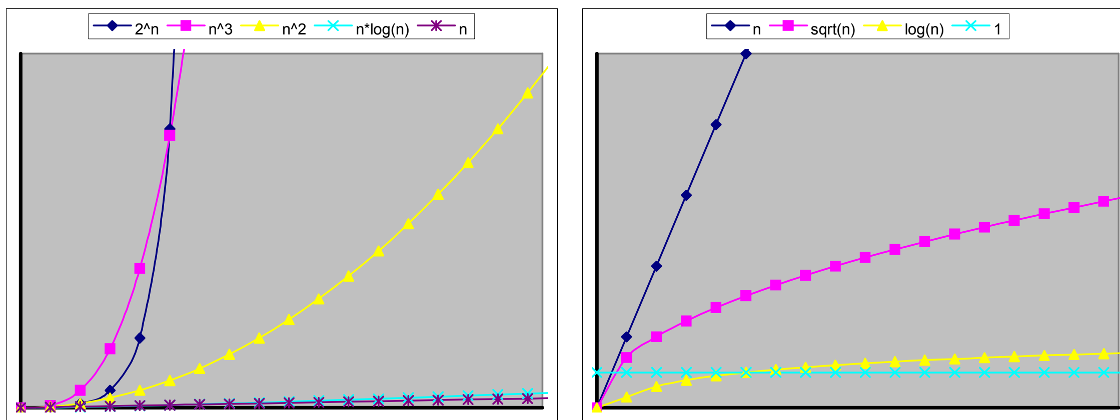


Figure: Typical Growth Rate Functions for Computer Algorithms

Example:

Below are a variety of methods, each using an integer `n` as input. Determine the order of growth for each method, considered as a function of `n`.

```
void method1(int n)
{   for (int i=1; i <= n; i++)
        System.out.println(n);
}

void method3(int n)
```

```
void method2(int n)
{   for (int i = 0; i < 2*n; i++)
        for (int j = 0; j < n; j++)
            System.out.println(i*j);
}
void method4(int n)
```

```
{  if (n > 0)
   {  method3(n-1);
      System.out.println(n);
      method3(n-1);
   }
}
```

```
{  if (n > 0)
   {  System.out.println(n);
      method4(n / 2);
   }
}
```

```
void method5(int n)
{  for (int i = -2; i < n; i++)
      for (int j = 4; j < n-3; j++)
         System.out.println(j+i);
}
```

```
void method6(int n)
{  for (int i = 0; i < 10; i++)
      System.out.println(i*n);
}
```

```
void method7(int n)
{  if (n > 2)
   {  method7(n-2);
      System.out.println(n);
   }
}
```

```
int method8(int n)
{  int sum = 0;
   for (int i = 0; i < n; i++)
      sum += i;
   return sum;
}
```

So, finally we see how this big-O notation is applied to methods in Java as opposed to mathematical functions. We will describe what each method is doing, and depending on that description determine the order of growth for each method. We will deal with these methods in order of difficulty, which is not necessarily the order in which they are presented.

Our first method is `method1`: it simply prints out the numbers from 1 to `n`. Therefore, it will take exactly `n` steps to complete, for whatever value `n` has. Therefore, this method is of order `n`, or O(n).

Next, let's look at `method2`: it consists of two loops, an inner and an outer loop. For every fixed `i` in the outer loop, the inner loop will run from `j = 0` to `j = n-1`, in steps of 1. Therefore, for every fixed `i` there are `n` steps in the inner loop. Since the outer loop runs from `i = 0` to `i = 2n-1`, it will execute `2n` times. Together with the inner loop, the inside call to `System.out.println` will therefore execute `2n * n` times, or $2n^2$ times. That, of course, means that the method is of order n^2, or $O(n^2)$.

We will for now skip `method3` and `method4`. Instead, `method5` again has two loops, an inside and an outside loop, just as `method2`. This time the outer loop executes `(2+n)` times, while the inner loop executes `n-7` times. Therefore, the inner most `System.out.println` statement will execute exactly `(2+n)*(n-7)` times. But that means again that `method5` is of order n^2, or $O(n^2)$.

Next, `method6` is really a trick method. The loop executes exactly 10 times, regardless of what the input value is. Of course, the actual output of the method does depend on `n`, but the number of times the method executes does not. Therefore, the method has no growth, or constant growth 0, and is therefore O(1).

The only remaining non-recursive method is `method8`. But that's actually a method we have seen before: it adds the numbers from 1 to `n`. The answer, as we already know, is $\frac{n(n-1)}{2}$, but the method reaches that answer in `n` steps. Therefore, this method is O(n). Note that the answer itself is $O(n^2)$, but the number of steps to reach that answer is O(n).

Finally, the remaining methods are recursive, and perhaps somewhat harder to explain. We will start with `method7`. That method recursively calls itself with input 2 less than the current value of `n`, then prints out the value of `n`. For example, if the method was called as in `method7(8)`, then it would call `method7(6)`, then `method7(4)`, then `method7(2)`. In other words, if `n = 8` as input, the method would call itself 4 times in total (counting the current version as well). It's easy to see that if `n = 10`, the method would call itself 5 times. Therefore, it seems that for a given `n`, the method calls itself `n/2` times, which means that `method7` is O(n).

That leaves `method3` and `method4` to analyze. Let's take a look at `method4` first: suppose the input value `n` is, say 16. Then the recursive calls would call `method4(8)`, then `method4(4)`, then `method4(2)`, and finally `method4(1)`. Thus, if `n = 16`, the method makes 4 recursive calls to itself. If `n = 64`, say, the method would call itself with arguments of 32, 16, 8, 4, 2, and 1, i.e. for `n = 64` the method calls itself 6 times. Now we see a pattern emerging, at least if the input is a power of 2: if $n = 2^k$, the method calls itself `k` times. Mathematically, how does one transform the equation $n = 2^k$ to get `k`, the number of times the function calls itself? The answer is to take the logarithm function with base 2 on both sides:

$$n = 2^k \Leftrightarrow \log_2(n) = k$$

That means, of course, that our method is O(log(n)). Actually, we have cheated somewhat: we only verified this equation for powers of 2, not for an arbitrary value of `n`. However, for powers of 2 this answer is clear, and we will take this as evidence that the method really is O(log(n)), even for arbitrary values of `n`. We can actually prove this mathematically using a technique called induction, but as promised we will leave "real" mathematics to mathematics courses.

That, finally, leaves `method3`. We will proceed in stages, starting with "easy" cases, until we see a pattern emerging:

- `method3(1)` calls `method3(0)` twice, so if `n = 1`, the method makes two recursive calls
- `method3(2)` calls `method3(1)` twice, each of which calls `method3(0)` twice (as we already figured out); therefore, the method makes 2 * 2 = 4 recursive calls
- `method3(3)` calls `method3(2)` twice. That method, as we already say, makes 4 recursive calls, so that `method3(3)` will make 2*4 = 8 recursive calls.
- `method3(4)` calls `method3(3)` twice. That method, from before, makes 8 recursive calls, so that `method3(4)` will make 2*8 = 16 recursive calls.

Now, however, the pattern should be clear: if the input to `method3` is `n`, the method will make 2^n recursive calls. Therefore, this method is $O(2^n)$. Again, instead of relying on "pattern recognition" one could use the principle of induction to mathematically prove that this is indeed the proper order of growth.

To summarize our answers, here is a table listing the growth rates of the above methods:

Method	Order of Growth
method1: single loop	O(n)
method2: double loop	$O(n^2)$
method3: double recursion	$O(2^n)$
method4: recursion with half the input	O(log(n))
method5: double loop	$O(n^2)$
method6: no dependence on n	O(1)
method7: simple recursion	O(n)
method8: single loop	O(n)

Table: Selected Methods and their Growth Order

Now that we have seen examples of order of growth for various methods, we should investigate what this exactly means to the performance of an algorithm.

Example:

Suppose we have created four methods that are functionally equivalent (i.e. they produce the same results for the same input), but they are using different algorithms to achieve their results. The algorithm for method A is, say, $O(n^2)$, method B is O(n), method C is O(log(n)), and method D is $O(2^n)$. Which method would you pay money for, i.e. which method performs best?

If the task the methods are completing depend on the value of `n`, the last method D will take the most steps. Method A will take the second-most steps, method B the third-most, and method C will take the least amount of steps. For example:

n	method A	method B	method C	method D
10	100	10	1	1024
100	10000	100	2	1.27E+30
1000	1000000	1000	3	1.1E+301
10000	1E+08	10000	4	≈infinity
100000	1E+10	100000	5	≈infinity

This table does not reflect the *actual* number of steps for each task. After all, a method that is, say, O(n) could really take `10 n + 7` step, or `512 n - 99` because all of these are O(n). But, the table does reflect the order of growth if `n` gets larger and larger. And it is clear that method C completely outperforms the other three, and would therefore be considered the best method.

■

Order of Growth for Searching and Sorting

Now we have the necessary background to analyze the search and sort algorithms in terms of their order of growth. That discussion will be more complicated, because the actual performance of these algorithms does not only depend on `n` (which is synonymous to the length of the input array) but also on the specific composition of the input arguments. For example, the algorithm to search an array for a particular value will finish very quickly, regardless of the length of the array, if the value is found in the first step. Therefore, we will work out the order of growth for the algorithms in a "best", "worst", and in case of sorting also an "average" situation.

Example:

What is the order of growth for the simple search and the binary search algorithm as a function of the length of the array to be searched, in a best and worst case situation.

For both algorithms, the best case is to find the value at the first step. This, obviously, does not depend on the length of the array, so for both algorithms the order of growth in the best possible situation is O(1).

The worst case situation is when the value you are searching for is not contained in the array so that the search will return false. The analysis will be different for both situations:

The simple search algorithm looks at every element in the array in a simple loop and compares it to the one to find. Therefore, there are `n` elements to look at if the input array is of length `n`. Hence, the simple search algorithm is O(n) in a worst case situation.

The binary search algorithm is recursive, and is more complicated to figure out. Recall the algorithm searches an array from `start` to `end`:

```
public static boolean binarySearch(double A[], double key,
                int start, int end)
{  int mid = (start + end) / 2;
   if (start > end)
      return false;
   else if (A[mid] == key)
      return true;
   else if (A[mid] < key)
      return binarySearch(A, key, mid + 1, end);
   else
      return binarySearch(A, key, start, mid-1);
}
```

Let's say the input array is of length 64. The algorithm picks the middle element, and continues to search either the left or right half of the array. In either case, the new array that's searched has length 32, approximately. Again, the middle element is looked at, and the array is split in half again, leaving 16 items. In the next step, the number of elements to still consider is cut to 8, then to 4, then to 2, then to 1, and then we finally know that the element we searched for is not available (in the worst case scenario). Therefore, for $n = 64$ it would take approximately 6 steps before we could be sure that the element is not contained in the array. This works, however, for every array whose length is initially a power of 2: if the length of the array $n = 2^k$, then we would need $\log(n) = k$ steps to complete our search. That, of course, means that the binary search algorithm is $O(\log(n))$.

Therefore, the binary search algorithm is *significantly* better than simple search. Of course, it requires that the array be initially sorted. If not, we could apply one of our sort methods to sort it first, then use binary search. That, however, would mean that the order of growth would be determined by the order of growth of sorting, not of searching. There are many situations, however, where arrays are already sorted, or you can simply make sure that new items are always inserted into an array so that the array remains sorted at all times. Then binary search clearly outperforms simple search.

■

Next, let's try to analyze our sorting algorithms. We have already determined that Quick Sort performs very well for "average" arrays so now we want to compare these algorithms objectively via our big-O notation.

Example:

What is the order of growth of the selection sort algorithm as a function of the length of the array to be sorted, in a best, worst, and average situation.

