Module Title: Informatics 2 – Introduction to Algorithms and Data Structures
Exam Diet: April 2020
Brief notes on answers:
PART A:

1. (a) The diagram (which I cannot upload to the exam machine) has stack entries for A pointing to 'array 1' on the heap, for B and C both pointing to 'array 2', and for D containing the value 5. Both array 1 and array 2 have two elements. The elements of array 1 contain references to distinct copies of the array [2,3,5]; those of array 2 both contain a reference to array 1.

   [Roughly 2 marks each for A and B, 1 mark each for C and D.]

   (b) Address of first copy of [2,3,5]; address of second copy; address of second copy; the integer 5. [1 mark each.]

2. (a) insert(16): Try $h(16, 0) = 6$ which is free, so 16 goes in cell 6.

   insert(36): Try $h(36, 0) = 6$ which is occupied, then try $h(36, 1) = 9$ which is free, so 36 goes in cell 9.

   insert(29): Try $h(29, 0) = 9$ which is occupied, then try $h(29, 1) = 2$ which is free, so 29 goes in cell 2.

   lookup(46): Look in cell $h(46, 0) = 6$, which is occupied but not by 46. Then look in $h(46, 1) = 9$ (ditto), and $h(46, 2) = 2$ (ditto). Finally look in $h(46, 3) = 5$ which is unoccupied. We conclude that 46 isn't present in the set.

   Final state of table T has $T[2] = 29, T[6] = 16, T[9] = 36$ and all other cells contain -1.

   [4 marks for the three insertions, 2 marks for the lookup, 1 mark for final state of table.]

   (b) For example, suppose we try delete(16), lookup(36). The deletion will locate the cell 6 containing 16 and put -1 there. Then lookup(36) will first probe cell 6 and see -1, so will then stop probing and wrongly conclude that 36 isn't present.

   [2 mark for a correct example, 1 mark for explanation.]

3. (a) Not true that $f_1 = O(f_2)$, the other three are all true. [2 marks]

   (b) True that $f_2 = o(f_1)$, the other three are all false. [2 marks]

   (c) True that $\sqrt{n} = O(n/4)$: e.g. taking $C = 1$ and $N = 16$, we have for all $n \geq N$ that $\sqrt{n} \leq n/4 = Cn/4$. [2 marks for good choice of C, N, 1 mark for the right inequality.]

   (d) False that $f_1 = o(f_1)$. This would be saying

   $$\forall c > 0.\ \exists N.\ \forall n \geq N.\ f_1(n) < c.f_1(n)$$

   But even with $c = 1$, we never have $f_1(n) < f_1(n)$ for any n. [1 mark for showing an understanding of o, 1 mark for good choice of c, 1 mark for the rest.]

4. **(worked example, 10 marks)**

For the given graph, we first write down $D^{<0}$, which is

$$D^{<0} = \begin{bmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 4 & -2 & 5 \\ \infty & \infty & 0 & \infty & \infty \\ 1 & 3 & \infty & 0 & \infty \\ \infty & \infty & 4 & 1 & 0 \end{bmatrix}.$$

Note we index the rows by $0, \ldots, 4$ to match the vertices.

To compute $D^{<1}$ we allow ourselves to consider intermediate visits through V_1, which is $\{0\}$. The vertex 2 row will not change, as vertex 2 has no outgoing edges (and this remains true for all iterations). Otherwise the only row which can make use of paths through 0 is the vertex 3 row (as $3 \to 0$ is an arc), but using this will not improve any distances. So $D^{<1}$ is $D^{<0}$.

For $D^{<2}$, we consider paths which might visit 1 as an intermediate point. Rows for vertices 3 and 0 have non-∞ values for $3 \to 1, 0 \to 1$, so we consider updating these.

$$D^{<2} = \begin{bmatrix} 0 & 5 & 9 & 2 & 20 \\ \infty & 0 & 4 & -2 & 5 \\ \infty & \infty & 0 & \infty & \infty \\ 1 & 3 & 7 & 0 & 8 \\ \infty & \infty & 4 & 1 & 0 \end{bmatrix}.$$

Next we consider routes with 2 as an intermediate point - however, 2 has no outgoing arcs/paths, so considering 2 will not change any distances. So $D^{<3}$ is $D^{<2}$.

Next consider routes with 3 as an intermediate point, and this results in improvements/updates.

$$D^{<4} = \begin{bmatrix} 0 & 5 & 9 & 2 & 10 \\ -1 & 0 & 4 & -2 & 5 \\ \infty & \infty & 0 & \infty & \infty \\ 1 & 3 & 7 & 0 & 8 \\ 2 & 4 & 4 & 1 & 0 \end{bmatrix}.$$

Next routes with 4 as an intermediate point - however, given the high values in the final column, this does not result in any improved paths. So $D^{<5}$ is $D^{<4}$.

marking: 2 marks for setting-up $D^{<0}$, 2 marks for computing/explaining $D^{<1}$, 2 marks for computing/explaining $D^{<2}$, 1 mark for explaining why $D^{<3}$ is $D^{<2}$, 2 marks for computing/explaining $D^{<4}$, 1 mark for explaining why $D^{<5}$ is $D^{<4}$.

