Algorithm mergeSort(S,C)

Input:sequence S with n elements, Comparator C.

Ouput: Sequence S sorted according to C.

    if(S.size() > 1)

        S1 <- partition(S,0,n/2-1)

        S2 <- partition(S,n/2,n)

        mergeSort(S1,C)

        mergeSort(S2,C)

        S->merge(S1,S2)

Algorithm partition(S,i,j)

Input:Sequence S with the position of start and end for making partition

Output: Sequence S1 with the elements from i to j

    S1<-empty sequence

    while i <= j do

        S1.insertLast(S.elementAtRank(i))

        i <- i + 1

    return S1

Algorithm merge(A, B, C)

    Input:sequences A and B with n/2 elements each,comparator C

    Output:sorted sequence of A U B

    S <- empty sequence

    while not A.isEmpty() ^ not B.isEmpty() do

        if C.isLessThan( B.first().element(),A.first().element() ) then

            S.insertLast(B.remove(B.first()))

        else

            S.insertLast(A.remove(A.first()))

        while not A.isEmpty() do

            S.insertLast(A.remove(A.first()))

        while not B.isEmpty() do

            S.insertLast(B.remove(B.first()))

    return S

=========================

Algorithm removeDuplicateAndUnion(A, B)

    Input:sequences A and B with n elements each

    Output:sorted sequence of A U B

    S <- empty sequence

    while !A.isEmpty() ^ !B.isEmpty() do

        if B.first().element() < A.first().element() then

            S.insertLast(B.remove(B.first()))

        else B.first().element() > A.first().element() then

            S.insertLast(A.remove(A.first()))

        else

            S.insertLast(A.remove(A.first()))

            B.remove(B.first())

    return S

=================

Algorithm electionWinner(S)

    Input: n-element sequence S where each element represent a different vote

    Ouput: ID of winning candidate

    mergeSort(S,C)

    winnerId <- S.first()

    maxVote <- 0

    previousId <- S.first()

    while !S.isEmpty() do

        currentId <- S.remove(S.first())

        if currentId != previousId then

            if maxVote < noOfVote

                maxVote <- noOfVote

                winnerId <- currentId

        else

            previousId <- currentId

            noOfVote<- noOfVote + 1

==================================================

Algorithm getSmallerEqualKey(T,x)

    Input: Vector based heap T, stroing n keys and query key x

    Ouput: Sequence S, containing all the keys which are less than or equal to x

    S <-empty sequence

    i<-1

    while i < T.size()

        key<-elementAtRank(i)

        if key <= x

            S.insertLast(key)

            i <- i + 1

        else

            return S

===================

Algorithm getSmallerEqualKey(T,key,i)

    Input: Vecotr T representing a Heap,index i of an element in the heap and a query key

    Output: Vecotr V contining all the keys which are less than or equal to given key

    if i < T.size() ^ T.elementAtRank(i) <= key

        V.insertLast(T.elementAtRank(i))

        getSmallerEqualKey(T,key,2 \* i)

        getSmallerEqualKey(T,key,2 \* i + 1)

        return

===================

Algorithm parent(p)

    return p / 2

=================

Algorithm leftChild(p)

    return 2 \* p

==========

Algorithm rightChild(p)

    return 2 \* p + 1

==========

Algorithm isInternal(p)

    Input : a position p of a node in the vector S

    Output:position of the leftChild of the tree T in S

    if 2 \* p <= S.size() or (2 \* p + 1) <= S.size()

        return true

    return false

=========================

Algorithm isExternal(p)

    Input : a position p of a node in the vector S

    Output:position of the leftChild of the tree T in S

    if 2 \* p > S.size() ^ (2 \* p + 1) > S.size()

        return true

    return false

===============

Algorithm putSequenceInRandomOrder(S)

    Input:Sequence S with n elements

    Output: S in random order

    r <- n - 1

    while r > 0 do

        rand <- randomInt(r)

        S.swapElements(S.atRank(r),S.atRank(rand))

        r <- r - 1

    return S

===================================

Algorithm insertBefore(p,e)

    Input : position of node p where  newnode will be inserted before this node and element of newNode

    Output: newNode with element e will be inserted into list.

