

Chapter 2 Recursion: The Mirrors

1 The problem is defined in terms of a smaller problem of the same type:

Here, the last value in the array is checked and then the remaining part of the array is passed to the function.

Each recursive call diminishes the size of the problem:

The recursive call to `getNumberEqual` subtracts 1 from the current value for `n`, passing this as the parameter `n` in the next call, effectively reducing the size of the unsearched remainder of the array by 1.

An instance of the problem serves as the base case:

Here, the case where the size of the array ≤ 0 results in the return of the value 0: an array of size 0 can have no instances of the `desiredValue`. This terminates the recursion.

As the problem size diminishes, the base case is reached:

`n` is an integer and is decremented by 1 with each recursive call. After `n` recursive calls, the parameter `n` will have the value 0 and the base case will be reached.

2a The call `rabbit(5)` produces the following box trace:

```
n = 5
rabbit(4) = ?
rabbit(3) = ?
return ?
```

Follow the `rabbit(4)` call

```
n = 5
rabbit(4) = ?
rabbit(3) = ?
return ?
```

```
n = 4
rabbit(3) = ?
rabbit(2) = ?
return ?
```

Follow the `rabbit(3)` call

```
n = 5
rabbit(4) = ?
rabbit(3) = ?
return ?
```

```
n = 4
rabbit(3) = ?
rabbit(2) = ?
return ?
```

```
n = 3
rabbit(2) = ?
rabbit(1) = ?
return ?
```

Follow the `rabbit(2)` call

```
n = 5
rabbit(4) = ?
rabbit(3) = ?
return ?
```

```
n = 4
rabbit(3) = ?
rabbit(2) = ?
return ?
```

```
n = 3
rabbit(2) = ?
rabbit(1) = ?
return ?
```

```
n = 2
return 1
```

Base case: `n = 2`

```
n = 5
rabbit(4) = ?
rabbit(3) = ?
return ?
```

```
n = 4
rabbit(3) = ?
rabbit(2) = ?
return ?
```

```
n = 3
rabbit(2) = ?
rabbit(1) = ?
return ?
```

```
n = 2
return 1
```

The `rabbit(2)` call completes

n = 5 rabbit(4) = ? rabbit(3) = a? return ?	n = 4 rabbit(3) = ? rabbit(2) = ? return ?	n = 3 rabbit(2) = 1 rabbit(1) = ? return ?	Follow the rabbit(1) call
n = 5 rabbit(4) = ? rabbit(3) = ? return ?	n = 4 rabbit(3) = ? rabbit(2) = ? return ?	n = 3 rabbit(2) = 1 rabbit(1) = ? return ?	n = 1 return 1 Base case: n = 1
n = 5 rabbit(4) = ? rabbit(3) = ? return ?	n = 4 rabbit(3) = ? rabbit(2) = ? return ?	n = 3 rabbit(2) = 1 rabbit(1) = 1 return 2	n = 1 return 1 The rabbit(1) call completes
n = 5 rabbit(4) = ? rabbit(3) = ? return ?	n = 4 rabbit(3) = 2 rabbit(2) = ? return ?	n = 3 rabbit(2) = 1 rabbit(1) = 1 return 2	n = 1 return 1 The rabbit(3) call completes
n = 5 rabbit(4) = ? rabbit(3) = ? return ?	n = 4 rabbit(3) = 2 rabbit(2) = ? return ?	Follow the rabbit(2) call	
n = 5 rabbit(4) = ? rabbit(3) = ? return ?	n = 4 rabbit(3) = 2 rabbit(2) = ? return ?	n = 2 return 1	Base case: n = 2
n = 5 rabbit(4) = ? rabbit(3) = ? return ?	n = 4 rabbit(3) = 2 rabbit(2) = 1 return 3	n = 2 return 1	The rabbit(2) call completes
n = 5 rabbit(4) = 3 rabbit(3) = ? return ?	n = 4 rabbit(3) = 2 rabbit(2) = 1 return 3	n = 2 return 1	The rabbit(4) call completes
n = 5 rabbit(4) = 3 rabbit(3) = ? return ?	Follow the rabbit(3) call		
n = 5 rabbit(4) = 3 rabbit(3) = ? return ?	n = 3 rabbit(2) = ? rabbit(1) = ? return ?	Follow the rabbit(2) call	
n = 5 rabbit(4) = 3 rabbit(3) = ? return ?	n = 3 rabbit(2) = ? rabbit(1) = ? return ?	n = 2 return 1	Base case: n = 2
n = 5 rabbit(4) = 3 rabbit(3) = ? return ?	n = 3 rabbit(2) = 1 rabbit(1) = ? return ?	n = 2 return 1	The rabbit(2) call completes

