Chapter 2-4, 5, 6, & 7, Lecture notes

2.4 Recursion with Array – working with an array, recursion can be practical and powerful

2.4.1 Writing an Array's Entries in Backward Order – similar to writing a string backwards

2.4.1.1 Don't actually remove the character from a character array but change where the last character is

2.4.1.2 So in printing array[0 … last] you could pass the array[0 … last-1]

2.4.1.3 So the function would look like

```
/** Writes the characters in an array backward.
   @pre     The array anArray contains size characters, where size >= 0.
   @post    None.
   @param  anArray   The array to write backward.
   @param  first    The index of first character in array.
   @param last      The index of last charater in the array. */
void writeArrayBackward(const char anArray[], int first, int last)
{
   if (first <= last)
   {
      // Write last character
      cout << anArray[last];

      // Write rest of array backward
      writeArrayBackward, first, last – 1);
   }  // end if

   // first > last is base case – do nothing
} // end writeArrayBackward
```

2.4.2 Binary Search

2.4.2.1 Searching occurs frequently – typically for a particular entry

2.4.2.2 Take the dictionary problem with words in sorted order and change it slightly so that it will search the array of words for a **target** – like so:

```
binarySearch(anArray: ArrayType, target: ValueType)

   if (anArray is of size 1)
      Determine if anArray's value is equal to target
   else
   {
      Find the midpoint of anArray
      Determine which half of anArray contains target
```

```
  if (target is in the first half of anArray)
    binarySearch(first half of anArray, target)
  else
    binarySearch(second half of anArray, target)
}
```

2.4.2.3 The concept is sound but what about the details? Like so:

2.4.2.3.1 **How will you pass half of anArray to recursive calls to binarySearch?**

2.4.2.3.1.1 In C++, passing the array costs nothing (it is a call by reference) – need to have the first and last index and the target passed, like so:

```
binarySearch(anArray, first, last, target)
```

2.4.2.3.1.2 To get the middle point or midpoint, you do:

```
mid = (first + last) / 2
```

2.4.2.3.1.3 Then call the function for the *first half of* anArray like:

```
binarySearch(anArray, first, mid – 1, target)
```

2.4.2.3.1.4 Or call the function for the *second half of* anArray like:

```
binarySearch(anArray, mid + 1, last, target)
```

2.4.2.3.2 **How do you determine which half of array contains target?**

2.4.2.3.2.1 With *if (target is in the first half of* anArray) should be:

2.4.2.3.2.2 if (target < anArray[mid])

2.4.2.3.2.3 But there is no test for equality, therefore you need to check for equality BEFORE testing for the correct half of anArray

2.4.2.3.2.4 Also need to rethink the base case

2.4.2.3.3 **What should the base case be?**

2.4.2.3.3.1 Size of the array becoming a 1 can be a base case

2.4.2.3.3.2 However, there are two tests for base case that is clearer

- first > last. You will reach this base case when target is not in the original array
- target == anArray[mid]. You will reach this base case when target is in original array

2.4.2.3.3.3 The base cases are a bit different from those encountered thus far – in a sense, the algorithm determines the answer to problem from the base case it reaches

2.4.2.3.4 **How will binarySearch indicate result of search?**

2.4.2.3.4.1 If binarySearch successfully locates the target in the array, it could return index of array value that is equal to target

2.4.2.3.4.2 Since the index cannot be negative, binarySearch could return a negative value if it doesn't find target in array

2.4.2.4 The C++ function implements the above details:

```
/** Searches the array anArray[first] through anArray[last]
    for a given value by using a binary search.
 @pre   0 <= first, last <= SIZE – 1, where SIZE is the
        maximum size of the array, and anArray[first] <=
        anArray[first + 1] <= … <= anArray[last].
 @post  anArray is unchanged and either anArray[index] contains
        the given value or index == -1.
 @param anArray   The array to search.
 @param first     The low index to start searching from.
 @param last      The high index to stop searching at.
 @return   Either index, such that anArray[index] == target, or -1
           for not found */
int binarySearch(const int anArray[], int first, int last, int target)
{
   int index;
   if (first > last)
      index = -1;   // target not in original array
   else
   {
      // If target is in anArray,
      // anArray[first] <= target <= anArray[last]
      int mid = first + (last – first) / 2;
      if (target == anArray[mid])
         index = mid; // target found at anArray[mid]
      else if (target < anArray[mid])
         // Point X
         index = binarySearch(anArray, first, mid – 1, target);
      else
         // Point Y
         index = binarySearch(anArray, mid + 1, last, target);
   } // end if
   return index;
```

