Chapter 7

Functions

Introduction

- During the 1970s and into the 80s, the primary software engineering methodology was *structured programming.*

- Structured programming makes
  - programs more comprehensible.
  - programming errors less frequent.

- A *function* is a self-contained block of program statements that performs a particular task.

Why are Functions Needed?

- It breaks up a program into easily manageable chunks and makes programs significantly easier to understand.
- Well written functions may be reused in multiple programs. e.g. the C standard library functions.
- Functions can be used to protect data.
- Different programmers working on one large project can divide the workload by writing different functions.

Why are Functions Needed?

- All C programs contain at least one function, called main(), where execution starts.
- When a function is called, the code contained in that function is executed.
- When the function has finished executing, control returns to the point at which that function was called.

Function Prototype Declaration

- The general form of function declaration statement is as follows:

return_data_type function_name (data_type variable1,...);

Or

return_data_type function_name (data_type_list);

Function Prototype Declaration

- function_name :
  - This is the name given to the function
  - It follows the same naming rules as that for any valid variable in C.
- return_data_type:
  - This specifies the type of data given back to the calling function after it executes its specific task.
- data_type_list:
  - This list specifies the data type of each of the variables.

Function Prototype Declaration

- The name of a function is global.
- No function can be defined in another function body.
- Number of arguments must agree with the number of parameters specified in the prototype.
- The function return type cannot be an array or a function type.

Rules for Parameters

- The number of parameters in the actual and formal parameter lists must be consistent.
- Parameter association in C is *positional*.
- Actual parameters and formal parameters must be of compatible data types.
- Actual (input) parameters may be a variable, constant, or any expression matching the type of the corresponding formal parameter.

Call By Value Mechanism

- In call by value, a copy of the data is made and the copy is sent to the function.
- The copies of the value held by the arguments are passed by the function call.
- As only copies of the values held in the arguments are sent to the formal parameters, the function cannot directly modify the arguments passed.

An Example of Call by Value Mechanism

```
#include <stdio.h>
int mul_by_10(int num); /* function prototype */
int main(void)
{
  int result,num = 3;
  printf("\n num = %d before function call.", num);
  result = mul_by_10(num);
  printf("\n  result  =  %d  after  return  from
        function", result);
  printf("\n num = %d", num);
  return 0;
}
/* function definition follows */
int mul_by_10(int num)
{
  num *= 10;
  return num;
}
```

Output

```
num = 3, before function call.
result = 30, after return from function.
num = 3
```

Passing Arrays to Functions

- When an array is passed to a function, the address of the array is passed and not the copy of the complete array.
- During its execution the function has the ability to modify the contents of the array that is specified as the function argument.
- The array is not passed to a function by value.
- This is an exception to the rule of passing the function arguments by value.

Passing Arrays to Functions

- Consider the following example

```
#include <stdio.h>
void doubleThem(int [], int);
                        /* declaration of function */
int main(void)
{
  int myInts[10] = {1,2,3,4,5,6,7,8,9,10};
  int size=10;
  printf("\n\n The given numbers are :");
  for (i = 0; i < size; i++)
  printf("%d,",myInts[i]);
  doubleThem(myInts,size);   /* function call */
  printf("\n\n The double numbers are : ");
  for (i = 0; i < size; i++)
  printf("%d,",myInts [i]);
  return 0;
}
/******* function definition *******/
void doubleThem(int a[], int size)
{
  int i;
  for(i = 0; i < size; i++)
  {
      a[i] = 2 * a[i];
  }
}
```

Output

```
The given numbers are :1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
The double numbers are : 2, 4, 6, 8, 10, 12, 14, 16, 18, 20,
```

Scope Rules

- The region of the program over which the declaration of an identifier is accessible is called the *scope of the identifier.*

- The scope relates to the accessibility, the period of existence, and the boundary of usage of variables declared in a program.

- Scopes can be of four types.
  - Block
  - File
  - Function
  - Function prototype

Storage Classes

Storage class specifier	Place of storage	Scope	Lifetime	Default value
auto	Primary memory	Within the block or function where it is declared.	Exists from the time of entry in the function or block to its return to the calling function or to the end of block.	garbage
register	Register of CPU	Within the block or function where it is declared.	Exists from the time of entry in the function or block to its return to the calling function or to the end of block.	garbage
static	Primary memory	*For local* Within the block or function where it is declared. *For global* Accessible within the program file where it is declared	*For local* Retains the value of the variable from one entry of the block or function to the next or next call. *For global* Preserves value in the program file	0
extern	Primary memory		Exists as long as the program is in execution.	0

Storage Class Specifiers for Functions

- The only storage class specifiers that may be assigned with functions are extern and static.
- extern signifies that the function can be referenced from other files.
- static signifies that the function cannot be referenced from other files.
- If no storage class appears in a function definition, extern is presumed.

Linkage

- An identifier's linkage determines which of the references to that identifier refer to the same object.
- C defines three types of linkages – external, internal, and no linkage.
- Functions and global variables have external linkage.
- Identifiers with file scope declared as static have internal linkage.
- Local identifiers have no linkage and are therefore known only within their own block.
- The same identifier cannot appear in a file with both internal and external linkage.

Inline Function

- C99 has added the keyword *inline*, which applies to functions.
- By preceding a function declaration with *inline,* the compiler is instructed to optimize calls to the function.
- Here the function's code will be expanded in line, rather than called.
- Function definition:

```
inline int sum(int x, int y)
{
        return x+y;
}
```

Recursion

- *Recursion* in programming is a technique for defining a problem in terms of one or more smaller versions of the same problem.
- A function that calls itself directly or indirectly to solve a smaller version of its task until a final call which does not require a self-call is a recursive function.
- The following are necessary for implementing recursion:
  - Decomposition into smaller problems of same type
  - Recursive calls must diminish problem size
  - Necessity of base case
  - Base case must be reached
    - It acts as a terminating condition. Without an explicitly defined base case, a recursive function would call itself indefinitely.
    - It is the building block to the complete solution. In a sense, a recursive function determines its solution from the base case(s) it reaches.

