Dynamic Data Structures with C

Terry Marris October 2010

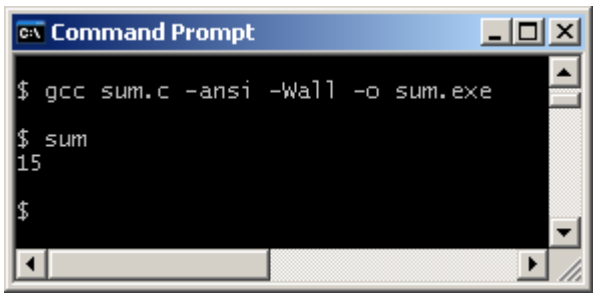
2 Recursion

Previously, we have looked at pointers to functions. Now we take a look at recursion. The use of recursion helps to simplify some complex algorithms, but at the expense of increased memory usage and slower execution speed.

2.1 Recursion

Look at how we might sum the first five integers, 1 + 2 + 3 + 4 + 5 = 15

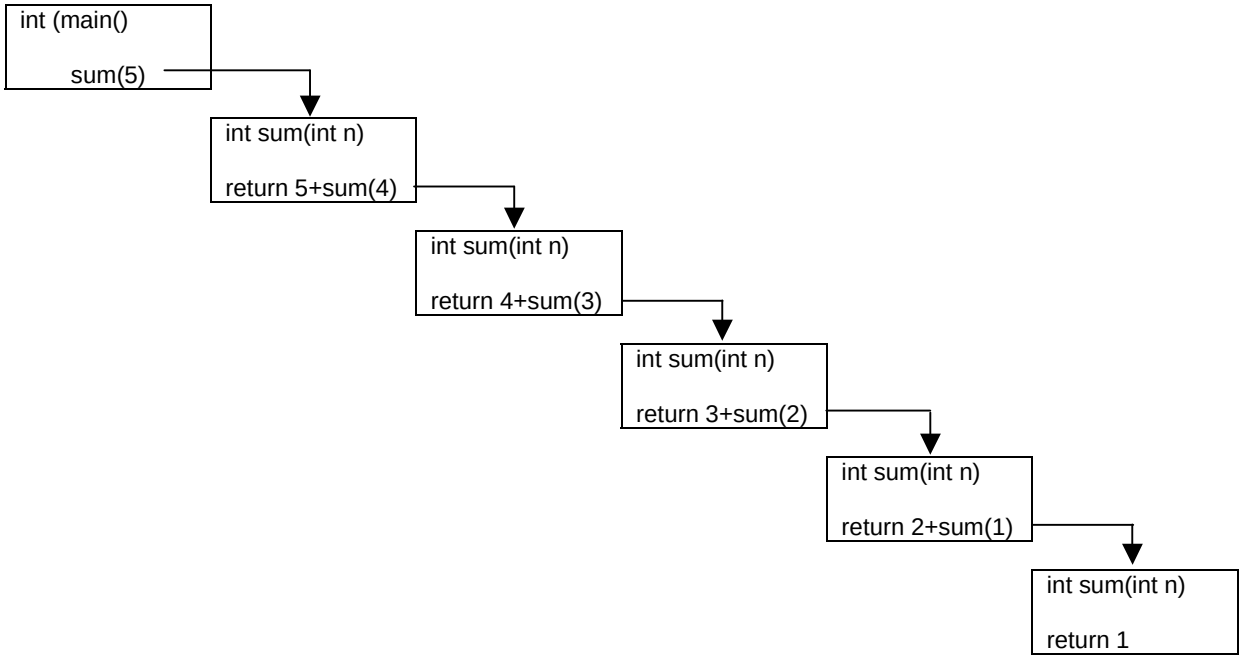
```
/* sum.c */

#include <stdio.h>

int sum(int n)
{
  if (n == 1)  /* stopping case */
    return 1;
  return n + sum(n-1);  /* recursive call to sum() */
}

int main()
{
  printf("%d\n", sum(5));  /* initial call to sum() */
  return 0;
}
```



