**Unit – 1: Introduction**

Algorithm Specification, Performance analysis, Performance Measurement. Arrays: Arrays, Dynamically Allocated Arrays. Structures and Unions. Sorting: Motivation, Quick sort, How fast can we sort, Merge sort, Heap sort

**1.1 Algorithm Specification:**

Algorithm Definition: An algorithm is a finite set of instructions which, if followed, accomplish a particular task. In addition every algorithm must satisfy the following criteria:

(i) input: there are zero or more quantities which are externally supplied;

(ii) output: at least one quantity is produced;

(iii) definiteness: each instruction must be clear and unambiguous;

(iv) finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

(v) effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite as in (iii), but it must also be feasible.

Two Examples should help to illustrate the process of translating a problem into an algorithm.

Example1 :Problem statement: (selection sort): Suppose we must devise a program that sorts a set of n>=1 integers.simple solution is given by the following;

For(i=0; i<n; i++)

{

Examine list[i] to list[n-1] and suppose that the

smallest integer is at list[min];

Interchange list[i] and list[min];

}

**Program: Selection sort program**

To Turn program into real C Program, two clearly defined subtasks:Finding smallest integer and interchanging it with list[i].

Void swap(int \*x, int \*y)

{

int temp = \*x;

\*x = \*y;

\*y = temp;

}

**Program: Swap Function**

Using the function, suppose and b are declared as ints. To swap their values one would say:

Swap(&a,&b);

Passing to swap the addresses of a and b.The macro version of swap is

#define SWAP(x,y,t) ( (t) = (x), (x) = (y) , (y) = (t) ) )

The functions code is easier to read than that of the macro but the macro works with any data type. The function’s code is easier to read than that of the macro but the macro works with any data type.

Using Swap macro writing the selection sort program.

#include<stdio.h>

#include<math.h>

#define MAX\_SIZE 101

#define SWAP(x,y,t) ((t) = (x), (x) = (y), (y) = (t) )

Void sort(int [],int ); /\* Selection Sort \*/

Void main(void)

{

int i,n;

int list[MAX\_SIZE];

printf(“ \n Enter the number of numbers to generate \n”);

scanf(“%d”,&n);

if(n<1 || n > MAX\_SIZE)

{

printf(“\n Improper value of n \n”);

exit(0);

}

for(i=0;i<n;i++)

{

list[i]= rand() %100

printf(“%d”, list[i]);

}

sort(list,n);

printf(“\n Sorted array \n”);

for(i=0;i<n;i++)

printf(“%d “.list[i]);

printf(“\n”);

}

void sort(int list[],int n)

{

int i , j, min,temp;

for(i=0;i<n-1 ;i++)

min=i ;

for(j = i+1;j<n;j++)

if(list[j] < list[min])

min = j;

SWAP(list[i],list[min],temp);

}

}

Example2: Assume that we have n>=1 distinct integers that are already sorted and stored in the array list. That is list[0]<=list[1]<=…list[n-1]. We must figure out if an integer searchnum is in this list.If we compare list[middle]

with searchnum, we obtain one of three results.

(1). Searchnum<list[middle], then right = middle-1

(2) searchnum = list[middle] then return middle

(3) searchnum>list[middle] then left = middle+1

The below program implement the above strategy

while(there are more integers to check)

{

middle = (left+right)/2;

if(searchnum<list[middle])

right = middle-1;

else if(searchnum == list[middle])

return middle;

else

left = middle+1

we can handle the comparisons through either a function or a macro.

* We return a negative number(-1) if the first number is less than the second.
* We return zero if the two numbers are equal.
* We return positive ineger number(1) if the first number is greater than the second.

int compare(int x, int y)

{

if(x<y) return -1;

else if(x==y) return 0;

else return 1;

}

Program: Comparision of two integers

The macro version is

#define COMPARE(x,y) ((x>y)<(y))?-1:( (x) ==(y))?0:1)

Putting all this information together gives us binary search

int binsearch(int list[], int searchnum, int left, int right )

{

int middle;

while(left <= right )

{

middle = (left+right)/2;

Switch(COMPARE(list[middle],search num))

{

case -1: left= middle+1;

break;

case 0 : return middle;

case 1 : right = middle-1;

}

}

return -1;