} // end binarySearch

2.4.2.4.1 Note that the target has to be in the correct half like:

anArray[first] target anArray[last]

2.4.2.4.2 The following is a box trace for binarySearch

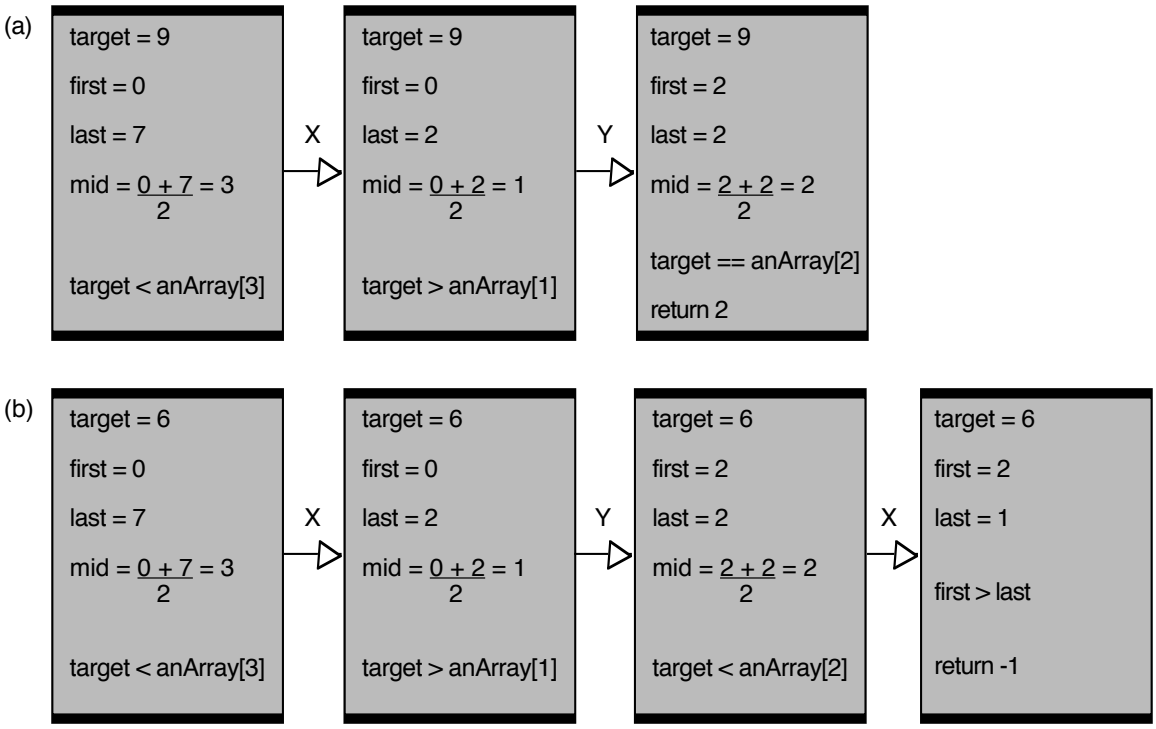

Figure 2-10, Box traces of binarySearch with anArray = <1, 5, 9, 12, 15, 21, 29, 31>:
(a) a sucessful search for 9; (b) an unsuccessful search for 6