What is Needed for Implementing Recursion?

- Decomposition into smaller problems of same type
- Recursive calls must diminish problem size
- Necessity of base case
- Base case must be reached
- It acts as a terminating condition. Without an explicitly defined base case, a recursive function would call itself indefinitely.
- It is the building block to the complete solution. In a sense, a recursive function determines its solution from the base case(s) it reaches.

Recursion

- A recursive function is one that calls itself directly or indirectly to solve a smaller version of its task until a final call which does not require a self-call.
- Recursion is like a top–down approach to problem solving; it divides the problem into pieces or selects one key step, postponing the rest.
- On the other hand, iteration is more of a bottom–up approach; it begins with what is known and from this constructs the solution step by step.
- What is a base case? An instance of a problem the solution of which requires no further recursive calls is known as a base case. It is a special case whose solution is known. Every recursive algorithm requires at least one base case in order to be valid.

```
if(this is a base case) then
   solve it directly
else
   redefine the problem using recursion.
```

Recursion

Four questions can arise for constructing a recursive solution.

- How can the problem be defined in terms of one or more smaller problems of the same type?
- What instance(s) of the problem can serve as the base case(s)?
- As the problem size diminishes, will this/these base case(s) be reached?
- How is/are the solution(s) from the smaller problem(s) used to build a correct solution to the current larger problem?

It is not always necessary or even desirable to ask the above questions in strict order.

Recursion

Linear recursion

- This term is used to describe a recursive function where at most one recursive call is carried out as part of the execution of a single recursive process.

Non-linear recursion

- This term is used to describe a recursive function where more than one recursion can be carried out as part of the execution of a single recursive process.

Mutual recursion

- Two functions are called mutually recursive if the first function makes a recursive call to the second function and the second function, in turn, calls the first one.

Fibonacci Sequence

fib(n) = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

- How can the problem be defined in terms of one or more smaller problems of the same type?

```
fib(n) = fib(n-2) + fib(n-1) for n>2
```

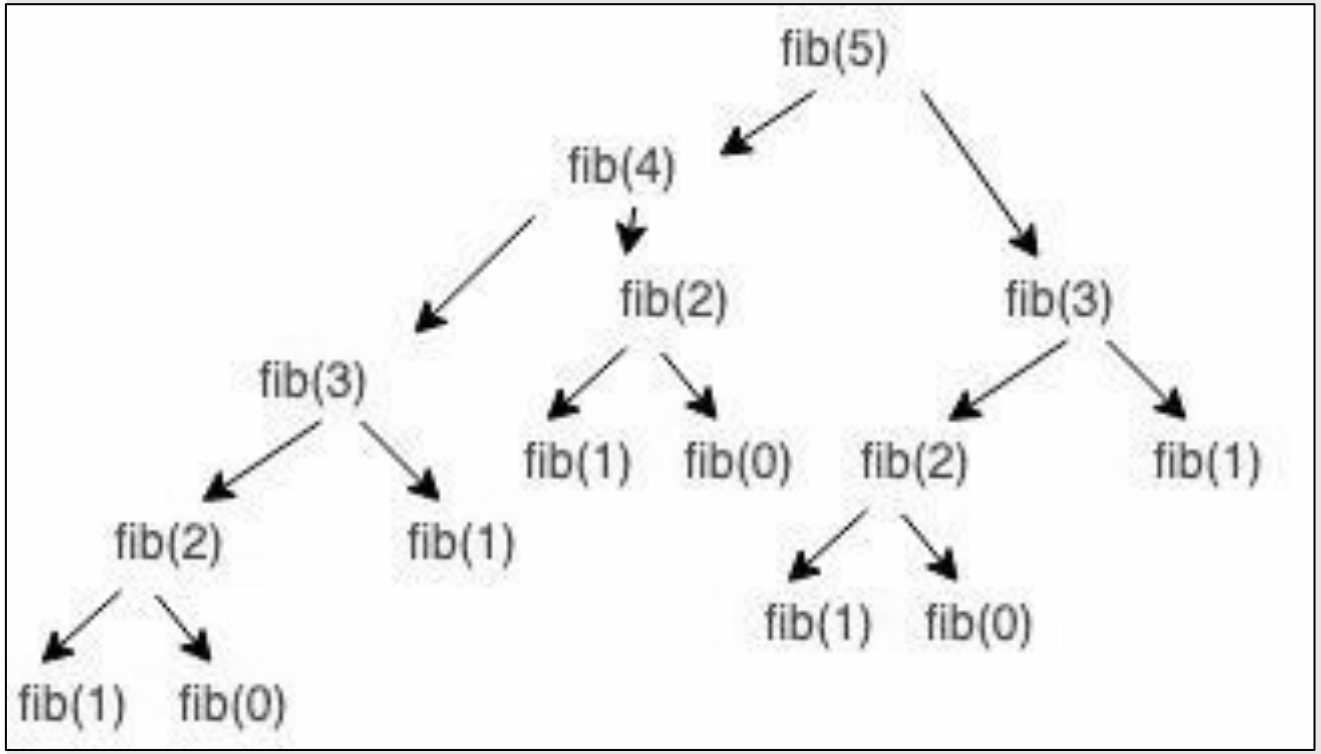
- What instance of the problem can serve as the base case?