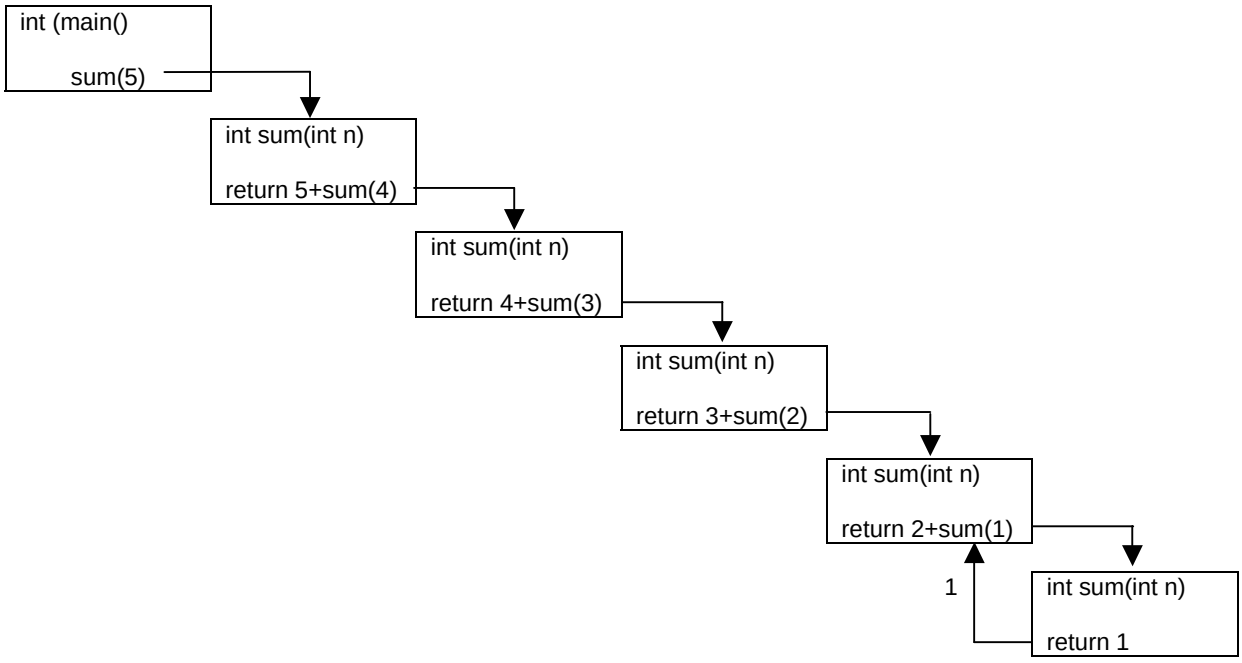
A recursive function is one that calls itself.

```
int sum(int n)
{
      ...
  return n + sum(n-1);  /* recursive call to sum() */
}
```

We trace the sequence of function calls to *sum()*.

