Write a Algorithm for creating max heap using ~~IAS~~ INSERT.

Procedure INSERT(A,n).

Description :- This Procedure rearranges elements such that maximum elements is should be at the root or A(1) where A(1:n) is array & n is the number of elements in array.

Declaration :-

integer A(1:n)

integer i,j,n.

j ← n, i ← [n/2], item ← A(n).

while i>0 and A(i) < item, do.

A(j) ← A(i)

j ← i

i ← i/2.

repeat.

A(j) ← item.

END INSERT.

for i ← 2 to n, do.

call INSERT (A, i)

repeat.

Write a Algorithm for ...

INSERT.

Procedure INSERT (A, n).

Description :- This procedure rearrange ~~Description~~ A(1:n) such that minim element is should be at the or at A(1). where n is the of elements in array.

Declaration :-

integer A(1:n)

integer i, j, n.

```
j <- n , i <- [n/2]; item <- A(n).
while i>0 and A(i) > item, do.
    A(j) <- A(i).
    j <- i
    i <- i/2.
repeat.
    A(j) <- item.
END INSERT.

for i <- 2 to n, do.
    call INSERT (A, i).
repeat.
```

Returned on

write Algorithm for ~~everything~~ creating max heap using ADJUST || HEAPIFY.

Procedure HEAPIFY (A,n).

Description :- This Procedure readjust the elements in A such that to form an heap (max). where n is number of elements and A(1:n) is an array.

Declaration :-

```
            integer n,;
for i <— [n/2] to 1 by -1 do.
      call ADJUST (A,i,n)
  repeat
END HEAPIFY.
```

Procedure ADJUST (A,i,n)

Description :- This procedure readjust the n elements, such that to form max heap, where n is number of elements in A(1:n) array.

Declaration :-

```
             integer i,j,n.
   j <— 2*i
   item <— A(i)
   while j ≤ n ,do.
       if j<n and A(j)<A(j+1) ,then
            j <— j+1.
       endif.
```

repeat
A([j/2)] ← item.
END ADJUST.

Write Algorithm for creating min heap using ADJUST | HEAPIFY.

Procedure HEAPIFY (A, n).

Description :- This Procedure is for any binary tree whose root is at location 'i'. n is the number of elements in array (1:n).

Declaration :- A, n, i.

```
for i <- [n/2] to 1 by -1. do
    call ADJUST (A, i, n)
repeat.
END HEAPIFY.
```

Procedure ADJUST (A, i, n).

Description :- This Procedure readjust the n elements such that to form min heap, where n is number of elements in A(1:n) array.

Declaration :-

```
        integer A, n.
        integer i, j, item.

j <- 2 * i , item <- A(i).
while j <= n . do.
    if j < n and A(j) > A(j+1).
        j <- j+1.
    endif.
```

```
if item < A(j), then
    EXIT LOOP.
else
    A([j/2]) <— A(j).
    j <— j*2.
endif.
repeat.
A([j/2]) <— item.
END ADJUST.
```

Procedure U(i,j).

// Description -

Replace the disjoint sets with roots (i,j), $i \neq j$ by their union.

// Declaration -

integer i,j

// Algorithm.

PARENT (i) ← j.

END_U.

* Simple find Algorithm -

Procedure F(i).

// Description - find the root of the tree containing element i.

// Declaration -

integer i,j.

// Algorithm :-

j ← i.

while PARENT (j) > 0.

j ← PARENT (j).

repeat

return(j).

END-F.

for :

rt

ne Listing

s

REDMI NOTE 5 PRO
MI DUAL CAMERA

2023/7/17 10:06

Modifying ——— Returned on ——— ... operation.

Procedure UNION (i,j)

Write a algorithm ~~for~~ to implement union operation.

Description —

Union sets with roots i & j, $i \neq j$. Using the weighting rule.

PARENT (I) = ~~count~~ -COUNT(I) and

PARENT (J) = -(COUNT(J)).