2.4.2.4.3 The parameter of anArray, is not a value argument nor a local variable

2.4.2.4.3.1 It is a reference parameter (default mode with a C++ array)

2.4.2.4.3.2 The only thing that is passed with anArray is the address to the array

2.4.2.4.3.3 Figure 2-11 shows the box trace that has anArray pointing to the same array

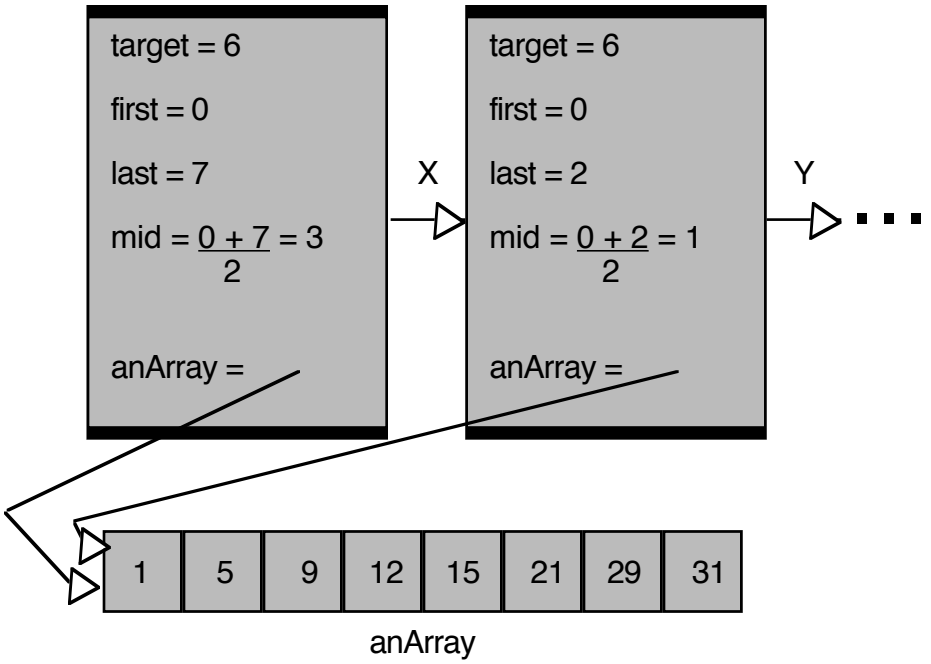

FIgure 2-11, Box trace with a reference argument

2.4.3 Finding the Largest Value in an Array

2.4.3.1 If you have an array of anArray you could easily make an iterative solution to find the largest value

2.4.3.2 But let's do a recursive solution, like so:

if (anArray has only one item)
    maxArray(anArray) is the item in anArray
else if (anArray has more than one item)
    maxArray(anArray) is the maximum of
        maxArray(left half of anArray) and
        maxArray(right half of anArray)

