Solutions for Inf2-IADS Sample Exam 2020/21 (prepared April 2021):

PART A:

1. (a) The diagram is essentially

   [3,9,8,2,4,1,6,5]
   [3,9,8,2] [4,1,6,5]
   [3,9] [8,2] [4,1] [6,5]
   [3] [9] [8] [2] [4] [1] [6] [5]
   [3,9] [2,8] [1,4] [5,6]
   [2,3,8,9] [1,4,5,6]
   [1,2,3,4,5,6,8,9]

   with the obvious arrows added.

   [5 marks; roughly 2 for the splittings and 3 for the mergings.]

   (b) Best, worst and average case times are all $\Theta(m * 2^m)$.

   [1 mark each.]

   (c) To mergesort a list of length 2^m we only need an array of size $2 * 2^m$ (proof is an easy induction). So space needed is $\Theta(2^m)$.

   [1 mark for the answer; 1 mark for any reasonable supporting point.]

2. This is the question about the different f_i; we have $f_1(n) = (\lg(n))^{\lg(n)}$, $f_2(n) = n^{\lg(n)}$, $f_3(n) = n^2$ and $f_4(n) = n$.

   note: this is an example of where it's important to know the "rules of logs".

   (a) We are asked to identify the $i \in \{1, 2, 3, 4\}$ such that

   $$f_i(n) \in O(f_j(n)) \text{ for all } j = \{1, 2, 3, 4\}.$$

   Basically we want the f_i that is "asymptotically slowest".

   This function is $f_4 = n$.

   The (informal) justification for this ...

   - n *certainly* grows slower than $f_3 = n^2$.
   - Also n certainly grows slower than $f_2 = n^{\lg n}$ ($\lg n \geq 2$ for every $n \geq 4$)
   - To consider relative growth of f_4 against f_1, we will take the lg of both functions: then $\lg(f_1(n)) = \lg(\lg(n)^{\lg(n)})$ which is equal to $\lg(n) \cdot \lg\lg(n)$ by "rules of logs" and $\lg(f_4(n)) = \lg(n)$. Hence $\lg(f_1(n)) = \lg(n) \cdot \lg(f_4(n))$ so $\lg(f_1(n)))$ grows asymptotically faster than $\lg(f_4(n))$, and $f_1(n)$ grows much much faster than $f_4(n)$. Hence $f_4 = O(f_1)$.

   [5 marks: 2 for identifying the correct i as 4, 1 mark each for the justification wrt each $j \neq 4$].

(b) We have to identify the $k \in \{1,2,3,4\}$ such that

$$f_k(n) \in \Omega(f_j(n)) \text{ for all } j = \{1,2,3,4\},$$

and *formally* justify this.

The function f_k is $f_2 = n^{\lg(n)}$.

To *prove* the $f_k = \Omega(f_j)$ for another j we need to find a $n_0 \in \mathbb{N}$, $c \in \mathbb{R}^{>0}$ such that $n^{\lg(n)} \geq c \cdot f_j$ for all $n \geq n_0$.

- For $j = 1$, we can set $c = 1, n_0 = 2$, then $n \geq \lg(n)$ for $n \geq 2$ and hence $n^{\lg(n)} \geq \lg(n)^{\lg(n)}$ (as the exponent $\lg(n)$ is a positive value > 1 for $n \geq 2$)
- For $j = 3$, we set $c = 1, n_0 = 4$, then for $n \geq 4$ we have $\lg(n) \geq 2$ and hence $f_3(n) = n^{\lg(n)} \geq n^2$.
- For $j = 3$, we set $c = 1, n_0 = 2$, then for $n \geq 2$ we have $\lg(n) \geq 1$ and hence $f_3(n) = n^{\lg(n)} \geq n$.

[5 marks: 2 for identifying the correct i as 4, 1 mark each for the formal justification wrt each $j \neq 4$].