First, recall the methods making up that algorithm:

```
public static int selectMin(double A[], int start)
{  int smallest = start;
   for (int i = start+1; i < A.length; i++)
      if (A[i] < A[smallest])
         smallest = i;
   return smallest;
}

public static void sortSelect(double A[])
{  for (int i = 0; i < A.length-1; i++)
      swap(A, i, selectMin(A, i));
}
```

Actually, selection sort also involves the `swap` method. However, that method does not depend on the length of the array, because it always swaps exactly two elements. Therefore, swap is O(1) and that can be neglected as long as the remaining methods are of higher order of growth.

The `sortSelect` method contains a loop that goes from 0 to `n-2`, where `n` is the length of the array. For each value `i` in that loop `selectMin` is called, which in turn contains a loop that goes from `i+1` to `n-1`. Therefore:

Outer loop	Inner loop	Count
0	1, 2, 3, …, n-1	n-1 times
1	2, 3, …, n-1	n-2 times
2	3, 4, …, n-1	n-3 times
3	4, 5, …, n-1	n-4 times
…	…	…
n-3	n-2, n-1	2 times
n-2	n-1	1 times

Table: Counting the number of times the selection sort executes

Therefore, the total number of times the inside code in the `selectMin` method executes is the sum of the counts in the third column:

$$1 + 2 + 3 + \ldots + (n-3) + (n-2) + (n-1) = \frac{(n-1)n}{2}$$

Therefore, the algorithm is $O(n^2)$. Note that we wanted to analyze the algorithm in a best, worst, and average situation. However, this algorithm always performs the same, regardless of how the numbers in the array are distributed originally. Therefore, in all cases this algorithm is $O(n^2)$. Also note that we have actually computed the exact number of times the code inside the `selectMin` method executes. To find the order of growth, we could have estimated everything just as well. A simple analysis, for example, simply notes that the selection sort is based on a double-loop, and both loops depend, more or less, on the length of the array. Hence, that algorithm must be $O(n^2)$.

■

Example:

What is the order of growth of the bubble sort algorithm as a function of the length of the array to be sorted, in a best, worst, and average situation.

Again, let's review the methods (aside from swap which is O(1) and can be ignored) that make up this algorithm:

```
public static boolean bubble(double A[], int stop)
{  boolean sorted = true;
   for (int i = 0; i < stop; i++)
   {  if (A[i+1] < A[i])
      {   swap(A, i, i+1);
          sorted = false;
      }
   }
   return sorted;
}

public static void sortBubble(double A[])
{
```

```
        int stop = A.length-1;
        boolean sorted = false;
        while ( (stop >= 0) && (!sorted))
        {  sorted = bubble(A, stop);
           stop--;
        }
     }
```

The first method, `bubble`, consists of a loop from 0 to stop-1, hence it executes `stop` number of times, regardless of the composition of the array. But `sortBubble` depends not only on the length of `A` but also on the return value returned by `bubble`. In particular, if `bubble` returns true, `sortBubble` is done. This calls for a best/worst/average analysis:

The best case, in terms of performance, is if `bubble` returns `true` on the first call. That would mean that bubble executes approximately `n` times, and `sortBubble` would finish right away, making the algorithm O(n) in this case. And bubble returns `true` if the array is already sorted completely in ascending order. Thus, if `A` is already sorted, then bubble sort is O(n).

The worst case is when `bubble` never returns `true`. That means that the full loop of `sortBubble` will execute, namely `n` times. Inside, each call to `bubble` will produce another loop executing `stop` number of times. In total, that would make - well, at this point it should be clear that we can simply say the algorithm is $O(n^2)$ in this case. The actual number of times is the same as for the selection sort, but the nice thing about big-O notation is that this level of details is not necessary at all. All we care for is the order of growth, and that does not require us to figure out the exact number of times an algorithm executes.

For an average situation, it turns out that bubble sort is also $O(n^2)$. The exact details of such a computation would require some probability theory, so we will not go into detail.

■

So, according to these two examples, selection sort and bubble sort should perform comparably, except for arrays that are already sorted where bubble sort is O(n) and thus a lot faster in this situation. But, bubble sort achieves this slight advantage by moving a lot more elements than selection sort in every pass. Since moving elements can be relatively slow, the bubble sort algorithm is in fact slower than selection sort in almost every situation. Therefore, aside from a great name, this algorithm really is not a good one to use.

Example:

What is the order of growth of the insertion sort algorithm as a function of the length of the array to be sorted, in a best, worst, and average situation.

This time we leave the analysis of the best and worst case as an exercise. It is very similar to the previous example, and the answer is, actually disappointing.

For average arrays, this algorithm is again $O(n^2)$. However, the algorithm does outperform the selection and bubble sort algorithms virtually every time because it does not "indiscriminately" move elements, but instead carefully moves specific elements long distances and only moves those elements out of the way that really are out of place. Thus, it really is the best of the sorting algorithms aside from quick sort. But since all three algorithms analyzed so far are $O(n^2)$, insertion sort is not *substantially*.

■

We will not analyze the quick sort algorithm in detail since it is a little more complicated than the previous ones. But for completeness, here are the results anyway.

Example:

What is the order of growth of the quick sort algorithm as a function of the length of the array to be sorted, in a best, worst, and average situation.

Recall the code of the `quickArrange` method (which is the essential part of the algorithm):

```
public static void quickArrange(double A[], int left, int right)
{  if (left < right)
   {  int j = left+1;
      int k = right-1;
      if (A[left] > A[right])
         swap(A, left, right);
      while (j <= k)
      {  while (A[j] < A[left])
            j++;
         while (A[k] > A[left])
            k--;
         if (j < k)
            swap(A, j, k);
      }
      swap(A, left, k);
      quickArrange(A, left, k-1);
      quickArrange(A, k+1, right);
   }
}
```

We should at least try to determine what the best and worst case situation is. The algorithm is different from the others because it is recursive and it partitions the original array into two pieces that can be rearranged further. Therefore, the best situation will be if the array splits approximately in half at each step, while the worst case situation will be if the array does not really split at all, i.e. one of the two pieces has almost the same length as the original array. The splitting up of the array is somewhat reminiscent of the binary search algorithm, which leads us to believe that splitting the array into two equal pieces should be the best possible situation.

The array is split at `k`, so the worst situation will be if `k` is close to either `left` or `right`. Suppose, for example, the array is originally sorted in ascending order. Then certainly `A[left] <= A[right]`, and `A[j]` is never bigger than `A[left]` if `j > left`. Hence, `j = left + 1`. On the other hand, all `A[k] > A[left]`,as long as `k > left`, so that `k` will scan down all the way to `A[left]` before the loop is over. That, in fact, takes approximately `(right - left)` times. Then the recursive calls are made, but the first one will exit right away, because `left > k-1`. The second call, on the other hand, will execute with `left+1` and right as input. Hence, in effect this will mean, eventually, about `n` loops altogether. But then we have:

- If the original array is already sorted in ascending order, it is the worst possible situation for our quick sort algorithm. In fact, this "worst" case situation is $O(n^2)$.

Because the algorithm is recursive, this order of growth presents a particular problem Aside from just taking longer, this algorithm, being recursive, also requires a *lot* of memory in the worst, or $O(n^2)$, case. In fact, on my particular computer I could use selection, insertion, and bubble sort just fine for arrays of size 20,000, but the quick sort algorithm crashed the JVM for arrays of that size

(only in the worst case situation of initially sorted arrays - in the average situation, quick sort outperformed the others nicely).

In the best and average possible situation it turns out that quick sort is O(n log(n)). That is a significant speed improvement over the other algorithms which are $O(n^2)$. To see the difference, consider sorting an array of 100 elements and another one with 10,000 elements:

N	O(n log(n))	$O(n^2)$
100	460	10,000
10,000	92,103	100,000,000

Table: Comparing execution steps between O(n2) and O(n log(n)) algorithms

In other words, sorting an array of 100 elements takes little time with either algorithm, because today's computers can perform 10,000 operations so quickly that it is hardly noticeable. But when the length of the array increases to, say, 10,000, the number of steps for the $O(n^2)$ algorithm increases to 100 million steps - even modern computers will take time to process that many steps. The O(n log(n)) algorithm, on the other hand, increases to 90,000 steps. That number of steps, again, executes very quickly, so the advantage of this algorithm will become more pronounced for large array sizes.

■

To conclude this chapter, we summarize the previous discussion about performance of sorting algorithms in one convenient location.

Definition: Summary of Performance of Sort Algorithms

We have introduced four sort algorithms, and the table below summarizes their order of growth:

	Average Case	Best Case	Worst Case
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n^2)$	O(n)	$O(n^2)$
Insertion Sort	$O(n^2)$	O(n)	$O(n^2)$
Quick Sort	O(n log(n))	O(n log(n))	$O(n^2)$
Java Build-in	O(n log(n))	O(n log(n))	O(n log(n))

In addition:

- *Selection Sort performs lots of comparisons, and few moves. Therefore, it is best suited for data that compares quickly.*
- *Insertion Sort performs lots of moves and fewer comparisons. Therefore, it is best suited for data that can be moved quickly.*
- *Bubble Sort does not have many advantages, and is generally the least desirable algorithm.*
- *Quick Sort is recursive, and general memory considerations associated with recursion apply. It can, however, be improved in a variety of ways to yield the best possible general purpose sorting algorithm known today, including creating a non-recursive version.*

Note: The theoretically best possible sort algorithm can not perform better than O(n log(n)).

There are other sort algorithms we did not discuss, such as Tree Sort, Heap Sort, Radix Sort, and Merge Sort, etc., that may give better performance in particular situations. In general, if

performance and speed is essential, a sorting algorithm should be selected carefully for the particular task at hand.

Finally, all sorting algorithms discussed sort data that is contained entirely in main memory, where it is quickly accessible. An entirely different discussion applies to data that is located on disk and that can not fit completely into memory. That discussion, however, is beyond the scope of this chapter, and in fact this book.

Recursion

We will start our discussion of new ideas with the concept of recursion, a and powerful technique to create programs or classes than can perform a tremendous amount of work while being defined in only a few lines of code.[1]

Recursion, by nature of the word, means that something is occurring over and over again. In particular, recursion in a programming context is the definition of a method that calls itself. Recursion and recursive definitions, in fact, are also a standard staple of mathematics. At this time, however, we will consider recursion only in a programming context and refer the reader to a mathematical text on induction and recursive definitions. Our basic definition of recursion is straightforward:

Definition 7.1.1: Recursion

Recursion occurs when a method is defined in such a way that it calls itself at least once inside its method body. For recursion to be successful, two stages, or cases, are necessary.

- *a "base case", i.e. a statement that does not contain a recursive call to the method itself*
- *one or more "recursive calls", i.e. one or more statements that call the very method in which the current body of code is defined*

For recursion to be valid the parameters in the recursive call must be modified so that they are moving towards the base case of the recursion and eventually the base case must apply.

It might seem strange to have a method call upon itself to define itself. At first glance, it seems that it will result in a circular definition resulting at best an infinite loop. However, when recursion is properly implemented no infinite loop will result and the recursive method seems to perform its task "by magic". An improper recursive method, on the other hand, can very well lead to an infinite loop.

Single Recursion

The first type of recursion we will encounter is simple and is in fact related to single loops.

[1] As in real life, with every good thing there's bad news: the short and concise code that recursion allows you to create comes at the cost of using a lot of system resources. While these resources are allocated entirely automatically, it can frequently happen that a recursive method runs out of resources and will crash (see example 7.1.20)

Definition 7.1.2: Single Recursion

Single recursion refers to recursive methods where the recursive call contains one instance of a call to the method itself.

