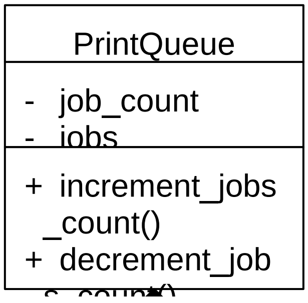
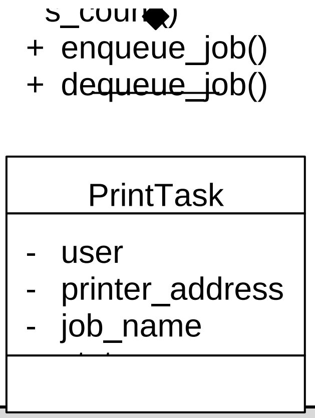
Qn	Task	Sample Solution
1a	**Networking, TCP/IP [6]**	
i	Layer 1: Application Layer	
i	Layer 3: Internet Layer	
i	Layer 4: Network/Link Layer	
ii	identifies the source and destination **port/program/process** of the data packet	
ii	determines how the data is to be sent (e.g. with acknowledgement), OR ensures data is sent/arrives in sequence	
iii	Reserved ports: 0 to 1023	
1b	**Data representation [5]**	
i	Appropriate conversion of binary digits to decimal values: 2^10 = 1024, 2^4 = 16, 2^3 = 8	2^15 --> 00000100 00011000 <-- 2^0 2^10 + 2^4 + 2^3
i	Summed up values: 1024 + 16 + 8 = 1048	= 1024 + 16 + 8
i	Correct ans	= 1048
ii	All binary values < 1024 will have first 6 binary digits == 0 (1023 in binary = 00000011 11111111)	
ii	So check if the **first 6 binary digits** are 0	
1c	**DNS [3]**	
i	Application Layer	
ii	**Map/associate** domain/subdomain names (not URLs) with IP addresses	
ii	Enable web browsers/clients to access servers/services using domain/subdomain names instead of IP addresses	
2a	**Validation, verification [2]**	
i	Ensure that the **input** to a function/program meets **a set of criteria/conditions**	
ii	Ensure that the **output** from a function/program matches the **expected** output	
2b	**Validation/verification & TCP [2]**	
	Validation: the program takes in a **port number** ... / **Verification:** the program passes the port number to another program .../	
	Validation: and has to ensure the port number is **valid (type) for use** / meets a **range check** / will **not cause an error** **Verification:** and has to ensure the port number is what the other program expects	
2c	**Testing [3]**	
	Normal: integers 0-65535	10, 65000
	Extreme: integers < 0, > 65535	-1, 65536
	Abnormal: hex, binary values, ...	"65000"
3a	**Class diagram, OOP**	
i	**Class** Task	 PrintQueue - job_count - jobs + increment_jobs_count() + decrement_jobs_count()
i	**Properties** (valid attribute names) - user - address - job_name - status	
i	**Class** Queue	
i	**Properties** (valid attribute names) - job_count	
i	Properties are private	



i	Getters and setters for properties (some setters might be optional)

i	**Queue:** task adding/sending: appropriate interface/method names (property optional: might not be stored as property)
	+ enqueue()
	+ dequeue()

i	Job count updating: appropriate methods/properties
i	Appropriate relation: Association: -- Directed association: --> Aggregation: --<> Composition: --<> (filled diamond)
ii	**Inheritance:** a **subclass**/**child class** inherits/can **access** attributes & methods ...
ii	**Inheritance:** ... from a **superclass**/**parent class** (but cannot modify/delete those attributes/methods from the superclass/parent class)
ii	**Polymorphism:** multiple classes/objects provide the **same interface**/set of methods ...
ii	**Polymorphism:** ... but with **different underlying implementation**
3b	**Data structures, OOP [6]**
i	Circular queue has **fixed size** (no. of slots), linear queue does not / CQ is **statically allocated**, linear queue can be dynamically allocated /
i	next **element after tail is head**, in a linear queue there is no element after tail / (other valid ans: CQ usually uses static memory allocation, CQ involves constant-time insertion/removal)
ii	**Polymorphism:** CQ & LQ share **same interface** ...
ii	... but **do not share** same underlying **implementation**
ii	they **use different data structures**/different **attributes**/different **algorithms**
ii	... hence **cannot make use of inherited attributes/methods**
3c	**Data structure, programming [4]**
	check for full array (e.g. head = -1), handle case
	item stored in array using head/tail as index
	increment tail/head **after insertion** ...
	... **with wraparound** (mod by size, or similar method)
4a	**Algorithms [7]**
i	bubble sort
ii	O(n^2)
iii	1. [2, 3, 4, 5, 1, 6]
iii	2. [2, 3, 4, 1, 5, 6]
iii	3. [2, 3, 1, 4, 5, 6]
iii	4. [2, 1, 3, 4, 5, 6]
iii	5. [1, 2, 3, 4, 5, 6]
4b	**Programming, optimisation [4]**
1	After each iteration, **largest element is at end of array**
	inner loop can therefore **exclude largest element(s)** in iteration (i.e. FOR j = 1 to Array.LENGTH - i)