3. (a) The hash table will have better average case performance. If 10,000 integers are stored, the average bucket size will be 5, so on average, 5 key comparisons will be required for an insertion or unsuccessful lookup, and fewer for a successful lookup. A binary tree storing 10,000 integers, even if perfectly balanced, will have depth around $\lg 10,000 \simeq 13$, so a lookup will involve 12 comparisons on average.

However, the hash table has terrible worst case performance: if all 10,000 integers hash to the same value, a lookup might require 10,000 comparisons. For red-black trees, the depth (hence the number of comparisons) will be bounded by something around $2 \lg 10,000 \simeq 26$.

[3 marks for discussion of average case, 3 marks for worst case. Ballpark figures are acceptable, as in the above answer. Any other reasonable points are acceptable.]

(b) The 1 is first added as the left child of 2, and initially coloured R [1 mark]. We then apply the 'red uncle' rule, turning 2 and 4 B and 3 R [2 marks]. Finally, since 3 is the root, we turn it B [1 mark]. So in the end, 2,3,4 are B and 1 is R.

4. A *decimal counter of length* n consists of an array D indexed by $0, \ldots, n-1$, in which each cell contains one of the decimal digits $0, \ldots, 9$. The *value* $v(D)$ of such a counter is the sum $\sum_{i=0}^{n-1} D[i] * 10^i$.

The following operation increments the current value of the counter modulo 10^n:

```
Inc():
    i = 0
    while i<n and D[i]==9
        D[i] = 0
        i = i+1
    if i<n
        D[i] = D[i]+1
```

(a) Best case time is $\Theta(1)$ [1 mark]. This is illustrated by any scenario where $D[0] \neq 9$ [1 mark].

(b) Worst case time is $\Theta(n)$ [1 mark]. Illustrated by the case where $D[i] = 9$ for all i [1 mark].

(c) Suppose we gauge runtime cost by the number of times the while-test is evaluated (the total number of line executions will be within a factor of 6 of this). Each of the 10^n Inc's will do this test at least once; a tenth of them will do it at least twice; a hundredth at least three times, etc. So total runtime cost will be

$$10^n(1 + \frac{1}{10} + \frac{1}{100} + \cdots + \frac{1}{10^n}) \; < \; 10^n * \sum_{i=0}^{\infty} 10^{-i} \; = \; 10^n * 1.1111\cdots$$

Thus total cost is between 10^n and $10^n * 10/9$, and so is $\Theta(n)$. Amortized average cost per operation is thus between 1 and 10/9, hence $\Theta(1)$. [1 mark for each of the answers; 4 marks for the justification.]

5. Trying to reduce Independent Set to 3-SAT.

(a) Given $C = (\ell_1 \vee \ell_2)$, we can replicate the exact conditions under which C is true by introducing any logical variable x_C and replacing C by the conjunction of the following two 3-CNF clauses: $C' = (\ell_1 \vee \ell_2 \vee x_C)$ and $C'' = (\ell_1 \vee \ell_2 \vee \neg x_C)$

[2 marks for the correct construction]

(b) For a k-literal clause $C = (\ell_1 \vee \ldots \vee \ell_k)$ for $k \geq 4$, we define $k-2$ variables $x_{C,2}, \ldots, x_{C,k-2}$ and replace C by the conjunction of the following clauses:

$$\begin{array}{rcl} C(2) & = & (\ell_1 \vee \ell_2 \vee \neg x_{C,2}) \\ C(3) & = & (x_{C,2} \vee \ell_3 \vee \neg x_{C,3}) \\ \cdots & \cdots & \cdots\cdots\cdots \\ C(k-1) & = & (x_{C,k-2} \vee \ell_{k-1} \vee \ell_k) \end{array}$$

[2 marks for the correct construction]

(c) The graph $G = (V, E)$ $(n = |V|, m = |E|)$ will have an Independent Set of size ≥ 4 if and only if for every subset $V' \subseteq V, |V'| = n - 3$, V' has at least one vertex in $\mathcal{I}$. Using the boolean variable x_v to indicate "v is in $\mathcal{I}$" ($x_v = 1$ in this case), we can represent "V' has at least one vertex in $\mathcal{I}$" as the length-$(n-3)$ clause $\bigvee_{v \in V'} x_v$.