Example 7.1.3:

Define a static method that has one positive integer input `N` and `void` as return type. The method should be recursive with a base case `N = 1`. The recursive call should be a call to the method itself with the input being one less than the current value of `N`. In either the base case or the recursive call the method should simply print the current value of `N`. Finally, place this method in a standard executable program and call it from the `main` method with input argument `5`.

Following these instructions, we need to define a method that differentiates between two cases, a base case and a recursive call. Therefore, an `if` statement seems appropriate:

```
if (condition for base case)
/* do what is required */
else
/ * implement recursive call as well as what is required */
```

Since `N = 1` determines the base case and since we are supposed to print out the value of `N`, our code looks like:

```
if (N == 1)
   System.out.println(N);
else
{  System.out.println(N);
   /* recursive call */
}
```

To finish the method, we need to give it a proper header so that we know how to call it in the recursive call. The method is supposed to have one integer input argument, so:

```
public static void recursiveExample(int N)[2]
{  if (N == 1)                        // base case
      System.out.println(N);
   else
   {  System.out.println(N);
      recursiveExample(N-1);          // recursive call
   }
}
```

To see what this method will do, we will put it in a complete program and execute it:

```
public class RecursiveTest
{  public static void recursiveExample(int N)
   {  if (N == 1)                        // base case
         System.out.println(N);
      else
      {  System.out.println(N);
         recursiveExample(N-1);          // recursive call
      }
```

[2] The method is marked `static` so that it can be called directly from the `static void main` method without having to instantiate any objects.

```
    }
    public static void main(String args[])
    {  recursiveExample(5); }
}
```

This program, when compiled and executed, will print out the numbers from 5 down to 1, one number per line.

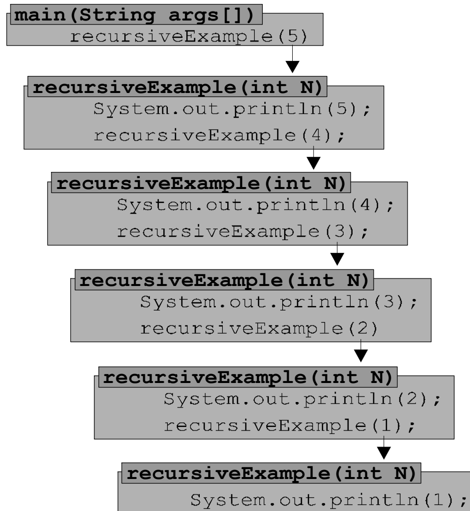


Figure 7.1.1: Resulting calls to `recursiveExample` *for initial input of* `N = 5`

■

The above example was a valid recursion since the argument in the recursive case is moving towards the base case as long as the initial value of `N` is a positive integer[3]. It is easy to create an invalid recursive method:

Example 7.1.4:

Explain why the method `invalidRecursion` defined below constitutes an invalid recursive method if called with positive integer input.

```
public static void invalidRecursion(int N)
{  if (N <= 1)
      System.out.println(N);
   else
      invalidRecursion (N+1);
}
```

This method will compile fine, but when you execute it with a positive integer as input it will result in an infinite loop. The base case of the recursion would be reached if `(N <= 1)`. However, if the method is called with a positive integer input, the recursive call *increases* whatever the input value is and calls itself with that new value. That version of the method, of course, will again increase the input value, call itself, and so on. Therefore, the input parameter `N` never reaches the base case – which would end the recursion – so that the method will forever call itself with higher and higher input values. Of course the method will eventually run out of resources, or the input will go above the largest possible integer, so in practical terms the method *will* end one way or another. It is not, however, a proper recursion.

---

[3] If the input were a negative integer, the recursive case would continue to subtract one from the previous number. The base case would never be reached, hence it would not be a valid recursion in this case.

The output of the numbers 5, 4, 3, 2, 1 in our first example is perhaps not surprising. Before we illustrate in detail what happens during recursion, let's consider another example.

Example 7.1.5:

Create two versions of the recursive method in our first example: the first version should *first* print out the value of `N`, and *then* place the recursive call (as before). The second should *first* place the recursive call, *then* prints the current value of `N`. Execute both methods from a standard `main` method with an input value of `5` and explain any differences that you see.

The code is simple: we just duplicate the above method, but switch the recursive call and the `System.out.println` statements around in the second version of the method.

```
public class RecursiveTest2
{  public static void recursiveExampleA(int N)
   {  if (N == 1)                      // base case
         System.out.println(N);
      else
      {  System.out.println(N);       // first, print
         recursiveExampleA(N-1);      // second, recursive call
      }
   }
   public static void recursiveExampleB(int N)
   {  if (N == 1)                      // base case
         System.out.println(N);
      else
      {  recursiveExampleB(N-1);      // first, recursive call
         System.out.println(N);       // second, print
      }
   }
   public static void main(String args[])
   {  System.out.println("Calling method A:");
      recursiveExampleA(5);
      System.out.println("Calling method B:");
      recursiveExampleB(5);
   }
}
```

When this program is compiled and executed, the output will be:

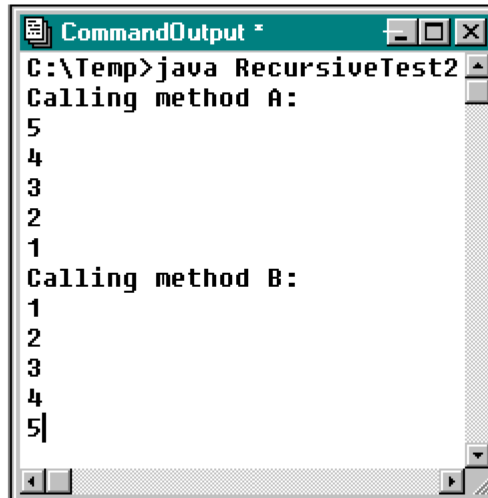


Figure 7.1.2: Output of two Recursive Test methods

This *is* surprising: the first method, as before, prints the numbers from 5 down to 1, while the second version of that method counts *up* from 1 to 5. We need to explain carefully how this small change in code of switching two lines can make such a difference in the way the methods work.

Let's take a close look at the first method `recursiveExampleA`. That method starts with input `N = 5` and executes its recursive call by first printing out the current value of `N (5)`, then calling itself with input `N = 4`. That means, however, that the first version of `recursiveExampleA` can not finish until the second version finishes. That version, in turn, prints `N (4)` and starts another version of `recursiveExampleA`, this time with input value `N = 3`. Now two versions of the method are put on hold, and the third version prints `N (3)` and executes itself yet again. Now, with three versions on hold, `recursiveExampleA` prints `N (2)`, and calls itself one more time with input `N = 1`. That version, on the other hand, executes the base case (printing `N = 1`) and finishes. Since it finishes, the "immediate predecessor " of the method (where `N` was `2`) can now finish, causing the version "above" *it* with `N = 3` to finish, and so on, until the first version finishes last. Here is the chain of execution of this method:

`recursiveExampleA(5)`

- `prints 5`
- `calls recursiveExampleA(4) and waits`

  `recursiveExampleA(4)`

  - `prints 4`
  - `calls recursiveExampleA(3) and waits`

    `recursiveExampleA(3)`

    - `prints 3`
    - `calls recursiveExampleA(2) and waits`

      `recursiveExampleA(2)`

      - `prints 2`
      - `calls recursiveExampleA(1) and waits`

        `recursiveExampleA(1)`

        - `prints 1 (BASE CASE)`
        - `exits`

      `recursiveExampleA(2) continues`

      - `exits`

    `recursiveExampleA(3) continues`

    - `exists`

  `recursiveExampleA(4) continues`

  - `exists`

```
recursiveExampleA(5) continues
•   exits
```

Figure 7.1.3: Valid recursion counting down

The second version of the method, `recursiveExampleB`, works similarly, but with a subtle difference: The method starts with `N = 5` and executes its recursive call. Before the print statement is reached, however, the method `recursiveExampleB` is called with input `4`, putting the original version on hold before it can print anything. That version, on the other hand, can not print out *its* value of `N (4)` because it calls `recursiveExampleB` first with `N = 3`. That goes on until the method `recursiveExampleB` calls itself with `N = 1`. That version receiving `N = 1` as input executes its base case, prints out `N = 1`, and finishes. Since it is now finished, its "immediate predecessor" can continue its execution, print out its value of `N (2)` and quit. Then the version above it can continue, print out its value of `N (3)` and quit. This goes on until all versions of the `recursiveExampleB` method are finished, resulting in the printout counting up from `1` to `5`. Here is the chain of execution of this method:

```
recursiveExampleB(5)
•   calls recursiveExampleB(4) and waits
    recursiveExampleB(4)
    •   calls recursiveExampleB(3) and waits
        recursiveExampleB(3)
        •   calls recursiveExampleB(2) and waits
            recursiveExampleB(2)
            •   calls recursiveExampleB(1) and waits
                recursiveExampleB(1)
                •   prints 1 (BASE CASE)
                •   exits
            recursiveExampleB(2) continues
            •   prints 2
            •   exits
        recursiveExampleB(3) continues
        •   prints 3
        •   exits
    recursiveExampleB(4) continues
    •   prints 4
    •   exits
recursiveExampleB(5) continues
•   prints 5
•   exits
```

Figure 7.1.4: Valid recursion counting up

Before we show some useful recursive methods, we should try to determine a rule of thumb as to when and how to use a recursive method:

Definition 7.1.6: Rule of Thumb for Using Recursion

Suppose you are dealing with a problem of the following type:

- *one particular situation of your problem is easy to deal with*
- *every other situation* would *be easy, if you only new the answer to the previous case*

Then a recursive method will solve the problem nicely and quickly:

- *convert the easy case to the base case of the recursive method*
- *convert the other case to the recursive call, where you use the call to the method itself as if it would already deliver the intended answer perfectly well*

It also helps to write down the complete method header, including a comment regarding what your method wants to accomplish. Then you can use it in the recursive call as if the method already did *accomplish its task.*

That rule of thumb probably sounds mystifying – as indeed recursive methods seemingly provide a solution to a problem in mysterious ways. The following examples will, hopefully, explain this rule of thumb and demystify recursive methods.

Some Useful Recursive Methods

From the above examples it may be apparent that recursion is actually quite demanding on system resources: each time a recursive call is made, the currently executing method is put on hold. That means that the JVM needs to store the current state of the method before calling the next version. But before we mention the "bad" news using recursion, here are a couple of examples where recursion actually produces useful methods.

Example 7.1.7:

Recall the definition of `N!` (read as `N` factorial): $N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (N-1) \cdot N$. For example, $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. Create and test a method that computes and returns `N!` for any positive integer `N`.

Before attempting to solve this problem, let's rephrase what we have to do:

- `1!` is easy, it's simply `1`.
- `N!` would be easy if we already knew `(N-1)!`, because $N! = N \cdot (N-1)!$

But that means of course that our rule of thumb applies. First, we will write down the method header, with comments indicating what the method will accomplish:

```
public static int factorial(int N)
{  / returns the value for N! */ }
```

Now we can create the method easily, using our rule of thumb:

```
public static int factorial(int N)
{  if (N == 1)
      return 1;                   // the "easy" base case
   else
      return N * factorial(N-1); // recursive call, uses what the
                                 // method will accomplish
}
```

And that's already it: the method will work (almost) perfectly, as in the following complete program:

```
public class FactorialTest
{  public static int factorial(int N)
   {  if (N == 1)
         return 1;
      else
```

```
            return N * factorial(N-1);
      }
      public static void main(String args[])
      {  System.out.println(" 5! =" + factorial(5));
         System.out.println("10! =" + factorial(10));
         System.out.println("20! =" + factorial(20));
         System.out.println("50! =" + factorial(50));
      }
}
```

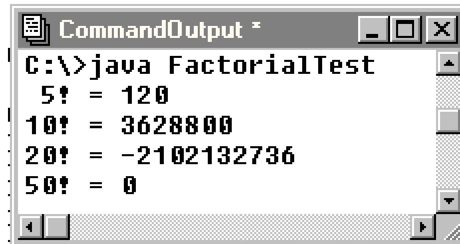


Figure 7.1.5: Output for factorial method

As you can see in figure 7.1.5, the first two invocations of `factorial` work great, verifying that in principle our method is correct. However, `factorial(20)` returns a negative number, and `factorial(50)` returns `0`. Both answers are clearly incorrect. This happens because the answers to the last two computations are larger than the maximum integer possible[4], and we should redefine the `factorial` method using either `long` or `double` as input and output type (see exercises). The method, in principle, works fine.