2.4.3.3 That is the divide-and-conquer model that the binary search used

2.4.3.4 Also there is more than one recursive call but instead of calling only one or the other half, it does both

2.4.3.4.1 Binary search only calls one, the largest value does both

2.4.3.4.2 That is called **multipath recursion**

2.4.3.4.3 Figure 2-13 shows the box trace for finding the largest value

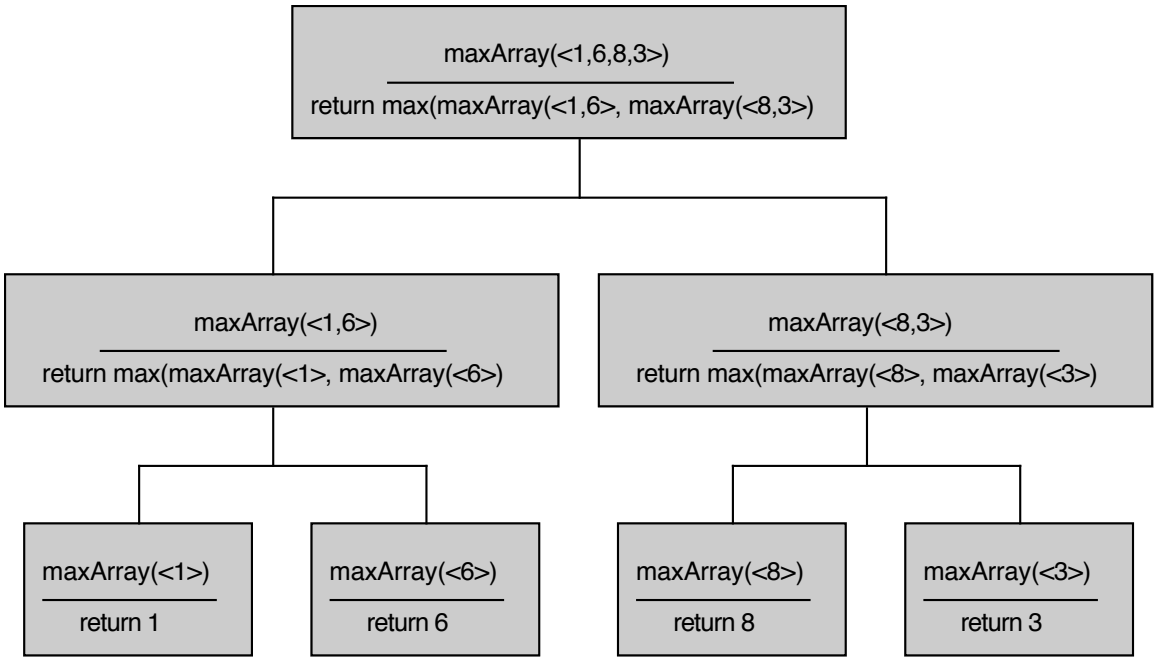

Figure 2-13, The recursive calls that maxArray(<1,6,8,3>) generates

2.4.4 we'll skip the "Finding the *k*th Smallest Value of an Array" for now

2.5 Organizing Data (data can be organized in one way, but you might want to organize in another way)

2.5.1 Towers of Hanoi is a classic problem in organizing though it has no practical application

2.5.1.1 Story behind it is that a long time ago at the Vietnamese city of Hanoi, the emperor's wiseperson joined his ancestors (i.e. croaked)

2.5.1.2 Emperor (a wiseperson himself) devised a puzzle that, if solved, would determine who would be the next wiseperson

2.5.1.2.1 Puzzle had three pegs and an unspecified number of disks of different sizes with holes in the center

2.5.1.2.2 Because they were heavy, the disks could only be placed on a disk that was larger than it

2.5.1.2.3 Initially, all the disks went on pole A

2.5.1.2.4 The puzzle is to move all the disks to pole B, one-by-one, without putting a large disk on a smaller disk and with having only one disk at a time in transit (all other disks MUST be on a pole).

2.5.1.3 Emperor got a lot of solutions – some with thousands of steps and deeply nested loops and control structures

2.5.1.4 The emperor wanted a SIMPLE solution

2.5.1.5 One great Buddhist monk came out of the mountains and stated that the solution was so simple "it solves itself".

2.5.1.5.1 If you have 1 disk, you move it from pole A to pole B

2.5.1.5.2 But for disk n > 1, you ignore the bottom disk and solve it for n-1

2.5.1.6 Therefore:

2.5.1.6.1 Ignore the bottom disk and solve for n – 1 disks

2.5.1.6.2 Move it from pole A to pole C and have pole B be considered the "spare".

2.5.1.6.3 After the n – 1 disks have been moved to pole C, you can move the largest disk from pole A to pole B.

2.5.1.7 Now move all the n – 1 disks from pole C to pole B with pole A being the spare.

2.5.1.8 Well, the emperor blankly looked at the Monk then said, impatiently, "Are you going to tell us your solution or not?"