We need to consider *every subset of size* $(n-3)$, so we have $\binom{n}{3}$ of these length-$(n-3)$ clauses, joined by a $\bigwedge$. By (b) we know any one of the length-$(n-3)$ clauses can be re-cast as $(n-5)$ 3-CNF clauses. Doing this for each of the big clauses gives a 3-CNF Φ which has $(n-5)\binom{n}{3}$ clauses in total.

To ensure adjacent vertices can't belong to $\mathcal{I}$, we add $(\neg x_u \vee \neg x_v)$ for every $e = (u, v) \in E$. By (a), these can be coded as 2 3-CNF clauses, giving $2m$ extra clauses.

[4 marks - 1 for the length-$(n-3)$ clause, 1 for the characterisation as 3-CNF, 1 for the counting of all big clauses, 1 for the "non-adjacent" clauses.]

(d) If we consider generalising the reduction in (c) to the case where we were interested in an Independent Set of size $\geq h$, for an arbitrary h, we would find ourselves considering taking $\binom{n}{h-1}$ different subsets of V, and adding a "big clause" (eventually realised as the $\wedge$ of a number of 3-CNF clauses) for *each* of these. If h is variable, then the number of "big clauses" is no longer polynomially-sized. So no, this approach doesn't work.

[2 marks for a decent explanation.]

FOR PART B:

1. (a) After the first Heap-Extract-Max, the array has $18, 11, 8, 1, 6, 5$.
   After calling Max-Heap-Insert(17), the array then has $18, 11, 17, 1, 6, 5, 8$
   After calling Heap-Extract-Max again, the array then has $17, 11, 8, 1, 6, 5$
   After calling Max-Heap-Insert(7), the array then has $17, 11, 8, 1, 6, 5, 7$
   After calling Max-Heap-Insert(19), the array then has $19, 17, 8, 11, 6, 5, 7, 1$
   [5 marks for the right answer]

   (b) We are considering the implementation of *ExtendedQueue*.

   i. There is not really any intelligent way to implement findElement on a Heap. If we try to search from the root, then whenever we are exploring a node v, if we find that $v.key < k$, we are allowed to ignore all nodes below v. However, if $v.key > k$ then we must search in both of the subtrees below v, since the Heap structure does not tell us anything more about where k might be.
   We can however implement an algorithm which does a traversal of the Heap, and which stops exploring when $v.key < k$. In the worst-case this has a running time of $\Theta(n)$. Consider the case where k lies is the lowest level of the Heap... In this case k is no more likely to be in any spot (on this level) than another. However the lowest level of the Heap may contain $n/2$ elements in total (if the total size of the Heap is n).
   [3 marks for a correct findElement and 2 marks for justifying the $\Theta(n)$ (1 for each of $O(n)$ and $\Omega(n)$).]

   ii. To implement maxElement on a Red-Black tree, we observe that the element with the Maximum key in a R-B tree is the rightmost "near-leaf", giving the following algorithm for maxElement:
   I. Start at the root of the R-B tree and keep walking to the right-hand child of the current node, until the next right child is a trivial leaf.
   II. Return the element stored in this position.
   This little algorithm has worst-case running-time $\Theta(h) = \Theta(\lg(n))$, given what we know about the height of R-B trees.
   For removeMax, we initially perform the same steps as for maxElement. Then we *delete* the node we arrived at - this will potentially require the re-colouring of nodes up to the root of the tree (in the case that we deleted a previously-black node). There are a number of cases to patch-up the tree, but all can be executed in time proportional to the height which is $\Theta(\lg(n))$.
   [1 mark for removeMax and 1 for its running time, 2 marks for removeMax and 1 for its running time.]

   iii. The operations removeMax and insertItem both have worst-case running time $\Theta(\lg(n))$ on a Heap, and on a R-B tree. The operations findElement and maxElement have different asymptotics on the different structures.