    newNode<-createNewNode(e)

    tmp<-p.prev

    tmp.next <- newNode

    newNode.next <- p

    newNode.prev <- tmp

    p.prev <- newNode

=============

Algorithm findMiddle(L)

    Input : List L with odd number of nodes

    Output : middle position of L

    p<-L.first()

    q<-L.last()

    while p != q do

        p<-L.after(p)

        q<-L.before(q)

    return p

    ===========================

Algorithm insertFirst(e)

    Input : element e,which will be inserted into first of the linked list

    Output: newNode with element e will be inserted into first position of list.

    newNode<-createNewNode(e)

    tmp<-head.next

    head.next<-newNode

    newNode.next<-tmp

    tmp.prev <- newNode

    newNode.prev <- head

    ====================

Algorithm insertLast(e)

    Input : element e,which will be inserted into last position of the linked list

    Output: newNode with element e will be inserted into last position of list.

    newNode<-createNewNode(e)

    tmp<-tail.prev

    tail.prev <-newNode

    newNode.prev<-tmp

    tmp.next <- newNode

    newNode.next <- tail

============================

Algorithm insertAtRank0(obj)

    Input: the object obj for inserting

    if V.size() = n-1

        throw fullException

    f<-(f - 1 + n ) mod n

    V[f] <- obj

============================

Algorithm insertAtRankEnd(obj)

    Input: the object obj for inserting

    if V.size() = n-1

        throw fullException

    V[r] <- obj

    r <- (r + 1) mod n

===================

Algorithm removeAtRank0()

    if !V.isEmpty()

        f<-(f + 1 ) mod n

    else

        throw emptyVectorException

====================

Algorithm removeAtRankEnd()

    if !V.isEmpty() then

        r<-(r - 1 + n) mod n

    else

        throw emptyVectorException

========================

S1<-Empty Stack

S2<-Empty Stack

enqueue(val)

    if size() = N - 1 then

        throw FullQueueException

    S1.push(val)

dequeue()

    if S2.isEmpty() then

        while !S1.isEmpty() do

            S2.push(S1.pop())

    if !S2.isEmpty() then

        return S2.pop()

    else

        throw EmptyStackException

=================================

Q1<-Empty Queue

Q2<-Empty Queue

push(val)

    if size() = N - 1 then

        throw FullStackException

    Q1.enqueue(val)

pop()

    if Q2.isEmpty() then

        while !Q1.isEmpty()

            Q2.enqueue(Q1.dequeue())

    if !Q2.isEmpty() then

        Q2.dequeue()

    else

        throw EmptyQueueException

===========================

Algorithm perm(S, int n)

    Input: Sequence S with n elements

    Output: List L containing all the permutation

    if n = 1 then

        L.insertLast(S)

        return;

    while i < S.size() do

        S.swapElements(S.rankOf(i),S.rankOf(n-1))

        perm(S,n-1)

        S.swapElements(S.rankOf(i),S.rankOf(n-1))

Let the (S, min, max) be an instance of Subset Sum. The transformation

would use the following algorithm:

Algorithm reduceSSto0-1K(S, min, max)

    Input: a Sequence S of numbers and the limits min and max from Subset Sum

    Output: a Sequence P of pairs (representing benefit and weight) and the

    values of w and b for 0-1 Knapsack

P ? new empty Sequence

for i ? 0 to S.size()-1 do

 val ? S.elemAtRank(i)

 P.insertLast( (val, val) )

return (P, max, min) {pairs, maximum weight, minimum benefit}

Algorithm checkSS(S,min,max,T)

    sum<- 0

    for each e of T do

        sum <- sum + e

    if sum >= min ^ sum <= max then

        return yes

    else

        return no

=============

Algorithm MST2SS(G,k)

    Input:

    Output:

    T<-MST(G)

    sum <- 0

    for each e of T.edges() do

        sum <- sum + weight(e)

    S<-new empty sequence

    S.insertLast(2)

    if sum <= k then

        return (S,2)

    return (S,1)

==============

HP2LP(G,u,v)

    Input:

    Output:

    for each e of G.edges() do

        setWeight(e,1)

    return (G,u,v,G.noOfVertices()-1)

=========

Algorithm SP2MST(G,k)

    G' <- new Graph

    u<-G'.insertVertex("u")

    v<-G'.insertVertex("v")

    e<-G'.insertEdge(u,v,2)

    p<-shortestPath(G')

    sum<- 0

    for each e of P do

        sum <- sum + weight(e)

    if sum <= k then

        return (G',2)

    else

        return (G',1)

13(2)

a.