2.5.1.8.1 The Monk just gave an all-knowing smile and vanished … The emperor was not a recursive thinker

2.5.1.8.2 Figure 2-16 shows the initial state

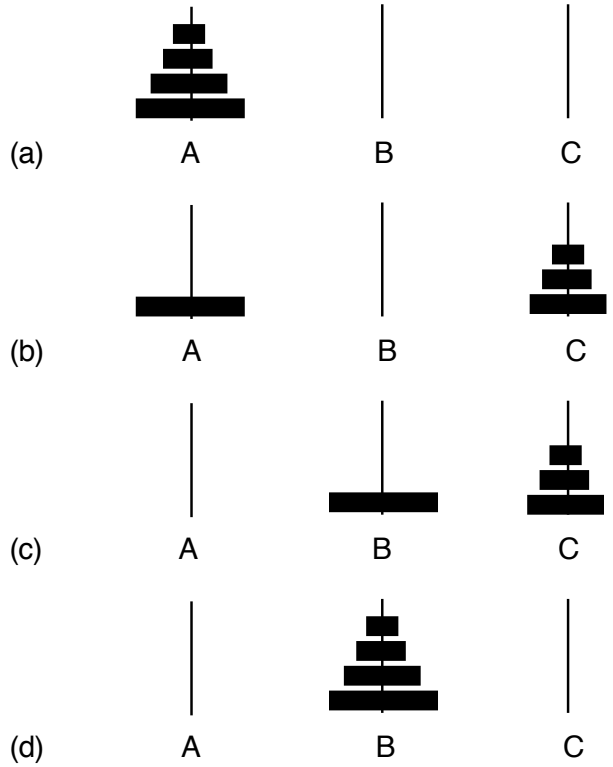

Figure 2-16, (a) Initial state; (b) move n - 1 disks from A to C; (c) move 1 disk from A to B; (d) move n - 1 disks from C to B

Restate the Emperor's problem like: with n disks on pole A and zero disks on poles B and C, solveTowers(n, A, B, C) – like so:

Step 1: Move the n – 1 disks from A to B like so:
solveTowers (n-1, A, C, B)

Step 2: Move the largest disk from A to B like so:
solveTowers (1, A, B, C)
Note: it's important that the number of disks be 1

Step 3: Move the n – 1 disks from C to B like so:
solveTowers (n – 1, C, B, A)

The code for the function is:

```
/** Towers of Hanoi
 * @pre  count > 0, if count < 1, will end function immediately
 * @post  None
 * @param count   The number of disks to move
 * @param source  The name of the source disk
 * @param dest    The name of the destination disk
 * @param spare   The name of the spare disk
 * @return  No value – just print the moves  */
void solveTowers(int count, char source, char dest, char spare)
{
        if (count < 1)
                return;
        if (count == 1)
                cout << "Move top disk from " << source
                     << " to " << dest << endl;
        else
        {       solveTowers(count-1, source, spare, dest); // X
                solveTowers(1, source, dest, spare);       // Y
                solveTowers(count-1, spare, dest, source); // Z
        } // end if
}
```

2.5.1.9 The order of recursive calls is shown in figure 2-17 – the output is:

The output for 3 disks would be:
Move top disk from A to B
Move top disk from A to C
Move top disk from B to C
Move top disk from A to B
Move top disk from C to A
Move top disk from C to B
Move top disk from A to B

2.5.1.10 The number of moves is $2^n - 1$. But moving more than 15 disks can be a problem (you can run out of RAM, literally).

