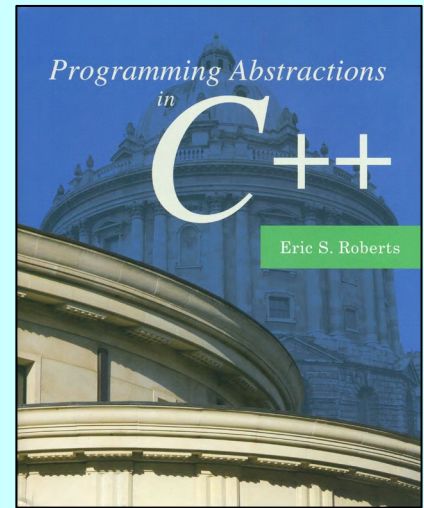


CHAPTER 7

Introduction to Recursion

*And often enough, our faith beforehand in a certain result is
the only thing that makes the result come true.*

—William James, *The Will to Believe*, 1897



[7.1 A simple example of recursion](#)

[7.2 The factorial function](#)

[7.3 The Fibonacci function](#)

[7.4 Checking palindromes](#)

[7.5 The binary search algorithm](#)

[7.6 Mutual recursion](#)

[7.7 Thinking recursively](#)

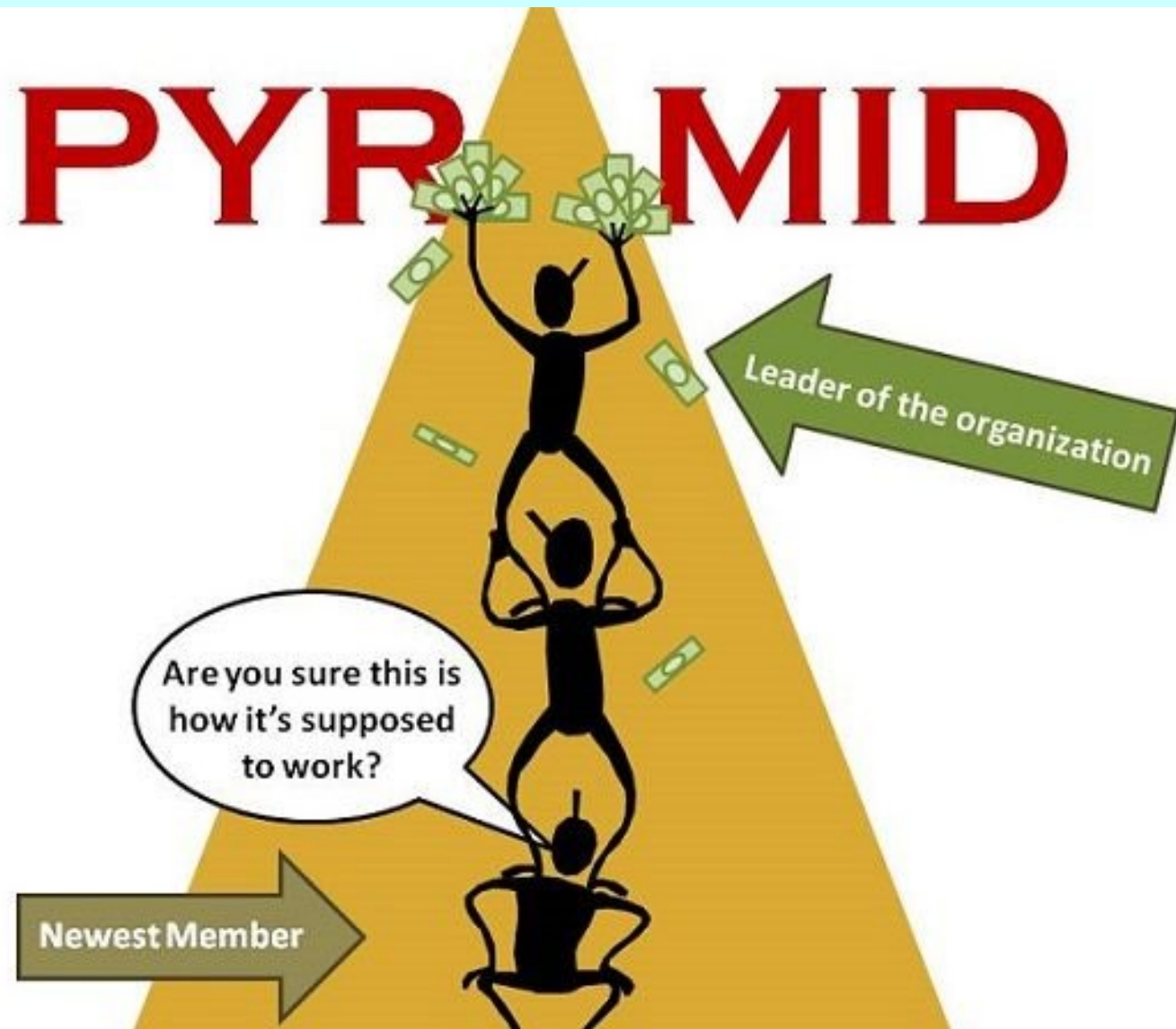
Recursion

- One of the most important “Great Ideas” in programming is the concept of *recursion*, which is the process of solving a problem by dividing it into smaller sub-problems of *the same form*. The emphasized phrase is the essential characteristic of recursion; without it, all you have is a description of *top-down design/stepwise refinement* we have studied earlier. It is a special kind of *divide and conquer*.
- The fact that recursive decomposition generates sub-problems that have the same form as the original problem means that recursive programs will *use the same function or method* to solve sub-problems *at different levels of the solution*. In terms of the structure of the code, the defining characteristic of recursion is having *functions that call themselves*, directly or indirectly, as the decomposition process proceeds.

A Simple Illustration of Recursion

- Suppose that you are the national fundraising director for a charitable organization and need to raise \$1,000,000.
- One possible approach is to find a wealthy donor and ask for a single \$1,000,000 contribution. The problem with that strategy is that individuals with the necessary combination of means and generosity are difficult to find. Donors are much more likely to make contributions in the \$100 range.
- Another strategy would be to ask 10,000 friends for \$100 each. Unfortunately, most of us don't have 10,000 friends.
- There are, however, more promising strategies. You could, for example, find ten regional coordinators and charge each one with raising \$100,000. Those regional coordinators could in turn delegate the task to local coordinators, each with a goal of \$10,000, continuing the process reached a manageable contribution level.

