Solution

SECTION A – Answer Sheet

1.	A	B	C	D
2.	A	B	C	D
3.	A	B	C	D
4.	A	B	C	D
5.	A	B	C	D
6.	A	B	C	D
7.	A	B	C	D
8.	A	B	C	D
9.	A	B	C	D
10.	A	B	C	D
11.	A	B	C	D
12.	A	B	C	D
13.	A	B	C	D
14.	A	B	C	D
15.	A	B	C	D
16.	A	B	C	D
17.	A	B	C	D
18.	A	B	C	D
19.	A	B	C	D
20.	A	B	C	D

SECTION A – Answer Sheet

1.	A	B	C	D
2.	A	B	C	D
3.	A	B	C	D
4.	A	B	C	D
5.	A	B	C	D
6.	A	B	C	D
7.	A	B	C	D
8.	A	B	C	D
9.	A	B	C	D
10.	A	B	C	D
11.	A	B	C	D
12.	A	B	C	D
13.	A	B	C	D
14.	A	B	C	D
15.	A	B	C	D
16.	A	B	C	D
17.	A	B	C	D
18.	A	B	C	D
19.	A	B	C	D
20.	A	B	C	D

Section B

Instructions for Section B
Answer **all** questions in the spaces provided.

Question 1 (6 marks)
Consider the following algorithm:

```
ALGORITHM: find_max
INPUT: A[] - array with integer values
       start - first index in array to sort
       end - last index in array to sort
OUTPUT: maximum value in A[]
BEGIN
if start == end then   // we're looking at only one item in array
   return A[start]
mid = floor(start + end) / 2
max1 = find_max(A, start, mid)     // sort lower half of array
max2 = find_max(A, mid+1, end)  // sort upper half of array
if max1 > max2 then
   return max1
else
   return max2
END
```

a) What algorithmic design pattern is this an example of? Explain your answer.

Divide and conquer.
We find the overall maximum by dividing the problem in 2 halves each time & 2
finding the maximum of that half.

b) Write a recurrence relation for the runtime of this algorithm.

$T(n) = 2T(\frac{n}{2}) + O(1)$ 2

c) Solve the recurrence relation found in (b) to write an expression for the asymptotic runtime of this algorithm using Big-O notation.

$a = 2,\ b = 2,\ d = 0$

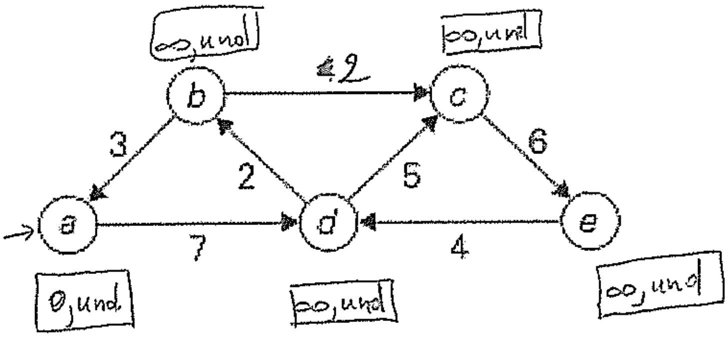
$b^d = 1$

$a > b^d$ 2

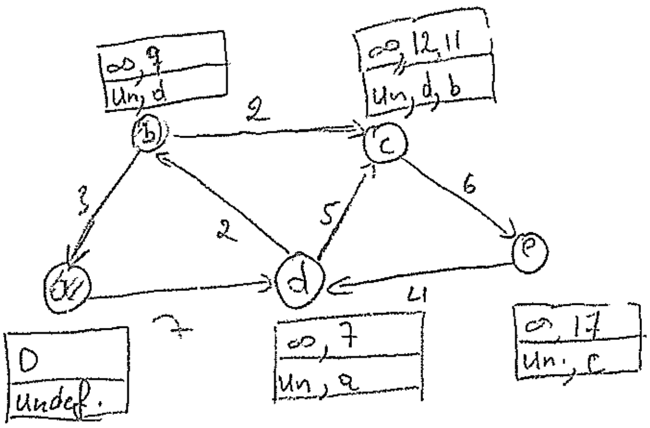
$T(n) \in O(n^{\log_2 2})$ or $O(n)$

Question 2 (4 marks)
Use Dijkstra's algorithm to find the shortest distances from vertex **a** to all others in the graph below. Show all your steps, assuming that each node has two attributes: distance and prev_node.

initialise:



steps:



	a	b	c	d	e
init	0	∞	∞	∞	∞
1		∞	∞	7	∞
2		9	12		∞
3			11		∞
4					17

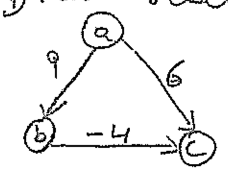
Question 3 (6 marks)

a) What type of graph is Bellman-Ford's algorithm able to solve which Dijkstra's algorithm is not able to?

a graph with one or more negatively-weighted edges.