}

1.1.2: Recursive Algorithms:

The below program will explain iterative version of binary search. Here, we can see that there are two ways to terminate the search one signaling a success (list[middle] == searchnum) the other signaling faiure.

int Binarysearch(int list[], int searchnum, int left, int right)

{

int middle;

if(left<=right)

{

middle= (left+right)/2;

switch( compare(list[middle], searchnum))

{

case -1: return binsearch(list,searchnum,middle+1,right);

case 0: return middle;

case 1: return;

binsearch(list, searchnum,left,middle-1);

}

}

return -1;

}

Program : Recursive Implementaion of Binary Search

**1.2 Performance analysis:**

There are many criteria upon which we can judge a program including :

1. Does the program meet the original specifications of the task ?
2. Deos it work correctly ?
3. Does the program contain documentation that shows how to use it and how it works ?
4. Does the program effectively use functions to create logical units
5. is the program code readable ?
6. Does the programs running time acceptable for the task ?

Performance evaluation divided into two distinct fields.First field is performance analysis and second is performance measurement.

* Performance analysis depend upon space and time complexities.

Space and time Complexity definitions: Space Complexity of a program is the amount of memory that it needs to run to completion. The time complexity of a program is the amount of computer time that it needs to run to completion.

Space Complexity:

Fixed Space Requiremnets: This component refers to space requirements that do not depend on number and size of the programs inputs and outputs. It includes the instruction space(space needed to store the code), space for simple variables. Fixed size structured variables (such as structs) and constants.

Variable space requirements: This component consists of the space needed by structured variables whose size depends on the particular instance, I of the problem being solved. Additional space required when a function uses recursion. The variable space requirements of a program P working on an instance I is denoted Sp(I)

Total Space requirements S(p) = c+Sp(I)

Here c represents fixed size, Sp(I) represents variable space.

Example:We have a function , abc which accepts three simple variables as input and returns a simple value as output. According to classification given , this function has only fixed space requirements therefore

Sabc (I) = 0

float abc(float a,float b,float c)

{

return a+b+b\*c+(a+b-c)/(a+b)4.00;

}

Example: we want to add a list of numbers although the output is a simple value, the input includes an array.

Float sum(float list[ ],int n)

{

float tempsum=0;

int I;

for(i=0;i<n;i++)

tempsum += list[i];

return tempsum;

}

When an array is passed as an argument to a function, c interprets it as passing the address of first element of the array.

Example:

float rsum(float list[], int n)

{

if(n) return rsum(list,n-1)+list[n-1];

return 0;

}

Program: Recursive function for summing a list o numbers

Table: space needed for one recursive call of program

|  |  |  |
| --- | --- | --- |
| Type | Name | Number of bytes |
| Array pointer(Parameter) | List[ ] | 4 |
| Integer(parameter) | N | 4 |
| Return address(used internally) |  | 4 |
| Total per recursive call |  | 12 bytes |

**1.3 Performance Measurement**

The functions we need to time events are part of c’s standard library, and are accessed through statement:

#include<time.h>. There are actually two different methods for timing events in c. The below table shows the major differences between those two methods.

|  |  |  |
| --- | --- | --- |
| Method1 Method2 | | |
| Start timing | Start = clock(); | Start = time(NULL); |
| Stop timing | Stop = clock); | Stop = time(NULL); |
| Type returned | Clock\_t | Time\_t |
| Result in Seconds | Duration =  ((double)(stop-start))/  CLOCKS\_PER\_SEC | Duration =  (double)difftime(stop,start); |

Table: Event timing in C

* The below two examples shows the worst case for selection sort. This worst case occurs when the elements are in reverse order.