A Simple Illustration of Recursion



A Pseudocode Fundraising Strategy

- If you were to implement the fundraising strategy in the form of a C++ function, it would look something like this:

```
void collectContributions(int n) {  
    if (n <= 100) {  
        Collect the money from a single donor.  
    } else {  
        Find 10 volunteers.  
        Get each volunteer to collect n/10 dollars.  
        Combine the money raised by the volunteers.  
    }  
}
```

what makes this strategy **recursive** is that the line:

Get each volunteer to collect n/10 dollars.

will be implemented using the following recursive call:

`collectContributions(n/10);`



The Recursive Paradigm

- Most recursive methods you encounter in an introductory course have bodies that fit the following general pattern:

```
if (test for a simple case) {  
    Compute and return the simple solution without using recursion.  
} else {  
    Divide the problem into one or more sub-problems that have the same form.  
    Solve each of the sub-problems by calling this method recursively.  
    Return the solution by reassembling the results of the various sub-problems.  
}
```

- Finding a recursive solution is mostly a matter of figuring out how to break it down so that it fits the paradigm. When you do so, you must do two things:
 1. Identify **simple cases** that can be solved **without recursion**.
 2. Find a **recursive decomposition** that breaks each instance of the problem into **simpler** sub-problems of **the same type**, which you can then solve by applying the method recursively.

Recursive Functions

- The easiest examples of recursion to understand are functions in which the recursion is clear from the definition. As an example, consider the factorial function, which can be defined in either of the following ways:

$$n! = n \times (n - 1) \times (n - 2) \times \dots \times 3 \times 2 \times 1$$

$$n! = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ n \times (n - 1)! & \text{otherwise} \end{cases}$$

- The two definitions lead directly to different implementations, one **iteration**-based and the other **recursion**-based:

```
int fact(int n) {  
    int result = 1;  
    for (int i = 1; i <= n; i++) {  
        result *= i;  
    }  
    return result;  
}
```

