Recursion - II

Binary search
Fibonacci sequence
Printing string
Palindrome
GCD of two numbers

Binary search (recursive version)
Divide and conquer implementation

We have already looked at the iterative version of binary search algorithm. Given an array with elements in the ascending order, the problem is to search for a target element in the array. Here is the recursive version of the binary search. It returns the position of the element which matches the target. If the target is not found, it returns -1.
Use this code to implement binary search program, and test for arrays containing odd numbered elements as well as arrays containing even numbered elements. In order to understand the working of the program introduce a statement to print the middle element every time. Modify the program so that it can handle an array containing elements in descending order.

```
#include <stdio.h>


int targetsearch ( int target, int A[], int  n )
{
       int temp;
       temp = binary(target, A, 0, n-1) ;
       return temp ;
}


int binary ( int target, int A [ ], int left, int right)
{
   int mid;
   if ( left > right ) return -1 ;
   mid =  ( left + right )/ 2;
   if   ( target ==  A [mid] ){
           return (mid);}
   if  ( target < A [mid])
           return (binary (target, A, left, mid - 1 ));
   else
           return (binary (key, A, mid + 1, right )) ;
}
```

The time complexity of the algorithm can be obtained by noting that every time the function is called, the number of elements to be handled reduces to half the previous value. Assuming it takes one operation for each of the 3 IF statements, and 2 operations for computing mid, the total number of operations can be expressed through the recurrence relation

$$T(n) = T(n/2) + 5$$

Note this relation is very similar to the one that we had for the efficient version of the exponentiation algorithm, and it can be shown that the complexity works out to O(log n).

Fibonacci Sequence

This is a very famous sequence which can be generated through a recursive solution. It has a small history. In 1202, Italian mathematician Leonardo Fibonacci posed a problem that has had a wide influence on many fields. The problem is related to growth in population of rabbits, generation to generation. It is assumed that the rabbits are reproduced according to the following rules:

1. Each pair of fertile rabbits produces a new pair of offspring each month.
2. Rabbits become fertile in their second month of life.
3. No rabbit dies.

Suppose we start with a pair introduced in January. Thus
on Jan 1 , there are no rabbits and
on Feb 1 , there is one pair. Since they are not yet fertile
on March 1, it is still single pair. In march they produce another pair, so
on April 1, the count is 2 pairs. In April the original pair produces another pair, so
on May 1 there are 3 pairs.

We continue further with the same logic and find that the number of rabbits in each month is given by the sequence 0,1,1,2,3,5,8… This sequence is known as the *Fibonacci sequence*:

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
0	1	1	2	3	5	8	13	21	34 …

A very interesting point about this sequence is that each element in this sequence is the sum of the two preceding elements. Further no two consecutive elements are even numbers. The sequence can be mathematically described through the relations:

$$t_n = \begin{cases} n & \text{if } n \text{ is } 0 \text{ or } 1 \text{ (i.e. } n < 2) \\ t_{n-1} + t_{n-2} & \text{otherwise} \end{cases}$$

How fast do the numbers grow? If 4^{th} term is 2, 7^{th} term is 8, 12^{th} term is 89, 20^{th} term is 4181, and so on. The numbers grow exponentially, and it can be shown by induction that

$t_n > (3/2)^n$

$t_n = t_{n-1} + t_{n-2}$

$t_n > (3/2)^{n-1} + (3/2)^{n-2}$

$> (2/3)(3/2)(3/2)^{n-1} + (2/3)(3/2)(2/3)(3/2)(3/2)^{n-2}$

$> (2/3)(3/2)^n + (2/3)(2/3)(3/2)^n$

$> [(2/3) + (4/9)](3/2)^n$

$> (10/9)(3/2)^n$

$t_n > (3/2)^n$

The fibonacci sequence can be generated through the following recursive function.

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

What is the complexity of this function? At every function call, two more fibonacci functions are being called. The same function is being called twice and there is an overlap of computations between the two function calls.
This results in excessive computation. In fact we shall see that the complexity turns out to be exponential in nature.