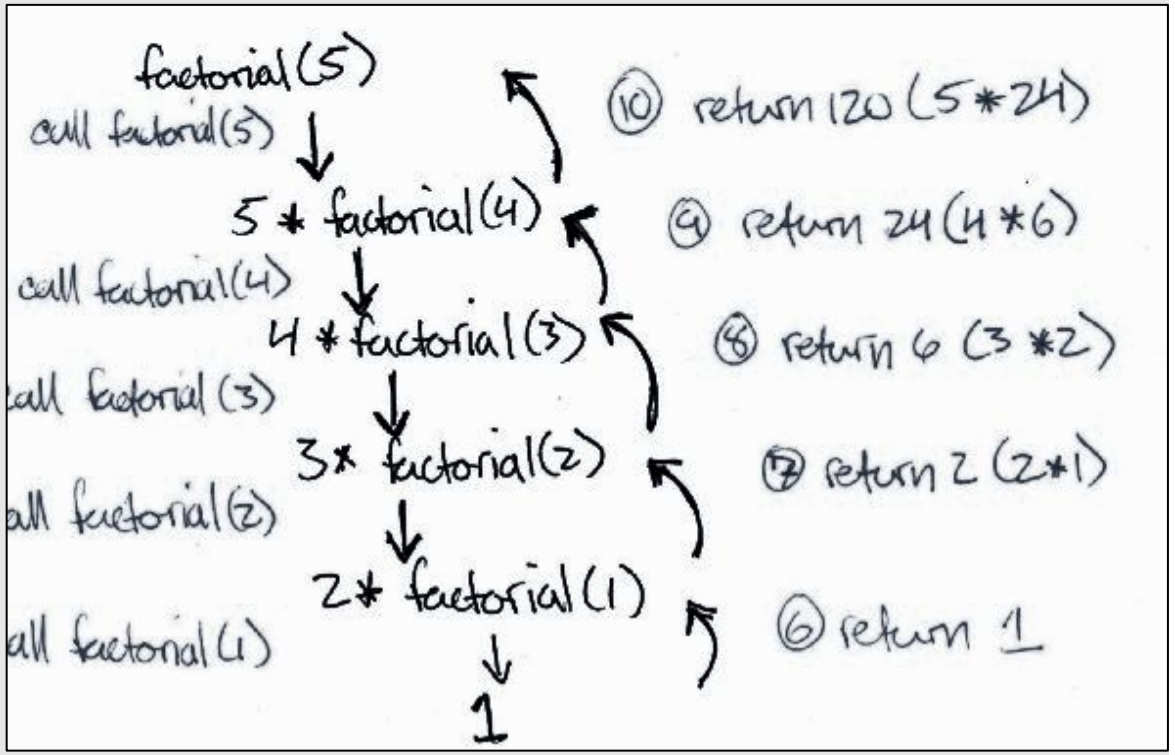
```
fib(1) = 1 for n = 1
fib(2) = 1 for n = 2
```

- As the problem size diminishes, will one reach these base cases?
  - n = non-negative integer - each call to the function will reduce the parameter n by 1 or 2
- How are the solutions from the smaller problems used to build a correct solution to the current larger problem?[fib(n) = fib(n-2) + fib(n-1)]
  - function uses non-linear recursion.

Fibonacci Sequence

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


fib(5)
fib(4)
fib(2)
fib(3)
fib(3)
fib(1)
fib(0)
fib(2)
fib(1)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)


factorial(5)
call factorial(5)
5 * factorial(4)
call factorial(4)
4 * factorial(3)
call factorial(3)
3* factorial(2)
call factorial(2)
2* factorial(1)
call factorial(1)
1
⑩ return 120 (5*24)
⑨ return 24 (4*6)
⑧ return 6 (3*2)
⑦ return 2 (2*1)
⑥ return 1

Greatest Common Divisor

The greatest common divisor of two integers is the largest integer that divides them both.

```
int gcd(int a,int b)
 {
    int remainder;
    remainder = a % b;
    if(remainder == 0)
      return b;
    else
      return gcd(b, remainder);
 }
```

Examples

Iterative:

f(n) = 1 + 2 + 3 +……..+ n

Recursive:

f(n) = 1 n=1

f(n) = n + f(n-1) n>1

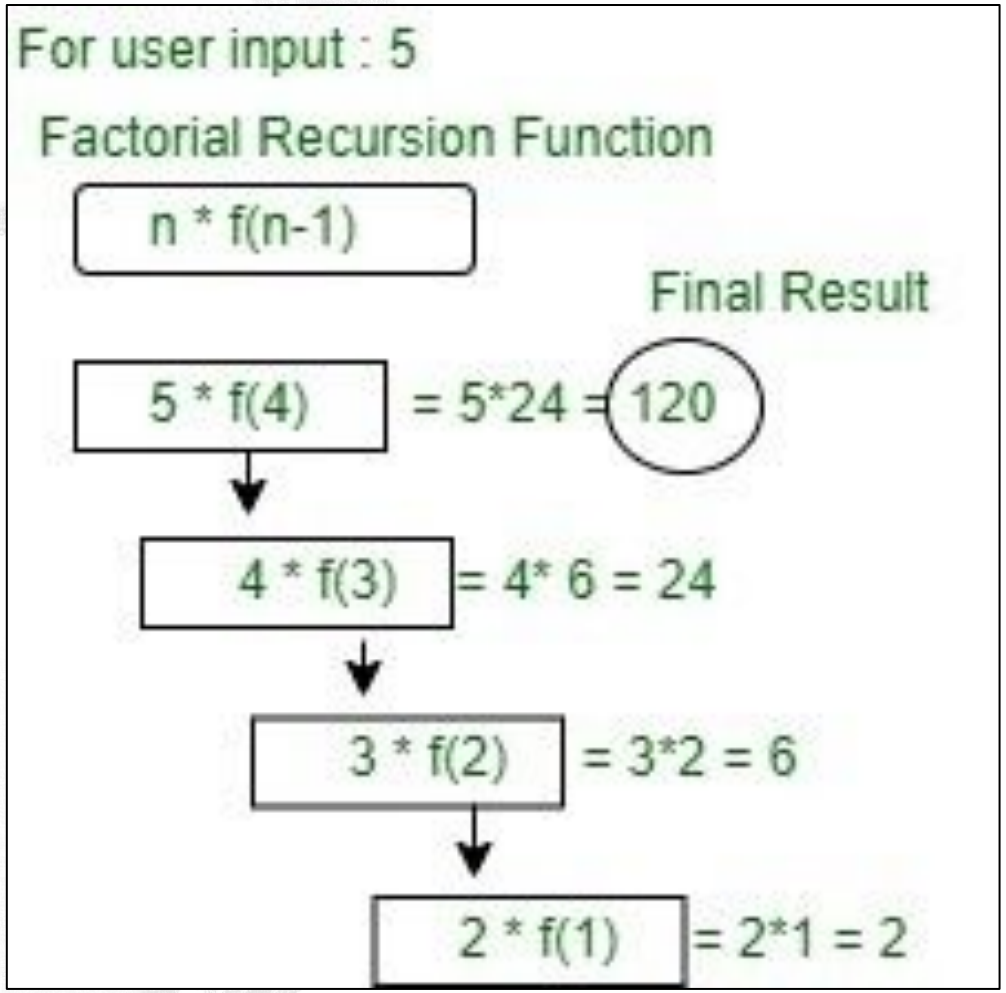
Iterative:

f(n) = 1 * 2 * 3 *……..* n

Recursive:

f(n) = 1 n=1

f(n) = n * f(n-1) n>1



Example

```
INPUT: n - a natural number.
OUTPUT: true if n is even; false otherwise

odd(n)
   if n = 0 then return FALSE
   return even(n-1)
even(n)
   if n = 0 then return TRUE
   else return odd(n-1)
```

Recursion and Iteration

- Recursion is a very powerful tool for solving complex problem, particularly when the underlying problem or data to be treated are already defined in recursive terms.
- Depending on the implementation available and the algorithm being used, recursion can require a substantial amount of runtime overhead.
- Thus, the use of recursion illustrates the classic trade off between time spent in constructing and maintaining a program and the cost in time and memory of execution of that program

Recursion and Iteration

- Two factors contribute to the inefficiency of some recursive solutions.
  - The overhead associated with function calls
  - The inefficient utilization of memory

- Every time a new recursive call is made a new set of local variables is allocated to function.
- Moreover it also slows down execution speed, as function calls require jumps, and saving the current state of the calling function onto stack before jump.
- Recursion is of value when the return values of the recursive function are used in further processing within the calling version of the function

Searching

- Among the searching algorithms, only two of them will be discussed here:
  - Sequential search
  - Binary search

Sequential Search

- Here is an implementation of this simple algorithm:

```
int Lsearch(int  ArrayElement[], int key, int ArraySize)
  {
    int i ;
    for (i = 0; i < ArraySize; i++)
        if (ArrayElement[i] == Key)
    return (i) ;
    return (-1);
}
```

Binary Search

- The C code for binary search is given below.

```
#include <stdio.h>
int binarysearch(int a[], int n, int key)
{
     int beg,mid;
     beg=0; end=n-1;
     while(beg<=end)
     {
       mid=(beg+end)/2;
      if(key==a[mid])
             return mid;
       else if(key>a[mid])
                  beg=mid+1;
        else
                  end=mid-1;
     }
   return -1;
}
```

Sorting

- Arranging elements of an array in a particular order is called sorting.
- Sorting algorithms are divided into two categories:
  - ***Internal sort:***
    - Any sort algorithm, which uses main memory exclusively during the sort.
    - This assumes high-speed random access to all memory
  - ***External sort:***
    - Any sort algorithm, which uses external memory, such as tape or disk, during the sort.