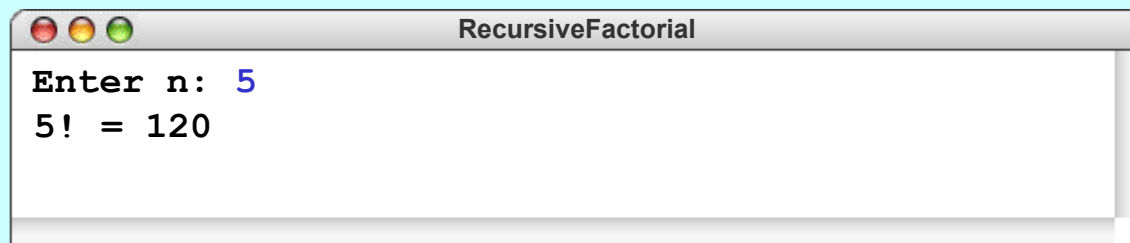
```
int fact(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fact(n - 1);  
    }  
}
```

Simulating the **fact** Function

```
int main() {  
    int fact(int n) {  
        int fact(int n) {  
            int fact(int n) {  
                int fact(int n) {  
                    int fact(int n) {  
                        int fact(int n) {  
                            if (n == 0) {  
                                return 1;  
                            } else {  
                                return n * fact(n - 1);  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

n

0



The Recursive “Leap of Faith”



recursive leap of faith.

Mathematical Induction

- Algorithmic analysis often requires proving properties that apply to all nonnegative integers. For example, the analysis of **selection sort** depended on the correctness of the following formula:

$$1 + 2 + 3 + \cdots + (N - 1) + N = \sum_{i=1}^N i = \frac{N \times (N + 1)}{2}$$

- The most important mathematical technique for proving the correctness of such formulas is **mathematical induction**, which consists of the following steps:
 - Prove the base case.* In the base case, you prove that the formula holds when N has the **basis value**, which is typically 0 or 1.
 - Prove the inductive case.* The second step is to show that assuming the formula holds for N , you can prove that it holds for $N + 1$.

Example of Mathematical Induction

To prove: $1 + 2 + 3 + \cdots + (N - 1) + N = \frac{N \times (N + 1)}{2}$

Base case: $0 = \frac{0 \times (0 + 1)}{2}$

Inductive case:

$$1 + 2 + 3 + \cdots + (N - 1) + N + (N + 1)$$

$$= \frac{N \times (N + 1)}{2} + (N + 1)$$

$$= \frac{N \times (N + 1)}{2} + \frac{(2N + 2)}{2}$$

$$= \frac{N^2 + 3N + 2}{2}$$

$$= \frac{(N + 1) \times (N + 2)}{2}$$

Recursion vs. Induction

- Both **induction** and **recursion** require you to make **a leap of faith**.
- When you write a recursive function, this leap consists of believing that all simpler instances of the function call will work without your paying any attention to the details.