Declaration —

int i,j.x.

step 1 - x = PARENT (I) + PARENT (J).

if (PARENT (i) > PARENT (j), then

PARENT (i) ← j // i has fewer nodes.

~~else~~ PARENT (j) ← x

else

PARENT (j) ← i

PARENT (i) ← x

endif.

end UNION.

REDMI NOTE 5 PRO
MI DUAL CAMERA
2023/7/17 10:06

Submitted on 3-9-22
Returned on

maximum in given array -

max-Min.
procedure MAXMIN (p, q, max, min)

Description - A(1:n) is a global array. the effect is ~~to~~ to set max and min to the largest & smallest values in A[p:q], ~~respd~~ respectively.

Declaration -

integer p, q.
$i \leq p \leq q \leq n$.

Algorithm -

```
if p=q, then.
    max <- min <- A(p).
else.
    if p=q-1, then
        if A(p) > A(q), then.
            max <- A(p).
            min <- A(q).
        else
            max <- A(q).
            min <- A(p).
        endif.
    else.
        m <- (p+q)/2.
        call MAXMIN (p, m, fmax, fmin).
        call MAXMIN (m+1, q, hmax, hmin).

        if fmax > fmax, then.
            max <- fmax.
        else. max <- hmax.
```

```
if fmin < hmin, then.
        min ← fmin.
else
        min ← hmin.
    endif
  endif
 endif.
END MAXMIN.
```

Procedure ~~BINARY~~ BISEARCH

Description -

Given an array A[1:n] of element in non-descring order, $n \geq 0$, determine whether x is present, and if so, return j such that $x = A[j]$; else return 0.

Declaration - integer low, high, mid, j, n.

Algorithm -

```
low <-- ; high <-- n.
while low <= high , do.
    mid <- [(low+high)/2].
    case.
    : x < A(mid): high <- mid-1.
    : x > A(mid): low <- mid+1.
    : else.
        j <- mid ; return.
    endcase.
repeat.
j <- 0.
```

END_BISEARCH.

Procedure BIN-SEARCH1(A,n,x,j)

Description -

Declaration - integer low, high, mid, j, n.

Algorithm -

```
low ← 1; high ← n+1.
While low < high-1 do.
    mid ← [(low+high)/2].
    if x < A(mid). then.
        high ← mid.
    else.
        low ← mid.
    endif.
repeat.
if x = A(low), then.
    j ← low.
else
    j ← 0.
endif.
BIN-SEARCH1.
```

REDMI NOTE 5 PRO
MI DUAL CAMERA
2023/7/17 10:06

Write a algorithm to sort an array in ascending order using HEAP SORT.

Procedure HEAPSORT (A,n)

Description :-

This Procedure Sorts the n elements of A(1:n). HEAPSORT rearrange them in-place into non-decreasing order. where (1:n) is array contains n number of elements.

// Transforms the elements into a heap call HEAPIFY (A,n).

// interchange the maximum with the elements at the end n and adjust root.

```
for i<-n to 2 by -1 do
    call EXCHANGE (A(i), A(1))
    call ADJUST (A, i, i-1)
repeat.

END HEAP_SORT.
```

Procedure HEAPIFY (A, n).

Description :- This Procedure readjust elements in A(1:n) such form on heap (max).

```
for i ← [n/2] to 1 by -1, do
    call ADJUST (A, i, n)
repeat.
END HEAPIFY.
```

Procedure ADJUST (A, i, n).

Description :- This procedure sort the elements of A(1:n). Heap rearrange them in-place into n decending order. where A(1:n) arra contains n number of elements.

Declaration :-

```
        integer i, j, item.
    j ← 2 * i
    item ← A(i)
    while j ≤ n, do
        if j < n and A(j) < A(j+1), th
            j ← j+1.
        endif.
        if item ≥ A(j), then.
            EXIT LOOP
        else
            A([j/2]) ← A(j).
            j ← j * 2.
        endif.
    repeat.
    A([j/2]) ← item.

END ADJUST.
```

Submitted on
Returned on

Algorithm for sorting given array in descending order using HEAP_SORT

Procedure ADJUST (A, i, n).