1

b) ~~Give an example~~ Draw such a graph



1

c) Complete the pseudocode for the algorithm (only consider distances)

```
Algorithm Bellman-Ford
Input: an edge weighted directed graph G; two nodes A, B;
Output: the shortest path distance from A to B
Assumption: B is reachable from A

BEGIN
   // initialise the distance attributes of all nodes
     foreach node in allNodes(G)
        distance of node ⇐ INFINITY                          1
     end foreach
     distance of A ⇐ 0
   // work out the distances from A to all other nodes
   for i from 1 to length(allNodes(G))-1
      foreach eddge in allEdges(G)
         distance of endNode(edge) ← min(dist. of edge, distance of startNode(e)+w(e))
      end foreach
                                                               2
   end for

   // return the shortest path distance from A to B
   return distance of B                                        1
END
```

Question 4. (4 marks)
Consider an algorithm for medical diagnosis. The algorithm takes the following inputs:

- Smoker: True, False
- Weight: Underweight, Normal, Overweight, Obese
- Exercise: Low, Medium, High
- Drinker: None, Social, Regular

a) Explain why we might want to use pair-wise testing on this algorithm.

There can be too many combinations of inputs to test in a realistic time frame. Pair-wise testing makes sure that all possible pairs of values are tested.

1

b) Complete the following table, showing 12 test cases that would guarantee that every pair of inputs has been considered

Smoker	Weight	Exercise	Drinker
True	Under	Medium	None
True	Under	High	Social
True	Normal	High	None
True	Over	Medium	Regular
True	Over	High	Social
True	Obese	Low	Social
False	Under	Low	Regular
False	Normal	Low	Regular
False	Normal	Medium	Social
False	Over	Low	None
False	Obese	Medium	None
False	Obese	High	Regular

3

Look for:

True, Under
~~True~~, Normal
True, Over
True, Obese

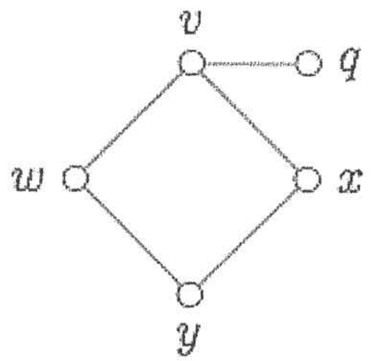
False, Under
False, Normal
False, Over
False, Obese

Under, Low
Under, Med
Under, High
Normal, Low
Normal, Med
Normal, High
Over, Low
Over, Med
Over, High
Obese, Low
Obese, Med
Obese, High

[illegible]

etc...

Question 5 (8 marks)



a) Complete the definition of this graph in graph theoretic notation:

V = {q, v, w, x, y }

E = {(q, v), (v, w), (w, y), (x, y), (x, v)} 2

b) Is this graph a tree? Explain your answer

NO. It has a cycle v, w, y, x, v.

2

c) How many edges would need to be added for this graph to be a complete graph?

5 edges. 1

d) What is the distance between vertices ~~p~~ w and q of the graph?

2

1

e) What is the vertex with the highest degree? What is the degree of that vertex?

v, degree - 3

2

Question 6 (4 marks)

Inspect the following algorithm and answer the questions below:

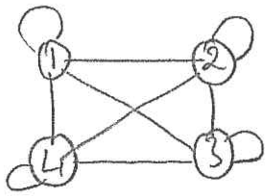
Input: A natural number n
Output: A complete graph with n vertices
for i=*1 to n* **do**
  addNode ~~n~~ i
  foreach *node in allthenodes* **do**
    addEdge (~~n~~ i,*node*)
  end
end

a) Describe what this algorithm does and the mistake that's in it.

It creates a complete graph of size n. The mistake is that it creates a loop at each vertex.

2

b) Show the result of running this algorithm with the input $n = 4$.