- Making the inductive hypothesis requires much the same mental discipline.
- In both cases, you have to restrict your thinking to one level of the solution and not get sidetracked trying to follow the details all the way to the end.

The Fibonacci Function

- In 1202, the Italian mathematician Leonardo Fibonacci published a treatise entitled *Liber Abaci* in which he argued for the adoption of the Hindu-Arabic numbering system over the far more cumbersome system of Roman numerals still in use at the time.
- As one of the examples in *Liber Abaci*, Fibonacci asked a question about the growth pattern in a population of rabbits under the following, admittedly fanciful assumptions:
 - Rabbit pairs become fertile two months after their birth.
 - Fertile rabbit pairs produce a new pair every month.
 - Old rabbits never die but keep on reproducing.
- The sequence of values recording the population of rabbits turns up in a surprising number of contexts and is called the *Fibonacci sequence*.

Fibonacci's Rabbits



Jan		1
-----	---	---

Feb		1
-----	---	---

Mar	 	2
-----	---	---

Apr	 	3
-----	---	---

May	 	5
-----	---	---

Jun	 	8
-----	--	---

Jul	 	13
-----	--	----

Recursive Formulation of **fib**(n)

- The n^{th} term in the Fibonacci sequence has the following recursive definition:

$$\text{fib}(n) = \begin{cases} n & \text{if } n \text{ is 0 or 1} \\ \text{fib}(n - 1) + \text{fib}(n - 2) & \text{otherwise} \end{cases}$$

- As with factorial, the recursive definition leads directly to an implementation, as follows:

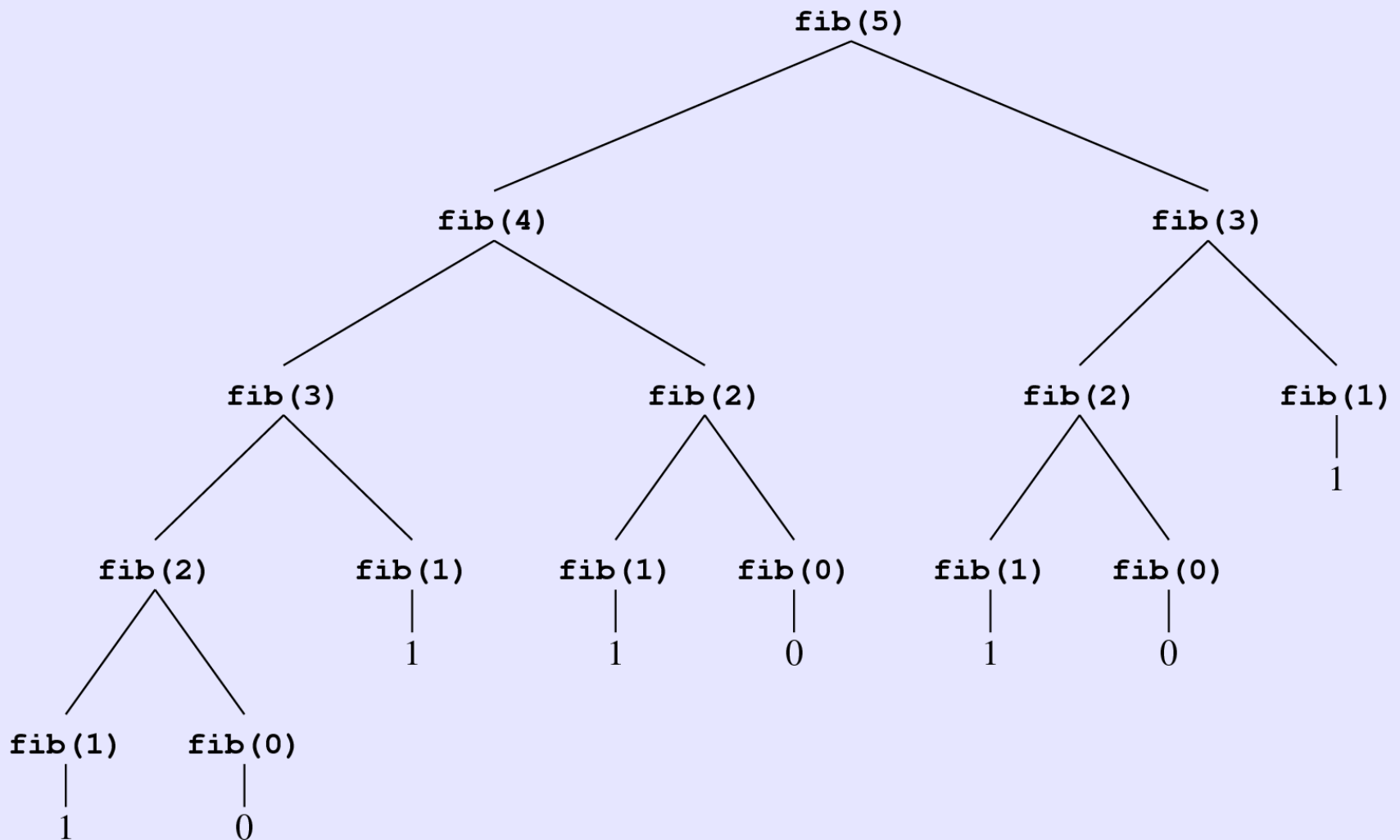
```
int fib(int n) {  
    if (n < 2) {  
        return n;  
    } else {  
        return fib(n - 1) + fib(n - 2);  
    }  
}
```

- Is this a good program?

Extremely inefficient! Why?

Recursive Formulation of `fib(n)`

FIGURE 7-2 Steps in the calculation of `fib(5)`



Improve `fib(n)`

- Use the analytical form?

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n .$$

- Use iterations?