Let T(n) be running time for calling the fibonacci Function. The recurrence relations for this function are

T(n) = T(n-1) + T(n-2) +3
T(2) = 3

In order to solve this recurrence relation, we note that

T(n-1) = T(n-3) + T(n-2) +3

Substituting for T(n-1) in the first relation

$T(n) = 2\,T(n- 2) + T(n-3) + 6$

Even if you try to reduce it further, it will be difficult to come up with a pattern where we can involve the parameter k. We shall use a small trick here to work out the complexity. Since we are interested in working out the worst case complexity, let us try to simplify our work. We observe that

$T(n) > 2\,T(n -2)$

Now we can try reducing this by replacing n with n-2, to get
$T(n-2) > 2\,T(n-4)$

Replacing for T(n-2) in the previous relation we get

$T(n) > 4\,T(n-4)$

Continuing the process, it is easy to show that

$T(n) > 8\,T(n-6)$

$T(n) > 16\,T(n-8)$

And in general,

$T(n) > 2^k\,T(n - 2k)$

Since we know the result for T(2), let us try to reduce the term on the right hand side to T(2). Thus we let

$n-2k = 2$

Solving for k, we get

$k = (n-2)/2$

$T(n) > 2^{(n-2)/2}\,T(2)$

Thus we find that

$T(n) > 3.\ 2^{(n-2)/2}$

which shows that the complexity of this algorithm is $O(2^n)$.

Recursive function to print a string in reverse order

Here is a recursive function that reads the characters of a string from the keyboard, as they are being typed, but prints them in the reverse order. The function needs to know how many characters would be read before printing starts. Obviously, printing cannot start until all the characters have been read. The function uses an internal stack of the computer to store each character as it is being read.

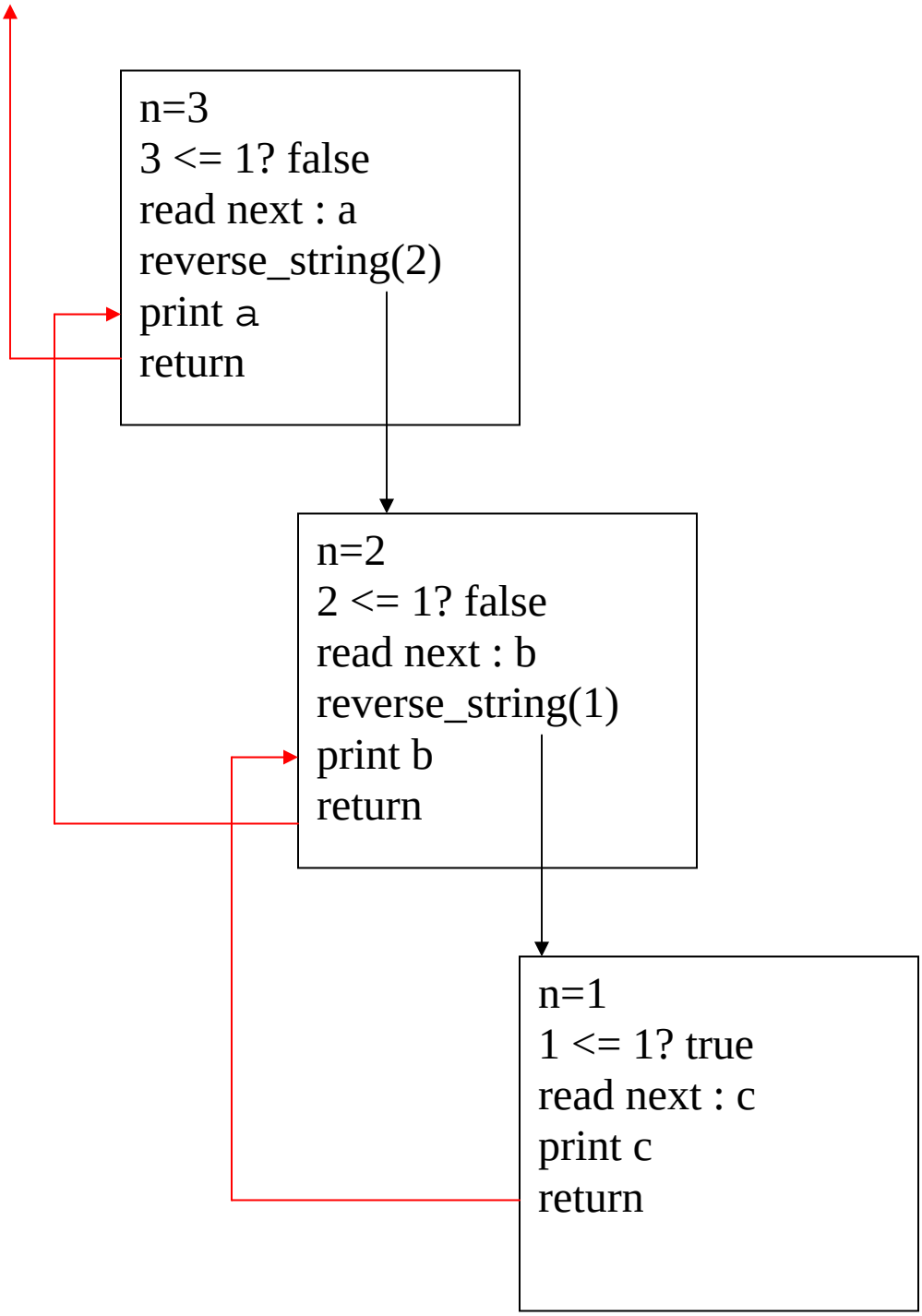
```
void print_reverse(int n)
{
     char next;

     if (n == 1) {       /* stopping case */
          scanf("%c",&next);
          printf("%c", next);
    }
     else {
          scanf("%c", &next);
          print_reverse(n-1);
          printf("%c",next);
     }
     return;
}
```