■

Note: The above method factorial will *not* work correctly if the input is not a positive integer. In other words, calling that method as in

```
System.out.println(FactorialTest.factorial(-10));
```

will result in an infinite loop because the base case of the recursion is never reached (verify). While mathematically the factorial of a negative number may not make much sense, we should modify our method so that it will not inadvertently result in an infinite loop. A simple change to the condition leading to the base case will do the trick:

```
public static int factorial(int N)
{  if (N <= 1)
      return 1;
   else
      return N * factorial(N-1);
}
```

Incidentally, this new definition of our method will result in `factorial(0) = 1`, which is indeed the mathematically correct definition of `0!`.

Example 7.1.8:

Define a method that computes a^n, i.e. `a` raised to the power `n`, where `n` is a non-negative integer.

---

[4] See chapter 1, table 10, which list the largest positive and smallest negative integer values. If an integer exceeds those maximum values it will "wrap around", i.e. the next integer after the largest positive value is the smallest negative value.

We have to compute $a^n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{n \text{ times}}$. That, of course, is easy for `n = 0`, because any number raised to the 0th power is 1. And a^n would be easy to compute if we knew a^{n-1}, because $a^n = a \cdot a^{n-1}$. Finally, the method header of the method we want to create is:

```
public static double power(double base, int exp)
{ /* returns base raised to the exp power */ }
```

Now we have all ingredients for our Rule of Thumb in definition 7.6.1 and it is easy to create the complete method:

```
public static double power(double base, int exp)
{  if (exp == 0)
      return 1;
   else
      return base * power(base, exp-1);
}
```

You should make sure that this method indeed works as advertised by embedding it in a suitable program.[5]

■

Before defining our next simple but useful recursive method we need to make a quick excursion into the land of binary numbers.

As you may recall from you math classes, every number can be uniquely represented as a sequence of 1's and 0's. That fact is the basis of every computer, including the Java Virtual Machine. Computers are, after all, comprised of lots of on/off switches called transistors and hence everything that the computer handles must eventually be stored in that format of on/off, or 1 and 0, sequences. As an example, here is the binary representation of an integer:

XXX -> Moved to chapter 1 <- XXX

Definition 7.1.9: Binary Representation of an Integer

Take any positive integer `X`*. Then* `X` *can be represented as a finite sum of powers of* `2` *in the form:* $X = a_n 2^n + a_{n-1} 2^{n-1} + \ldots + a_1 2^1 + a_0 2^0$*, where each* a_j *is either* `0` *or* `1` *and* `n` *is a suitable integer. The sequence* $a_n, a_{n-1}, a_{n-2}, \ldots a_2, a_1, a_0$ *is called the binary representation of the integer* `X`*, i.e. a representation that requires only* `0`*'s or* `1`*'s. We sometimes write such a binary representation as* $(a_n\ a_{n-1}\ \ldots\ a_2\ a_1\ a_0)_2$ *to indicate that the number in parenthesis is in binary form.*[6]

Example 7.1.10:

Convert the numbers 44 and 63 to their binary representation.

---

[5] To be picky, the above `power` method is only a valid recursion if the exponent is a non-negative integer. What would happen for a negative exponent as input? Can you redefine it so that it takes care of this problem? Does the method work for positive as well as negative base input values?

[6] This is really no different that our "usual" numbers, which can be uniquely represented as a finite sum of powers of 10. For example, the number 12,345 can be written as: $12{,}345 = 1*10^4 + 2*10^3 + 3*10^2 + 4*10^1 + 5*10^0$. Incidentally, in a binary representation $12{,}345 = (11000000111001)_2$.

Here is how to find the binary representation of the number `44`. To start, we write down the first few powers of 2 in a table:

64	32	16	8	4	2	1

Next, we can see that the highest power of 2 that fits into 44 is 32, with 12 remaining. Hence, we enter a 1 into the "32" slot of the table:

64	32	16	8	4	2	1
	1					

Since the remainder is 12, the next power of two that fits that remainder is 8, with a remainder of 4. We therefore enter a 1 into the "8" slot of the table:

64	32	16	8	4	2	1
	1		1			

Since the last remainder was 4 it is itself a power of 2, so that we enter a 1 into the "4" slot. Since the other powers of 2 were not used, we enter zeros into their slots:

64	32	16	8	4	2	1
0	1	0	1	1	0	0

Thus, we have arrived at the binary representation $(101100)_2$ of 44 (we have dropped the leading zero), which can easily be confirmed: $1\ 2^5 + 0\ 2^4 + 1\ 2^3 + 1\ 2^2 + 0\ 2^1 + 0\ 2^0 = 44$.

As another example, the binary representation of 63 is:

64	32	16	8	4	2	1
0	1	1	1	1	1	1

or, alternatively $(111111)_2 = 63$.

The converse is also easy: to find out which integers are represented by $(100010010)_2$ and $(1101011)_2$, for example, we enter them into appropriate tables:

256	128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	1	0

Then we add those powers of 2 with a non-zero entry: 2 + 16 + 256 = 274.

Similarly, $(1101011)_2$ as an integer is 1 + 2 + 8 + 32 + 64 = 107

Now we want to create methods that convert "regular" integers to their binary representation and visa versa.

Example 7.1.11:

Create a method that converts an integer from its standard decimal system based on powers of 10 into its representation as a binary number, i.e. a number in a base-2

number system. Also, create a method that does the opposite, i.e. convert a binary number into a "regular" integer.

The first question is harder than the second so we will tackle it first. If `N` is a "regular" integer, we could follow the above approach and first find the largest power of 2 that is still less than `N`. Then we could find the remainder and repeat the process until we are done. Here is some code to determine the highest power of 2 that is still less than or equal to `N`:

```
int power = 0;
while (Math.pow(2, power) <= N)
   power++;
power--;
```

With the help of this code our method to convert "regular" integers to their binary representations could look as follows:

```
public static void toBinary(int N)
{  // First, find the "appropriate integer n" mentioned in the definition
   // (we call that number 'power' in this method)
   int power = 0;
   while (Math.pow(2, power) <= N)
      power++;
   power--;
   // Now, determine whether to enter 0's or1's in the appropriate table
   for (int i = power; i >= 0; i--)
   {  int powerOfTwo = (int)Math.pow(2, i);
      if (powerOfTwo <= N)
         System.out.print("1");
      else
         System.out.print("0");
      N = N % powerOfTwo;
   }
}
```

This method follows pretty closely the previously explained procedure of entering either 0's or 1's into an appropriate table. But while certainly working the method is rather complicated for a simple task. Perhaps using recursion would give a better method.

To attempt a recursive solution to this problem, we need to rephrase it in a form where our Rule of Thumb from definition 7.1.6 would apply: First, let's assume that `n` is even. If we then knew the binary representation of `n / 2` then it is easy to find the representation of `n`, because:

> if $n/2 = a_k 2^k + a_{k-1} 2^{k-1} + ... + a_1 2^1 + a_0$ then $n = a_k 2^{k+1} + a_{k-1} 2^k + ... + a_1 2^2 + a_0 2^1$ because we can simply multiply both sides by two. But in terms of binary representation, that means that given that $n/2 = (a_k a_{k-1} ... a_1 a_0)_2$ we know that $n = (a_k a_{k-1} ... a_1 a_0 0)_2$ because multiplication by 2 amounts to adding a zero at the end of a binary representation.

As an example, let's find the binary representation of, say, 10, given that we know the one for 5, which is $(101)_2$ (because 5 = 1*4 + 0*2 + 1*1). Then the binary representation of 10 is simply that of 5 with a 0 appended at the end, i.e. 10 = $(1010)_2$ (because 10 = 1*8 + 0*4 + 1*2 + 0*1).

A similar argument applies when n is odd:

> if `n` is odd then `n - 1` is even, so that the previous argument means that adding a zero at the end of the binary representation of `(n-1)/2` gives us the representation of `n-1`. However, that

leaves us short by 1, so that instead of appending a zero to the representation of `(n-1)/2` we need to append a 1 (because a 1 in the last slot represents $2^0 = 1$).

As an example, let's find the binary representation of, say, 13, given that we know the one of (13 - 1) / 2 = 6 which is $(110)_2$ (because 6 = 1*4 + 1*2 + 0*1). Then the binary representation of 13 is simply that of 6 with a 1 appended at the end, i.e. 13 = $(1101)_2$ (because 13 = 1*8 + 1*4 + 0*2 + 1*1).

Now we have determined that it is easy to find the binary representation of `N`, given that we know the one for `N/2` (integer division automatically takes care of our even/odd case):

- first, find the binary representation of `N/2`
- then, append a 0 to that if `N` was even or a 1 if `N` was odd

That, of course, is exactly what our rule of thumb asks for. We still need to determine a base case, however. But in our situation, there really is no base case, the above procedure simply works, as long as `N` is 1 or higher. An empty base case is simply a particularly easy case: do nothing.

Therefore, our complete recursive method to convert a positive integer `N` to its binary representation is:

```
public static void toBinary(int n)
{  if (n > 0)
   {  toBinary(n/2);
      System.out.print((n % 2));
   }
}
```

This example illustrates what frequently happens when you suspect that a recursive method might be useful:

> The original phrasing of a problem may not seem to call for a recursive solution and a complicated program or method could be created, following the original phrasing of the problem. After spending some time thinking about the problem and rephrasing it, a *considerably* easier method can be created using recursion. In fact, the recursive method might be so easy that it is hard to believe that it will actually accomplish the desired task.

But it does and the second version of this method is by far more elegant than the first. Creating a method that converts a number from its binary representation to a "regular" integer is left as an exercise. Again there are at least two possible solutions, one that does not use recursion and a recursive solution. You should try to provide both.

■

The last example showed how a method containing a loop was converted into a recursive method without any loop. That, in fact, is always possible:

Definition 7.1.12: Single Recursion is equivalent to a Simple Loop

> *Every single-recursive method can be converted into an equivalent non-recursive method using a loop, and every loop can be converted into a recursive method.*[7]

[7] A slight modification of the method header may be necessary during this type of conversion.

Example 7.1.13:

In the previous exercises we created recursive methods to compute `N!` and a^n. Redo these methods so that they produce the same results but are not using recursion.

The value of `N!` is computed by multiplying together all numbers from 1 to `N`. That, of course, is a construction we have seen before, so that it is easy to create a loop-based version of the factorial method:

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

Similarly, to compute a^n we need to multiply a with itself `n` times (unless `n` is 0). Therefore, a loop-based version of the previous `power` method is:

```
public static double power(double base, int exp)
{  if (exp == 0)
      return 1.0;
   else
   {  double prod = base;
      for (int i = 1; i < exp; i++)
         prod *= base;
      return prod;
   }
}
```

You should, of course, make sure that both methods work as advertised.[8]

In both of these examples we first had the recursive method and we then created an appropriate non-recursive method. The procedure also works the other way around:

Example 7.1.14:

In chapter two we dealt with one-dimensional arrays, and it was straightforward to create a method that uses a simple `for` loop to compute the sum of an input array of doubles. According to definition 7.1.12 that method can be turned into a recursive method that does not contain any direct loop. Provide that recursive method (note that you may need to slightly alter the method header).