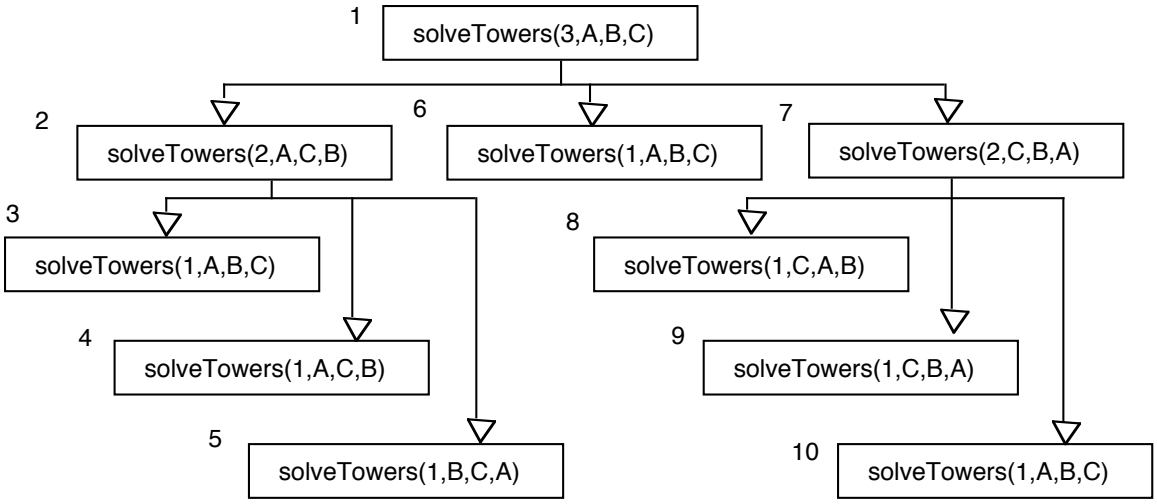

Figure 2-17, Order of recursive calls that results from solveTowers(3, A, B, C)

2.6 More Examples (three problems return you count events or combinations of events or things – good examples of recursive solutions with more than one base case … They can be inefficient and thus not practical – though that is not true for all recursive solutions – but these are still worth studying)

2.6.1 The Fibonacci Sequence (Multiplying Rabbits)

2.6.1.1 Rabbits are very prolific – we can study the rabbit population with the following assumptions:

- Rabbits never die
- A rabbit reaches sexual maturity exactly two months after birth – beginning of its third month of life
- Rabbits always born in male-female pairs. At the beginning of every month each sexually mature male-female pair gives birth to exactly one male-female pair

If you started with a single newborn male-female pair, how many pairs would there be in month 6 – like so:

Month 1: 1 pair, original rabbits
Month 2: 1 pair still, because rabbits are not yet sexually mature
Month 3: 2 pairs: original pair reached sexual maturity and gave birth to a second pair
Month 4: 3 pairs; original pair has given birth again, but the pair born at beginning of month 3 are not yet sexually mature
Month 5: 5 pairs; all rabbits alive in month 3 (2 pairs) are now sexually mature. Add their offspring to those pairs in month 4 (3 pairts) to give 5 pairs

Month 6: 8 pairs; 3 newborn pairs from pairs alive in month 4 pluse 5 pairs in month 5

2.6.1.2 The recursive solution for computing *rabbit*(n) with n being the month would be:

rabbit(n) = rabbit(n – 1) + rabbit(n – 2)

2.6.1.3 Figure 2-18 shows it:

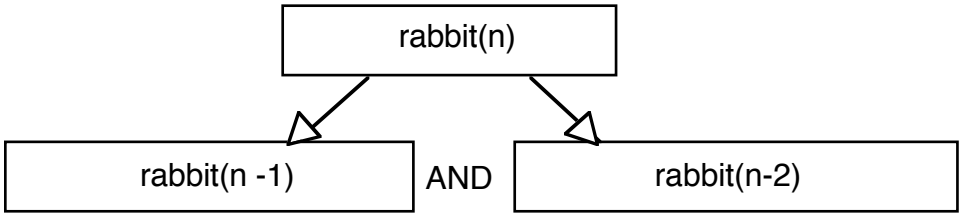

Figure 2-18, Recursive solution to rabbit solution

2.6.1.4 The full recursive definition is:

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

2.6.1.5 The rabbit solution is actually the **Fibonacci Sequence** which models many naturally occurring phenomena

```
/** Computes a tern in Fibonacci sequence
 * @pre       n is a positive number
 * @post      None
 * @param n   The given integer
 * return     The nth Fibonacci number. */
int rabbit(int n)
{
        if (n <= 2)
           return 1;
        else // n > 2, so n – 1 > 0 and n – 2 > 0
           return rabbit(n – 1) + rabbit (n – 2);
} // end rabbit
```