<pre>n = 5 rabbit(4) = 3 rabbit(3) = ? return ?</pre>	<pre>n = 3 rabbit(2) = 1 rabbit(1) = ? return ?</pre>	Follow the rabbit(1) call		
<pre>n = 5 rabbit(4) = 3 rabbit(3) = ? return ?</pre>	<pre>n = 3 rabbit(2) = 1 rabbit(1) = ? return ?</pre>	<pre>n = 1 return 1</pre>	Base case: n = 1	
<pre>n = 5 rabbit(4) = 3 rabbit(3) = ? return ?</pre>	<pre>n = 3 rabbit(2) = 1 rabbit(1) = 1 return 2</pre>	<pre>n = 1 return 1</pre>	The rabbit(1) call completes	
<pre>n = 5 rabbit(4) = 3 rabbit(3) = 2 return 5</pre>	<pre>n = 3 rabbit(2) = 1 rabbit(1) = 1 return 2</pre>	<pre>n = 1 return 1</pre>	The rabbit(3) call completes	
<pre>n = 5 rabbit(4) = 3 rabbit(3) = 2 return 5</pre>	<pre>n = 3 rabbit(2) = 1 rabbit(1) = 1 return 2</pre>	<pre>n = 1 return 1</pre>	The rabbit(5) call completes and the value 5 is returned to the calling function	

2b The call `countDown(5)` produces the following box trace:

<pre>n = 5 cout << "5 ";</pre>	The value 5 is printed. Follow the call to <code>countDown(4)</code>				
<pre>n = 5 cout << "5 ";</pre>	<pre>n = 4 cout << "4 ";</pre>	The value 4 is printed. Follow the call to <code>countDown(3)</code>			
<pre>n = 5 cout << "5 ";</pre>	<pre>n = 4 cout << "4 ";</pre>	<pre>n = 3 cout << "3 ";</pre>	The value 3 is printed. Follow the call to <code>countDown(2)</code>		
<pre>n = 5 cout << "5 ";</pre>	<pre>n = 4 cout << "4 ";</pre>	<pre>n = 3 cout << "3 ";</pre>	<pre>n = 2 cout << "2 ";</pre>	The value 2 is printed. Follow the call to <code>countDown(1)</code>	
<pre>n = 5 cout << "5 ";</pre>	<pre>n = 4 cout << "4 ";</pre>	<pre>n = 3 cout << "3 ";</pre>	<pre>n = 2 cout << "2 ";</pre>	<pre>n = 1 cout << "1 ";</pre>	The value 1 is printed. Follow the call to <code>countDown(0)</code>
<pre>n = 5 cout << "5 ";</pre>	<pre>n = 4 cout << "4 ";</pre>	<pre>n = 3 cout << "3 ";</pre>	<pre>n = 2 cout << "2 ";</pre>	<pre>n = 1 cout << "1 ";</pre>	<pre>n = 0 cout << endl; return</pre>
					The end of line is printed and the <code>countDown(0)</code> call completes.
<pre>n = 5 cout << "5 ";</pre>	<pre>n = 4 cout << "4 ";</pre>	<pre>n = 3 cout << "3 ";</pre>	<pre>n = 2 cout << "2 ";</pre>	<pre>n = 1 cout << "1 ";</pre>	<pre>n = 0 cout << endl; return</pre>
					The <code>countDown(1)</code> call completes.

```
n = 5
cout << "5 ";
```

```
n = 4
cout << "4 ";
```

```
n = 3
cout << "3 ";
```

```
n = 2
cout << "2 ";
```

```
n = 1
cout << "1 ";
```

```
n = 0
cout << endl;
return
```

