Q.1

a)

input: C[] U[] N k

create int[] Q size k

create double[] ratio size k

create boolean switch = true

for i=0 →k-1 do{

ratio[i]= (double) U[i] / (double)C[i] // set an array of the ration U/C

}

while(switch){

while(N>0){

//getMax(int[]) returns the index at which key the max value of the array is,returns -1 if array is empty

i =getMax(ratio)

if(i == -1) switch =false //when no more thing to buy stop the while loop

Q[i] = N/C[i] //int division assumed, put in the number of things bought

N= N-Q[i]\*C[i] //computes remaining money

ratio[i]=null // we bought as many as possible of this object

}

}

return Q

b) my algorithm will fails if N=12 , C = {3 , 3 , 9 }, U= { 0, 2, 7}

it will return Q = { 1, 0, 1 } which give U= 7

while the best result is Q={ 0,4 ,0 } which give U= 8

C)

If N=0 Q[N]=0

For N>0 :

Q[N] = Max{ U(i)+ utility(N-C(i)) for i=0 -> k-1 }

Create Q[] size N

Create array O[] size N // will stock the first object for each cost

for c = 0 to N do

Q[c] = 0

O[c] = 0

for i = 1 -> k-1 do

if (c>= C[i] && U[i] + Q[ c-C[i] ] > Q[c]) then

X[w] = V[i] + X[ w-W[i] ]

choice[w] = i

// the best is Q[N]

c= N

while (O[c] > 0) do

c = c – C[ O[c] ]

return Q[N];

d)

from the example we get (0,1,4,0) that’s C1=1 C2=4 and utility is 1\*5+4\*8=37

Q2 .

With A : array of unsorted int from 1-→2n

int k =1

for int i=0-→n //find max, big O(n)

do {

if (A[i]>k)

k= A[i]

}

Int[] count size k+1 //declare an array to count the repetition of every ints in the initial array big O(n)

for int i=0 -->n

do count[A[i]] = count[A[i]] +1 // counts the numbers of repetition big O(n)

for int i=1 --> k+1

do count[i] = count[i] +count[i-1] // sets counters without zeros in order big O(n)

int[] sorted size n //initialize sorted array , big O (n)

for int i=0-→n

do {

sorted[count[A[i]]] = A[i]

count[A[i]] = count[A[i]] -1

}

we have a few big O(n) operation not nested in one another hence we have big O(C\*n) = big O (n)

b) we couldn’t use this algorithm to sort any int array since I have an implicit minimum already set in the algorithm which is zero, it wouldn’t work on negative number.

Q.3 this is in a pdf file.

Q.4

a)

Problem:

Write an algorithm to compute the excentricity of a given vertex in a graph.

Algorithm:

excentricity(vertex u)

Input:a vertex u from the graph

Output: the excentricity of u

//find all distances for a vertex : keep the biggest one

//iterate trough graph setting distance of each vertices (from the initial one)

//iterate with B/D FS putting every distance in an array

create queue

create int[] distances

int Acounter =0 //array counter

int Ncounter =0 // neighbor counter

for all vertices I in getNeighbors(u) do{

if(! Get visited(i))

Ncounter++

queue.add(i)

}

for i=0 → N-1 do{

u=queue.remove()

setVisited(u,true)

// fills the distances array with the max of each path taken recursivly

distances[Acounter] = 1+excentricity(u)

Acounter++

}

int max =findmax(distances) // returns the max value in distances array

return max

b) so that no two adjacent vertices have the same color.

Algorithm

is2colorable(vertex u)

Input: a graph vertex u

Output: true if the graph to which u belongs is 2-colorable, and false otherwise

1.set first vetex to 0

2.set all u’s neighbors to 1

3.iterate trhough all vertices setting color to opposite of current if not colored and comparing color if colored , return false if vertex with same color as neighbor return true if iterate trough all graph without returning false

Create queue Q

enqueue u in Q

while Q not empty do {

v <- q.dequeue

setVisited(v,true)

setColor(u,0)

for all vertices w in getNeighbors(v) do{

if getVisited==false

c = getColor ( v)

if getColor(w) is null {

if (c==0) setColor(w,1)

else setColor (w,0)

}

Else if (c= getColor(w) ) return false

enqueue(w)

}//for loop

}//while loop

Return true