**Program:First timing program for selection sort**

#include<time.h>

#include “selectionsort.h”

#define MAX\_SIZE 1001

void main(void)

{

int i,n,step = 10;

int a[MAX\_SIZE];

double duration;

clock\_t start;

/\* times for n=0,10…..100,200,…..1000 \*/

printf(“ \n time\n”);

for(n=0;n<=1000; n += step)

{

/\* get time for size n \*/

/\* initialize with worst case data \*/

for(i=0;i<n;i++)

a[i] = n-1;

start = clock();

sort(a,n);

duration = ((double) (CLOCK()- start))/ CLOCKS\_PER\_SEC;

printf(“ %6d %f\n”,n,duration);

if(n==100) step = 100;

}

}

Program: More accurate timing program for selection sort

#include<time.h>

#include “selectionsort.h”

#define MAX\_SIZE 1001

void main(void)

{

int i,n,step = 10;

int a[MAX\_SIZE];

double duration;

/\* time for n=0,10,…..100,200…..1000 \*/

printf(“ n repetitions time \n”);

for(n=0; n <=1000; n += step

{

long repetitions = 0;

clock\_t start = clock();

do

{

repetitions++;

/\* intitialize with worst-case data \*/

for(i=0;i<n;i++)

a[i] = n-1;

sort(a,b);

}

while(clock()-start<1000)

/\* repeat until enough time has elapsed \*/

duration = ((double) (clock()-start)/CLOCKS\_PER\_SEC;

duration /= repetitions;

printf(“%6d %9d %f\n”,n,repetitions,duratopm);

if(==100) step = 100;

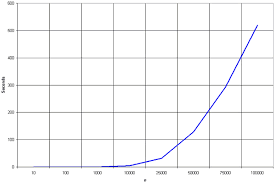
}

}

The results from above program are displayed in figures.

|  |  |  |
| --- | --- | --- |
| N | Repetitions | time |
| 0 | 8690714 | 0.000000 |
| 10 | 2370915 | 0.000000 |
| 20 | 604948 | 0.000002 |
| 30 | 329505 | 0.000003 |
| 40 | 205605 | 0.000005 |
| 50 | 145353 | 0.000007 |
| 60 | 110206 | 0.000009 |

Fig: Worst case performance of selection sort(seconds)



graph of worst-case performance of selection sort. X axis is n value and y axis is time .

**1.4 Arrays:**

1.4.1 Abstract Data Type

ADT concerns concerned with the operations that can be performedon array.Most languages provide only two standard operations for arrays, one that retrieves a value, and second that stores a value.

ADT Array is

objects: A set of pairs <index,value> where for each value of index there is a value from the set item.Index is a finite ordered set of one or more dimensions .For example ,{0,1,…n-1} for one dimension, {(0,0),(0,1),(0,2), (1,0),(1,1),(1,2),(2,0),(2,1),(2,2)} for two dimensions etc.

functions

for all A∈ Array, i∈index, x∈ itme,j,size∈integer

Array create(j,list) = return an array of j dimesions where list is a j-tuple, whose i**th** element is the

size of the i**th** dimension.items are undefined

item Retrieve(A,i) = if(i∈ index) return the item associated with index value I in an array A.

Array store(A,i,x) = if(i in index) returns an array that is indentical to array A except the new pair <i,x> has been inserted else return error.

1.4.2 Arrays in C

A one-dimensional array in c is declared implicitly by appending brackets to the name of a variable .

For example

int list[5], \*plist[5];

Example program1:

#define MAX\_SIZE 100

float sum(float [ ], int);

float input[MAX\_SIZE],answer;

void main(void)

{

int i;

for(i=0;i<MAX\_SIZE;i++)

input[i] = I;

answer = sum(input, MAX\_SIZE);

printf(“ The sum is : %f\n”, answer);

}

Example program2: one dimensional array accessed by address

void print1(int \*ptr, int rows)

{ /\* print one dimensional array using a pointer \*/

int i;

printf(“ Adress contents \n”);

for(i-0;i<rows;i++)

printf(“%8u%5d\n”,ptr+I, \*(ptr+i));

printf(“\n”);

}

**1.5 Dynamically Allocated Arrays**

1.5.1: one dimensional Arrays

Without mentioning the size we can declare the arrays with the help of dynamic memory allocation functions.the following program is example for dymanically allocated one dimensional arrays.

#include<stdio.h>

void main()

{

int i,n,\*list;

clrscr();

printf(“\n Enter n value \n”);

Scanf(“%d”,&n);

if(n<1)

{

printf(“\n Improper value of n \n”);

exit(0);

}

list = malloc(n\*sizeof(int));

printf(“\n Enter %d values”,n);

for(i = 0;i<n;i++)

scanf(“%d”,&list[i]);

printf(“\n print numbers\n”);

for(i=0;i<n;i++)

printf(“%d\t”,list[i]);

getch();

}

In the above program , we are not mentioning the array size, but reading and printing array values.