Description :- This procedure readjust the elements in A (1:n) such to term an heap (min).

Declaration :-

```
        integer i,j,n.
    j<—2*i ; item <—A(i)
    while j≤n , do.
        if j<n & A[j+1] ,then.
            j<— j+1 .
        endif
        if item >A[i]
        then exit loop.
        else
            A[j/2] <—A(j) .
            j<—2*j .
        endif.
    repeat
        A ([j/2]) <— item.
END ADJUST.
```

Procedure HEAPIFY (A, n).

Description :- This Procedure readjust the elements in A(1:n) such form min heap, where n is no. elements.

Declaration :-

```
        integer i,n
    for i <- [n/2] to 1 by -1 do.
        call ADJUST (A, i, n).
    repeat.
end HEAPIFY.
```

Procedure HEAP_SORT (A, i, n).

Description :-

This procedure sorts the n of A(1:n). Heap sort rearrange the place into decending order, where array contain n elements.

Declaration :-

```
        integer i,n
    for i <- n to 2 by -1 do
        call EXCHANGE (A[i], A[1])
        call ADJUST (A, 1, i-1)
    repeat
end HEAP_SORT.
```

Remarks ____ Expt. No. 10.3 ____ Submitted on ____ Returned on ____

Write an algorithm for Quick_sort.

~~Prode~~ Procedure QUICK-SORT (P,q).

Description - sort the elements A(P).... up to A(q) which decide in the global array n into 'ascending' order A(n+1) is consider to be define & must be greater than or equal to all elements in A(P:q) ; A(n+1) = +∞

Declaration -

```
        integer P,q.
        Global n.(A,n).
  if  P<a, then.
      j<-q+1.
      Call (P,i)
      call QUICK_SORT (P,j-1).
      Call QUICK-SORT (j+1,q).
endif.
END QUICK_SORT.
```

Procedure PARTITION (m,P).

Description.

within A(m), A(m+1).... A(p-1). ~~way~~ the elements are rearranged in such a way that if initially temp = A(m) then after completion A(q) = temp for some q between m & (p-1) A(k) ≤ temp for $m \leq k \leq q-1$, A(k) > temp for $(q > k \geq p-1)$. the final value of q is q.

2023/7/17 10:05

REDMI NOTE 5 PRO
MI DUAL CAMERA

Declaration –

```
Global A(m:P), integer m,P;
temp <— A(m);
i = m.
loop
  do
    i <— i+1.
  while A(i) < temp repeat
  do
    P <— P-1.
  while A(P) > temp repeat
  if i<P, then.
        EXCHANGE A(i), A(P).
  else.
        Exit loop.
  endif.
repeat.
A(m) <— A(P).
A(P) <— temp.
END PARTITION.
```

Write Algorithm to find solution of KNAPSACK instant.

Procedure GREEDY_KNAPSACK (P, ω, x, F, n).

Description: P(1:n) & ω(1:n) contains the profit & weights respectively of an objects. ordered so that P(i)/ω(i) ≥ P(i+1)/ω(i+1), M is the KNPSACK size x : & x(1:n) is the soln vector.
// Assuming that data given is sorted according to P/ω as describe above.

Declaration: P(1:n), ω(1:n), x(1:n), ~~M~~, M, cu.
integer i, n.

Algorithm
```
x ← 0        // initialize soln vector with zero.
cu ← M       // remaining maps of capacity.
for i ← 1 to n do.
    if ω[i] > cu, then.
        exit loop.
    else
        x(i) ← 1
        cu ← cu - ω(i)
    endif.
repeat.
  if i ≤ n, then.
  | x(i) ← cu/ω(i).
  endif.
END GREEDY_KNAPSACK.
```

Write a algorithm to find shortest path using single source shortest path.

Procedure SHORTEST_PATH(V,COST,DIST,n).

<u>Description</u> – DIST (j) . j . is $1 \le j \le n$ ....., is set to the length of the shortest path from vertex v to vertex j in a diagraph G with n vertices DIST(V) is set to zero. G is represented by its COST adjancecy matrix COST (n,n).

