|  |
| --- |
| Assignment |
| Lesson 10 to 12 |
| Rizk Hanna |

**Lesson 10**

**R-3.19**

Algorithm search(k)

Input key k

Output node containing k or NO\_SUCH\_KEY

p ← top

n ← after(p)

while below(p) ¬= null do

while key(n) < k do

p ← n

n ← after(p)

p ← below(p)

if key(k) = k then

return n

else

return No\_Such\_Key

Algorithm removeElement(k, L)

Input key k, and List L

Output list with the removed k or NO\_SUCH\_KEY

node ← search(k)

if node ¬= NO\_SUCH\_KEY

delete(node)

else

return NO\_SUCH\_KEY

**C-4.16**

Algorithm findDuplicate(S)

Input sequence S

output true if the sequence has a duplicated numbers and false if not

D ← hashtable based a dictionary

for i ← 1 to S.size() do

element ← last()

if D.findElement(element)

return true

else

S.remove(element)

The running time is O(n)

**C-4.18**

Algorithm inPlaceQuickSort(S, lo, hi)

Input Sequence S of distinct elements; integer lo and hi

output the pivot is now stored at its sorted rank

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

index ← lo

while j ≤ k do

while j ≤ k ^ S.elemAtRank(j) ≤ pivot do

j ← j + 1

**if elemAtRank(j) = pivot then**

**S.swapElements(S.atRank(j), ++index)**

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

k ← k - 1

**if elemAtRank(k) = pivot then**

**S.swapElements(S.atRank(k), ++index)**

if j < k then

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

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

**for i ← lo + 1 to index - low // index - low (how many duplicated found)**

**S.swapElement(S.atRank(--k), S.atRank(++low))**

return k

**C-4.19**

Algorithm mergeSort(S, C)

Input sequence S with n elements, comparator C

Output sequence S sorted according to C

if S.size() > 1 then

(S1, S2) ← partition(S, n/2)

mergeSort(S1, C)

mergeSort(S2, C)

inversion ← inversion + countInversion(S1, S2, C)

Algorithm countInversion(A, B, C)

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

Output sorted sequence of A ? B

S ← empty sequence

while ¬A.isEmpty() ^ ¬B.isEmpty() do

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

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

inversion ← inversion + 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 inversion

**C-4.25**

Algorithm nutsAndBolts(S1, S2)

Input sequence S1 with n elements of nuts, S2 with n elements of bolts

Output sequence S sorted according to C

D ← new hashtable based dictionary

S1 ← mergeSort(S1) O(n logn)

S2 ← mergeSort(S2) O(n logn)

for i ← 1 to S1.size() do O(n)

nut ← S1.remove(S1.last()) O(n)

bolt ← S2.remove(S2.last()) O(n)

D.insertItem(nut, bolt) O(n)

return D

The running time for this algorithm is O(n logn)

**Lesson 11(a)**

**R-5.1**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **a** | **b** | **c** | **d** | **e** | **f** | **g** |
| **Weight** | 4 | 6 | 5 | 7 | 3 | 1 | 6 |
| **Benefit** | 12 | 10 | 8 | 11 | 14 | 7 | 9 |
| **Value** | 3 | 1.7 | 1.6 | 1.57 | 4.66 | 7 | 1.5 |

|  |  |  |
| --- | --- | --- |
| **Space** | **Benefit** |  |
| 1 | 7 | f(7,1) |
| 4 | 21 | e(14,3) |
| 8 | 33 | a(12,4) |
| 14 | 43 | b(10,6) |
| 18 | 49.4 | c(8,5) |

**R-5.3**

**3**

6

3

**2**

9

5

2

**1**

8

7

4

1

**0 1 2 3 4 5 6 7 8 9**

**R-5.11**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| {} | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| (12,4) | 0 | 0 | 0 | 0 | **12** | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| ..(10,6) | 0 | 0 | 0 | 0 | 12 | 12 | 12 | 12 | 12 | 12 | **22** | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 |
| ..(8,5) | 0 | 0 | 0 | 0 | 12 | 12 | 12 | 12 | 12 | 20 | 22 | 22 | 22 | 22 | 22 | **30** | 30 | 30 | 30 |
| ..(11,7) | 0 | 0 | 0 | 0 | 12 | 12 | 12 | 12 | 12 | 20 | 22 | 22 | 22 | 22 | 22 | 30 | 31 | 33 | 33 |
| ..(14,3) | 0 | 0 | 0 | 14 | 14 | 14 | 14 | 26 | 26 | 26 | 26 | 26 | 34 | 36 | 36 | 36 | 36 | 36 | **44** |
| ..(7,1) | 0 | 7 | 7 | 14 | 21 | 21 | 21 | 26 | 33 | 33 | 33 | 33 | 34 | 41 | 43 | 43 | 43 | 43 | 44 |
| ..(9,6) | 0 | 7 | 7 | 14 | 21 | 21 | 21 | 26 | 33 | 33 | 33 | 33 | 34 | 41 | 43 | 43 | 43 | 43 | 44 |