```
int fib(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    int f0 = 0, f1 = 1, f2 = 1;  
    for (int i = 3; i <= n; i++) {  
        f0 = f1, f1 = f2, f2 = f0 + f1;  
    }  
    return f2;  
}
```

- Use another form of recursion?

```
int fib(int n) {  
    return additiveSequence(n, 0, 1);  
}  
int additiveSequence(int n, int t0, int t1) {  
    if (n == 0) return t0;  
    if (n == 1) return t1;  
    return additiveSequence(n - 1, t0, t1) + additiveSequence(n - 2, t0, t1);  
}
```

```
fib(5)  
= additiveSequence(5, 0, 1)  
= additiveSequence(4, 1, 1)  
= additiveSequence(3, 1, 2)  
= additiveSequence(2, 2, 3)  
= additiveSequence(1, 3, 5)  
= 5
```

Exercise: Greatest Common Divisor

Remember the Euclid's algorithm for computing the Greatest Common Divisor (GCD) of two integers, x and y ? Euclid's algorithm is usually implemented iteratively using code that looks like this:

```
int gcd(int x, int y) {  
    int r = x % y;  
    while (r != 0) {  
        x = y;  
        y = r;  
        r = x % y;  
    }  
    return y;  
}
```

Solution: A Recursive GCD Function

Rewrite this method so that it uses recursion instead of iteration, taking advantage of Euclid's insight that **the greatest common divisor of x and y is also the greatest common divisor of y and the remainder of x divided by y** , which shows the recursive nature of the problem.

```
int gcd(int x, int y) {  
    if (x % y == 0) {  
        return y;  
    } else {  
        return gcd(y, x % y);  
    }  
}
```

As is usually the case, the key to solving this problem is finding the **recursive decomposition** and defining appropriate **simple cases**.

Exercise: Recognizing Palindromes

- A *palindrome* is a word that reads identically backward and forward, such as “level” or “noon”.
- Write a C++ program `isPalindrome` that checks whether a string is a palindrome.

```
bool isPalindrome(string str) {  
    int n = str.length();  
    for (int i = 0; i < n / 2; i++) {  
        if (str[i] != str[n - i - 1]) return false;  
    }  
    return true;  
}
```

```
bool isPalindrome(string str) {  
    return str == reverse(str);  
}
```

- Efficiency vs. Readability

Palindromes

- Recursion is not limited to numeric examples and appears naturally in many other contexts. For example, we have written a program to check whether a string is a *palindrome* (e.g., “level”, “noon”, in Ch. 3), which can also be recognized recursively if one of the two following cases applies:
 - It is empty or contains a single character.
 - Its first and last characters match and enclose a palindrome.
- This decomposition gives rise to the following code:

```
bool isPalindrome(string str) {  
    int len = str.length();  
    if (len <= 1) {  
        return true;  
    } else {  
        return str[0] == str[len - 1] &&  
            isPalindrome(str.substr(1, len - 2));  
    }  
}
```

Palindromes

- You can improve the performance of `isPalindrome` by making the following changes:
 - Calculate the length of the argument only once.
 - Don't make a substring on each call (**use indices instead**).
- The improved code:

```
bool isSubstringPalindrome(string & str, int p1, int p2) {  
    if (p1 >= p2) {  
        return true;  
    } else {  
        return str[p1] == str[p2] &&  
            isSubstringPalindrome(str, p1 + 1, p2 - 1);  
    }  
}  
  
bool isPalindrome(string str) {  
    return isSubstringPalindrome(str, 0, str.length() - 1);  
}
```

Linear Search

- The simplest strategy for searching is to start at the beginning of the array and look at each element in turn. This algorithm is called *linear search*.
- Linear search is straightforward to implement, as illustrated in the following method that returns the first index at which the value **key** appears in **array**, or **-1** if it does not appear at all:

```
int linearSearch(int key, Vector<int> vec) {  
    for (int i = 0; i < vec.size(); i++) {  
        if (key == vec[i]) return i;  
    }  
    return -1;  
}
```

Searching for Area Codes

- To illustrate the efficiency of linear search, it is useful to work with a somewhat larger example.
- The example on the next slide works with an array containing many of the area codes assigned to the United States.
- The specific task in this example is to search this list to find the area code for the Silicon Valley area, which is 650.
- The linear search algorithm needs to examine each element in the array to find the matching value. As the array gets larger, the number of steps required for linear search grows in the same proportion.
- As you watch the slow process of searching for 650 on the next slide, try to think of a more efficient way in which you might search this particular array for a given area code.

Linear Search (Area Code Example)

201	202	203	205	206	207	208	209	210	212	213	214	215	216	217	218	219	224	225	228	229	231
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
234	239	240	248	251	252	253	254	256	260	262	267	269	270	276	281	283	301	302	303	304	305
22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
307	308	309	310	312	313	314	315	316	317	318	319	320	321	323	325	330	331	334	336	337	339
44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65
347	351	352	360	361	364	385	386	401	402	404	405	406	407	408	409	410	412	413	414	415	416
66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87
417	419	423	424	425	430	432	434	435	440	443	445	469	470	475	478	479	480	484	501	502	503
88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109
504	505	507	508	509	510	512	513	515	516	517	518	520	530	540	541	551	559	561	562	563	564
110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131
567	570	571	573	574	575	580	585	586	601	602	603	605	606	607	608	609	610	612	614	615	616
132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153
617	618	619	620	623	626	630	631	636	641	646	650	651	660	661	662	678	682	701	702	703	704
154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
706	707	708	712	713	714	715	716	717	718	719	720	724	727	731	732	734	740	754	757	760	762
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197
763	765	769	770	772	773	774	775	779	781	785	786	801	802	803	804	805	806	808	810	812	813
198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219
814	815	816	817	818	828	830	831	832	835	843	845	847	848	850	856	857	858	859	860	862	863
220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241
864	865	870	878	901	903	904	906	907	908	909	910	912	913	914	915	916	917	918	919	920	925
242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263
928	931	936	937	940	941	947	949	951	952	954	956	959	970	971	972	973	978	979	980	985	989
264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285

The Idea of Binary Search

- The fact that the area code array is in ascending order makes it possible to find a particular value much more efficiently.
- The key insight is that you get more information by starting at the middle element than you do by starting at the beginning.
- When you look at the middle element in relation to the value you're searching for, there are three possibilities:
 - If the value you are searching for is greater than the middle element, you can discount every element in the first half of the array.
 - If the value you are searching for is less than the middle element, you can discount every element in the second half of the array.
 - If the value you are searching for is equal to the middle element, you can stop because you've found the value you're looking for.
- You can repeat this process on the elements that remain after each cycle. Because this algorithm proceeds by dividing the list in half each time, it is called *binary search*.

Binary Search (Area Code Example)

Binary search needs to look at only eight elements to find 650.

201	202	203	205	206	207	208	209	210	212	213	214	215	216	217	218	219	224	225	228	229	231
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
234	239	240	248	251	252	253	254	256	260	262	267	269	270	276	281	283	301	302	303	304	305
22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
307	308	309	310	312	313	314	315	316	317	318	319	320	321	323	325	330	331	334	336	337	339
44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65
347	351	352	360	361	364	385	386	401	402	404	405	406	407	408	409	410	412	413	414	415	416
66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87
417	419	423	424	425	430	432	434	435	440	443	445	469	470	475	478	479	480	484	501	502	503
88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109
504	505	507	508	509	510	512	513	515	516	517	518	520	530	540	541	551	559	561	562	563	564
110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131
567	570	571	573	574	575	580	585	586	601	602	603	605	606	607	608	609	610	612	614	615	616
132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153
617	618	619	620	623	626	630	631	636	641	646	650	651	660	661	662	678	682	701	702	703	704
154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
706	707	708	712	713	714	715	716	717	718	719	720	724	727	731	732	734	740	754	757	760	762
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197
763	765	769	770	772	773	774	775	779	781	785	786	801	802	803	804	805	806	808	810	812	813
198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219
814	815	816	817	818	828	830	831	832	835	843	845	847	848	850	856	857	858	859	860	862	863
220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241
864	865	870	878	901	903	904	906	907	908	909	910	912	913	914	915	916	917	918	919	920	925
242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263
928	931	936	937	940	941	947	949	951	952	954	956	959	970	971	972	973	978	979	980	985	989
264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285

Binary Search