1.5.1: two dimensional Arrays

Without mentioning the size we can declare the two dimensional arrays with the help of dynamic memory allocation functions.the following program is example for dymanically allocated two dimensional arrays.

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

void main()

{

int r,c,i,j,\*\*list;

clrscr();

printf("\n Enter the number of rows and columns \n");

scanf("%d%d",&r,&c);

if(n<1)

{

printf("improper value of n\n");

exit(0);

}

list =malloc((r\*c)\*sizeof(int));

printf("\n Enter %d values \n",r\*c);

for(i=0;i<r;i++)

for(j=0;j<c;j++)

scanf("%d",&list[i][j]);

printf("\n print numbers \n");

for(i=0;i<r;i++)

for(j=0;j<c;j++)

printf("%d\t ",list[i][j]);

getch();

}

**1. 6 Structures and unions**

1.6.1: structures:

Structure definition: Structure is a collection of one or more variables of different data types grouped together under one single name. one can create structure within structure i.e is nested structure.

syntax of structure :

struct structure\_name {

data type variblename1;

data type variblename2;

.

.

data type variblenamen;

}objectname;

Here structure name is an optional.

Example :

struct {

char name[30];

int age;

float salary;

}person;

We can create our own structure data types by using the typedef statement below.

typedef struct {

char name[10];

int age;

float salary;

}humanbeing;

This says that humanbeing is the name of the type defined by the structure definition and we may follow this definition with declarations of variables such as:

humanbeing person1,person2;

we might have a program segment says

if(strcmp(person1.name,person2.name))

printf(“ The two people do not have the same name \n”);

else

printf(“The two people have the same name \n”);

We can also embed a structure within a structure. For example, associated with one humanbeing structure we may wish to include the date of his or her birth. We can do this by writing:

typedef struct {

int month,day,year;

}date;

typedef struct {

char name[10];

int age;

float salary;

date dob;

}humanbeing;

A person born on feb 11 1944 would have the values for the date struct set as

Person1.dob.month = 2; person1.dob.day = 11 ; person1.dob.year = 1944;

1.6.2:unions: Union declaration is similar to structure , but the fields of a union must share their memory space.

This means that only one field of the union is active at any given time

Example

typedef struct {

enum tagfield {female, male } sex;

union {

int children;

int beard;

}u;

}sexType;

typedef struct {

char name[10];

int age;

float salary;

date dob;

sexType sexinfo;

}humanbeing;

humanbeing peron1,person2;

we could assign values to person1 and person2 as

person1.sexinfo.sex = male;

person1.sexinfo.u.beard = FALSE;

and

person2.sexinfo.sex = female;

person2.sexinfo.u.children =4;

1.6.3: self referential structures:

A Self referential structure is one in which one or more of its components is a pointer to itself.

Example :

typedef struct {

char data;

struct list \*link;

}list;

Consider these statements which create three structures and assign values to their respective fields

list item1,item2,item3;

item1.data = ‘a’;

item2.data = ‘b’;

item3.data= ‘c’;

item1.link = item2.link = item3.link = NULL;

Structures item1,item2 and item3 each contain the data item a,b and c respectively, and the null pointer. We can attach these structures together by replacing the null link field in item2 with one that points to item3 and by replacing the null link field in item1 with one that points to item2.

item1.link = &item2;

item2.link = &item3;

**1.7.Sorting: Motivation**

1.7 :Motivation: We use the term list to mean a collection of records, each record having one or more fields. The fields used to distinguish among the records are known as keys. For instance, we may regard a telephone directory as a list , each record having three fields: name, address and phone number. The key is usually persons name.However, we may wish to locate the record corresponding to a given number in which case the phone number field would be the key. In yet another application we may desire the phone number at a particular address, so the address field could also be the key.

One way to search for a record with the specified key is to examining the list of records in left-to-right or right-to-left order. Such a search is known as a sequential search.