Algorsithm BFS(G)

    Input graph G

    Output labeling of the edges and partition of the vertices of G

    initResult( G )

    for all u of G.vertices()

        setLabel(u, UNEXPLORED)

    for all e of G.edges()

        setLabel(e, UNEXPLORED)

    for all v of G.vertices()

        if getLabel(v) = UNEXPLORED

            preComponentVisit(G, v)

            BFS(G, v)

            postComponentVisit(G, v)

    result(G)

Algorithm BFS(G, s)

    L <- new empty sequence

    L.insertLast(s)

    setLabel(s, VISITED)

    startVertexVisit(v)

    while !L.isEmpty() do

        v <- L.remove (L.first())

        for all e of G.incidentEdges(v) do

            if getLabel(e) = UNEXPLORED then

                w <- opposite(v,e)

                if getLabel(w) = UNEXPLORED then

                    preDiscoveryTraversal(G, v, e, w)

                    setLabel(e, DISCOVERY)

                    setLabel(w, VISITED)

                    L.insertLast(w)

                    postDiscoveryTraversal(G, v, e, w)

                else

                    setLabel(e, CROSS)

                    backTraversal(G, v, e, w)

    finishVertexVisit(G, v)

13.2(b)

Algorithm findPath(G, u, v)

    S <-new empty stack {S is a subclass field}

    z<-v  {z is a subclass field & is the target vertex}

    BFS(G,u)

    return(path)

Algorithm startVertexVisit(v)

    S.push(v)

Algorithm preDiscoveryTraversal(G,v,e,w)

    if !pathFound then

        S.push(e)

        S.push(w)

Algorithm postDiscoveryTraversal(G,v,e,w)

    if w = z then{z is a subclass varibale and target}

        pathFound = true

Algorithm finishVertexVisit(G, v)

    if !pathFound then

        return no\_such\_path

    return S.elements()

13.2(c)

Algorithm startVertexVisit(v)

    if !cycleFound then

        S.push(v)

Algorithm preDiscoveryTraversal(G,v,e,w)

    if !cycleFound then

        S.push(e)

        S.push(w)

Algorithm backTraversal(G, v, e, w)

    if !cycleFound then

        cycle<-new empty sequence

        cycle.insertLast(w)

        while o != w do

            o<-S.pop()

            cycle.insertLast(o)

        cycleFound <- true

Algorithm finishVertexVisit(G, v)

    while !S.empty()

        S.pop()

    if !cycleFound then

        return no\_cycle\_found

    else

        return cycle

13.2(d)

No.

Reason:DFS goes into the deep of a node and there is no guarantee that that path will be the minimum number of edges and it is hard to find minimum number of edges during backtracking.

=======================================

13(1)

Algorithm initResult(G)

    S<- new Empty Sequence

Algorithm preComponentVisit(G, v)

    S.insertLast(v)

Algorithm result(G)

    return S;

13(4)

Algorithm initResult(G)

    levelNo <- 1

Algorithm startVertexVisit(v)

    setLevel(v,levelNo){levelNo is a subclass field}

Algorithm postComponentVisit(G,v)

    levelNo <- levelNo + 1

    ==========

13(3)

Algorithm DijkstraDistances(G, s)

    Q <-new heap-based priority queue

    initGraph(G)

    for all v of G.vertices() do

        if v = s

            setDistance(v, 0)

        else

            setDistance(v, INF)

        l<-Q.insert(getDistance(v), v)

        setLocator(v,l)

    startVertexVisit(G,s)

    while !Q.isEmpty() do

        u <- Q.removeMin()

        for all e of G.incidentEdges(u)

            { relax edge e }

            z <- G.opposite(u,e)

            r <- getDistance(u) + weight(e)

            if r < getDistance(z) then

                preUpdate()

                setDistance(z,r)

                Q.replaceKey(getLocator(z),r)

                postUpdate(G,u,e,z)

    finishVertexVisit(G,s)