<u>Declaration</u> – boolen s(1:n)
real COST (1:n, 1:n)
DIST (1:n)
integer n, u, w, v, num, j.

```
//initialize set s to empty & DIST
using current edges.
    for i <— 1 to n, do.
        s (i) <— 0 ; DIST(i) <= COST (V,i)
    repeat.
    S(V) <— 1; DIST (V) <— 0 // add v in s
    for num <— 2 to n-1 ,do
      choose 'V' from among those
      vertices not in s such that
      DIST (u) = min{ DIST(w) and S(w)=0
      s(u) <—1 // put 'u' is set s.
      for all w with S(w)=0 ,do
            DIST (w) <— min (DIST(w),
                  DIST(u) + COST (u+w
      . repeat
   repeat
END SHORTEST PATH.
```

Class : 128 Expt. No. ______ Performed on 1-10-22.
Roll No. ______ Submitted on ______
Remarks ______ Returned on ______

Write Algorithm to find minimum cost spanning Trees (Prim's) Algorithm.

PRIM's Algorithm -

procedure : PRIM's (E, COST, n, T, min-cost).

Description -

E is set of edges in G. COST(n,n) is graph adjency matrix of the n,n veratices a graph such as that cost(i,j) is either a positive real number or +∞. if no edge(i,j) exist. A minimum spanning tree is computed & stored as a set of edges in the array T(1:n,2) T(i,1), T(i,2) is on edge in the minimum spanning tree. The final cost is assign to min cost.

Declaration -

real cost(n,n), min-cost;
integer NEAR(n), n, i, j, k, l.
T(1:n-1,2)

step-I :- (k,l) ← edge with min-cost
min-cost ← cost (k,l)
(T(1,1), T(1,2) ← (k,l).
step-II :- Fill up near array.
for i ← 1 to n do.
if COST(i,l) < COST(i,x)
NEAR(i) ← l.
else
NEAR(i) ← k.

```
repeat.
        NEAR(K) <-- 0
        NEAR (l) <-- 0.

// findout remaining (n-2) edges o ->
for i <-- 2 to n-1 do,
        let j be an index that NEAR
        ≠ 0 and COST (j, NEAR(j)) is

for i <-- 2 to n-1 do,
        let j be an index that NEA
//      3 -> j        1 2 3 4 5 6
        NEAR(j)     | 0 | 0 | 2 | 1 | 2 | 1 |

    T(i,1), T(i,2) <-- (j, NEAR(j)).
    min cost <-- min-cost + cost (j, NEAR(
        NEAR (j) <- 0.

    // update Near Array
        for k <-- 1 to n do
            if NEAR (k) ≠ and COST (k,
                                    COST (k,
                NEAR(K) <- j.
            endif
        repeat.
repeat
        if min-cost >= ∞ . then.
            print (" No spanning tree")
        endif.
end PRIMS.
```

Remarks ________ Returned on ________

Write a algorithm to find minimum cost-spanning Trees (Prim's & kruskal's) algorithm.

a) **KRUSKAL Algorithm** -

Procedure KRUSKAL(E, COST, n, T, mincost)

Description -

E is the set of edges in G. G has n vertices. COST (u,v) is the cost of edges is the set of edges in minimum spanning tree.

Declaration -

Real mincost, COST (1:n, 1:n).
integer PARENT(1:n), T (1:n-1,2), n.

Algorithm -

step 1 - Construct a heap out of edes. cost using heapify.

step 2 - PARENT = -1.

i ← mincost ← 0.
while i<n-1 and HEAP is not empty do.
→ delete a minimum cost edge (u,v). from a Heap & reheapify using Adjust.
→ j ← FIND (u); k ← FIND (V).
→ if j ≠ k, then.
i ← i+1.
T((i,1),(i,2)) ← (u,v).
mincost ← mincost + COST (u,v).
call UNION (j,k).
endif.
repeat.

if $i \neq n-1$, then
    print ("no minimum spanning tree").
endif.
END_KRUSKAL.