The countDown(2) call completes.

```
n = 5
cout << "5 ";
```

```
n = 4
cout << "4 ";
```

```
n = 3
cout << "3 ";
```

```
n = 2
cout << "2 ";
```

```
n = 1
cout << "1 ";
```

```
n = 0
cout << endl;
return
```

The countDown(3) call completes.

```
n = 5
cout << "5 ";
```

```
n = 4
cout << "4 ";
```

```
n = 3
cout << "3 ";
```

```
n = 2
cout << "2 ";
```

```
n = 1
cout << "1 ";
```

```
n = 0
cout << endl;
return
```

The countDown(4) call completes.

```
n = 5
cout << "5 ";
```

```
n = 4
cout << "4 ";
```

```
n = 3
cout << "3 ";
```

```
n = 2
cout << "2 ";
```

```
n = 1
cout << "1 ";
```

```
n = 0
cout << endl;
return
```

The countDown(5) call completes and returns to the calling function.

```
3  /** Returns the sum of the first n integers in the array anArray.
    Precondition: 0 <= n <= size of anArray.
    Postcondition: The Sum of the first n integers in the
                    array anArray are returned. The contents of
                    anArray and the value of n are unchanged. */
int computeSum(const int anArray[], int n)
{  // base case
  if (n <= 0)
    return 0;
  else  // reduce the problem size
    return anArray[n - 1] + computeSum(anArray, n - 1);
}  // end computeSum
```

```
4int sum (int start, int end )
{
  if (start == end)
    return end;
  else
    return start + sum(start + 1, end);
}
```

```
5#include <string>
   using namespace std;

   // -----
   // Writes a character string backward.
   // Precondition: The string str contains size characters,
   //               where size >= 1.
   // Postcondition: str is written backward, but remains
   //               unchanged.
   // -----
void writeBackward(string str, int size)
{   // base case
    if (size == 1)
        cout << str[0];

    // else, write rest of string
    else if (size > 1)
    {
        cout << str[size - 1];
        writeBackward(str, size - 1);
    }
    // size <= 0 do nothing;
}   // end writeBackward
```

6 The recursive method does not have a base case. As such, it will never terminate.

```
7   // -----
   // Prints out the integers from 1 through n as a
   // comma separated list followed by a newline.
   // Precondition: n >= 0 and limit == n.
   // Postcondition: The integers from 1 through n
   //               are printed out followed by a
   //               newline.
   // -----
void printIntegers(int n, int limit)
{
    if (n > 0)
    {   // print out the rest of the integers
        printIntegers(n - 1, limit);

        // now print out this integer
        cout << n;

        // test for end of string
        if (n != limit)
            cout << ", ";
        else
            cout << "." << endl;           // end of string
    }   // end if

    // n <= 0 do nothing
}   // end printIntegers
```

```

8int getSum(int n)
    {
        int result;
        if (n == 1)
            result = 1;
        else
            result = n + sum (n-1);
        return result;
    } // end getSum

```

```

9const int NUMBER_BASE = 10;

```

```

    /** Displays the decimal digits of number in reverse order.
        Precondition: number >= 0.
        Postcondition: The decimal digits of number are printed in reverse order.
                        This function does not output a newline character at the
                        end of a string. */
void reverseDigits(int number)
{ // check for input bounds
    if (number >= 0)
    { // base case
        if (number < NUMBER_BASE)
            cout << number;
        else
        { // print out rightmost digit
            cout << number % NUMBER_BASE;

            // pass remainder of digits to next call
            reverseDigits(number / NUMBER_BASE);
        } // end if
    } // end if
} // end reverseDigits