The below program explains sequential search function that examines the record in the list a[1:n] in left-to-right order.

int seqsearch(element a[ ], int k,int n)

{

/\* search a[1:n]; return the least i such that a[i].key = k; return 0, if k is not in the array \*/

int i;

for(i=1;i<=n && a[i].key!=k;i++)

;

if(i>n) return 0;

return i;

}

So, the average number of comparisons for a successful search is

( ∑ i)/n = (n+1)/2

1<=i<=n

Interpolation search can be used only when the list is ordered.

Let us now look at another example in whose the use of ordered list greatly comparing two list of records containing data that are essentially the same but have been obtained from two different sources .Such a problem can arise , for instance in the case of United states Internal Revenue Service (USIRS), which might receive millions of forms from various employers stating how much they paid their employees and then another set of forms from individual employees stating how much they received. So we have two list and there is no discrepancy between two tlist.

Here list1 be the employer list,

list2 be the employee list

Example1: program for two unordered list

void verifiy(element list1[ ], element list2[ ],int n, int m)

{

int i,j,marked[MAX\_SIZE];

for(i=1;i<=m;i++)

marked[i] = FALSE;

for(i=1;i<=n;i++)

if((j=seqsearch(list2,m,list1[1].key))==0)

printf(“%d is not in list2 \n”, list1[i].key);

else

marked[j] = TRUE;

for(i=1;i<=m;i++)

if(!Marked[i])

printf(“%d is not in list1 \n”,list2[i].key);

}

Example2: Verifying two unsorted lists using sequential search

Void verify2(element list1[ ], element list2[ ], int n, int m)

{

int i,j;

sort(list1,n); sort(list2,m);

i = j =1;

while(i<=n && j<=m)

if(list1[i].key <list2[j].key)

{

printf(“%d is not in list2 \n”,list1[i].key);

i++;

} else if(list1[i].key == list2[j].key ) {

i++;

j++;

}

else

{

printf(“ %d is not in list \n”,list2[j].key);

j++;

}

for(;i<=n;i++)

printf(“ %d is not in list2 \n”,list1[i].key);

for(;j<=m;j++)

printf(“%d is not in the list1 \n”,list2[j].key);

}

In the example1, we are searching element without sorting operation.But in the example2 , we are searching for the elements with sorting operations. If we are following first example, it will take long time to search element. If we are following second example, it will take less time to search the element.

**1.8 Quick Sort or Partitioning Sort**

Quick sort is based upon the divide and conquer technique.for example take the some data elements.

5 30 35 16 23 42 52

Intially second element 30 is comparing with first element , it its less element consider first element as left side element otherwise right side element

left side elements 30 right side elements

5 16 23 35 42 52

left side 16 right side leftside 42 right side

5 23 35 52

Program :

#include<stdio.h>

#include<conio.h>

main()

{

int a[50],i,n;

clrscr();

printf("\n How many elements do you enter \n");

scanf("%d",&n);

printf("\n Enter %d elements \n",n);

for(i=1;i<=n;i++)

scanf("%d",&a[i]);

printf("\n Elements after sorting \n");

quick\_sort(a,n);

getch();

}

quick\_sort(int a[ ],int n)

{

int left[30],right[30],l=0,r=0;

int i;

output : How many elements do you enter

7

Enter 7 elements

5 30 35 16 23 42 52

Elements after sorting

5 16 23 30 35 42 52

if(n>0)

{

for(i=2;i<=n;i++)

if(a[i]<a[1])

left[++l] = a[i];

else if(a[i]>a[1])

right[++r] = a[i];

quick\_sort(left,l);

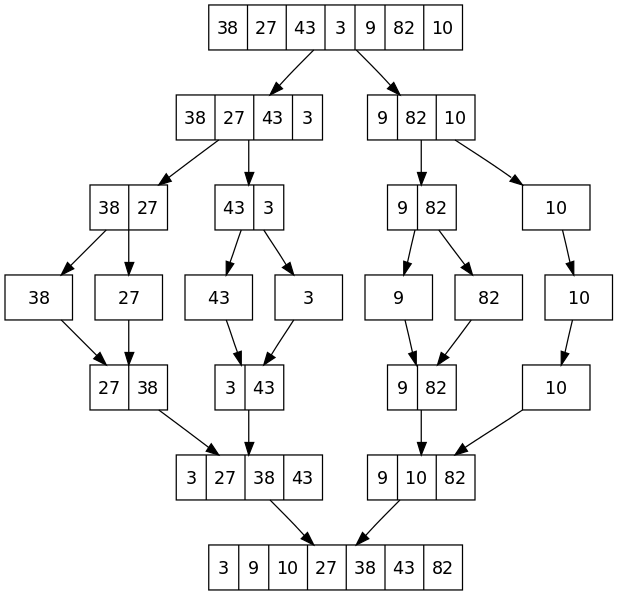
printf(" %d",a[1]);

quick\_sort(right,r);