Procedure ALL_PATHS (cost, A, n).

Description -

COST(n,n) is the cost-Adjency matrix of a graph with n vertices. A(1,i) is the cost of a shortest path from V_i to V_j.
COST (i,i) = 0. 1 ≤ i < n.

Declaration -

```
integer i, j, k, n.
real COST(n,n), A(n,n).
```

Algorithm -

```
for i <- 1 to n do.
    for j <- 1 to n do,
        A(i,j) <- COST(i,j).
    repeat.
repeat.
for k <- 1 to n, do.
    for i <- 1 to n, do.
        ~~A(i,j) <- COST(i,j).~~
        for j <- 1 to n, do.
            A(i,j) <- min{A(i,j)
                A(i,k) + A(k,j)
        repeat
    repeat
repeat.
END. ALL-PATHS.
```

Returned on

*longest Common subsequence.

The algorithm L.C.S is divided into two Parts one Part compute the length (L.C.S) & another Part constructs a L.C.S.

procedure L.C.S-LENGTH(x,y).

Description :-

$x = (x_1, x_2, x_3 \ldots x_n)$ & $y \in (y_1, y_2, y_3 \ldots y_m)$ are the two given subsequences the also uses two $m \times n$ matrix c(0:m.0:n). & B(0:m. 0:n) the matrix c stores the length of LCS & matrix B stores B the length of LCS & matrix B stores the symbol of S which symbol S. which are used to construct LCS the entries are computed in row order this Procedure return matrix B & C.

Declaration -

Global integer c(0:m,0:n)
char B(0:m)(0:n) char x(1:m)
y(1:n).
local integer m,n,i,j

Algorithm -

```
m <— LENGTH (x).
n <— LENGTH (y).
for i <— 0 to m do.
  | c(i, 0) <— 0.
repeat
for j <— 0 to n do.
  | c (0, j) <— 0.
repeat.
For i <— 1 to m. do.
   for j <— 1 to n, do.
      if x(i) = Y(j), do.
        | c(i,j) <— c(i-1, j-1) + 1
        | B(i,j) <— '↖'.
      else
         if (i-1, j) ≥ c(i, j-1) then
           c(i,j) <— c(i-1, j).
           B(i,j) <— '↑'.
      else
           c(i,j) <— c(i, j-1).
           B(i,j) <— '←'.
         endif.
      endif.
   repeat.
repeat
return x & y.
end LCS_LENGTH.
```

Declaration-

```
        Global Char B(0:m, 0,n)
        global x (x1, x2 ... xm)
        integer  m,n
        Local  integer i,j
        STACK  S (i:k)

    for i ← m  to 1 by -1, do
        for j ← n to 1 by -1 do
            if B (i,j) '↑' then
                PUSH (x(i))
                i ← i - 1
                ~~els~~
            else
                if B(i,j) - '↑' then
                   i ← i-1 & j ← j+1
                endif
            endif
        repeat
    repeat
    for j ← top to 1 do
            PRINT (STACK (i))
    repeat

End LCS_PRINT
```

Remarks ______ Submitted on ______ Returned on ______

Breadth First search:-

Procedure BFS(V)

Description :- A breadth first search of G is carried out begining at vertex V. All vertices visited are marked as ∀ VISITED (i) = 1. The graph G & array ~~visit~~ VISITED are global and VISITED is intialized to 0.

Declaration :-

```
VISITED (V) <— 1;
u <— V.
Initialize Q to be empty.
loop.
    for all vertices w adjecent from u do
        if VISITED (w)=0 , then.
            call ADDQ(w,Q).
            VISITED (w) <—1.
        endif.
    repeat
    if Q is empty then.
        return.
    endif.
    call DELETEQ(u,Q).
repeat.
```

Depth First search (Recursive) :-

Procedure DFS (V).

<u>Description</u> :- Given an undirected or graph G = (V, E) with n & an array VISITED (n) initially set to 0.