First, recall how to find the sum of all numbers in an array of double numbers:

```
public static double sum(double a[])
{  double tmpSum = 0.0;
   for (int i = 0; i < a.length; i++)
      tmpSum += a[i];
   return tmpSum;
}
```

---

[8] The non-recursive factorial method suffers from the same problem the recursive one had: if the input is too large, the resulting number will be larger than the largest possible integer and the values will "wrap around". As before, changing the types to `long` or `double` will solve that problem.

This method is simple enough and does not necessarily benefit from a recursive version. However, since we were asked to do it, here is how a recursive version would look like.

First, the method header is not sufficient for a recursive method. After all, such a method requires that some input parameter changes in the recursive call to eventually reach the base case, but in our situation the input is an array, and nothing should change that.

Therefore, we add another input variable `N`, indicating that the method will find the sum of the first `N` entries in the array `a`, i.e.:

```
public static double sum(double a[], int N)
{ /* finds the sum of the first N entries in the array a */ }
```

With this minor modification it is easy to determine a recursive solution:

- if `N = 0`, the method should return 0.0, since there is nothing to do
- otherwise, if we knew the sum of the entries up to `N-1`, then the sum of all entries would simply be the sum of the ones up to `N-1` plus the last entry.

Hence, our recursion is complete:

```
public static double sum(double a[], int N)
{  /* finds the sum of the first N entries in the array a */
   if (N == 0)
      return 0.0;
   else
      return a[N-1] + sum(a, N-1);
}
```

While this method is not really simpler that the previous version, it is certainly more elegant because it does not seem to really do anything, yet it still accomplishes its task perfectly.

■

Actually, many other methods dealing with arrays can be converted into equivalent recursive methods – see the exercises for details.

Multiple Recursion

While single recursion (i.e. methods that call themselves once) can be interesting, multiple recursive calls can lead to methods that perform very complex tasks and are not easy to understand.

Definition 7.1.15:

> *Multiple recursion refers to recursive methods where the recursive call contains more than one instance of a call to the method itself and all calls are made in one execution of the method.*[9].

Example 7.1.16:

Create a recursive method with an integer input that calls itself twice in its recursive case, each time with input one less than the current value of the input. It should also print out the current value of its input. If the input is one or less, the method should

[9] A method that contains several calls to itself but where each call is embedded in a nested `if-else` statement so that only one of them would execute is not a double recursive method.

print the value of the input without a recursive call. Test this method, starting with an input value of 4.

Clearly, the base case occurs when the input `N <= 1`. The recursive call must contain three statements: one statement to print the current value of `N` and two recursive calls with input `N-1`. But the order of these three statements is not specified ! Therefore, there are three possible versions of this method:

```
static void R1(int N)
{  if (N <= 1)
      System.out.print(N+" ");
   else
   {  System.out.print(N+" ");
      R1(N-1);
      R1(N-1);
   }
}
static void R3(int N)
{  if (N <= 1)
      System.out.print(N+" ");
   else
   {  R3(N-1);
      R3(N-1);
      System.out.print(N+" ");
   }
}
```

```
static void R2(int N)
{  if (N <= 1)
      System.out.print(N + " ");
   else
   {  R2(N-1);
      System.out.print(N + " ");
      R2(N-1);
   }
}
public static void main(String args[])
{  System.out.println("\nR1(4):");
   R1(4);
   System.out.println("\nR2(4):");
   R2(4);
   System.out.println("\nR3(4):");
   R3(4);
}
```

When we execute these methods we might expect to see the numbers from 1 to 4 in regular or perhaps reverse order. Instead, the output is shown in table 7.1.6:

R1(4)	4, 3, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 2, 1, 1
R2(4)	1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1
R3(4)	1, 1, 2, 1, 1, 2, 3, 1, 1, 2, 1, 1, 2, 3, 4

Table 7.1.6: Output of double-recursive methods

To understand this output, we need to carefully trace what happens when each method is called. Let's start with `R1`, but instead of figuring out what happens with input of 4, let's start "backwards", so that the easy case comes first:

- `R1(1)` simply prints out 1 because only the base case executes
- `R1(2)` prints `N = 2`, then calls `R1(1)`, then calls `R1(1)`. But we already know that `R1(1)` just prints 1, so that the output for `R1(2)` will be 2, 1, 1.
- `R1(3)` prints `N = 3`, then calls `R1(2)`, then calls `R1(2)`. But we already know that `R1(2)` prints out 2, 1, 1, so that `R1(3)` will print out 3, 2, 1, 1, 2, 1, 1
- `R1(4)` prints `N = 4`, then calls `R1(3)`, then calls `R1(3)`. But we already know that `R1(3)` prints out 3, 2, 1, 1, 2, 1, 1, so that `R1(4)` will print out 4, 3, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 2, 1, 1.

The point of this analysis is that we are actually using the very basis of recursion: if we only knew what happens for `N-1` the case for `N` is easy to determine. Therefore, finding out `R1(4)` would be easy if we knew `R1(3)`. That, in turn, would be easy if we knew `R1(2)`, which would be easy if we knew `R1(1)`, which *is* easy because it is the base case. So instead of trying to determine `R1(4)`, we start with `R1(1)` and work our way up until we reach the case we are interested in.

Similarly analyzing `R2` will now be easy, starting with the base case and working our way up:

- `R2(1)` is easy since it is the base case: it just prints 1.
- `R2(2)` calls `R2(1)`, prints out 2, then calls `R2(1)`. Hence, it would print out 1, 2, 1
- `R2(3)` calls `R2(2)`, prints out 3, then calls `R2(2)`. Hence, it would print out 1, 2, 1, 3, 1, 2, 1
- Finally, `R2(4)` calls `R2(3)`, prints out 4, then calls `R2(3)`. Therefore, the output is 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1.

Analyzing `R3` to determine why the output is they way it is should now be easy.

■

Of course, the above example(s) of multiple recursion are not really useful. So, here is an example of using multiple recursion to determine some rather well known mathematical numbers called the Fibonacci numbers.

Example 7.1.17:

The Fibonacci numbers are mathematically defined as follows:

$$x_1 = 1,\ x_2 = 1, \text{ and } x_n = x_{n-1} + x_{n-2} \text{ for } n \geq 3$$

i.e. the first two Fibonacci numbers are set to 1 and 1, respectively and every remaining number is the sum of its two predecessors. The first few Fibonacci numbers, therefore, are:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

Create a method that computes the nth Fibonacci number, then use it to find the first 20 Fibonacci numbers.

What we need to do first is create a method called, say, `Fibonacci` that takes a positive integer `n` as input and produces, somehow, the nth Fibonacci number. But just by saying this, we have almost solved the problem:

```
public static int Fibonacci(int n)
{  /* returns nth Fibonacci number */ }
```

The first and second numbers are easy: each is simply equal to 1. Any other number would be easy if we knew the previous and second-previous number: we would add those to get the next Fibonacci number. But now we are in exactly the situation that our "rule of thumb" in definition 7.1.6 asks for, so our recursive method is:

```
public static int Fibonacci(int n)
{  if (n <= 2)
      return 1;
   else
      return Fibonacci(n-1) + Fibonacci(n-2);
}
```

To finish the example, we embed this method into an entire program as follows:

```
public class Fibs
{  public static int Fibonacci(int n)
   {  if (n <= 2)
         return 1;
      else
```

```
            return Fibonacci(n-1) + Fibonacci(n-2);
      }
      public static void main(String args[])
      {  for (int i = 1; i <= 20; i++)
            System.out.println(Fibonacci(i));
      }
}
```

The 20th Fibonacci number turns out to be, for example, 6765.

■

While this is certainly a conceptually easy way to find Fibonacci numbers, it is not really an efficient method, as the next example will illustrate:

Example 7.1.18:

Use the above method to find the 100th Fibonacci number. Explain.

Of course this should be easy: just put a call to `Fibonacci(100)` in the `main` method, compile and execute, and you should see the answer. But when you execute `Fibonacci(100)`, your computer will become *very* busy, and you will not see anything for a *very* long time. In fact, for all intent and purposes, the computer seems to have locked up, and you should hit `CONTROL-C` to stop the calculation.

To explain why this calculation will take so extraordinarily long, let's analyze a comparable method that is slightly easier to figure out:

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

This is a double-recursive method that is almost the same as the Fibonacci method, but simpler to analyze. Our basic question is: how often does this function call itself when called with an input argument of n. Figure 7.1.7 illustrates the answer to this question for some small `n` = 1, 2, 3, and 4 (from left to right, respectively):

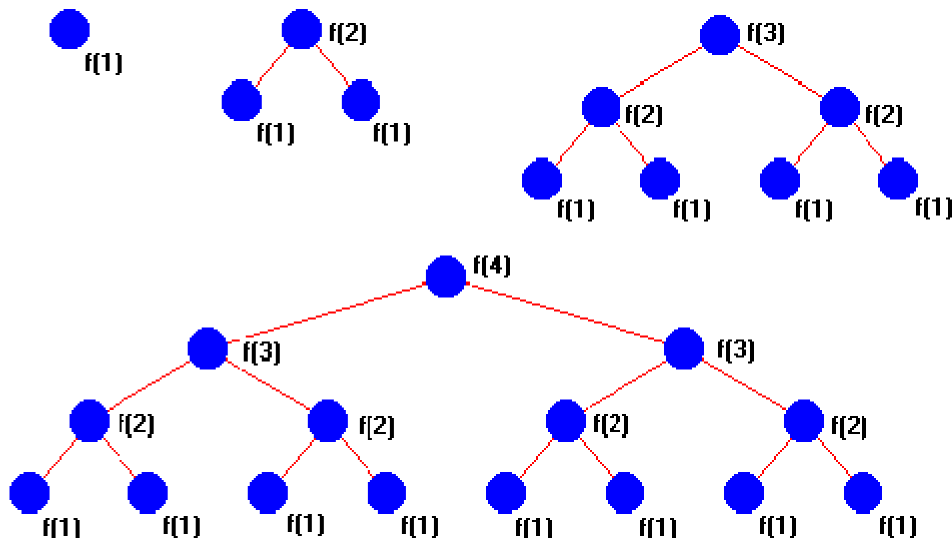


Figure 7.1.7: Counting Double Recursive Function Calls

In other words:

- for `n = 1` the function calls itself once
- for `n = 2` it calls itself `1 + 2 = 3` times
- for `n = 3` it calls itself `1 + 2 + 4 = 7` time
- for `n = 4` it calls itself `1 + 2 + 4 + 8 = 15` times.

In general, it seems that `F(n)` would call itself $1+2^1+2^2+2^3+2^4+\ldots+2^{n-1}$ times and since $1+2^1+2^2+2^3+2^4+\ldots+2^{n-1}=2^n-1$[10] we have figured out that `F(n)` would call itself exactly 2^n-1 times. In our case `n` was equal to 100, so `F(100)` would call itself $2^{100}-1$ times. That number, however, is huge: $2^{100}-1$ = 1,267,650,600,228,229,401,496,703,205,375 !!! Assuming that your computer can do 1 billion function calls per second (which it most likely can not do in the first place) it would take approximately 2 * 10^{27} seconds to figure out the answer. As an exercise, you can find out how many years that would take - certainly more time that we will have available.