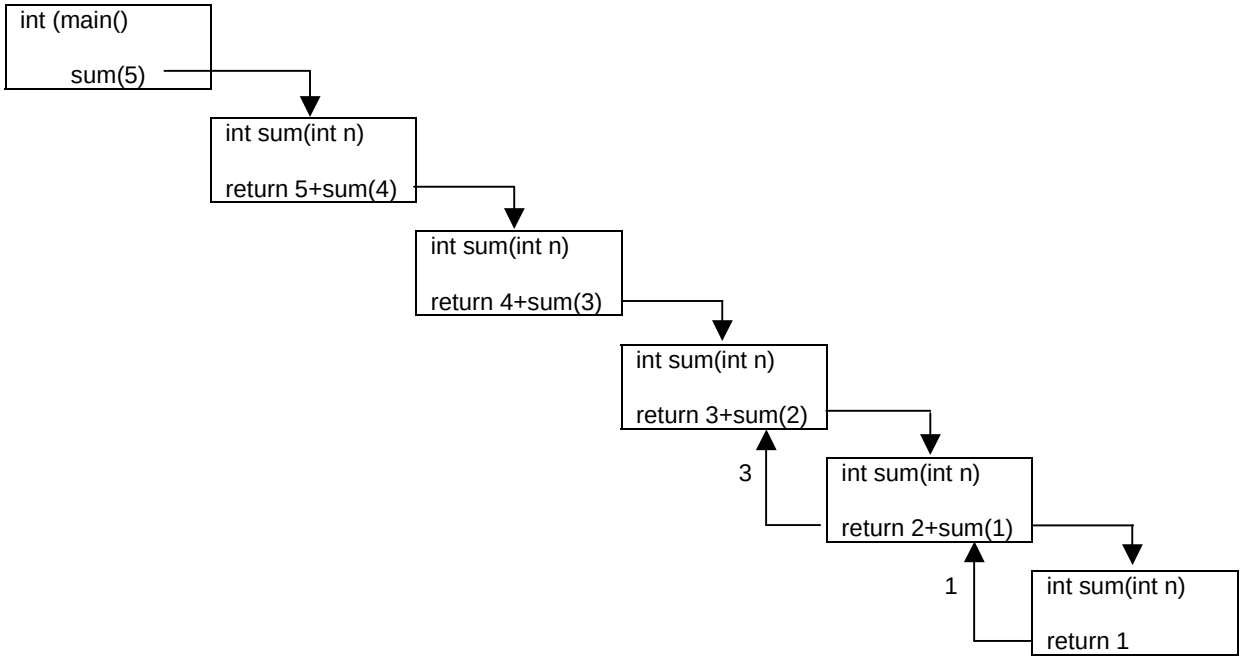


When *n = 1*, we have the stopping case which anchors the recursive calls and prevents them going on for ever (that is, until you run out of memory). Now we can trace the returns from each function call.

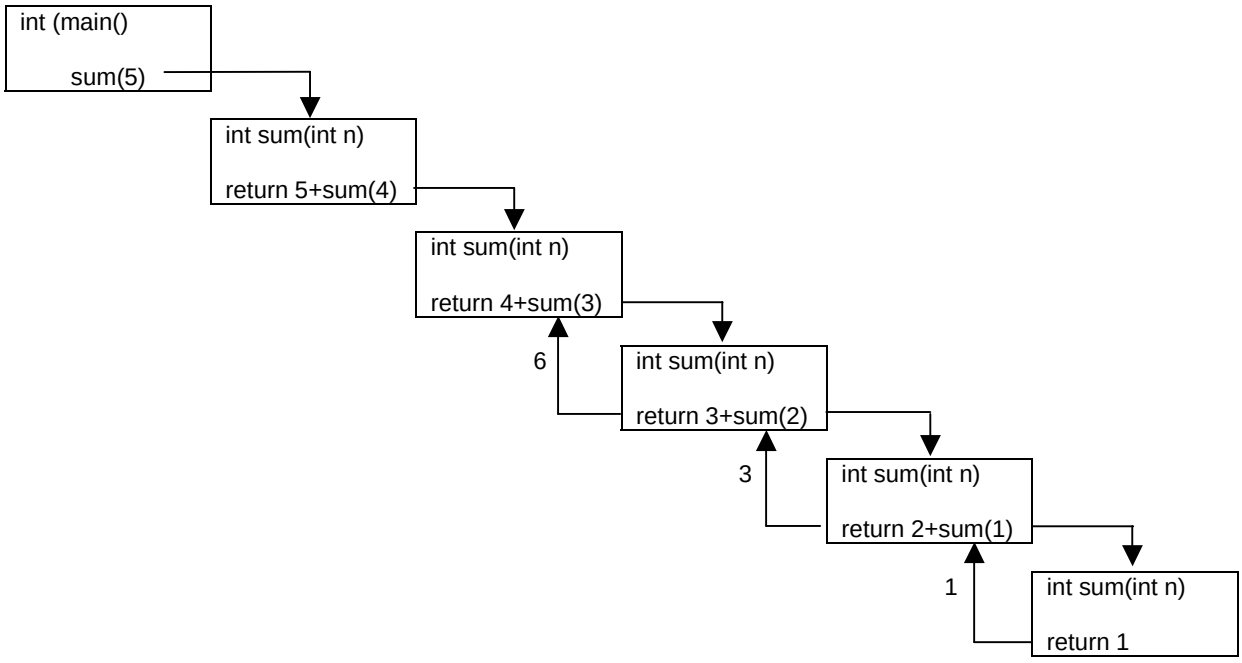


When *n = 1*, *1* is returned.