<u>Declaration</u> :-

```
VISITED(V) <— 1.
for each vertex w adjecent from
    if VISITED (w) = 0, then.
        call DFS (w)
    endif.
repeat
end DFS.
```

Depth First search (Non-Recursive) :-

Procedure NR-DFS(V).

```
VISITED (V) <— 1.
u <— V.
Initialize STACK to be empty.
loop
    for all adjecent from u do.
        PUSH (w).
        VISITED (w) <— 1.
    repeat
    if STACK u empty, then.
        return.
    else
        u <— POP ().
    endif.
repeat
end NR-DFS.
```

proc... (low, high).

Description - A [low: high] is a global array to be sorted in this case the list is already sorted.

Declaration - integer A, low, high.

```
if low < high then.
    mid <- (low+high)/2
    Call MERGE_SORT (A, low, mid).
    Call MERGE_SORT(A, mid+1, high).
    call MERGE (A, low, mid, high).
endif.
end MERGE_SORT.
```

Procedure MERGE (A, low, mid, high.).

Description :- This process merge two sublist A(low: mid) & ~~B(~~ m(mid+1: high) it uses auxillary B (low: high) & sorted.

Declaration :-

Global A(low: mid) & A(mid+1, hig

integer i, j, k.

```
i <- low.
j <- mid+1.
k <- low.
```

```
while i≤mid and j≤high, do
    if A(i) ≤ A(j), then.
        B(K) <— A(i).
        i <— i+1.
    else
        B(K) <— A(i).
        j <— j+1.
    endif.
repeat K <— K+1.
if i≤mid
    while i≤mid, do.
        B(K) <— A(i).
        i <— i+1.
        K <— K+1.
    repeat.
else.
    while j≤mid, do.
        B(K) <— A(i).
        j <— j+1.
        K <— K+1.
    repeat
endif
for K <— low to high, do.
    A(K) <— B(K).
repeat.
end MERGE.
```

*Strassen's matrix multiplication

//As can be seen P, Q, R, S, T, U, V
computed using 7-matrix mu
and 10 matrix additions or
The Cij's require an additi
additions or substractions.

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$Q = (A_{21} + A_{22}) B_{11}$$
$$R = A_{11}(B_{12} - B_{22})$$
$$S = A_{22}(B_{21} - B_{11})$$
$$T = (A_{11} - A_{12}) B_{22}.$$
$$U = (A_{21} - A_{11})(B_{11} + B_{12}).$$
$$V = (A_{12} - A_{22})(B_{21} + B_{22}).$$

$$C_{11} = P + S - T + V.$$
$$C_{12} = R + T.$$
$$C_{21} = Q + S.$$
$$C_{22} = P + R - Q + U.$$

Write an algorithm to find all solutions for 8-queen problem using backtracking.

procedure NQUEENS (n).

Description :- using backtracking, this procedure prints all possible placements of n queens on an nxn chessboard, so that they are nonattac

Declaration :-

```
        integer k, n, x(1:n).
k <-- 1 ; x(k) <-- 0   // start with first row
                          0th column.
while k > 0 do  // try all possible solution
    x(k) <-- x(k) + 1.
    while x(k) ≤ n and NOT PLACE (k) do
        x(k) <-- x(k) + 1   // try next column
                              & posn.
    repeat.
    if x(k) ≤ n, then.
        if k = n, then.
            print (x)
        else
            k <-- k + 1.
            x(k) <-- 0.
        endif
    else
        k <-- k - 1.
    endif.
repeat.
```

NQUEENS

procedure PLACE (k).

Description :- return true if a queen can b
in k^{th} row and $x(k)^{th}$ colu
otherwise it returns false
x is a global array whose fi
k values having set ABS(x)
returns absoulute value of

declaration :-

```
        global x(1:n).
        integer i,k).
for i <- 1 to k-1 do.
  if (x(i) = x(k) OR.
        ABS (i-k) = ABS(x(i) - x(k)), th
    return (false).
    endif
repeat
return(true).
end PLACE.
```