QUICK_sod (A, N, BEG, END, LOC)

- Here A is an arry with N elements. BEG & END are boundary values of sublist A to which procedure is applied
- LEFT & RIGHT are local variable contain boundary values of elements not been scanned

1. Set LEFT := BEG , RIGHT := END & LOC := BEG,

2. [Scan from right to left until smaller element is found]

   (a) Repeat while A[LOC] ≤ A[RIGHT] & LOC ≠ RIGHT

   RIGHT := RIGHT − 1

   [END of loop 2(a)]

   (b) If LOC = RIGHT, then: Return

   (c) If A[LOC] > A[RIGHT], then

   swaping { (i) { TEMP := A[LOC],
   A[LOC] := A[RIGHT] & A[RIGHT] := TEMP

   (ii) Set LOC := RIGHT

   (iii) Go TO step 3

3. [Scan from left to right until larger element is found]

   (a) Repeat while A[LEFT] ≤ A[LOC] & LEFT ≠ LOC

   LEFT := LEFT + 1

   [End of loop]

   (b) If LOC = LEFT, then RETURN

   (c) If A[LEFT] > A[LOC], then

   Swap { (i) TEMP := A[LOC], A[LOC] := A[LEFT]
   A[LEFT] := TEMP

   (ii) Set LOC := LEFT

   (iii) Go to step 2

[End of if structure]

QUICK SORT (An application of stack)

Quick sort is an algorithm of divide & conquer type. That is, the problem of sorting a set is reduced to the problem of sorting two smaller sets. This reduction step can be illustrate by following example

A[

	1	2	3	4	5	6	7	8	9	10	11	12]
	(44)	33	11	55	77	90	40	60	99	22	88	66
larger →	22	33	11	55	77				99	(44)	88	66
	22	33	11	(44)	77	90	40	60	99	55	88	66
larger →	22	33	11	40	77	90	(44)	60	99	55	88	66
	22	33	11	40	(44)	90	77	60	99	55	88	66

smaller ← (10) smaller ← (12) smaller ← (11)

First sublist — Second-sublist

→ The element (44) is correctly placed in its final position
→ Task of sorting original list has now been reduced to the task of sorting each of the above sublist
→ The above reduction step is repeated with each sublist containing 2 or more elements.
→ We process only one sub-list at a time. this is accomplished by using two stacks.
(i) LOWER & (ii) UPPER!
for temporarily hold such sublist. i.e.

LOWER : 1 , 6
UPPER : 4 , 12 } in our case (example above)

The algorithm is divided into two parts

(i) first part is procedure called QUICK_Red : which executes the above reduction step of algorithm.

& (ii) Second part uses QUICK to sort entire list, using the stacks i.e. LOWER & UPPER.

Complexity :

$$f(n) = n + (n-1) + (n-2) \ldots + 2 + 1 = \frac{n(n+1)}{2} = \boxed{O(n^2)}$$

RECURSION

Suppose P is a procedure containing either

→ a call statement to itself or a call statement

→ to a second procedure that may eventually result in call statement back to original procedure P

Then P is called recursive procedure.

Recursive procedure must have two properties:

(1) There must be certeria, called base criteria, for which the procedure does not call itself.

(2) Each time the procedure does call itself (directly or indirectly), it must be closer to the base criteria.

A recursive procedure with these properties is said to be well-defined.

i.e.

⟹ (Factorial function)

(a) If $n = 0$, then $n! = 1$ (Base value)

(b) If $n > 0$, then $n! = n \cdot (n-1)!$ (closer to base value)

Eg: $4! = 4 \cdot 3!$

$3! = 3 \cdot 2!$

$2! = 2 \cdot 1!$

$1! = 1 \cdot 0!$

$0! = 1$

$1! = 1 \cdot 1 = 1$

$2! = 2 \cdot 1 = 2$

$3! = 3 \cdot 2 = 6$

$4! = 4 \cdot 6 = 24$

Algorithm : QUICK SORT

1. TOP := NULL

2. [Push boundary values of A onto stack when A has 2 or more elements]

   If N > 1, then TOP := TOP + 1

   LOWER[1] := 1 & UPPER[1] := N

3. Repeat step 4 to 7 while TOP ≠ NULL

4. [Pop subList from stacks.]