**(12,4), (10,6), (8,5), (14,3)**

**R-5-12**

This is a knapsack problem, where the weight of the sack is n, and each bid i corresponds to an item of weight ki and value di. If each bidder i is unwilling to accept fewer than ki widgets, then this is a 0/1 problem. If bidders are willing to accept partial lots, on the other hand, then this is a fractional version of the knapsack problem

**Lesson 11(b)**

**(C)**

Algorithm largestSum(S, L):

Input: set S of n items with weight wi; maximum weight L

Output: best subset of S with weight at most L

let A and B be arrays of length L + 1

for w ← 0 to L do

B[w] ← 0

for k ← 1 to n do

copy array B into array A

for w ← wk to W do

if A[w-wk] > A[w] then

B[w] ← A[w-wk]

return B[W]

**(D)**

Algorithm 01Knapsack(s, k, W)

if k = 0 then

return 0

else

wk ← weight(s.elemAtRank(k-1))

if wk > W then

return 01Knapsack(s, k-1, W)

else

bk ← benefit(s.elemAtRank(k-1))

return max( 01Knapsack(s, k-1, W), 01Knapsack(s, k-1, W-wk) + bk)

Algorithm 01KnapsackMemoized(s, k, W)

if M[k, W] > 0 then

return M[k, W]

if k = 0 then

M[k, W] ← 0

else

wk ← weight(s.elemAtRank(k-1))

if wk > W then

M[k, W] ← 01Knapsack(s, k-1, W)

else

bk ← benefit(s.elemAtRank(k-1))

M[k, W] ← max( 01Knapsack(s, k-1, W), 01Knapsack(s, k-1, W-wk) + bk)

return M[k, W]

**C-5.9**

Algorithm subset(s, l)

arr ← 2 dimentional array to store result

for i ← 0 to s.size() do

arr[i, 0] ← 0

for i ← 0 to l.size() do

arr[0, i] ← 0

for m ← 1 to s.size() do

for n ← 1 to l do

if n < s[m] then

arr[m, n] ← arr[m-1, n]

else

arr[m, n] ← max(arr[m-1, n], arr[m-1, l-n] + n)

sol ← new sequence to track optimal solution

a ← s.size()

b ← l

if arr[a, b] ¬= arr [a-1, b] then

sol.insertLast(arr[a,b])

a ← a-1

b ← b-s[a]

else

a ← a-1

**Lesson 12**

**R-6.1**

In an undirected graph with no self-loops and no parallel edges m ≤ n(n-1)/2 so

So according to the above graph, we have 3 components with 6, 4, 2 vertices each. The maximum edges are

* 6(5)/2 = 15
* 4(3)/2 = 6
* 2(1)/2 = 1 so the total is 15+6+1 = 22 which is less than 66

Briefly, since we have three components, we can’t reach this value if G has 66 edges.

**R-6.4**

By traversing this graph using BFS, we can figure out this sequence

LA15, LA16, LA31, LA127, LA32, LA22, LA126, LA169, LA141

**R-6.7**

1. The adjacency list structure is preferable. As the matrix structure allocates entries for 100,000,000 edges while the graph has only 20,000 edges.
2. The adjacency list structure is better than the adjacency matrix in regards to space allocation, as they allocate 20.010.000 and 100.000.000 respectively.
3. The adjacency matrix structure is preferable. Indeed, it supports operation areAdjacent() in O(1)time, irrespectively of the number of vertices or edges.