1

c) Correct the mistake (Only rewrite the relevant part of the algorithm)

foreach node in allthenodes do
  if node is not n then
    add Edge (n, node)
end

1

Question 7 (8 marks)

a) Complete the signature for a simple List Abstract Data Type (ADT)

Name: list

Import: element, boolean

Operations:

also accepted: newList, emptyList

empty : $\longrightarrow$ list

isEmpty: list $\longrightarrow$ boolean

prepend: list $\times$ element $\longrightarrow$ list

head: list $\longrightarrow$ element

tail: list $\longrightarrow$ list

4

b) Let *L* be an empty list. Show what L looks like after the following operations are performed:

L $\leftarrow$ append(L, 1)

L $\leftarrow$ append(L, 5)

L $\leftarrow$ prepend(L, 8)

L $\leftarrow$ append(L, L)

8	1	5	8	1	5

c) What list operations would be required to transform the following list

1	2	3

Into the this one:

3	4

L $\leftarrow$ tail(L)

L $\leftarrow$ tail(L)

L $\leftarrow$ append(L, 4)

Question 8 (6 marks)
Consider the algorithm below for calculating Catalan numbers:

```
1  Algorithm: Catalan
   input  : A natural number n
   output: The n-th Catalan number
2  begin
3  |  Let c be an array of Catalan numbers of length (n + 1)
4  |  c[0] ← 1
5  |  i ← 0
6  |  while i < n do
   |  |  Calculate the next Catalan number
7  |  |  value ← Σ_{k=0}^{i} c[k] × c[i − k]
8  |  |  i ← i + 1
9  |  |_ c[i] ← value
10 |_ return c[n]
```

a) What algorithm design pattern is this algorithm an example of?

Dynamic programming

1

b) Explain the benefits and trade-offs involved in using this algorithmic design pattern

It results in algorithms with much better runtime complexity than their recursive counterparts (the latter are often exponential).
There is usually a space complexity tradeoff

2

c) Replace line 7 with a loop that would calculate *value*

for k = 0 to i do
    value ← value + c[k] × c[i−k]
end for

1

d) Conduct a time complexity analysis of this algorithm and express your answer in Big-O notation.

The essential operation is the addition or mult, in the loop above. This occurs i+1 times for each value of i, that is n times.
Total executions of the essential op = 1 + 2 + 3 + ... + n−1
$= \frac{n(n-1)}{2}$
Therefore, time complexity $T(n) \in O(n^2)$

2

Question 9 (4 marks)
Write an algorithm that takes an array as input and returns an array with the same elements reversed.
Eg: INPUT

THUS	SPOKE	YODA

OUTPUT

YODA	SPOKE	THUS

1

```
ALGORITHM : reverse-array
INPUT     : Array A[] of size n
OUTPUT    : Array A[] with elements reversed
BEGIN
      S ← newStack()
      for i = 1 to n do
          S ← push(S, A[i])
      end for
      for i = 1 to n do
          A[i] ← top(S)
            S ← pop(S)
      end for
      return A[]
END
```

1 mk for how it's written
+1 mk for correct logic

1

This is very finicky!

Question 10 (7 marks)

Consider the following graph search algorithm:

ALGORITHM: Graph_Search

INPUT: G, a connected graph

start_v: a vertex to start from

target_v: the key being searched for

OUTPUT: true: if target_v is found

false: otherwise

BEGIN

```
Stack S ← emptyStack()
foreach node in all_nodes do
    node[visited] ← false
end foreach
push(S, start_v)
while S is not empty do
    u ← top(S)    // return element on top of the stack
    S ← pop(S)    // remove the top element of the stack
    if u == target_v then
        return true
    if u[visited] == false then
        u[visited] ← true
        foreach neighbour w of u do
            if w[visited] == false then
                push(S, w)
        end foreach
end while
return false
```

END

a) Does Graph_Search proceed in a depth-first or breadth-first direction?

depth first 1

b) What different ADT would you use to make it proceed in the other direction?

a queue instead of a stack 1

c) Rewrite the algorithm so that it returns the path from start_v to target_v if the target is found and an empty array otherwise.

1. Stack S ← empty Stack
2. Array P ← empty Array 1 - init path
3. foreach node in all_nodes do
4. node [visited] ← false
5. node [prev] ← undef

```
end foreach
push(s, start-v)
while S is not empty do
    u ← top(s)
    S ← pop(s)
    if u == target-v then
        While u[prev] != undef do
            append u[prev] to P[]
            u ← u[prev]
        end while
        append u to P[]
        return P[]
    if u[visited] == false then
        u[visited] ← true
        foreach neighbour w of u do
            if w[visited] == false then
                w[prev] ← u
                push(S, w)
        end foreach
end while
return P[] //empty
```

1 ← cond. to stop looping

1 ← code in loop

1 - returning path

1 → returning empty if we get here

Question 11 (3 marks)

The following algorithm for finding the sum of positive integers from 1 to x is iterative. Rewrite the algorithm using tail recursion:

```
function iter_sum(x)
 running_total ← 0
 while (x != 0) do // loop terminates when x == 0
        running_total ← running_total + x
        x ← x - 1
 end while
 return running_total
end function
```

1

```
function tail-rec-sum(x, running_total)
    if (x == 0) then
        return running_total
    else
        return tail-rec-sum(x-1, x+running_total)
```

1 (base case)

1 (rec. call)