2	If in a given iteration of j, **no swaps were needed**, the array is **already sorted**	
	subsequent iterations **can be skipped** (use a **sentinel value** e.g. swapped = False)	
4c	**Time efficiency [2]**	
	Time efficiency describes **how execution time increases with data size** / **number of elements / does not indicate actual performance**	
	insertion sort is **still faster** than bubble sort (across all sizes**)**	
5a	**Data normalisation [6]**	
i	3NF **requires** the tables be in 2NF, which **requires** they be in 1NF	
i	1NF **requires data to be atomic**	
i	The Subjects column contains multiple items and is thus **not atomic**	
i	Therefore the table is **not in 3NF**.	
ii	Normalised data: changes only **need to be made in one place**, vs multiple places in redundant data (more error-prone) / **simpler to enforce** data integrity	
ii	Normalised data: **takes up less space/storage**	
5b	**SQL [8]**	
i	**Level(id[PK], name)**	
i	**Class(id[PK], name,** levelId[FK]**)**	
i	**Student(id[PK], name,** classId[FK]**)**	
i	appropriate FK for class-level relation: **FK in class**	
i	appropriate FK for student-class relation: **FK in student**	

ii	Appropriate SQL command (SELECT) & syntax (esp. single-quote for literals)	SELECT Student.name, Class.name FROM Student
		INNER JOIN Class on Student.classId = Class.id
		INNER JOIN Level on Class.levelId = Level.id
		WHERE Level.name = 'JC2';

ii	Appropriate column names (dot notation, table names to disambiguate); irrelevant columns not retrieved	
ii	Tables appropriately joined	
5c	**SQL vs NoSQL [6]**	
i	NoSQL since data schema is likely to **undergo further changes** / **need frequent modification** ...	
i	NoSQL provides a more flexible way for the database to grow since it **does not require/enforce constraints** on data fields and types (database does not need to be recreated)	
i	**Startup is fast-growing** and will need the database to scale (in performance/storage) as they grow	
i	NoSQL databases are **able to scale horizontally** (by adding more machines to the database) to provide increased performance	
ii	NoSQL **does not enforce constraints**: the startup will have to implement more **validation code** in their service app	
ii	There is likely to be **more data duplication**/it is **harder to normalise data** with NoSQL since collections cannot be joined in a query: startup has to be more careful to avoid breaking data integrity	

5d	**Backup/archival [4]**	
	establish a **backup plan** / **back up data regularly** ...	
	backup location should be **safe from disasters** / have copies in sufficiently **different locations** (to avoid correlated risk) / backup to cloud storage in addition to external storage ...	
	backup plan should be **tested regularly** ...	
	... to ensure that backup data **can be restored** in case of data loss	
6a	**Hashing [2]**	
	take in a (variable-length string) **key** ...	
	and return a **hash index** / **hash value** unique to the key (to be used in hash table)	
6b	**Programming (pseudocode) [4]**	

	loop through data string	sum = 0
		FOR i ← 1 TO data.LENGTH
		  sum = sum + (i * (31 ** Ord(data[i])))
		ENDFOR

	while keeping track of index (starting from 1)	
	calculate i*(31**ascii) correctly	
	sum result and return	
6c	**BST, hash table [4]**	
i	faster lookup for DNS results **as entries grow** ...	
i	... because **lookups are O(1)** / involve **constant number of operations** regardless of hashtable size	
ii	BST is able to maintain DNS cache in **sorted** (domain name) **order** ...	
ii	... which is simpler/easier to iterate through / retrieve / export / search for partial match / other suitable advantage	
6d	**BST [9]**	
i	**in-order** (tree) **traversal**	

ii	recursive algorithm: handle base case (empty/no node) by returning empty list	FUNCTION in_order(node)
		  IF node == None THEN
		    RETURN []
		  ENDIF
		  left ← in_order(node.LEFT)
		  right ← in_order(node.RIGHT)
		  RETURN CONCATENATE(left, [node.VALUE],

ii	**recursively** retrieve contents of left child node	
ii	**recursively** retrieve contents of right child node	
ii	concatenate retrieved contents with root (with appropriate wrapping of root in array)	
ii	in the correct order: left-root-right	
iii	BST **performs optimally** when balanced	
iii	as entries are added and removed, the tree might **become unbalanced** (because more entries are ordered before/after the root)	
iii	recreating the BST allows it to **rebalance** / **become balanced** again (with a new root node from the median element).	