If we are implementing an *Extended Queue* on a Heap, then the $\Theta(n)$ running-time of findElement on a Heap might be prohibitive. If we expect findElement to be used relatively frequently for a particular application of *ExtendedQueue*, then we should use the Red-Black implementation.
However, the R-B implementation of maxElement is less efficient than the $\Theta(1)$ implementation of the Heap. So if findElement calls are infrequent, we might choose to work with the Heap implementation.
[5 marks, awarded based on the quality of the arguments.]

(c) To implement UpdateKey(o, k'), we assume that o is a direct reference to the object $o = (e, k)$, so we don't need to carry out findElement. We want to update the key value of o to become k', but doing this may violate the Max-Heap property. There are two cases:

- If $k' > k$, then k' may be larger than the parent of o.
  This scenario is very similar to the situation in Max-Heap-Insert after the new key gets inserted at the available leaf - the solution is to "swap with the parent" until o sits at a position where k' is less than the parent's key. This is $O(h) = O(\lg(n))$ time.
- If $k' < k$, then k' may be smaller than the key(s) at o's child nodes. This can be resolved by making the call Max-Heapify at o, and again this will run in $O(\lg(n)$ time.

[5 marks, with 3 going for a correct solution to *either* of the two cases, and the other 2 for the second case.]

2. (a) The CYK chart is:

	cows	,	goats	and	sheep
cows	I,AL,CL		CL		CAL,L
,					
goats			I,AL,CL		AL,CAL,L
and					
sheep					I,AL,CL

[7 marks. Approximately 0.5 per correct entry.]

(b) The grammar is ambiguous [1 mark]. For instance, *goats and sheep* can be parsed as either AL or CAL(this example is encountered in the course of constructing the CYK chart above). [1 mark for any example.]

(c) The LL(1) parsing algorithm will fail at the very first step. We wish to expand the start symbol L. Suppose the first input token is *cows.* We cannot tell whether to expand L to AL or to CAL, as both are consistent with this input information (e.g. the string could be *cows and goats and sheep* or *cows, goats and sheep.* [2 marks for identifying the point of failure, 2 marks for an explanation showing clear understanding.]

(d) The parser executes as follows:

Operation applied	Remaining input	Stack state
	I , I and I	L
Lookup L, I	I , I and I	I Rest
Match I	, I and I	Rest
Lookup Rest, ,	, I and I	, I CTail and I
Match ,	I and I	I CTail and I
Match I	and I	CTail and I
Lookup CTail, and	and I	and I
Match and	I	I
Match I	end of input	empty stack

[10 marks; roughly 1 mark per line.]

(e) In general, LL(1) grammars aren't appropriate for NLP. Even in cases where an NL grammar can be made LL(1), doing so may artificially eradicate genuine ambiguities: a single interpretation will be selected and others ignored, and the ambiguity won't even be flagged up. [2 marks; any reasonable point accepted.]

3. (a) **Algorithm** hamilton($G = (V, E)$)

```
1. $len \leftarrow 1$
2. Initialize $visited$ array of length $|V|$ to FALSE in every cell.
3. $visited[v_1] \leftarrow$ TRUE
4. return hamiltonFromVertex($G = (V, E)$, $v_1$)
```

Algorithm hamiltonFromVertex($G = (V, E)$, w)

```
1. if (($len = |V|$) and($(w, v_1) \in E$)
2.       then return TRUE
3. else if (($len = |V|$) and($(w, v_1) \notin E$)
4.       then return FALSE
5. else
6.       $len \leftarrow len + 1$
7.       for all ($x \in Adj(w)$ such that $visited[x]$ = FALSE) do
8.           $visited[x] \leftarrow$ TRUE
9.           if hamiltonFromVertex($G = (V, E)$, $x$)
10.              return TRUE
11.          $visited[x] \leftarrow$ FALSE
12.      $len \leftarrow len - 1$
```

There are two main differences from depth-first search. The first is the lack of a loop in the "top-level" method iterating over all vertices (for a HC in a undirected graph it makes no difference where we start, since we require a single connected component). The second difference that when we meet a new vertex x, we mark it visited, and we subsequently continue out exploration from there, but if this fails to generate a HC, we will revert the "visited" status of x as we backtrack.