```

10a

```

    /** Displays a line of n characters, where ch is the character.
        Precondition: n >= 0.
        Postcondition: A line of n characters ch is output
                        followed by a newline. */
void writeLine(char ch, int n)
{ // base case
    if (n <= 0)
        cout << endl;

    // write rest of line
    else
    {
        cout << ch;

        writeLine(ch, n - 1);
    } // end if
} // end writeLine

```

10b

```

/** Displays a block of m rows by n columns of character ch.
    Precondition: m >= 0 and n >= 0.
    Postcondition: A block of m rows by n columns of
                    character ch is printed. */
void writeBlock(char ch, int m, int n)
{  if (m > 0)
    {
        writeLine(ch, n);           // write first line
        writeBlock(ch, m - 1, n);   // write rest of block
    }

    // base case: m <= 0 do nothing.
} // end writeBlock

```

11Running the given program produces the following output:

```

Enter: a = 1 b = 7
Enter: a = 1 b = 3
Leave: a = 1 b = 3
Leave: a = 1 b = 7
2

```

12Running the given program produces the following output:

```

Enter: first = 1 last = 30
Enter: first = 1 last = 14
Enter: first = 1 last = 6
Enter: first = 4 last = 6
Leave: first = 4 last = 6
Leave: first = 1 last = 6
Leave: first = 1 last = 14
Leave: first = 1 last = 30
5

```

13The algorithm first checks to see if n is a positive number: if not it immediately terminates. Otherwise, an division of n by 8 is taken and if the result is greater than 0 (i.e., the function is called again with an argument. This call processes that portion of the number composed of higher powers of 8. After this call, the residue for the current power, is printed.

The given function computes the number of times 80, 81, 82, . . . n . These values are stacked recursively and are printed out in the reverse of the order of computation. The following is the hand execution:

```

with
n = 100
displayOctal(12)
    n = 12
    displayOctal(1)
        n = 1
        cout << 1
    cout << 4
cout << 4

```

Output: 144

14 Even though the precondition states that n is nonnegative, there is no actual code to keep a negative value from being used as the argument in the function.

A call to the function `f` will produce a further call to `f` with a negative argument. Because 3 is not within the subrange of 0 to 2, the default case will execute, and the function will attempt to evaluate `f(1)`. Because the value for `f(n)` is based on the values for `f(n-2)` and `f(n-4)`, if n is even, its addends will be the next two smaller even integers; likewise, if n is odd, `f(n)`'s addends will be the next two smaller odd integers. Thus any odd nonnegative integer will eventually cause `f` to evaluate `f(3)`.

Theoretically, calling `f` with an odd integer will cause an infinite sequence of function calls. On a practical level, the computer's run-time stack will overflow, or an integer underflow will happen.

The following is the exact output of the program:

```
Function entered with n = 8
Function entered with n = 6
Function entered with n = 4
Function entered with n = 2
Function entered with n = 0
Function entered with n = 2
Function entered with n = 4
Function entered with n = 2
Function entered with n = 0
The value of f(8) is 27
```

15 The following output is produced when `x` is a value argument:

```
6 2
7 1
8 0
8 0
7 1
6 2
```

Changing `x` to a reference argument produces:

```
6 2
7 1
8 0
8 0
8 1
8 2
```

16a The call `binSearch(5)` produces the following box trace:

```
value = 5
first = 1
last = 8
mid = 4
value < anArray[4]
```

```
value = 5
first = 1
last = 3
mid = 2
value = anArray[2]
return 2
```


16b The call `binSearch(13)` produces the following box trace:

```
value = 13
first = 1
last = 8
mid = 4
value > anArray[4]
```

```
value = 13
first = 5
last = 8
mid = 6
value < anArray[6]
```

```
value = 13
first = 5
last = 5
mid = 5
value < anArray[5]
```

```
value = 13
first = 5
last = 4
first > last
return 0
```