2.6.1.6 Figure 2-19 illustrates recursive calls for rabbit(7)

2.6.1.7 Note that the function rabbit is inefficient – quite a few calls (impractical for large n)

2.6.1.8 Sometimes it is better to use an iterative solution

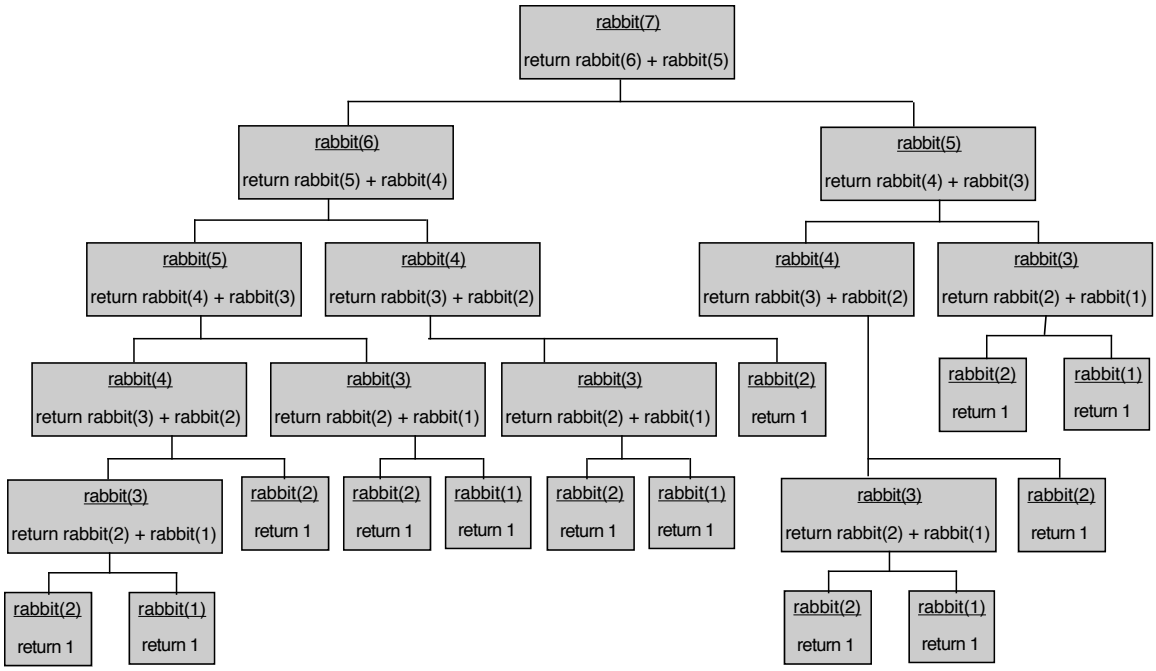

Figure 2-19, Recursive calls that rabbit(7) generates

2.6.2 Organizing a Parade

2.6.2.1 The bands in last year's parades of floats and bands tried to outdo each other

2.6.2.1.1 To avoid that, need to have no band following another

2.6.2.1.2 How many ways to organize a parade of length n?

2.6.2.1.3 Parades can be band-float or float-band and count as two ways

2.6.2.2 Assume the following:

P(n) be the number of ways to organize a parade of length n
F(n) be the number of parades of length n that end with a float
B(n) be the number of parades of length n that end with a band

Then: $P(n) = F(n) + B(n)$

2.6.2.3 With F(n), can get a float at the end of any acceptable parade of n – 1

2.6.2.3.1 Number of acceptable parades of length n ending with a float is equal to the number of acceptable parades of length n – 1

2.6.2.3.2 Or:

$F(n) = P(n – 1)$

2.6.2.4 With B(n), only way a parade can end with a band is that the unit just before the end is a float

2.6.2.4.1 Need to have a parade of n-1 end with a float THEN add a band
2.6.2.4.2 Number of acceptable parades of length n ending with a band is equal to the number of acceptable parades of length n – 1 ending with a float
2.6.2.4.3 Or:

$$B(n) = F(n - 1)$$

2.6.2.4.4 You can also use the fact of $F(n) = P(n - 1)$ to get:

$$B(n) = P(n - 2)$$

2.6.2.4.5 So you can have the following:

$$P(n) = P(n - 1) + P(n - 2)$$

2.6.2.5 Recurrence relation is identical to the rabbits problem in that need two base cases (n = 1 and n = 2)
2.6.2.6 But the values of P(n) are not the same as rabbit(n)
2.6.2.6.1 Consider:

P(1) = 2 Or the parade would either be a float or a band
P(2) = 3 Or the parade would be float-band, band-float, or float-float

2.6.2.6.2 So the recursive solution is:

P(1) = 2 base case
P(2) = 3 base case
$P(n) = P(n - 1) + P(n - 2)$ for $n > 2$

- Sometimes you can solve a problem by breaking it up into cases – like parades that end with a float and parades that end with a band
- Values you use for base cases are important – though the code is similar to rabbit, the base cases are different (n = 1 or n = 2), the rabbit(20) gives the value of 6,765 while the value of parade(20) gives the value of 17,711.

2.6.3 We'll skip "Choosing k Out of n Things"

2.7 Recursion and Efficiency
2.7.1 Recursion is a powerful problem-solving tool that frequently produces clean solutions to complex problems
2.7.1.1 Recursive solutions can be easier to understand and describe than iterative solutions: often write simple, short implementations

2.7.1.2 But recursion is not the proverbial hammer to nail everything down
2.7.1.3 Many recursive functions can be very inefficient can should not be used
2.7.1.4 The binarySearch and solveTowers are the exception since they are efficient.

2.7.2 There are two factors that make a recursive solutions inefficient:
- Overhead associated with function calls
- Inherent inefficiency of some recursive algorithms

2.7.2.1 First of the factors apply to all functions, not just the recursive ones
2.7.2.2 A function call does invoke bookkeeping overhead – sort of like what happens in a box trace
2.7.2.3 The factorial(n) call would invoke n recursive calls
2.7.2.4 But the inefficiency can be sometimes be compensated for by clarifying a complex problem.

2.7.3 But don't use recursion just for the sake of using recursion
2.7.3.1 The factorial function can be written just as easily as an iterative solution
2.7.3.2 Remember that "recursion is truly valuable when a problem has no simple iterative solutions".
2.7.4 For example, the rabbit problem can be done readily as iterative, like so:

```
/** Iterative solution to the rabbit problem
 * @pre       n is a positive number
 * @post      None
 * @param n   The given integer
 * return     The nth Fibonacci number. */
int iterativeRabbit(int n)
{
        // initalize base cases:
        int previous = 1;   // initially rabbit(1)
        int current = 1;    // initially rabbit(2)
        int next = 1;       // result when n is 1 or 2

        // compute next rabbit values when n >= 3
        for (int i = 3; i <= n; ++i)
        {
                next = current + previous;   // rabbit(i)
                previous = current;          // get ready for
                current = next;              // next iteration
        } // end for
        return next;
} // end iterativeRabbit
```

2.7.5 An iterative solution can be more efficient than a recursive one
- 2.7.5.1 But in certain cases it may be easier to discover a recursive solution than an iterative one
- 2.7.5.2 Then convert the recursive solution to an iterative one
- 2.7.5.3 That can be done easier if the recursive solution calls itself once like with the binarySearch (there are two calls in the code but only one is called during execution of a particular level).
- 2.7.5.4 Converting a recursive solution to an iterative solution is even easier when the solitary recursive call is the last action the function does
  - 2.7.5.4.1 That is "tail recursion".
  - 2.7.5.4.2 The writeBackward does tail recurtion
  - 2.7.5.4.3 The fact function does multiplication as it's last action so it is not tail recursion
- 2.7.5.5 Rewriting writeBackward is like so:

```
/** Writes a character string backward.
 *  @pre  The string s contains size characters, where size >= 0
 *  @post  None
 *  @parm s  The string to write backward.
 *  @parm size  The length of s.  */
void writeBackward(string s, int size)
{
      while (size > 0)
      {      cout << s.substr(size-1, 1);
            --size;
      } // end while

}      // end writeBackward
```

- 2.7.5.6 Tail recursion can be frequently converted into an iterative solution. Some compilers actually do that automatically.

2.7.6 Some recursive algorithms are very inefficient like rabbit and some are very efficient like binarySearch.