[The 7 marks are divided as 2 for the top-level method, and up to 5 marks for accurate details for the recursive method. Their structure approach may be slightly different, marks will be awarded fairly.]

(b) We assume that competing adjacent nodes are explored in order of increasing index. If we start our exploration at node 0, then the lowest-indexed adjacent edge is $(0, 1)$, and we continue our exploration from 1.

Note that $\{1, \ldots, \frac{n}{2} - 1\}$ is a *complete* graph, so hamiltonFromVertex(G, 1) (with 0 already visited), will explore every permutation of $\{2, \ldots, \frac{n}{2} - 1\}$ $((\frac{n}{2} - 2)!$ of these) before returning to 0. All are guaranteed to be explored, as we know there is no HC with 0 adjacent to 1, there is a unique HC which contains $(0, n - 1)$ and $(0, \frac{n}{2} - 1)$.

(note we can show a similar result for exploring the other edges $(0, i)$, $i = 2, \ldots, \frac{n}{2} - 2$ from 0, but we don't need this for the result.)

It is well-known that $k! \geq 2^k$ for $k \geq 4$, therefore $(\frac{n}{2} - 2)! \geq 2^{\frac{n}{2}-2}$. And using $c = \frac{1}{4}$ for the 2^{-2}, and $n_0 = 4$, clearly $2^{\frac{n}{2}-2} = \Omega(2^{\frac{n}{2}})$.

[3 marks for noticing the exploration of the complete graph on the top needs to be done in full, and explaining why wrt the indices, 2 marks for relating this to the unique HC and adjacencies of the 0 vertex, 2 marks for working out a lower bound wrt $(\frac{n}{2} - 2)!$, 1 mark for relating this to power of 2.]

(c) i. We can set $k = n$ and then a tour of value $\leq k$ (for this graph) can only be achieved with a tour where every edge has weight 1, as appropriate (note we require $n \geq 2$ for this!)
[2 marks for a correct answer]

ii. The constructed tour will *not* correspond to a HC of the original graph. This is because the first step of Greedy (breaking ties for lower indices) will be to add $(0, 1)$ to the tour, and commit to that edge. We already know there is no HC with the edge $(0, 1)$ in it.
[2 marks for the answer, 2 marks for justifying]

iii. The initial tour $0, 1, \ldots, \frac{n}{2} - 1, \frac{n}{2}, \ldots, n - 1$ consists of $(n - 1)$ edges of weight 1 (this includes the "wraparound" edge $(n - 1, 0)$) plus the weight n for edge $(\frac{n}{2} - 1, \frac{n}{2})$.
The only "swap" moves which have the potential to change the status of the weight-n edge are $i = \frac{n}{2} - 2, i = \frac{n}{2} - 1$ and $i = \frac{n}{2}$. In the first case vertex $(\frac{n}{2} - 2)$ will end up adjacent to $\frac{n}{2}$, and this also has weight n, so no improvement is made. For $i = (\frac{n}{2} - 1)$, the swap would bring $\frac{n}{2}$ adjacent to $(\frac{n}{2} - 2)$ and also $(\frac{n}{2} - 1)$ adjacent to $(\frac{n}{2} + 1)$, *two* edges of weight n, so strictly worse ... and again for $i = \frac{n}{2}$ the proposed change would result in two edges of weight n. None of these options has (strictly) lower tourValue than our original tour, so no changes will be made.
[2 marks for the answer, 2 marks for justifying]