=========

Algorithm findPath(G, u, v)

    S <-new empty stack {S is a subclass field}

    z<-v  {z is a subclass field & is the target vertex}

    DijkstraDistances(G,u)

    return(path)

Algorithm startVertexVisit(G,s)

    setParent(s,NULL)

Algorithm postUpdate(G,u,e,z)

    setParent(z,e)

Algorithm finishVertexVisit(G,s)

    path<-empty sequence

    u <- z{z is a subclass varibale and target}

    path.insertLast(u)

    while u != s do

        u<-getParent(u)

        path.insertLast(u)

==============

Algorithm anatjariWaterRefill(S, m)

    Input : Sequence S of water refilling place, m max distance for one bottle

    Output : Sequence S2 as path for anatjari to cross the desert with less stops for refill possible

    Q <- new heap-based priority queue

    for each x of S do

        Q.insert(x,x)

    t<-0

    while !Q.isEmpty() do

        t <- t + Q.removeMin()

        if t + Q.peekMin() < m then

            S2.insertElement(t)

        else

            S2.insertElement(temp)

            temp <-0

    return S2

============

Algorithm anatjariWaterRefill(G, k, s)

    Input : Graph G of map for water refilling place, k max distance for one bottle, s as anitjari's starting vertices

    Output : Sequence S2 as path for anatjari to cross the desert with less stops for refill possible

    Q <- new heap-based priority queue

    for each v on G.vertices() do

        if v = s then

            Q.insert(s,v)

        else

            Q.insert(v, v)

    while !Q.isEmpty() then

        t <- Q.removeMin()

        currWeight <- 0

        for each e on G.incidentEdges(vTemp) do

            if weight(e) < k and weight(e) > currWeight then

                currWeight <- weight(e)

            QQ.removeItem(vTemp)

            S2.insertElement(vTemp)

    return S2

Algorithm removeElement(x)

    Input: key x for remove item

    Output: item deletion with the key

    p<-first position top list

    p<-find(x,p)//return position of the key

    if p != NULL

        while p != NO\_SUCH\_KEY do

            if p.before = MINUS\_INF ^ p.after = PLUS\_INF

                p.before.below.above <- NULL

                p.after.below.above <- NULL

            a<-p.below

            tmp <- p.before

            tmp.after = p.after

            p.after.before = tmp

            p<-a

    ================

Algorithm find(x,p)

    Input:key x,postion p

    Output: postion of the key x

    y<-key(p.after)

    if x = y

        return p.after

    else if x > y then

        return find(x,p.after)

    else

        return find(x,p.below)

    return NULL

==================

Algorithm isExistTwoEqualElement(S)

    Input: Sequence S with n elements

    Output: true or false if two equal elements exis in the sequence,otherwise false

    D<-new Dictionary(HashTable)

    for each x of S do

        cnt <-D.findElement(x)

        if cnt != NO\_SUCH\_KEY then

            D.insertItem(x,0)

        else

            return true

    return false

====================

Algorithm inPlacePartition(S, lo, hi)

    Input: Sequence S and ranks lo and hi, 0 <= lo,hi < S.size()

    Output: Skip duplicate keys in the next partition

    p <-- a random integer between lo and hi

    S.swapElements(S.atRank( lo ), S.atRank( p ))

    pivot <-- S.elemAtRank(lo)

    j <-- lo + 1

    k <-- hi

    while  j < k  do

        while k >= j ^ S.elemAtRank(k) > pivot do

            k <-- k � 1

        while j <= k ^ S.elemAtRank(j) < pivot do

            j <--  j + 1

        if  j < k  then

            S.swapElements(S.atRank( j ), S.atRank( k ))

    S.swapElements(S.atRank( lo ), S.atRank( k )) {move pivot to sorted rank}

    return k

=========================

Algorithm countInversion(S, C)

    Input : sequence S with total order n elements, comparator C

    Output: number of Inversion

    if S.size() > 1 then

        (S1, S2)<-partition(S, n/2)

        countInversion(S1, C)

        countInversion(S2, C)

        (S,cnt) <-merge(S1, S2, C)

        count <- count + cnt

    return count

Algorithm merge(A, B, C)