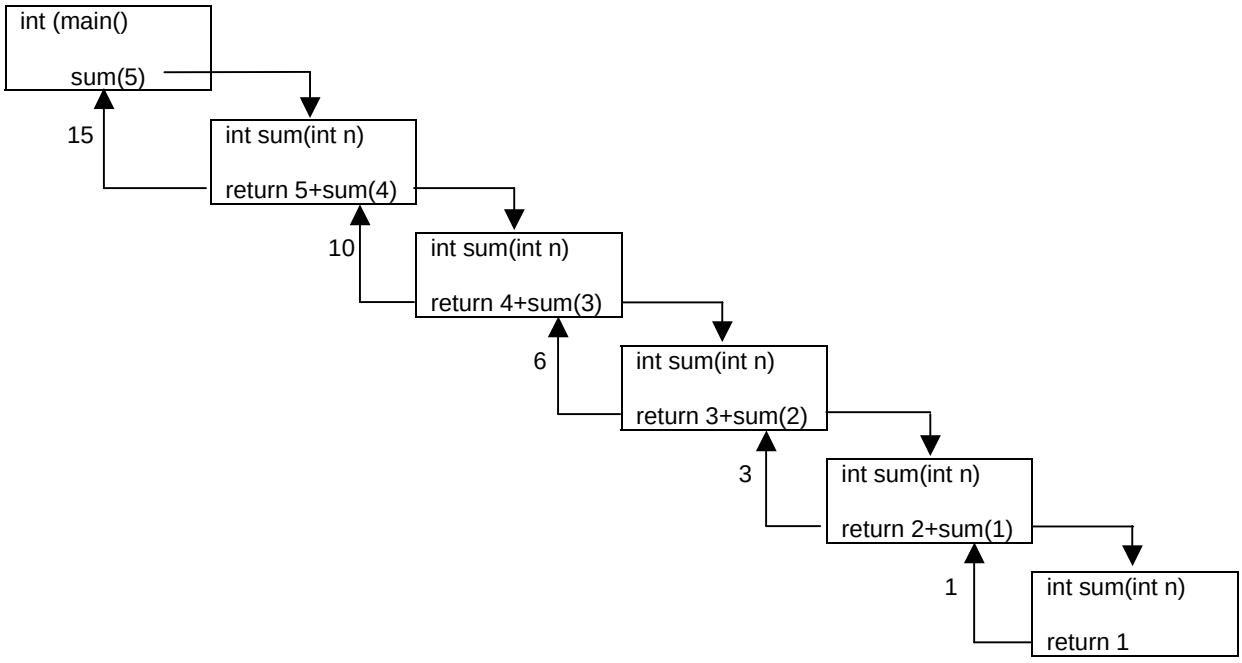
Then, *1 + 2 = 3* is returned.



Then *3 + 3 = 6* is returned.



Then *6 + 4 = 10*, and *10 + 5 = 15* are returned.




```
int sum(int n)
{
...
  return n + sum(n-1);  /* recursive call to sum() */
}
```

causes the function to repeatedly call itself with ever decreasing argument values until the anchor or stopping case is reached

```
int sum(int n)
{
  if (n == 1)  /* stopping case */
    return 1;
...
}
```

Each recursive call gets you one step closer to the stopping case.

2.2 Factorial

A classic example often used to illustrate recursion is the factorial function. Factorial 5, written as 5!, is the result of 5 x 4 x 3 x 2 x 1 = 120. The factorial function is defined as

$$n! = \begin{cases} 1 \text{ if } n = 0 \\ 1 \times 2 \times 3 \times \ldots \times n \text{ if } n > 0 \end{cases}$$

Notice that the function is not defined for integers less than zero.

That is

0! = 1
1! = 1 x 0! = 1
2! = 2 x 1! = 2
3! = 3 x 2! = 6
4! = 4 x 3! = 24
5! = 5 x 4! = 120

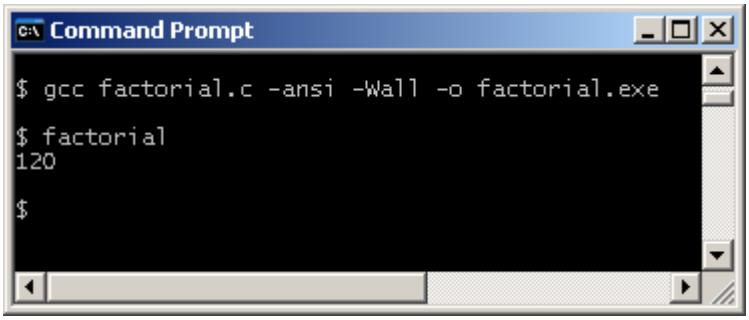
The recursive definition of the factorial function is

0! = 1	*anchor*
if n > 0, n! = n x (n - 1)	*recursive step*

We can write

```
/* factorial.c */
#include <stdio.h>

int factorial(unsigned n)
{
  if (n == 0)  /* anchor */
    return 1;
  return n * factorial(n - 1); /* recursive call */
}

int main()
{
  printf("%d\n", factorial(5));
  return 1;
}
```