**Templates**

**(1)**

initResult(G)

S[]← Create new sequence array, S is a class member

componentNo ← -1

preComponentVisit(v)

componentNo ← componentNo + 1;

startVertexVisit(v)

S[componentNo].InsertLast(v)

preDiscoveryTraversal(v, e, w)

S[componentNo].InsertLast(e)

result(G)

return S

**(2)(a)**

Algorithm BFS(G)

Input graph G

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

**initResult(G)**

for all u э G.vertices()

setLabel(u, UNEXPLORED)

for all e э G.edges()

setLabel(e, UNEXPLORED)

for all v э G.vertices()

if getLabel(v) = UNEXPLORED

**preComponentVisit(v)**

BFS(G, v)

**postComponentVisit(v)**

**Result(G)**

Algorithm BFS(G, s)

L0 ← new empty sequence

L0.insertLast(s)

setLabel(s, VISITED)

i ← 0

**startVertexVisit(s)**

while ¬Li.isEmpty()

Li+1 ← new empty sequence

for all v э Li.elements()

for all e э G.incidentEdges(v)

if getLabel(e) = UNEXPLORED

w ← opposite(v,e)

if getLabel(w) = UNEXPLORED

**preDiscoveryTraversal(v, e, w)**

setLabel(e, DISCOVERY)

setLabel(w, VISITED)

Li+1.insertLast(w)

**postDiscoveryTraversal(v, e, w)**

else

setLabel(e, CROSS)

**crossEdgeTraversal (v, e, w)**

i ← i + 1

**finishVertexVisit(v)**

**(2)(b)**

Algorithm BFS(G, s, d)

L0 ← new empty sequence

L0.insertLast(s)

setLabel(s, VISITED)

i ← 0

**startVertexVisit(s, d)**

while ¬Li.isEmpty()

Li+1 ← new empty sequence

for all v э Li.elements()

for all e э G.incidentEdges(v)

if getLabel(e) = UNEXPLORED

w ← opposite(v,e)

if getLabel(w) = UNEXPLORED

**preDiscoveryTraversal(v, d, e, w)**

setLabel(e, DISCOVERY)

setLabel(w, VISITED)

Li+1.insertLast(w)

**postDiscoveryTraversal(v,d, e, w)**

else

setLabel(e, CROSS)

**crossEdgeTraversal (v, e, w)**

i ← i + 1

**finishVertexVisit(v)**

initResult(G)

path ← new sequence

pathFound ← false

preDiscoveryTraversal(v, d, e, w)

if ¬pathFound

if d = w then

parent ← getParent(w)

repeat

path.insertLast(parent)

parent ← getParent(w)

until parent = null

pathFound ← true

else

setParent(w, v)

Result(G)

if pathFound = true then return path

else return No\_Such\_Path\_Exist

**(2)(c)**

crossEdgeTraversal(v, e, w)

cycle ← new sequence

cycle.insertLast(w)

cycle.insertLast(v)

parent = getParent(v)

while parent ¬= null

cycle.insertLast(parent)

parent ← getParent(parent)

preDiscoveryTraversal(v, e, w)

if cycle ¬= null

setParent(w, v)

**(2)(d)**

Yes it is possible. The idea is to apply the level approach to this algorithm, and get the minimum path to the parent to create the shortest path. I will hand-in the code separately.

**(3)**

Algorithm DijkstraDistances(G, s, d)

Q ← new heap-based priority queue

for all v э G.vertices()

if v = s

setDistance(v, 0)

else

setDistance(v, Ø)

l ← Q.insert(getDistance(v), v)

setLocator(v,l)

while ¬Q.isEmpty()

u ← Q.removeMin()

for all e э G.incidentEdges(u)

{ relax edge e }

z ← G.opposite(u,e)

r ← getDistance(u) + weight(e)

if r < getDistance(z)

**preChangeDistance(u, z, r)**

setDistance(z,r)

Q.replaceKey(getLocator(z),r)

**finish(s, d)**

preChangeDistance(u, z, r)

// set the u’s parent to z

setParent(z, u)

finish(s, d)

parent = getParent(d)

if parent = null then

return No\_Such\_Path

else

E ← new sequence

while parent ¬= null do

E.insertLast(parent)

parent ← getParent(parent)

return E.elements()

**(4)**

Using DFS

initResult(G)

counter ← 0

preComponentVisit(v)

counter ← counter + 1

startVertexVisit(v)

setLabel(v, counter)

preDiscoveryTraversal(v, e, w)

setLabel(e, counter)

backTraversal(v, e, w)

setLabel(e, counter)

**HW1:**

initResult(G)

counter ← 0

preComponentVisit(v)

counter ← counter + 1

startVertexVisit(v)

setLabel(v, counter)

**HW2:**

initResult( )

S ← new sequence

preComponentVisit(v)

Q ← new heap based priority queue

postComponentVisit(v)

S.insertLast(Q.minKey())

edgeVisit(v, e, w)

Q.insertItem(weight(e), e)

result( )

return S