   Set BEG := LOWER[TOP], END := UPPER[TOP]

   TOP := TOP - 1

5. Call QUICK_Red (A, N, BEG, END, LOC)

6. [Push left sublist onto stacks when it has 2 or more elements]

   If BEG < LOC - 1, then

   TOP := TOP + 1, LOWER[TOP] := BEG

   UPPER[TOP] = LOC - 1

   [End of if structure]

7. [Push right sublist onto stacks when it has 2 or more elements]

   If LOC + 1 < END, then:

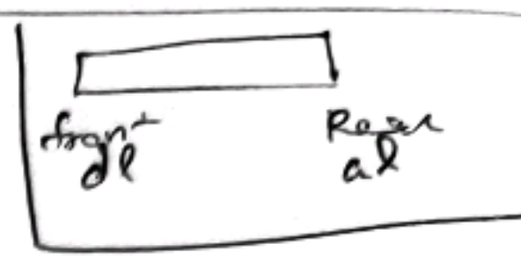
   TOP := TOP + 1,

   LOWER[TOP] := LOC + 1 & UPPER[TOP] := END

   [End of if structure]

   [End of step 3 loop]

8. EXIT




```
#include <iostream.h>
#define MAX 5  // max. values in Queue

class queue
{
  Private:
    int t[MAX]
    int al;  // Addition End (REAR)
    int dl;  // Deletion End (FRONT)
  Public:
     queue ()
{  dl = -1 ; al = -1 ; }
   void del()
{  int tmp;
   if (dl == -1)
{cout << "Queue is Empty ";}
else
{
for (int j=0; j < al ; j++)
  {
    if ( (j+1) <= al)
      {  tmp = t[j+1];
         t[j] = temp;
      }
  else
  {   al--;
  if ( al == -1)
      dl = -1;
  else
      dl = 0;
  }
}
}
```

```
add ( int item)
{
  if (dl == -1 && al == -1)
     {  dl++; al++; }
else
{   al++;
    if (al == MAX)
    {
    cout<<" queue is Full \n";
     al---; return ;
    }
}
  t[al] = item;
}
```

```
void display ()
{      if (dl != -1)
  { for( int iter=0; iter<=al; iter++)
      cout << t[iter] << "  ";
  }
   else
      cout << "EMPTY";
}
  }; // class clos
```

```
int main ()
{ queue a;
 int data[5] = { 1,2,3,4,5 };
a. display ();  // Befor data in Que
 for (i  0 to <5)
{ a. add (data[i]);  a.display ();
  cout << endl;
}
for ( i  0 to <5)
 { a.del (); a.display ();
 }
  return 0;
}
```

// STACK

```
# include <iostream.h>
# define MAX 10
 class stack
{
 private:
 int top, a[MAX]
 public:
 stack()
 {
    top = -1;     // to empty
 }
 void push (int x)
 {
      top++;
      if( top < MAX)
      {
       a [top]= x;
       }
      else
      {
       cout<<"STACK FULL";
       top--;
      }
    }

 int pop ()
{
        if (top==-1)
        {
         cout <<"STACK EMPTY \n";
         return NULL;
        }
        else
        {
         int data = a[top];
         a[top] = NULL;
         top--;
         return data;
        }
    }
    void display()
    for (int i=0; i<MAX; i++)
    cout<< a[i];
};    }
```

```
int main ()
{
  stack  s;

 s. push (3);
 s. push (10);
 s. push (5);

cout<<" Elements in stacksare:";

 s. display();

 s. pop ();
 s. pop();
 s. pop();
}
```