And obtain



Exercise 2.1

1. Trace the sequence of function calls, and the values returned, for *factorial(5)* shown above.

2.3 Power

We know that $2^5 = 2 \times 2 \times 2 \times 2 \times 2 = 32$

And

$2^0 = 1$
$2^1 = 1 \times 2^0 = 2$
$2^2 = 2 \times 2^1 = 4$
$2^3 = 2 \times 2^2 = 8$
$2^4 = 2 \times 2^3 = 16$
$2^5 = 2 \times 2^4 = 32$

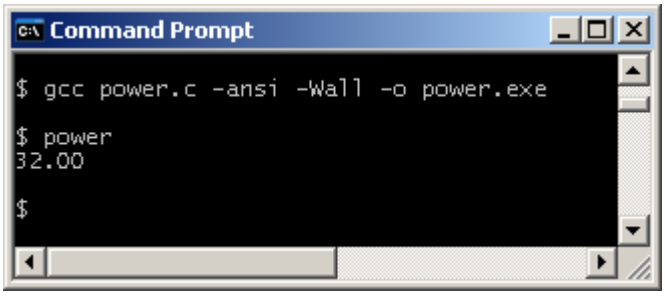
The recursive definition of power is

$x^0 = 1$	*anchor*
if $n > 0$, $x^n = x * x^{n-1}$	*recursive step*

where $x : \mathbb{R}$ and $n : \mathbb{N}$, i.e. x is a real number and n is an integer >= 0

The program and its run are shown below.

```
/* power.c */
#include <stdio.h>

double power(double x, unsigned n)
{
  if (n == 0)                          /* anchor */
    return 1;
  return x * power(x, n - 1); /* recursive call */
}

int main()
{
  printf("%0.2f\n", power(2.0, 5));
  return 0;
}
```



Exercise 2.2

1. Trace the sequence of function calls, and the values returned, for *power(2.0, 5)* shown above.

2.4 Searching

We see how recursion could be used to search through an array.

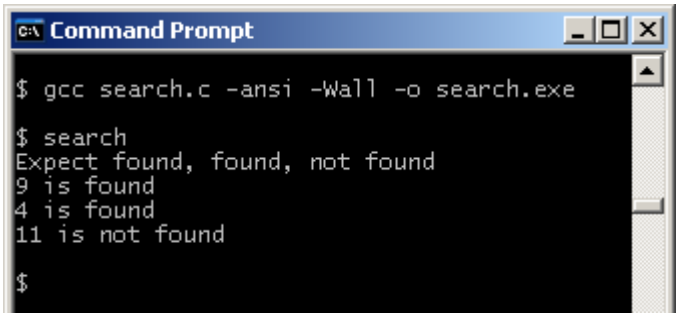
We look for a *target* in an array of integers and return its index in the array if found, or return -1 if not found. We (arbitrarily|) define the array as

```
int array[] = { 9, 0, 8, 1, 7, 2, 6, 3, 5, 4 };
```

We let *i* index the array. *i* is initialised with the last index value (9) (not to be confused with the array value 9) and decremented for each recursive call. We have two stopping cases. A stopping case occurs if *i < 0* when the target is not found in the array. The other stopping case occurs when *array[i] == target*.

Here is the entire program and its run.

```
/* search.c */
#include <stdio.h>

int find(int array[], int target, int i)
{
  if (i < 0)                                /* anchor */
    return i;
  else if (array[i] == target)              /* anchor */
    return i;
  else
    return find(array, target, i-1);        /* recursive call */
}

int main()
{
  int lastIndex = 9;
  int array[] = { 9, 0, 8, 1, 7, 2, 6, 3, 5, 4 };
  printf("Expect found, found, not found\n");
  find(array, 9, lastIndex) >= 0?
    printf("9 is found\n") : printf("9 is not found\n");
  find(array, 4, lastIndex) >= 0?
    printf("4 is found\n") : printf("4 is not found\n");
  find(array, 11, lastIndex) >= 0?
    printf("11 is found\n") : printf("11 is not found\n");
  return 0;
}
```