5. **(worked example for 5, thinking for 5)**

   (a) **marking:** Up to 5 marks depending on details.

   (b) In this second part of the question we are asked to consider the changes to the asymptotics for "best case" and "worst case" when we can assume all keys are from $\{1, 2, \ldots, 10\}$.

   For "worst case", the answer is still $\Theta(n^2)$. We can see this by studying the case where every single item in the array is the same - in that case, PARTITION will result in one subarray of length reduced by 1 (with the pivot forked-off). Applying this iteratively gives $\Theta(n^2)$.

   For "best case", the answer is also $\Theta(n^2)$, considerably worse than the unrestricted. The reason is because if we only have 10 possible key values, then *one of those values* appears at least $n/10$ of the time, therefore whenever Partition is called on that subarray of identical keys, the execution will take $\Omega((n/10)^2)$ time, which is also $\Theta(n^2)$.

   marking: 1 mark each for the correct answers, 2 marks for explaining why the "worst case" has this value, 1 mark for best case.

6. (a) False, true, true, false, true.

   [1 mark each.]

   (b) $f(n)$ is bounded below by $(n^2 - n + 1)\lg n$, and above by $(n^2 - n + 1)\lg(n^2) = 2(n^2 - n + 1)\lg n$. Hence clear that $f(n) = \Theta(n^2 \lg n)$.

   [2 marks for the answer, 3 for the justification. The above would suffice: explicit values for c, C, N not required.]

7. (a) In the first step, a new red node bearing 4 (with two trivial children) is added as the right child of '3'. In the second step, the red uncle rule is applied, so '3' and '7' turn black and '5' turns red. In the third step, the red root is turned black.

   [2 marks for first step, 3 marks for second step, 2 marks for third step.]

   (b) 4 times. Each application of red-uncle pushes a non-trivial black downwards in order to cure a double-red; this may introduce another double-red at the next level up. Since there are 4 non-trivial blacks along each path, this can happen at most 4 times, and it's clear that this max can be attained.

   [1 mark for answer, 2 for explanation.]

8. The course of computation is:

Operation	Input left	Stack state
	(n)	Exp
Lookup (, Exp	(n)	(Exp) Ops
Match (	n)	Exp) Ops
Lookup n, Exp	n)	n Ops) Ops
Match n	)	Ops) Ops
Lookup Ops,)	)	) Ops
Match)		Ops
Lookup Ops, $		Stack empties

[2 marks for attempting something of the right form, and 1 mark per correct line.]

PART B:

1. (a) Something like the following (minor variations allowed):

```
MergeSort3(A,p,q):
   if q-p == 1
      return [A[p]]
   else if q-p == 2
      if A[p] <= A[p+1] then return [A[p],A[p+1]]
      else return [A[p+1],A[p]]
   else
      r = floor((2p+q)/3),  s = floor((p+2q)/3)
      B = MergeSort3(A,p,r)
      C = MergeSort3(A,r,s)
      D = MergeSort3(A,s,q)
      return Merge3(B,C,D)
```

   [3 marks for something of broadly the right form; 2 marks for correct base cases; 2 marks for details of split.]

   (b) The recurrence relation is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 2 \\ 3T(n/3) + \Theta(n) & \text{otherwise} \end{cases}$$

   The Master Theorem therefore applies with $a = b = 3, k = 1$. Since $\log_3 3 = 1$, we are in the 'balanced case' and the solution is $T(n) = \Theta(n \lg n)$ (as for binary mergesort).

   [3 marks for recurrence relation, 1 for solution, 2 for explanation.]

   (c) For a subarray of size 9, there are 1+3+9 = 13 calls to MergeSort3. In general, for $n = 3^d$, there are $(3n-1)/2$ calls (obtained as the geometric sum $\sum_{i=0}^{d} 3^i$). For ordinary mergesort, the corresponding formula is $(2n-1)$ where $n = 2^d$.

   [2 marks for the size 9 case, 2 for the general formula, 2 for the binary analogue.]

   (d) We reason informally. If we first ignore the $O(1)$ 'error terms', then at each level of the recursion, the sizes of all calls to Merge 3 sum to n, so the number of comparisons for this level is estimated by $5n/3$. And there are about $\log_3 n$ recursion levels. So total number of comparisons is about $(5n/3)\log_3 n = (5/(3 \lg 3))n \lg n \simeq 1.05n \lg n$.

   For the error terms, there is an $O(1)$ contribution for each call, and by part (c) there are $\Theta(n)$ calls. So we have an upper bound of $(5/(3 \lg 3))n \lg n - \Theta(n)$ for the number of comparisons. For binary mergesort, we know from lectures, the corresponding formula is just $n \lg n - \Theta(n)$ (we know from lectures that a binary merge of size n requires at most $n-1$ comparisons).