}}

**1.10 Merge Sort**

Merge sort: Merge sort also based upon divide and conquer technique.Here we are dividing the elements as left side elements and right side elements base upon mid.



Program :

#include<stdio.h>

#include<conio.h>

int a[50],n;

void merge(int low,int mid,int high)

{

int b[50],i=low,j=low,k=mid+1;

while(i<=mid && k<=high)

if(a[i]<a[k])

b[j++] = a[i++];

else

b[j++] = a[k++];

while(i<=mid)

b[j++] = a[i++];

while(k<=high)

b[j++] = a[k++];

for(i=low;i<=high;i++)

a[i] = b[i];

}

void mergesort(int low,int high)

{

int mid;

if(low<high)

{

mid =(low+high)/2;

mergesort(low,mid);

mergesort(mid+1,high);

merge(low,mid,high);

}

}

main()

{

int i;

clrscr();

printf("\n how many elements do u want \n");

scanf("%d",&n);

printf("\n Enter %d elements \n",n);

for(i=1;i<=n;i++)

scanf("%d",&a[i]);

mergesort(1,n);

printf("\n Elements after sorting \n");

for(i=1;i<=n;i++)

printf(" %d",a[i]);

getch();

}

output :

How many elements do u want

7

Enter 7 elements

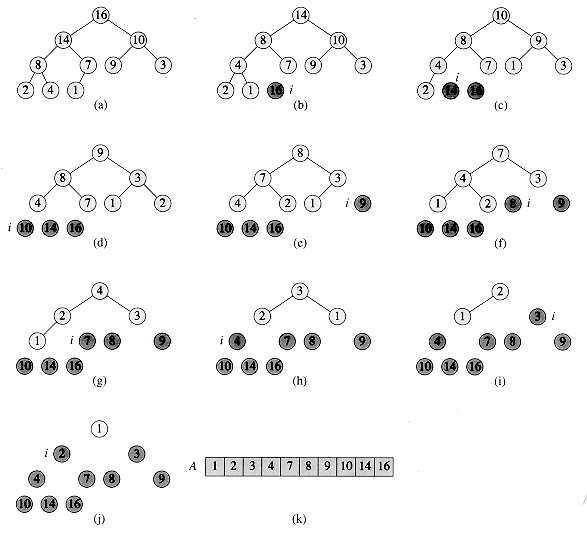
38 27 43 3 9 82 10

Elements after Sorting

3 9 10 27 38 43 82

**Heap Sort**

Before going to heap sort, first we have to know the Tree and binary trees. Here tree having root and child nodes. Binary tree having not more than two Childs. Initially take the input data and need to create the binary tree. Next we have to apply Max Heap Tree property, here max heap property means root node is always greater or equal to child nodes. next exchange the root node value and maximum position value.



By using adjust, heap sort and swap function implementing heap sort program

Void adjust(element a[ ], int root, int n)

{

/\* adjust the binary tree to establish the heap \*/

int child,rootkey;

element temp;

temp = a[root];

rootkey = a[root].key;

child = 2 \* root; /\* left child \*/

while(child <=n) {

if((child <n) &&

(a[child].key < a[child+1].key))

child++;

if(rootkey > a[child].key) /\* compare root and max.child \*/

break;

else {

a[child/2] = a[child]; /\* move to parent \*/

child \*= 2;

}

}

a[child/2] = temp;

}

**Program : Adjusting a max heap**

void heapsort(element a[ ], int n )

{

/\* perform a heap sort on a[1:n] \*/

int i,j;

element temp;

for(i=n/2;i>0;i--)

adjust(a,I,n);

for(i=n-1;i>0;i--)

{

SWAP(a[1],a[i+1],temp);

adjust(a,1,i);

}

}

**Program: Heap sort**