2.5 Sorting

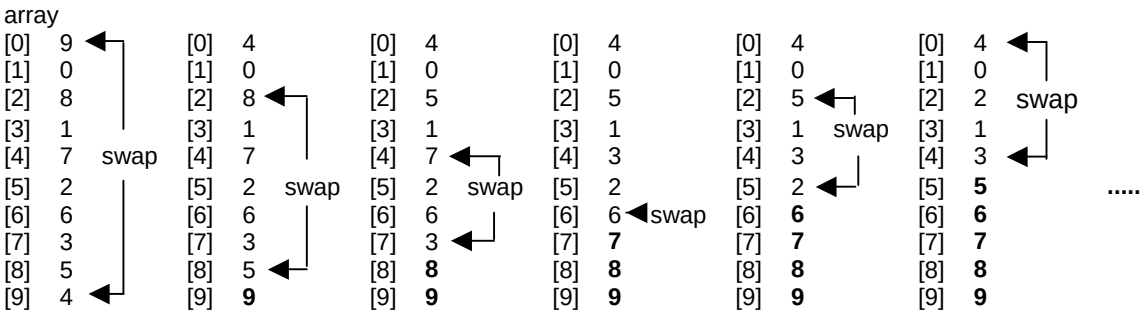
We see how recursion could be used to sort the elements of an array into ascending order. We start with an array such as

```
int array[] = { 9, 0, 8, 1, 7, 2, 6, 3, 5, 4 };
```

and seek to end up with

```
array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

We look for the largest item in the array and place it at the end. Then we consider the array without its last element: we look for the largest item in this sub-array and place it at its end. Then we consider this array without its last element: we look for the largest item in this sub-array and place it at its end. We repeat this process until the sub-array has just one element.



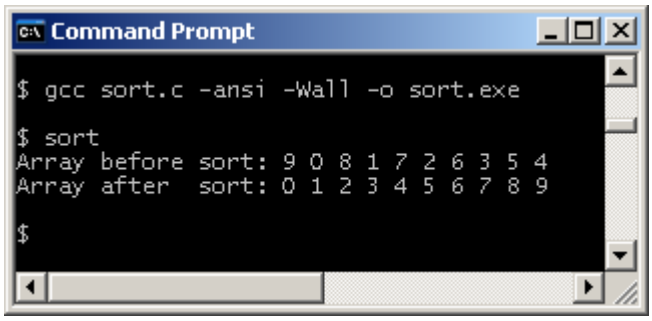
As you can see, the size of the sorted portion increases from the bottom for each swap made.

The recursive *sort()* function is

```
/* sort: sorts the given array of integers into ascending order */
void sort(int array[], int arraySize)
{
  if (arraySize == 1)                          /* anchor */
    return;
  int largest = indexOfLargest(array, arraySize);
  swap(array, largest, arraySize-1);
  sort(array, arraySize-1);                    /* recursive call */
}
```

The entire program and its run are shown below.

```
/* sort.c */
#include <stdio.h>

/* indexOfLargest: returns the index of the largest value in the
   array */
int indexOfLargest(int array[], int arraySize)
{
  int j;
  int i;
  for (i = 0, j = 1; i < arraySize; i++)
    if (array[i] > array[j])
      j = i;
  return j;
}

/* swap: exchanges values in the array at indices i and j */
int swap(int array[], int i, int j)
{
  int t = array[i];
  array[i] = array[j];
  array[j] = t;
  return 0;
}

/* sort: sorts the given array of integers into ascending order */
void sort(int array[], int arraySize)
{
  if (arraySize == 1)
    return;
  int largest = indexOfLargest(array, arraySize);
  swap(array, largest, arraySize-1);
  sort(array, arraySize-1);
}

/* printArray: displays the contents of the array */
void printArray(int array[], int arraySize)
{
  if (arraySize == 1)
    printf("%d ", array[0]);
  else {
    printArray(array, arraySize-1);
    printf("%d ", array[arraySize-1]);
  }
}

int main()
{
  enum { arraySize = 10 };
  int array[] = { 9, 0, 8, 1, 7, 2, 6, 3, 5, 4 };

  printf("Array before sort: ");
  printArray(array, arraySize);
  printf("\n");

  printf("Array after  sort: ");
  sort(array, arraySize);
  printArray(array, arraySize);
  printf("\n");

  return 0;
}
```



Exercise 2.3

1. Trace the recursive calls for the *printArray()* function shown above. You could use an example array of just three elements.

2. Write and test a recursive function that sums the elements in an array of integers.

3. Demonstrate how recursion could be used to implement a binary search function on an ordered array of integers.

Bibliography

Kernighan B, Ritchie D, *The C Programming Language*, Prentice Hall, 1988
Mark Williams Company, *ANSI C - A Lexical Guide*, Prentice Hall 1988
Koffman E, Problem *Solving & Structured Programming in Modula 2* Addison Wesley 1988
Stubbs D, Webre N, *Data Structures with Abstract Data Types & Modula 2* Wadsworth 1987
Leestma S, Nyhoff L, *Programming & Problem Solving in Modula 2* Macmillan 1989