16c The call `binSearch(16)` produces the following box trace:

```
value = 16
first = 1
last = 8
mid = 4
value > anArray[4]
```

```
value = 16
first = 5
last = 8
mid = 6
value < anArray[6]
```

```
value = 16
first = 5
last = 5
mid = 5
value > anArray[5]
```

```
value = 16
first = 6
last = 5
first > last
return 0
```

17a For a binary search to work, the array must first be sorted in either ascending or descending order.

b The index is $(0 + 101)/2 = 50$.

c Number of comparisons = $\lceil \lg 101 \rceil = 6$.

18a

```
/** Returns the value of x raised to the nth power.
    Precondition: n >= 0
    Postcondition: The computed value is returned. */
double power1(double x, int n)
{ double result = 1;           // value of x^0

    while (n > 0)               // iterate until n == 0
    { result *= x;
      n--;
    }
    return result;
} // end power1
```

18b

```
/** Returns the value of x raised to the nth power.
    Precondition: n >= 0
    Postcondition: The computed value is returned. */
double power2(double x, int n)
{ // base case
  if (n == 0)
    return 1;

  // else, multiply x by rest of computation
  else
    return x * power2(x, n-1);
} // end power2
```

```

18c  /** Returns the value of x raised to the xth power.
      Precondition:  n >= 0
      Postcondition: The computed value is returned. */
double power3(double x, int n)
{
    if (n == 0)
        return 1;
    else
    { // do this computation only once!!
        double halfPower = power3(x, n/2);

        // if n is even...
        if (n % 2 == 0)
            return halfPower * halfPower;

        // if n is odd...
        else
            return x * halfPower * halfPower;
    }
} // end power3

```

18d The following table lists the number of multiplications performed by each of the algorithms for computing the values on the top line:

	332	319
power1	32	19
power2	32	19
power3	7	8

18e The following table lists the number of recursive calls made by each of the algorithms indicated in order to perform the computation on the inputs given on the top line:

	332	319
power2	32	19
power3	6	5

19 Maintain a count of the recursive depth of each call by passing this count as an additional parameter to the function call and print that many spaces or tabs in front of each message:

```

/** Computes a term in the Fibonacci sequence.
Precondition: n is a positive integer and tab = 0.
Postcondition: The progress of the recursive function call is displayed
as a sequence of increasingly nested blocks. The function
returns the nth Fibonacci number. */
int rabbit(int n, int tab)
{
    int value;

    // Indent the proper distance for this block
    for (int i = 0; i < tab; i++)
        cout << '\t';

    // Display status of call
    cout << "Enter: n = " << n << endl;

    if (n <= 2)
        value = 1;

    else // n > 2, so n-1 > 0 and n-2 > 0
        // indent by one for next call
        value = rabbit(n-1, tab+1) + rabbit(n-2, tab+1);

    // Indent the proper distance for this block
    for (i = 0; i < tab; i)
        cout << '\t';

    // Display status of call
    cout << "Leave: n = " << n << " value = " << value << endl;

    return value;
}

```

20a $f(6)$ is 8; $f(7)$ is 11; $f(12)$ is 95; $f(15)$ is 320.

20b Since we only need the five most recently computed values, we will maintain a "circular" five-element array indexed modulus 5.

```

// Pre: n > 0.
int fOfN(int n)
{
    int last5[5] = {1, 1, 1, 3, 5};

    for (int i = 5; i < n; i++)
    {
        int fi = last5[(i - 1) % 5] + 3 * last5[(i - 5) % 5];

        // Replace entry in last5
        last5[i % 5] = fi; // f(i) = f(i - 1) + 3 x f(i - 5)
    } // end for

    return last5[(n - 1) % 5];
} // end fOfN

```

21aA function to compute $n!$ iteratively:

```
long fact(int n)
{  int i;
   long result;

   if (n < 1)      // base case
       result = 0;
   else
   {
       result = 1;
       for (i = 2; i <= n; i++)
           result *= i;
   }    // end if

   return result;
}    // end fact
```