Sorting

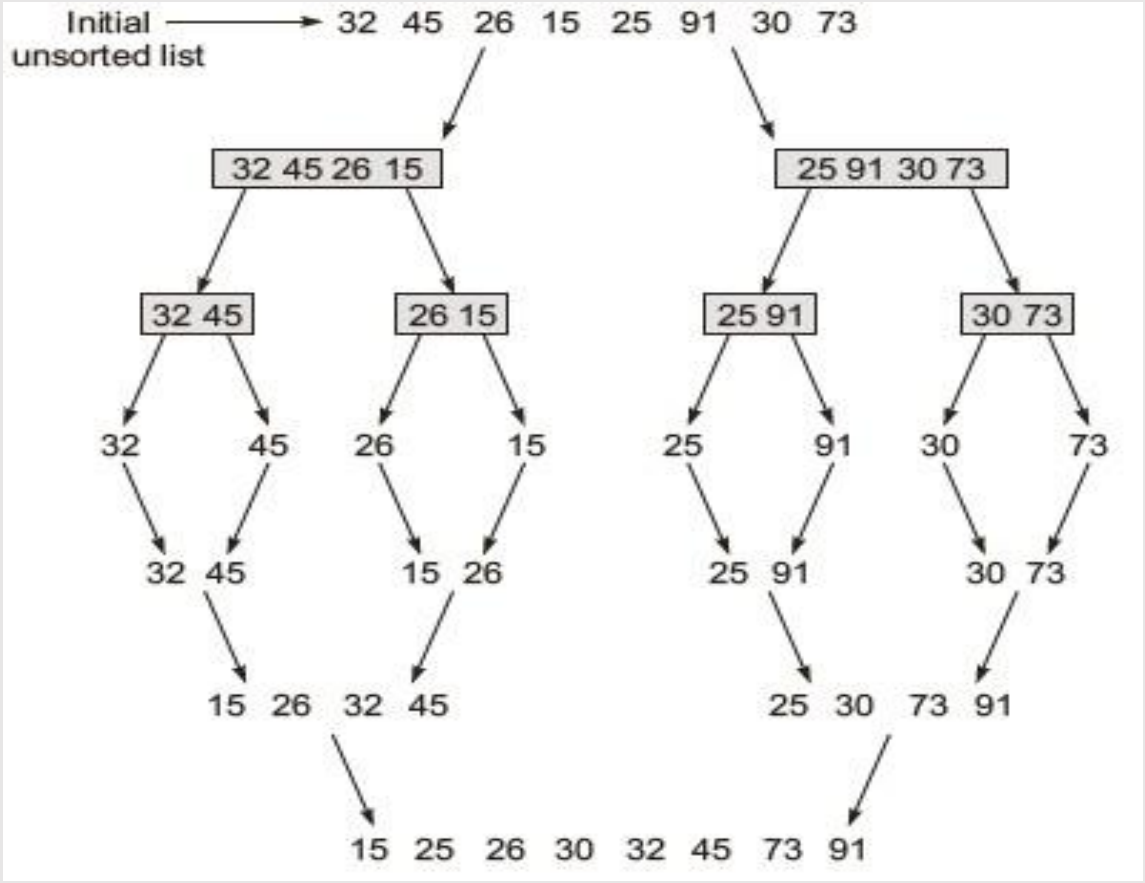
- Some of the sorting methods include:

  - Bubble sort
  - Selection sort
  - Insertion sort
  - Merge sort
  - Quick sort

Merge Sort

- The merge sort splits the data list to be sorted into two equal halves, and places them in separate arrays.
- This sorting method uses the divide-and-conquer paradigm.

Merge Sort


Initial
unsorted list
32 45 26 15 25 91 30 73
32 45 26 15
25 91 30 73
32 45
26 15
25 91
30 73
32
45
26
15
25
91
30
73
32 45
15 26
25 91
30 73
15 26 32 45
25 30 73 91
15 25 26 30 32 45 73 91

Merge Sort

- A function that implements the merge sort algorithm is given as follows:

```
 void mergesort(int array[], int n)
{
    int j,n1,n2,arr1[n],arr2[n];
       if (n<=1)return;
    n1=n/2;
    n2 = n - n1;
    for(j = 0; j<n1; j++)
        arr1[j]= array[j];
    for(j = 0; j<n2; j++)
        arr2[j]= array[j+n1];
    mergesort(arr1, n1);
    mergesort(arr2, n2);
    merge(array, arr1, n1, arr2, n2);
 }
```

Merge Sort

```
void merge (int array[], int arr1[], int n1,int
arr2[], int n2)
{
     int j, p=0, p1=0,p2=0;
     printf("\n After merging [");
     for(j=0; j<n1; j++)
     printf("%d ",arr1[j] );
     printf("] [");
     for(j=0; j<n2; j++)
     printf("%d",arr2[j] );
     printf("]");
     while ( p1 < n1 && p2 < n2 )
     {
       if( arr1[p1] < arr2[p2])
         array [p++] = arr1[p1++];
       else
         array[p++] = arr2[p2++];
      }
     while ( p1 < n1 )
       array [p++] = arr1[p1++];
     while ( p2 < n2 )
       array[p++] = arr2[p2++];
     printf(" merged array is [");
     for(j=0; j<n1+n2; j++)
       printf("%d", array[j]);
     printf("]\n");
    }
```

Quick Sort

- Quick sort is a recursively defined procedure for rearranging the values stored in an array in ascending or descending order.