The number of times that the actual `Fibonacci` method calls itself is slightly different from the above number. However, the order of magnitude[11] remains the same so there is no hope of computing `Fibonacci(100)` on a PC using the above double-recursive method.[12]

■

The Cost of Recursion

The above analysis shows that double-recursive methods can take a long time to compute, but it neglects one important aspect: computer memory is limited. If a method with one input variable calls itself $2^{100}-1$ times, it also needs to make $2^{100}-1$ copies of the input variable (as well as keep track of additional information). No computer has that kind of memory, so the computation of the 100th Fibonacci number would not finish but rather crash. That, in fact, is the potential drawback when using recursive methods:

Definition 7.1.19: The Cost of Recursion

> *Recursive methods are usually elegant and can perform complex tasks with a few lines of code. But for every recursive call the underlying operating system may need to place a copy of the current state of the method on a "stack"[13] for later retrieval to continue the method. Therefore, recursion can be very memory consuming, especially if the method contains a lot of information or if there is a multiple recursive call.*

---

[10] To find out why $1+2^1+2^2+2^3+2^4+\ldots+2^{n-1}=2^n-1$ we compute the answer twice (compare chapter 1, example 29): $1+2^1+2^2+2^3+2^4+\ldots+2^{n-1}=X$ and $2^1+2^2+2^3+2^4+\ldots+2^n=2X$. Therefore $X=2X-X=2^n-1$

[11] See the Appendix on Big-O notation for determining the order of magnitude of a method.

[12] There are much faster methods for computing Fibonacci numbers. You might want to ask a Mathematics teacher for help.

[13] See chapter 8 for a definition of a Stack.

Example 7.1.20:

Consider the following two methods:

```
public static void Expensive(int n, double x)
{  if (n >= 1)
   {  Expensive(n-1, x-1);
      System.out.println("n = " + n + ", x = " + x);
   }
}
public static void Cheap(int n, double x)
{  x = x - n + 1;
   for (int i = 1; i <= n; i++)
   {  System.out.println("n = " + i + ", x = " + x);
      x++;
   }
}
```

Are these methods functionally equivalent, i.e. do they produce the same results if given the same input? Which method uses less memory if called with input `n = 5` and `x = 5.0`?

When you execute these methods with various input parameters it is easy to see that they always seem to produce the same results. For example, when given the same input parameters as in `Expensive(5, 5.0)` and `Cheap(5, 5.0)`, both methods will display the numbers 1, 1.0, 2, 2.0, 3, 3.0, 4, 4.0, and 5, 5.0. Therefore, they are indeed functionally equivalent.

However, if `n = 5` the first method will call itself 4 additional times and display the original input values *last*. Therefore, it must temporarily store the additional input values somewhere, proceed to call itself 4 times, then print out the values that were temporarily stored. Therefore, the method needs at least as much memory to store 5 integers and 5 doubles.

The second, equivalent method uses a loop to accomplish the same task. It does not use recursion so the only memory requirements are those specifically indicated by the declared variables. Hence, the second version uses only one integer and one double variable as well as one additional loop control integer variable. Moreover, these memory requirements will not change for larger values of `n`.[14]

■

This discrepancy is, of course, even more drastic for double-recursive methods, where the number of calls the method makes to itself usually grows exponentially.

To illustrate this resource problem more drastically, here is another example that, depending on your version of the JVM, may even crash the JVM (it does for Sun's JVM 1.2 under Windows 98) instead of producing a suitable error or exception.

Example 7.1.21:

Consider the following example of defining and using a recursive method:

```
public class RecurseWithoutLimit
{  public void recurse(int N)
   {  double x[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

[14] If you measured the time it took for each method to complete a call with a large input value of n, the recursive version would be much slower than the non-recursive one. The JVM needs to allocate additional memory as the non-recursive method requires more space, and that will take additional time not needed by the non-recursive method.

```
            System.out.println(N);
            for (int i = 0; i < x.length; i++)
               x[i] = Math.sin(x[i]) + Math.cos(x[i]);
            recurse(N+1);
         }
         public static void main(String args[])
         {  RecurseWithoutLimit rc = new RecurseWithoutLimit();
            rc.recurse(1);
         }
      }
```

Is this a valid recursion? What do you expect to see when you execute this class? Execute this class and observe what happens. Replace the recursive method by an equivalent non-recursive one, run that new method and see if anything changes.

Clearly the method `RecurseWithoutLimit` is a recursive method without a base case, and the input to the recursive calls will keep increasing without bounds. Therefore, this is not a valid recursion. When the method executes, we would expect to see an infinitely increasing list of integers displayed on the screen. The part of the code that uses the `sine` and `cosine` functions is irrelevant, and is only there to simulate some computational behavior as well as occupy some memory within the recursive method. To be sure, we don't really expect to see infinitely large integers. After all, there is a largest integer, after which point the numbers should "wrap around" to the smallest, negative integer.

The reality, however, is different. When this program executes on my particular system, a laptop computer with 96MB RAM, using Sun's JDK 1.2 under Windows 98, it actually crashes the JVM after displaying integers up to a particular (and relatively small) value depending on the actual memory available at the time. In my case, the program crashes after about 25,000 recursive calls (see figure 7.1.8). On your system, the result may be different, and instead of crashing it may produce an error message. But in any case, the program will not perform as expected.[15]

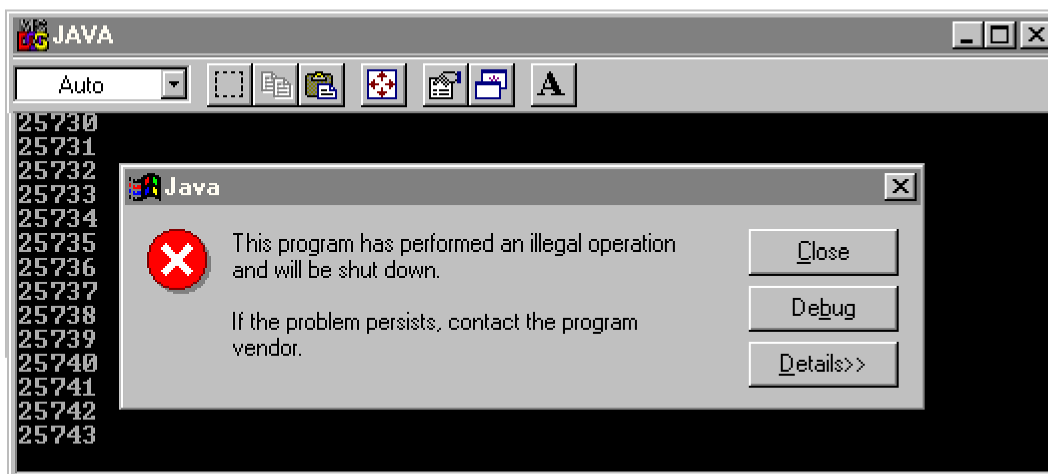


Figure 7.1.8: Excessive Recursion can crash the JVM

If we rewrote the method using a loop instead of recursion, it will *not* crash. The details are left as an exercise.

■

[15] Of course this example does not make much sense in the first place. But in example 7.2.17 we will see a meaningful recursive method that will crash when performing too many recursive steps.

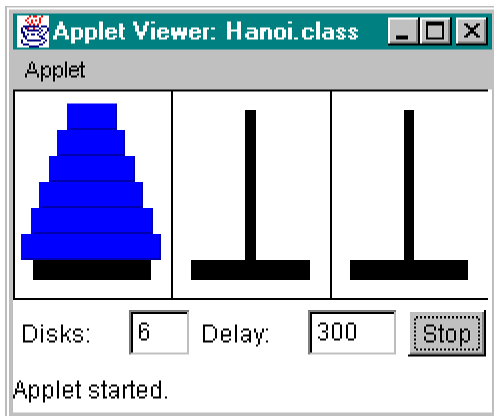
If memory considerations are an issue and the recursive method is expected to places several thousand calls to itself in the process of performing its task, it is worthwhile to convert the method into a non-recursive one using a loop. That usually works well for single recursive methods but double-recursive methods can not always be converted into equivalent methods using one (or more) simple loops. One could, however, make use of devices such as a "Stack" which we will introduce in chapter 8 to convert double-recursive methods into non-recursive methods, but that frequently makes the method a lot harder to code and may not eliminate the memory need anyway. In many cases, however, recursive methods do not need to make an unusually large number of recursive calls and recursive method can usually provided an elegant solution to many common "problems".

The Tower of Hanoi Applet

One problem where recursion indeed does provides an elegant solution to an otherwise complex task is the Tower of Hanoi problem. It would be quite difficult to solve that problem without recursion.

Example 7.2.1:

Suppose you have three pegs in front of you, the first one with 64 stacked disks of decreasing diameters, the other two are empty.



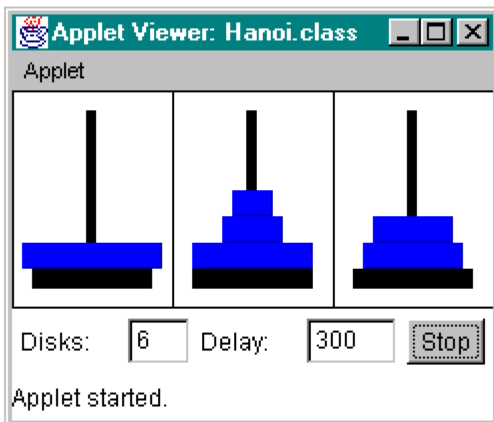


Figure 7.2.1: Tower of Hanoi with 6 Disks

The object of the game is to try to move all 64 disks from the first peg to the third peg, adhering to the following rules at all times:

- you can only move one disk at a time
- at no time can a smaller disk be underneath a larger disk

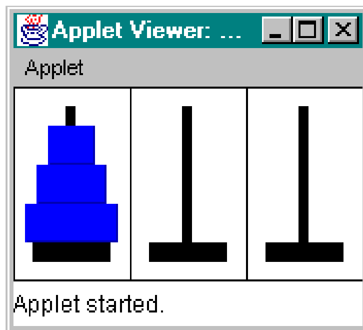
Actually, according to an ancient myth, when all 64 disks are moved to the third peg the world as we know it will come to an end. This is known as the "Tower of Hanoi" problem.

So, in that light we could rephrase our task at hand: make the world come to an end by moving the 64 disks to the third peg.

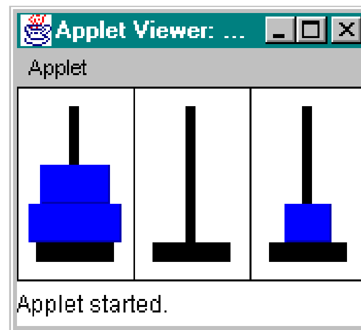
The more general Tower of Hanoi problem consists of moving `N` disks from the first peg to the last peg, following the rules at each step.

To the left is a picture of the general Tower of Hanoi problem with `N = 6` disks on the first peg, and an intermediate step where some disks have been moved around but not all are yet on the third peg in the proper order.

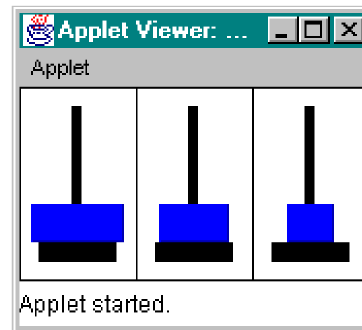
To get a feeling for this problem, let's try to solve the Tower of Hanoi problem with three disks "by hand":



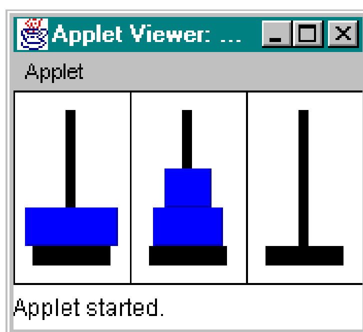
Disks in original position



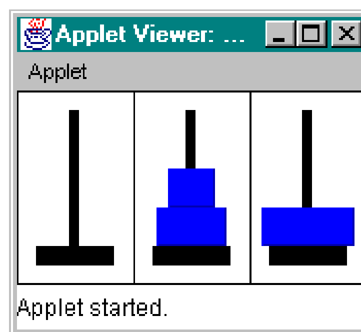
Move disk 1 from (a) to (c)



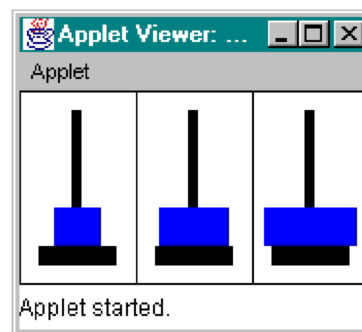
Move disk 2 from (a) to (b)



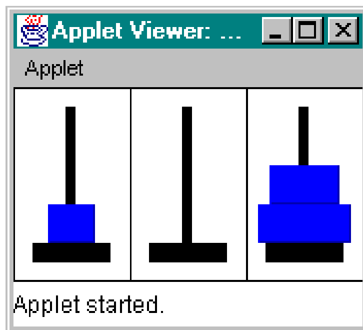
Move disk 1 from (c) to (b)



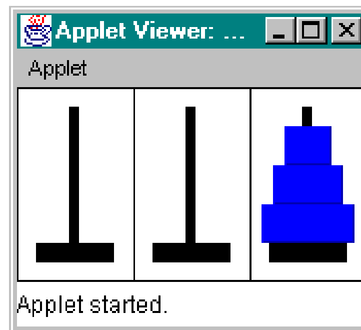
Move disk 3 from (a) to (c)



Move disk 1 from (b) to (a)



Move disk 2 from (b) to (c)



Move disk 1 from (a) to (c)

That was not so hard, but try moving 4, 5, or 6 disks "by hand" to understand how difficult this problem can get. To provide an algorithm for solving this problem we need to rephrase it somewhat:

- We have three pegs available to move `N` disks. Let's call these pegs `fromPeg` for the first one, `toPeg` for the last one, and `auxPeg` for the middle one.
- We want to move `N` disks from `fromPeg` to `toPeg`, using `auxPeg` as an auxiliary peg.
- If we knew how to move `N-1` disks from some peg to another peg, using the third one as helper, our problem would be easy:
  - First, move `N-1` disks from `fromPeg` to `auxPeg`, using `toPeg` as helper
  - Next, move the `N`-th disk from `fromPeg` to `toPeg`
  - Last, move the `N-1` disks from `auxPeg` to `toPeg`, using `fromPeg` as helper

But that's already the answer, thinking recursively: if we call our method to move disks from one peg to another using the third as helper, say:

```
void moveDisk(String fromPeg, String toPeg, String auxPeg, int N)
{  /* moves disk N from "fromPeg" to "toPeg", using "auxPeg" as helper */
```

then we can use this method to define the method itself. Of course, we first need a base case: if there was only one disk left on the `fromPeg` peg, we can simply move it without much ado:

```
void moveDisks(String fromPeg, String toPeg, String auxPeg, int N)
{  if (N == 1)   // base case
      System.out.println("Move disk 1 from " +fromPeg+ " to " +toPeg);
   else          // recursive call
   { /* some code */   }
}
```

But we have already stated how the recursive part should work:

- move `N-1` disks from `fromPeg` to `auxPeg`, using `toPeg` as helper
- move the `N`-th disk from `fromPeg` to `toPeg`
- move the `N-1` disks from `auxPeg` to `toPeg`, using `fromPeg` as helper

Since we do have a name for "move disks from one peg to another" we can complete the recursive method easily:

```
void moveDisks(String fromPeg, String toPeg, String auxPeg, int N)
{  if (N == 1)
      System.out.println("Move disk 1 from " +fromPeg+ " to " +toPeg);
   else
   {  moveDisks(fromPeg, auxPeg, toPeg, N-1);
      System.out.println("Move disk "+N+" from "+fromPeg+" to "+toPeg);
      moveDisks(auxPeg, toPeg, fromPeg, N-1);
   }
}
```

This algorithm will tells us which disk to move, and in what order, by simply calling the method as in:

```
moveDisks("A", "C", "B", 3);
```

You could declare the method static, to easily call it from a standard `main` method.

The answer is displayed in figure 7.1.10 and matches exactly the procedure we came up with "by hand" for three disks.

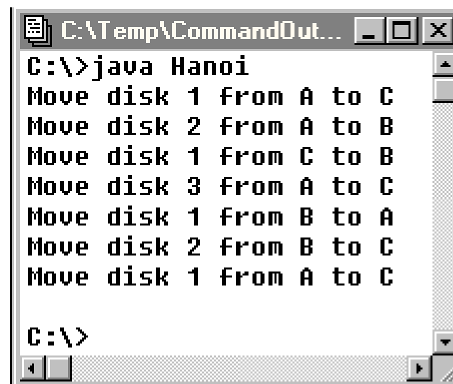


Figure 7.1.10: Solving Tower of Hanoi Problem

So, it looks like we have provided an answer for this ancient riddle, and we are just about ready to make the world come to an end by using our program to give us the instructions for moving 64 disks from the first to the third peg.

But while people in the past did not have computers, they were well aware that this riddle does have a theoretical solution (such as the algorithm we have come up with). But there are practical

considerations that will not allow us to implement our theoretically possible solution. The details are left as an exercise.

■

The basic problem underlying the Tower of Hanoi problems is mentioned in other old stories. For example, an Arab king once asked a Wise Man what he wanted to have as payment for a particular service rendered. The Wise Man answered that he wanted to have a single grain of rice placed on the first square of a chessboard and on each subsequent square should be double the amount of rice as on the previous square. The king imagined the first 8 of the 64 squares of a chessboard, did a quick mental calculation, and agreed to provide the rice to the Wise Man. However, that was a promise that the king was unable to keep – which is of course related to our problem of double-recursive methods. Can you figure it out?

Since you are still reading this, you obviously have not succeeded in stopping the world from existing. Perhaps it was not enough to provide the instructions for moving the disks, but we should have designed a program that actually moves the disks around graphically.

Example 7.2.2:

> Create a "Hanoi" applet that graphically solves the general Tower of Hanoi problem. In other words, the applet should display three pegs with disks, and it should graphically move the disks so that they eventually end up on the last peg.
>
> In addition, the applet should allow the user to set the number of disks moved, you should be able to start and stop the moving process, and there should be a user-definable delay so that the user can see which disk moves where at what time.

This time our problem is not to come up with the basic algorithm - we already have done that in exercise 7.2.1. Instead we need to worry about how to graphically implement this algorithm so that you can actually see the disks being moved around.

From an OOP standpoint, there are clearly two objects that make up this problem: a "disk" object, and a "peg" object. In addition, there should be at least one more class defining the graphical interface, buttons, etc., and controlling the entire process. Therefore, we decide to use four classes:

- A `HanoiDisk` class that models individuals disk. It should contain enough information so that the class can draw a representation of a disk in the appropriate height and width on a particular peg.
- A `HanoiPeg` class that models one peg. It should take care of adding and removing disks and it should draw its graphical representation together with any disks it may contain. This class will extend `Canvas` so that it has a drawing context to handle the graphics.
- A `HanoiApplet` applet class that defines all components and gives them a framework to run in. This class, of course, will extend `Applet`.
- A `HanoiControls` class that contains two text fields (for the number of disks and the delay) and at least one button to start and stop the moving process. It will extend `Panel` so that it can place the various control elements and implement `ActionListener` to react to action events.

Let's start with the `HanoiDisk` class. It needs to know which particular disk it is so that it can draw itself in the corresponding width. Therefore, it needs a field `ID` to store that information. In addition, disks belong to a peg, so it needs to have a field to determine the `peg` where it is currently located. Finally, it needs a `paint` method to draw itself. That method needs to draw the width of the disk according to its `ID`, but it also needs to be able to draw the disk at different heights. The `ID` is known

to the disk, so it can determine its own width, but the height depends on the level where the peg needs to put the disk. Therefore, the level needs to be an input argument to the `paint` method. In addition, the `paint` method needs to have a drawing context so that it gets a reference to a `Graphics` object as input.

```
HanoiDisk
HanoiPeg peg;
int      ID;
HanoiDisk(HanoiPeg _peg, int _ID)
void paint(Graphics g, int level)
```

Figure 7.2.2: Representation of `HanoiDisk` *class*

The rest consists of some simple computations to figure out the proper width of the disk, given the `ID`, the level, and the width and height of the peg canvas. Here is the class:

```
import java.awt.*;

public class HanoiDisk
{  private HanoiPeg peg   = null;
   private int      ID    = -1;

   public HanoiDisk(HanoiPeg _peg, int _ID)
   {  peg = _peg;
      ID = _ID;
   }
   public void paint(Graphics g, int level)
   {  int deltaH = peg.getSize().height / (HanoiApplet.DISKS+2);
      int deltaW = peg.getSize().width / (HanoiApplet.DISKS+2);
      g.setColor(Color.blue);
      g.fill3DRect(ID * deltaW / 2, peg.getSize().height-level*deltaH-20,
                   peg.getSize().width-ID*deltaW, deltaH, true);
   }
}
```

Next , let's look at the `HanoiPeg` class, which extends `Canvas`. This class needs to put the initial disks on a peg (a fixed number for the first peg and no disks on the other pegs), so we use a `boolean` variable for the constructor indicating whether `to` draw no disks or the appropriate number of disks. We use a `resetDisks` method to do the actual work. Next, a `HanoiPeg` needs to be able to keep track of the disks it contains, so it needs a field of type array of `HanoiDisk`. Finally, a `HanoiPeg` needs to add and remove disks as well as draw itself and the disks it contains. Therefore, we need an `addDisk`, `removeTopDisk`, and `paint` method.

```
HanoiPeg                 extends Canvas
int       top;
HanoiDisk disks[];
void resetDisks()
void resetDisks(boolean filled)
void addDisk(HanoiDisk disk)
HanoiDisk removeTopDisk()
void paint(Graphics g)
```

Figure 7.2.3: Representation of the `HanoiPeg` *class*

Actually, the disks themselves are drawn by the disk's `paint` method, but the peg's `paint` method needs to control which disk goes at what level. Here is the actual class:

```
import java.awt.*;

public class HanoiPeg extends Panel
{  private int top = 0;
```

```
   private HanoiDisk disks[];
   public HanoiPeg(boolean filled)
   {  resetDisks(filled); }
   public void resetDisks(boolean filled)
   {  disks = new HanoiDisk[HanoiApplet.DISKS];
      top = 0;
      if (filled)
      {  for (int i = 0; i < disks.length; i++)
            disks[i] = new HanoiDisk(this, i+1);
         top = disks.length;
      }
      repaint();
   }
   public void addDisk(HanoiDisk disk)
   {  disks[top] = disk;
      top++;
      repaint();
   }
   public HanoiDisk removeTopDisk()
   {  top--;
      repaint();
      return disks[top];
   }
   public void paint(Graphics g)
   {  g.setColor(Color.black);
      g.drawRect(0,0,getSize().width, getSize().height-1);
      g.fillRect(10, getSize().height - 20, getSize().width - 20, 10);
      g.fillRect((getSize().width -5)/2,10, 5, getSize().height - 20);
      for (int i = 0; i < top; i++)
         disks[i].paint(g, i+1);
   }
}
```

All methods are relatively easy: `addDisk` puts the input disk into the array of disks that the peg contains as a field and `removeDisk` returns the top-most disk from that array and removes it from the array. Both methods call `repaint` to update the graphics after moving a disk. The `paint` method draws a box around the drawing area, draws two fat lines representing the peg, and then calls on the `paint` method for all disks it contains, using the index of the disk as the level (or height) at which it should be drawn.

Finally, we need the applet to put everything together. In particular, this class should contain the recursive method `moveDisks` from example 7.2.1, suitably modified, to determine which disk moves where at what time. This method will use the `addDisk` and `removeDisk` of the appropriate pegs to move the disks. The pegs are implemented as a field of three `HanoiPegs` and therefore the new `moveDisks` method could look as follows:

```
private void moveDisks(int fromPeg, int toPeg, int auxPeg, int N)
{  if (N == 1)
      pegs[toPeg].addDisk(pegs[fromPeg].removeTopDisk());
   else
   {  moveDisks(fromPeg, auxPeg, toPeg, N-1);
      pegs[toPeg].addDisk(pegs[fromPeg].removeTopDisk());
      moveDisks(auxPeg, toPeg, fromPeg, N-1);
   }
}
```

Now our applet class could already be created as follows:

```
import java.applet.Applet;
import java.awt.*;
```

```
public class HanoiApplet extends Applet
{  protected static int DISKS = 12, SLEEP = 300;
   private HanoiPeg pegs[] = new HanoiPeg[3];
   public void init()
   {  Panel board = new Panel(new GridLayout(1,3));
      pegs[0] = new HanoiPeg(true);
      pegs[1] = new HanoiPeg(false);
      pegs[2] = new HanoiPeg(false);
      for (int i = 0; i < pegs.length; i++)
          board.add(pegs[i]);
      setLayout(new BorderLayout());
      add("Center", board);
      moveDisks(0, 2, 1, DISKS);
   }
   private void moveDisks(int fromPeg, int toPeg, int auxPeg, int N)
   {  if (N == 1)
         pegs[toPeg].addDisk(pegs[fromPeg].removeTopDisk());
      else
      {  moveDisks(fromPeg, auxPeg, toPeg, N-1);
         pegs[toPeg].addDisk(pegs[fromPeg].removeTopDisk());
         moveDisks(auxPeg, toPeg, fromPeg, N-1);
      }
   }
}
```

However, that will not quite work. For one thing, we did not implement any controls to set the number of disks and any delay in moving them. Even worse, the above applet, once started, will run without delay until it is done and we would not have any chance to observe the movement of the disks. It would simply show nothing for a while and then display the final image with all disks moved to the final peg.

Therefore, we need a thread that can be put to sleep for a specified amount of time in between moving each disk so that we have a chance to see that movement. Our improved applet will implement `Runnable` and use a separate thread to control the `moveDisks` method. We will follow the Rule of Thumb outlined in definition 5.2.11 to create the code including our thread, but we will have a slight problem stopping our recursive method. We will explain that problem in detail in the exercises, for now let's take a look at the complete `HanoiApplet` class.

```
import java.applet.Applet;
import java.awt.*;

public class HanoiApplet extends Applet implements Runnable
{  protected static int DISKS = 6, SLEEP = 300;
   private HanoiPeg pegs[] = new HanoiPeg[3];
   private Thread thread = null;
   private HanoiControls controls = null;
   private Thread currentThread   = null;
   public void init()
   {  pegs[0] = new HanoiPeg(true);
      pegs[1] = new HanoiPeg(false);
      pegs[2] = new HanoiPeg(false);
      Panel board = new Panel();
      board.setLayout(new GridLayout(1,3));
      for (int i = 0; i < pegs.length; i++)
          board.add(pegs[i]);
      controls = new HanoiControls(this);
      setLayout(new BorderLayout());
      add("Center", board);
      add("South", controls);
```

```
   }
   public void restart()
   {  thread = null;
      pegs[0].resetDisks(true);
      pegs[1].resetDisks(false);
      pegs[2].resetDisks(false);
      thread = new Thread(this);
      thread.start();
   }
   public void stop()
   {  thread = null; }
   public void run()
   {  currentThread = Thread.currentThread();
      moveDisks(0, 2, 1, DISKS);
   }
   private void moveDisks(int fromPeg, int toPeg, int auxPeg, int N)
   {  if (thread != currentThread)
         return;
      else if ((N == 1) && (thread == currentThread))
      {  pegs[toPeg].addDisk(pegs[fromPeg].removeTopDisk());
         delay();
      }
      else
      {  moveDisks(fromPeg, auxPeg, toPeg, N-1);
         if (thread == currentThread)
         {  pegs[toPeg].addDisk(pegs[fromPeg].removeTopDisk());
            delay();
         }
         moveDisks(auxPeg, toPeg, fromPeg, N-1);
      }
   }
   private void delay()
   {  try
      { thread.sleep(SLEEP); }
      catch(InterruptedException ie)
      { System.err.println("Error: " + ie); }
   }
}
```

Note that we added a `HanoiControl` class to the applet layout that we will create in a second. We made a few changes to the outline in definition 5.2.11 to make sure our movement of disks in `moveDisks` will indeed stop. First, we added a field `currentThread` of type `Thread` so that different methods can access that variable. It is initialized, as usual, in the `run` method but referred to several times in the recursive `moveDisks` method. That is necessary so that the `moveDisk` method will stop as soon as `currentThread` is different from thread. Since `moveDisk` is recursive, we need to check whether `currentThread == thread` several times to make sure that the method stops regardless at which level of the recursion it currently executes. We will see in detail why this is necessary in the exercises.

The piece missing from our applet is the user interface represented by the `HanoiControls` class that has already been added to `HanoiApplet`. It will contain two text fields to set the number of disks and the delay between moves and a button to start and stop the applet. Here is the corresponding `HanoiControls` class:

```
import java.awt.event.*;
import java.awt.*;

public class HanoiControls extends Panel implements ActionListener
{  private HanoiApplet  tower = null;
   private Button button = new Button("Start");
```

```
        private TextField disks =
                new TextField(String.valueOf(HanoiApplet.DISKS));
        private TextField sleep =
                new TextField(String.valueOf(HanoiApplet.SLEEP));

        public HanoiControls(HanoiApplet _tower)
        {   tower = _tower;
            setLayout(new FlowLayout());
            add(new Label("Number of disks:")); add(disks);
            add(new Label("Delay:"));           add(sleep);
            add(button);
            button.addActionListener(this);
        }
        public void setLabel(String label)
        {   button.setLabel(label); }
        public void actionPerformed(ActionEvent e)
        {   if (button.getLabel().equals("Start"))
            {   tower.DISKS = Integer.parseInt(disks.getText());
                tower.SLEEP = Integer.parseInt(sleep.getText());
                if (tower.SLEEP < 50)
                {   tower.SLEEP = 50;
                    sleep.setText(Integer.toString(tower.SLEEP));
                }
                tower.restart();
                button.setLabel("Stop");
            }
            else
            {   tower.stop();
                button.setLabel("Start");
            }
        }
    }
```

This applet will give the user the option to set the number of disks and to determine the delay so that they can easily follow the movements of disk. Note that `HanoiControls` ensures that delay can not be less than 50. If the delay is too small, the applet will not leave enough time for the user to click on the Start/Stop button.

Using this applet it is easy to verify our original algorithm of moving 3, 4, or 5 disks, but it is just as easy to use the applet to move, say, 10 disks. Before trying to bring the world to an end yet again, try executing the applet for 15 or 20 disks:

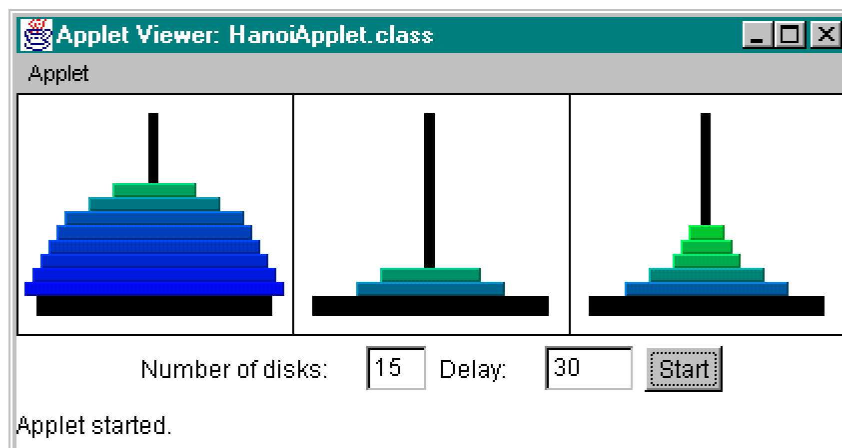


Figure 7.2.4: `HanoiApplet` in action, moving 15 disks

That should make it pretty clear that there will be very little chance of seeing the applet move all 64 disks around, at least not in our lifetime.

■

If you did try to set the number of disks to, say, 64, and the delay to, say, 50, you will see that the applet shows a lot of "flickering". That can be removed by using a technique called "double-buffering", as described in definition 6.5.15[16], or by converting the entire applet to Swing. The details are left as an exercise.

With this example we will conclude the section on recursion. We will next move to another "generally useful" topic, searching and sorting. In fact, some of the searching and sorting algorithms that we will introduce below are indeed defined recursively so that our previous discussions on recursive methods should come in handy.

Exercises

7.3. XXX – MOVED TO APPENDING Big-O Notation: the Cost of Computing

1. State, in your own words, what is meant by the following terms:

   a) limit of a function
   b) limit at infinity
   c) big-O
   d) f = O(g)
   e) growth rate of a function

2. Below are a variety of methods, each using an integer `n` as input. Determine the order of growth for each method, considered as a function of `n`.

```
void method1(int n)
{  for (int i=1; i <= 3*n; i++)
      System.out.println(n);
}

void method2(int n)
{  for (int i = 0; i < 2*n; i++)
      for (int j = 0; j < n; j++)
         System.out.println(i*j);
}

int method3(int n)
{  if (n <= 1)
      return 1;
   else
      return method3(n-2);
}

void method4(int n)
{  for (int i = 0; i < n/2; i++)
      for (int j = 0; j < i; j++)
         System.out.println(i*j);
}

void method5(int n, double x)
{  if (n <= 1)
      System.out.println(x);
   else
   {  method5(n-1, x+1);
      System.out.println(x);
      method5(n-2, x+2);
   }
}

void method6(int n)
{  if (n >= 1)
   {  System.out.println(n);
      method6(n/2);
   }
   else
      System.out.println(n);
}
```

3. What order of growth do the following mathematical functions have for large values?

[16] Definition 6.5.15 applies to Swing components that already use a default buffering mechanism. But it can easily be modified to implement double-buffering for AWT components as well (see exercises for details).

a) $f(n) = \frac{n(n+1)}{300} + 100^3 n$

b) $g(n) = n/2 + n^{1/2}$

c) $h(n) = n^2 + 2^n$

d) $k(n) = \log(n^5) + n^2 + n\log(n)$

e) $r(n) = 1^2 + 2^2 + 3^2 + \ldots + (n-1)^2 + n^2$ (consult a Calculus text book for help)

4. Suppose two algorithms are functionally equivalent (i.e. they produce the same output for the same input), but the first one is O(n log(n)), the second is O(n). Which one would you prefer?

5. Suppose two algorithms are functionally equivalent (i.e. they produce the same output for the same input), but the first one is O(n), the second is O(n^2). For some large value of n the first algorithm takes, approximately, 100 seconds to finish. Would the second algorithm require more or less time? Approximately how long would the second algorithm take to finish?

6. What is the order of growth for the insertion sort algorithm in the best and in the worst case situation?

7. The Java build-in sorting method for basic data types uses a modified quick search algorithm, while the same method for Object arrays uses a merge sort algorithm. Why are two different algorithms used?