On the next page, you can see the trace of the function for a sample case where it handles a string of length 3.

Trace of print_reverse for input string `abc`

reverse_string(3);



Checking for palindromes

Palindrome: a word, phrase, verse or sentence that reads the same backward or forward. Here are few palindromes:
borrow or rob
gateman sees name garageman sees name tag
murder for a jar of red rum
a nut for a jar of tuna
How do you mathematically define a "palindrome"? Here is a recursive definition:
It is a 'string' whose first and last characters match AND the remaining substring is also a palindorme. As per this definition the examples above would be termed palindromes if you ignore the spaces between the words.
anutforajaroftuna

The recursive definition can be easily coded into a computer code segment. It needs the length of the string to be supplied along with the substring itself. Every time the function is being called, it makes sure that the first and last characters match and then calls itself recursively after dropping the first and last characters from the substring.
nutforajaroftun
utforajaroftu
tforajaroft

```
int checkPalindrome (char string[])
{
        if ( check (string,  strlen (string) ))

           printf("\nyes, it is a palindrome\n");
        else printf("\nNo, it is not a palindrome");
}
```

Function *check* returns 1 if the remaining characters in the array str form a palindrome. */

```
int  check ( char str[],   int  len)
{
        if ( len <=  1)
                return 1 ;
        else

               return ( (str [0] == str [ len - 1 ] )
                && check(str + 1, len - 2 ) );

}
```

Note that array arithmetic is being used here. Since the first character of the string is located at memory location *str*, the new substring begins at memory location *str+1* and the length of the new substring is 2 less than the current string length .

The Greatest common Divisor (GCD)

The GCD of Two non-negative integers is the largest integer that divides evenly into both.

For example, numbers 2, 3, 4, 6 and 12 divide evenly into both the numbers 24 and 36 . So GCD of 24 and 36 is 12, the largest integer. The GCD can be obtained by factorizing the numbers and picking up all the common elements. However for very large numbers it would involve large number of operations. For example consider obtaining GCD of 129618 and 576234 by factorizing.

In 3rd century B.C., the great mathematician Euclid had proposed an algorithm to find GCD. It was based on the facts that GCD(p, q) = GCD(q, p), and GCD(p, 0) = p

Euclid's algorithm:

1. Make p the larger of two numbers p and q.

2. Divide p by q. Let r be the remainder. If r is zero, then q is the gcd.

3. Else, gcd of p and q equals gcd of q and r.

Using this algorithm we can find the gcd of 1296 and 576 as follows;

gcd(1296,576)

= gcd (576, 1296 % 576)

= gcd (576, 144)

= 144 as 144 divides 576 without leaving a remainder.

Write a recursive function **int GCD(int p, int q)** using the Euclid's algorithm.

Common Errors in Recursive Formulations

- It may not terminate if the stopping case is not correct or is incomplete (stack overflow: run-time error)

- Make sure that each recursive step leads to a situation that is closer to a stopping case.

Comparison of Iteration and Recursion

- In general, an iterative version of a program will execute more efficiently in terms of time and space than a recursive version. This is because the overhead involved in entering and exiting a function is avoided in iterative version.

- However a recursive solution can be sometimes the most natural and logical way of solving a problem.
  ⇨ Conflict: machine efficiency versus programmer efficiency

- It is always true that recursion can be replaced with iteration and a stack.