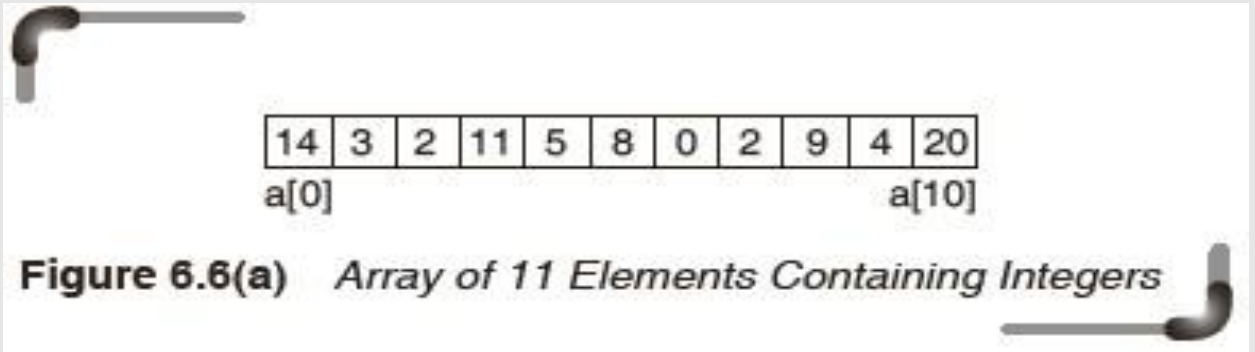


Figure 6.6(a) *Array of 11 Elements Containing Integers*

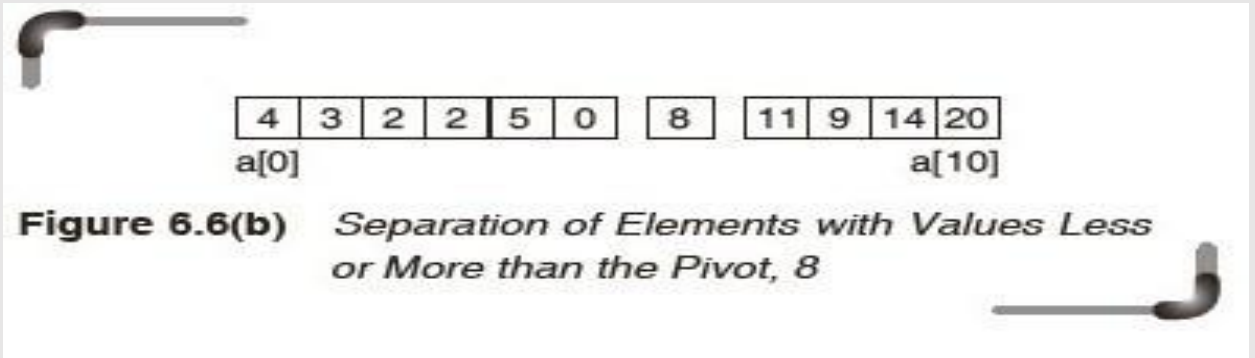


Figure 6.6(b) *Separation of Elements with Values Less or More than the Pivot, 8*

Quick Sort

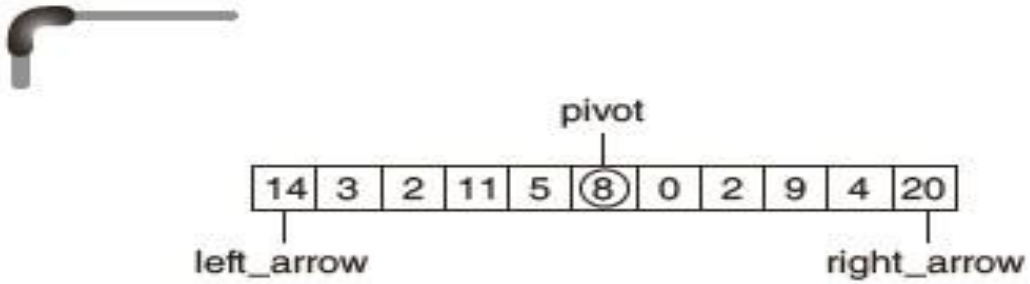


Figure 6.6(c) *Choosing the Index of the Pivot*

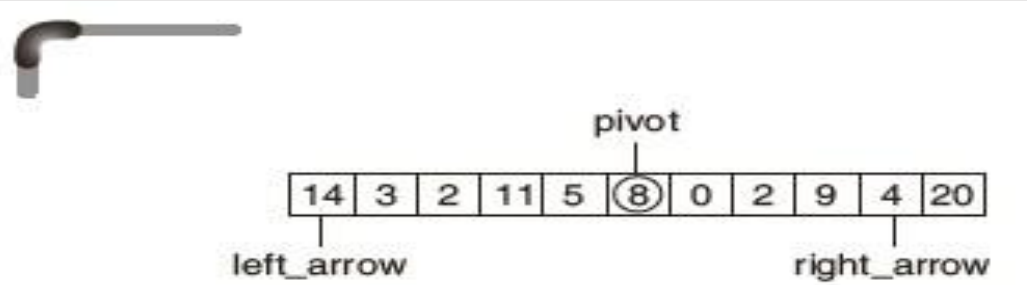


Figure 6.6(d) *Moving the right_arrow to the Left Until 'value <= pivot'*

Quick Sort

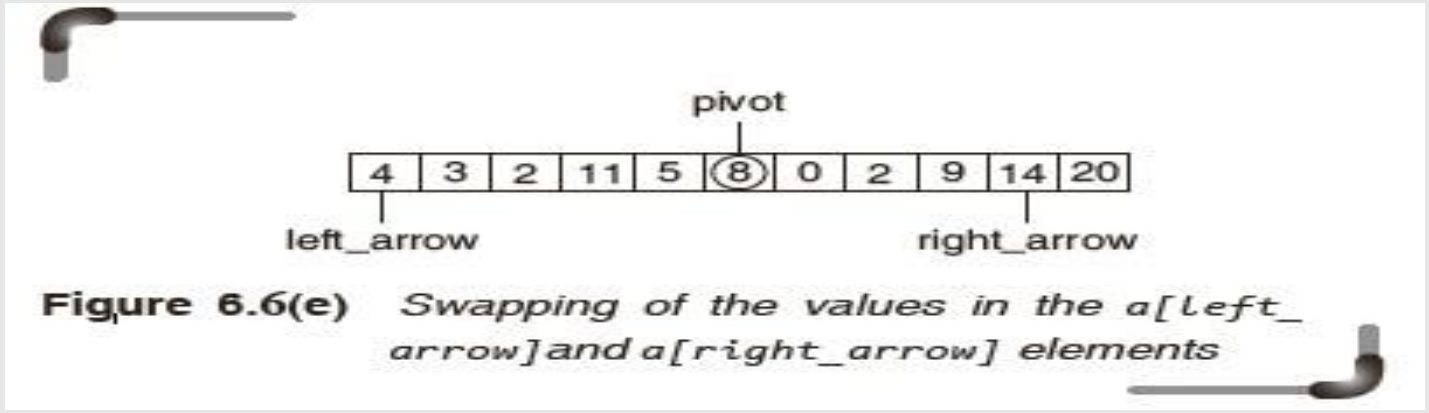


Figure 6.6(e) *Swapping of the values in the a[left_arrow] and a[right_arrow] elements*