```
/*
 * Function: binarySearch
 * Usage: int index = binarySearch(key, vec, p1, p2);
 * -----
 * Searches for the specified key in the Vector<string> vec, looking
 * only at indices between p1 and p2, inclusive. The function returns
 * the index of a matching element, or -1 if no match is found.
 */

int binarySearch(string key, Vector<string> & vec, int p1, int p2) {
    if (p1 > p2) return -1;
    int mid = (p1 + p2) / 2;
    if (key == vec[mid]) return mid;
    if (key < vec[mid]) {
        return binarySearch(key, vec, p1, mid - 1);
    } else {
        return binarySearch(key, vec, mid + 1, p2);
    }
}
```

Example: TwoLetterWords

```
/* Program to generate all the two-letter English words */
#include <iostream>
#include "lexicon.h"
#include "foreach.h" /* range-based for in Stanford */
using namespace std;
int main() {
    Lexicon english("EnglishWords.txt");
    string word("aa");
    for (char c0 = 'a'; c0 <= 'z'; c0++) {
        word[0] = c0;
        for (char c1 = 'a'; c1 <= 'z'; c1++) {
            word[1] = c1;
            if (english.contains(word)) cout << word << endl;
        }
    }
    foreach (string word in english) { /* Stanford */
        for (string word : english) { /* Standard C++11 */
            if (word.length() == 2) cout << word << endl;
        }
    }
    return 0;
}
```

- Question: which strategy is better?
depending on the speed of search.



Thinking Recursively

- Writing recursive programs requires developing a new way of thinking about problems that focuses more on **a holistic view** of the problem than on the details by adopting the **recursive leap of faith**.
- The following checklist may prove helpful:
 - Does your implementation begin by checking for simple cases?
 - Have you solved the simple cases correctly?
 - Does your recursive decomposition make the problem simpler?
 - Does the simplification eventually reach the simple cases?
 - Do the recursive calls in your function represent sub-problems that are truly identical in form to the original?
 - Do the solutions to the recursive sub-problems provide a complete solution to the original problem?
- Avoid **nonterminating recursion** (stack overflow), the recursive analogue of the infinite loop.

Mutual Recursion

- In some cases, recursive processes involve more than a single function. Instead, one function calls another, which calls another, which eventually circles back around to the original function. Such situations are called *mutual recursion*.
- The text uses the following simple example to illustrate mutual recursion, although it is *inefficient* and seems *unnecessary*!

```
bool isOdd(unsigned int n) {  
    return !isEven(n);  
}  
  
bool isEven(unsigned int n) {  
    if (n == 0) {  
        return true;  
    } else {  
        return isOdd(n - 1);  
    }  
}
```

Thinking Recursively

```
bool isOdd(unsigned int n) {  
    return !isEven(n);  
}  
  
bool isEven(unsigned int n) {  
    return !isOdd(n);  
}
```



```
bool isOdd(unsigned int n) {  
    return isEven(n-1);  
}  
  
bool isEven(unsigned int n) {  
    return isOdd(n-1);  
}
```



```
bool isOdd(unsigned int n) {  
    if (n == 1) return true;  
    return isEven(n-1);  
}  
  
bool isEven(unsigned int n) {  
    if (n == 0) return true;  
    return isOdd(n-1);  
}
```



```
bool isOdd(unsigned int n) {  
    if (n == 0) return false;  
    return isEven(n-1);  
}  
  
bool isEven(unsigned int n) {  
    if (n == 0) return true;  
    return isOdd(n-1);  
}
```


Thinking Recursively

```
bool isOdd(unsigned int n) {  
    return !isEven(n);  
}  
  
bool isEven(unsigned int n) {  
    if (n == 0) return true;  
    return isOdd(n-1);  
}
```

```
bool isOdd(unsigned int n) {  
    return !isEven(n);  
}  
  
bool isEven(unsigned int n) {  
    if (n == 0) return true;  
    return !isEven(n-1);  
}
```

```
bool isOdd(unsigned int n) {  
    return !isEven(n);  
}  
  
bool isEven(unsigned int n) {  
    if (n == 0) return true;  
    return isEven(n-2);  
}
```



```
bool isOdd(unsigned int n) {  
    return !isEven(n);  
}  
  
bool isEven(unsigned int n) {  
    if (n == 0) return true;  
    if (n == 1) return false;  
    return isEven(n-2);  
}
```

The End