    Input: sequences A and B with n/2 elements each, comparator C

    Output: count of number of inversion

    count<-0

    S <- empty sequence

    while !A.isEmpty() ^ !B.isEmpty() do

        if C.isLessThan( B.first().element(), A.first().element() ) then

            S.insertLast(B.remove(B.first()))

            count <- count + 1

        else

            S.insertLast(A.remove(A.first()))

        while !A.isEmpty() do

            S.insertLast(A.remove(A.first()))

        while !B.isEmpty() do

            S.insertLast(B.remove(B.first()))

    return count,S

=====================

Pseudo-code FindAllInRange(k1, k2)

    if k1 <= k2 then

        throw exception k2 is bigger than k1

    node1 <-- findElement(k1)

    node2 <-- findElement(k2)

    if node1 == null or node2 == null then

        throw exception key not found

    List <-- new List

    insertToList(D.rightChild(node1), node2, D, List)

    iterator <-- List.iterator()

    return iterator

Pseudo-code insertToList(node1, node2, Tree, List)

    if node1 < node2 && node1 != null then

        insertToList(Tree.leftChild(node1), node2, Tree, List)

        insertToList(Tree.rightChild(node1), node2, Tree, List)

        List.insertLast(Tree.items(node1))

==============

Algorithm findAllInRange(k1,k2)

    Input: key k1, k2

    Ouput: return iterator for all the elements in D within the range of k1 and k2

    T<- tree of D

    S<-findElements(T,T.root(),k1,k2)

    return S.iterator()

===================

Algorithm findElements(T,p,k1,k2)

    Input: Tree T, position of a node p, key k1, k2

    Output: Sequence S with all the elements between the range of k1 and k2 inclusive.

    S<-new Sequence

    k <- T.key(p)

    if k1 <= k ^ k <= k2 then

        S.insertLast(D.findElement(k))

        findElements(T,T.leftChild(p),k1,k2)

        findElements(T,T.rightChild(p),k1,k2)

    else if k < k1 then

        return findElements(T,T.leftChild(p),k1,k2)

    return S

====================

Algorithm electionWinner(S, C)

    Input: Sequence S of votes and Sequence SC of candidates

    Ouput:Id of winner candidate

    D<-new Dictionary(Hastable)

    for each id of C do

        D.insertItem(id,0)

    for each y of S do

        cnt<-D.findElement(x)

        D.insertItem(y,cnt + 1)

    winnerId <- -1

    for each (id,cnt) of D.items() do

        if cnt > winnerId then

            winnerId<-id

            winnerCount<-cnt

    return winnerId

===================

Algorithm isSumEqual(A,B,x)

    Input: Two Sequence A, B and integer x

    Ouput: true or false if such two intergers of A and B makes x

    T <- insert A into Redblack tree

    for each y of B do

        if T.findElement(x-y) != NULL then

            return true

    return false

===============

Algorithm isSameSetElements(A,B)

    Input: Sequence A, B

    Ouput: true if they are elements of same set,othewise false

    if A.size() = B.size() then

        D1<- Dictionary(hastable)

        D2<- Dictionary(hastable)

        for each x of A do

            cnt<-D1.findElement(x)

            if cnt != NO\_SUCH\_KEY then

                D1.insertItem(x, cnt + 1)

            else

                D1.insertItem(x, 0)

        for each x of B do

            cnt<-D2.findElement(x)

            if cnt != NO\_SUCH\_KEY then

                D2.insertItem(x, cnt + 1)

            else

                D2.insertItem(x, 0)

        for each x of A do

            if D1.findElement(x) != D2.findElement(x) then

            return false

    return true

=================

Algorithm removalLinearProbing(key)

    Input: key to remove from

    Output: remove and return the element

    key, element) <- findElement(key)

    If key != NO\_SUCH\_KEY then

        key <- AVAILABLE

        return element

    return NO\_KEY\_FOUND

===============

Algorithm nutsBoltsMatchup(A, B)

    Input : Sequence A of nuts, sequence B of bolts

    Output : Matched set of nuts and bolts

    T <- insertIntoRedBlackTree(B)

    PQ <- new Priority Queue Array

    for each x of A do

        PQ.insert(x, PQ.remove(x))

    return PQ