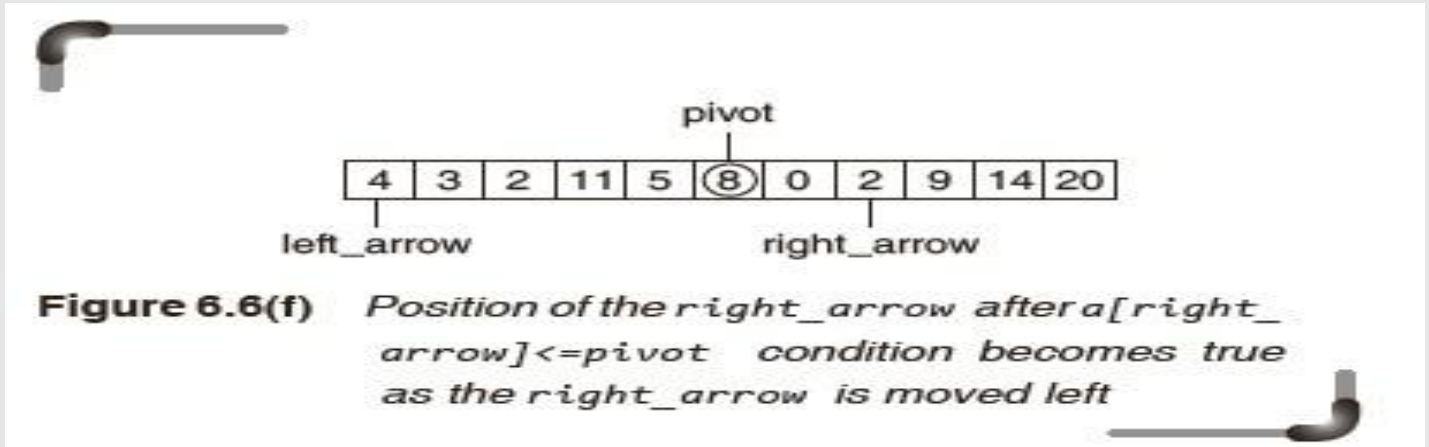


Figure 6.6(f) *Position of the right_arrow after a[right_arrow]<=pivot condition becomes true as the right_arrow is moved left*

Quick Sort

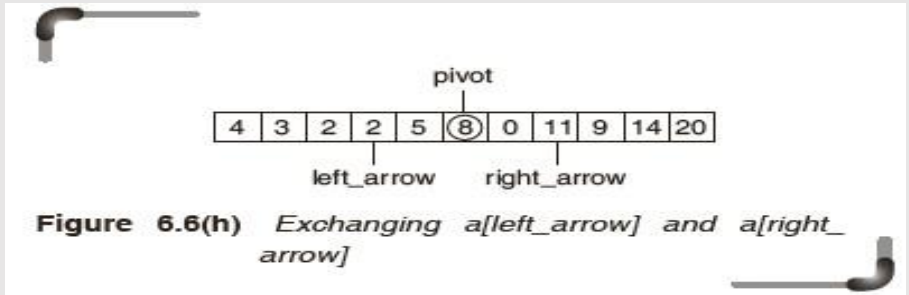


Figure 6.6(h) *Exchanging a[left_arrow] and a[right_arrow]*

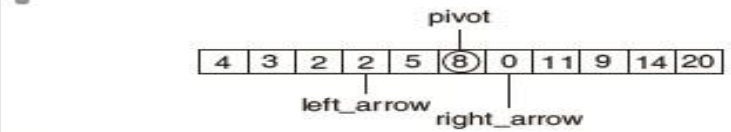


Figure 6.6(i) *Moving right_arrow to the Left till a[right_arrow]<=pivot*

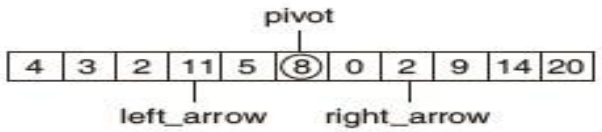


Figure 6.6(g) *Position of left_arrow after a[left_arrow]>=pivot condition becomes true as the left_arrow is moved right*

Quick Sort

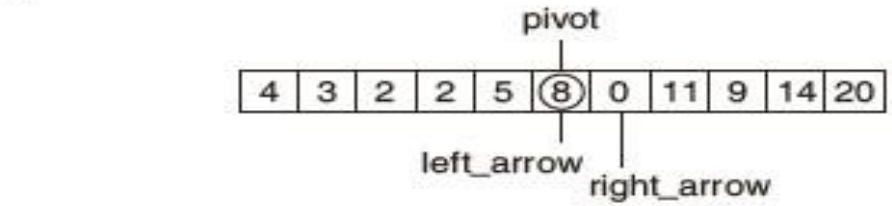


Figure 6.6(j) *Moving left_arrow Right till a[left_arrow]>= pivot*

Figure 6.6(k) *Exchanging Pivot with right_arrow Content*

Quick Sort

- The procedure's terminating condition left_arrow >right_arrow is now true, and the first sub-division of the list (i.e., array) is now complete.