21bA simple iterative solution to writing a string backwards:

```
#include <string>

void writeBackward(string str)
{
    for (int i = str.size() - 1; i >= 0; i--)
        cout << str[i];

    cout << endl;
}    // end writeBackward
```

21cA function to perform an iterative binary search:

```
/** Searches a sorted array and returns the index in the
    array corresponding to the value key if key is in the
    array, -1 otherwise.
    Precondition:  high is sorted in ascending order.
                   low = 0 and high = the size of high - 1.
    Postcondition: If key is found, its location index
                   in high is returned, else -1 is returned. */
int binarySearch(int anArray[], int key, int low, int high)
{
    int mid, result;
    while (low < high)
    {  mid = (low + high)/2;

       if (anArray[mid] == key)
       {  low = mid;
          high = mid;
        }
       else if (anArray[mid] < key)
           low = mid + 1;      // search the upper half
       else
           high = mid - 1;     // search the lower half
    }    // end while

    if (low > high)
        result = -1;          // if not found, return -1
    else if (anArray[low] != key)
        result = -1;
```

```

    else
        result = low;

    return result;
} // end binarySearch

```

21d We implement a function to find the kth smallest entry in an array using an integer array and a selection sort times. k We assume a standard integer swap function.

```

int kSmall(int k, int anArray[], int size)
{
    for (int i=0; i<k; i++)
        for (int j = i+1; j < size; j++)
            if (anArray[j] < anArray[i])
                swap(anArray[i], anArray[j]);

    return anArray[k-1];
} // end kSmall

```

22 The for loop invariant is:

$$3 \leq i \leq n$$

So the sum = $\sum_{i=3}^n i$ *irabbi il rabbit*

23a We must verify that the equation holds for both the base case and the recursive case.

For the base case, $\gcd(a, b) = b$. Then, $a \bmod b = 0$ and, since $0/0$ for all b , $\gcd(b, 0) = b$. Hence, $\gcd(b, a \bmod b) = b$.

For the recursive case, $\gcd(a, b) = d$, i.e. $a = dj$ and $b = dk$ for integers j and k . Now there exists integer q such that $a \bmod b = q$, where q is an integer. Then, $a = bq + r$ and, so $-dj = dkq + j$. Then, $(n/d) = kq + j$ where $kq + j$ is an integer. d divides a i.e. d divides $a \bmod b$.

To show that d is the greatest common divisor of a and $a \bmod b$, suppose for contradiction there exists integer g such that $b = gr$ and $a \bmod b = gs$ for integers r and s . Then, $g \mid (-a) \bmod b = q'$ where q' is an integer. So $a = grq'$ i.e. $a = g(s - rq')$. Thus g divides a and g divides b . But $\gcd(a, b) = d$ by hypothesis. Therefore, $\gcd(b, a \bmod b) = d$.

The proof is symmetrical when $\gcd(b, a \bmod b) = d$ is taken for the hypothesis.

23b If $b > a$ in the call $\gcd(a, b)$, $a \bmod b = a$ and so the recursive call effectively reverses the arguments.

23c When $a > b$, the argument associated with the parameter increases in the next recursive call. If the next recursive call will swap the arguments so that $a < b$, then, the first argument will eventually equal the second so eventually $a \bmod b$ will be 0. and

$$24a \quad c(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ \sum_{i=1}^{n-1} (c(n-i) + 1) & \text{if } n > 2 \end{cases}$$

$$24b \quad c(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ c(n-1) + c(n-2) & \text{if } n > 2 \end{cases}$$

$$25 \quad \text{Acker}(1, 2) = 4$$

```
int acker(int m, int n)
{
    int result;

    if (m == 0)
        result = n+1;

    else if (n == 0)
        result = acker(m-1, 1);

    else
        result = acker(m-1, acker(m, n-1));

    return result;
} // end acker
```