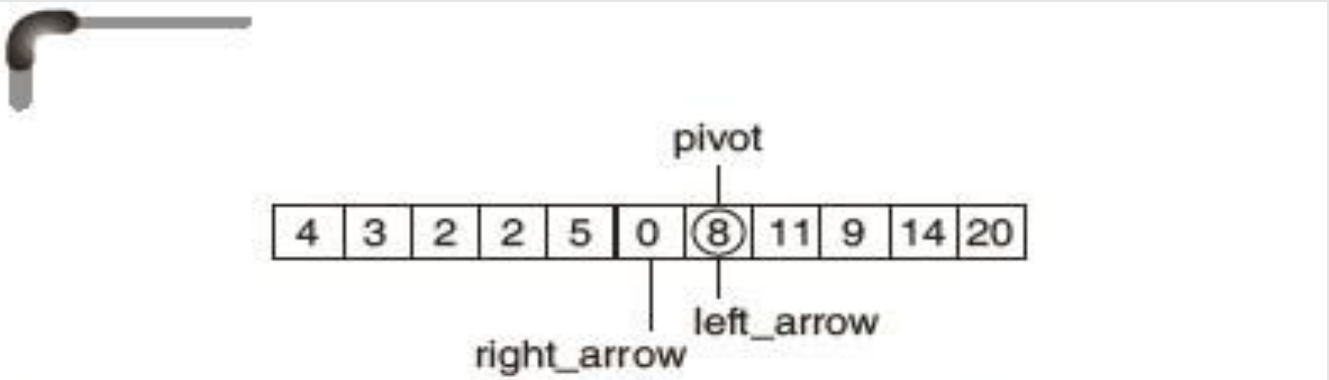


Figure 6.6(I) *The right_arrow is Moved Left and the left_ arrow is Moved Right*

Quick Sort

- The quick sort procedure is coded as a recursive C function. This can be seen as follows:

```
void quick_sort(int list[], int left, int right)
{
   int pivot, left_arrow, right_arrow;
   left_arrow = left;
   right_arrow = right;
   pivot = list[(left + right)/2];
   do
   {
     while(list[right_arrow] > pivot)
     right_arrow--;
     while(list[left_arrow] < pivot)
     left_arrow++;
     if(left_arrow <= right_arrow)
     {
       swap(list[left_arrow], list[right_arrow]);
       left_arrow++;
       right_arrow--;
     }
   }
   while(right_arrow >= left_arrow);
   if(left < right_arrow)
   quick_sort(list, left, right_arrow);
   if(left_arrow < right)
   quick_sort(list, left_arrow, right);
}
```

Analysis of Algorithms

- In analyzing an algorithm, rather than a piece of code, the number of times ‘the principal activity’ of that algorithm is performed, should be predicted.
- The efficiency of an algorithm is determined by the amount of time it takes to run the program and the memory space the program requires.

Complexity

- Complexity of an algorithm is a measure of the amount of time and/or memory space required by an algorithm for a given input.
- It is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.
- Usually there are natural units for the domain and range of this function.

Complexity

The following are the two main complexity measures of the efficiency of an algorithm:

- ***Time complexity:***
  - It is a function describing the amount of time an algorithm takes with respect to the amount of input provided to the algorithm
  - It is denoted as *T(n)* where *n* is the size of the input.
- ***Space complexity:***
  - It is a function describing the amount of memory (space) an algorithm takes with respect to the amount of input provided to the algorithm.
  - It is denoted as *S(n)* where *n is* the size of the input.

Complexity

- ***Worst-case complexity:***
  - It is defined by the maximum number of steps taken on any instance of input size *n*.
- ***Best-case complexity:***
  - The best-case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of input size *n.*
- ***Average-case complexity:***
  - It is defined by the average number of steps taken on any instance of input size *n.*

Big-O Notation

- It can be defined as, if $f(n)$ and $g(n)$ are functions defined for positive integers then $f(n) = O(g(n))$ if there exists a 'c' such that $|f(n)| <= c|g(n)|$ for all sufficiently large positive integers n.

$$f(n) = O(g(n)) \text{ is true if } \lim_{n \to a} f(n)/g(n) \text{ is a constant.}$$

Properties of the Big-O Notation

- O(k*f(n)) = O(f(n)), therefore, constants can be ignored
- O(f(n)*g(n)) = O(f(n)) * O(g(n))
- O(f(n)/g(n)) = O(f(n)) / O(g(n))
- O(f(n)+g(n)) = Max[O(f(n)), O(g(n))]
- The powers of n are ordered according to the exponent: $n^a=O(n^b)$ iff a <= b.
- The order of log n is independent of the base taken: $\log_a n=O(\log_b n)$ for all a, b > 1.

Analysis of Quick Sort

- Best case:
  - In the best case, a perfect partition is to be set every time.
  - If we let T(n) be the running time of quick sorting n elements, then

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

since partition runs in O(n) time.

Thus, the Best case running time is T(n)= O(n log n).

Analysis of Quick Sort

- Worst case:

  - The partition element was always the greatest value of the one remaining to be sorted.

  - Total run of partition is

    $1+2+3+...+(n-1)$

    $=(n-1)n/2.$

    Thus, the worst case running time is $O(n^2)$.

Analysis of Quick Sort

- Average case:
  - The average case running time is certainly difficult to ascertain because one cannot get any sort of partition.
  - It is assumed that each possible partition (0 and n-1, 1 and n-2, 2 and n-3, etc.) is equally likely.
  - Average case running time is T(n)= O(n log n).

Disadvantages of Complexity Analysis

- Many algorithms are simply too hard to analyze mathematically.
- There may not be sufficient information to know what the most important 'average' case really is, therefore analysis is impossible.
- Big-O analysis only specifies how it grows with the size of the problem, not how efficient it is.
- If there are no large amounts of data, algorithm efficiency may not be important.

Thank You!