   [This part is more challenging, though I'm not expecting anything more rigorous than the above. 3 marks for the coefficient $5/(3 \lg 3)$; 1 mark for the error term; 2 marks for the binary version.]

2. (a) Possible solution:

```
Insert(A,a):
   if A == nil then A = Cell(a,nil)
   else if a <= A.value then A = Cell(a,A)
   else A.next = Insert(A.next,a)
   return A
```

[4 marks. Any reasonable solution accepted, being lenient towards minor misunderstandings of the intended conventions re objects and references.]

(b) Possible solution:

```
InsertSort(A):
   B = nil
   C = A
   while C != nil
      Insert(B,C.value)
      C = C.next
   return B
```

[5 marks, guidelines as for (a).]

(c) `Insert(A,a)` clearly recurses at most length(`A`) times, with each call contributing $O(1)$ time, so runtime is $O(\text{length}(\texttt{A}))$. To insert-sort `A` of length n, we perform n such insertions into lists of length $0, 1, \ldots, n-1$ respectively, so total runtime is $O(n^2)$. Moreover, this upper bound is attained in the case that `A` was already sorted, so worst-case runtime is $\Theta(n^2)$. [1 mark for $O(n^2)$ upper bound, 2 marks for justifying this, 1 mark for the bound being attained.]

(d) For each n, let L_n be a reverse-sorted list such as $[n, n-1, \ldots, 2, 1]$. For such lists, each insertion will insert some number i into a linked list representing $[i+1, i+2, \ldots, n]$, which will only take constant time. So runtime of linked-list insert-sort on L_n takes time $\Theta(n)$. For the array-based insert-sort, however, runtime on L_n will be $\Theta(n^2)$, since to insert i at the head of $[i+1, \ldots, n]$ will require moving all the existing elements to the right. [1 mark for a suitable family of lists, 1 mark for each runtime.]

(e) Possible solution:

```
MergeSort (A,n):
   if n==1
      B = Cell(A.value,nil)
      return (B,A.next)
   else
      m = floor(n/2)
      (L1,A1) = MergeSort (A,m)
      (L2,A2) = MergeSort (A1,n-m)
      M = merge(L1,L2)
      return (M,A2)
```

[4 marks for some evidence of right idea, 8 marks for a perfect solution.]

(f) $\Theta(n \lg n)$ [1 mark].

Comment on B1: Although 'MergeSort for linked lists' is easily Googleable, I think the question is OK because the easy-to-find sources use the approach of splitting the

list into two halves first. In theory, the above approach ought to do better for large `n`, though I haven't yet implemented it to check... They may also be able to find some help online for linked list InsertSort — but I guess adapting an existing solution to the above framework would be at least as hard as just doing it?

3. (a)

	duck	egg	blue
duck	NP	NP	NP, NP, AP
egg		NP	NP, AP
blue			NP, A

(with the evident pointers added). Note the two ways of forming NP in the top-right cell. [4 marks for the entries, 2 marks for the pointers.]

(b) The entry in cell (i, j) should in effect be a table associating with each non-terminal X the number of syntax trees for $w_0 \cdots w_{j-1}$ with root X [2 marks for making this clear]. Let's write this number as $C[i, j, X]$.

For $j-i = 1$, the count for each X will be 1 if $X \to w_i$ is present in the grammar, 0 otherwise [1 mark]. Recall that a grammar is essentially a *set* of productions, so there's no danger of the same production 'appearing twice'.

For $j - i > 1$, we need to set

$$C(i, j, X) = \sum_{i<k<j} \sum_{(X \to YZ) \in \mathsf{G}} C(i, k, Y) \times C(k, j, Z)$$

[4 marks for any clear way of conveying this idea.]

(c) In the following chart, a cell entry m, n, p means that the relevant substring has m trees with root NP, n trees with root AP, and p trees with root A.

	duck	egg	blue	border
duck	(1,0,0)	(1,0,0)	(3,1,0)	(10,0,0)
egg		(1,0,0)	(1,1,0)	(4,0,0)
blue			(1,0,1)	(2,0,0)
border				(1,0,0)

The '10' at top right (the final answer) arises as $(1\times4)+(1\times2)+((3\times1)+(1\times1))$.

[2 marks if the meaning of the chart is clear. 6 marks if solution seems correct modulo minor slips; 6 marks for a perfect solution.]

(d) Base case $n = 2$: the string aa clearly has a syntax tree, and $2^0 = 1$.

Induction step $n > 2$: given any tree Δ for a^{n-1}, we may form one tree for a^n by first applying $S \to SS \to aS$ then using Δ on the remaining S; and another by first applying $S \to SS \to Sa$ then using Δ. So if there were m trees for a^{n-1}, this would give $2m$ trees for a^n, clearly all distinct. (There may of course be others.) But by the induction hypothesis, there are at least 2^{n-3} trees for a^{n-1}; hence there are at least 2^{n-2} trees for a^n.

[1 mark for base case, 3 marks for induction step.]

(e) No. In the case of the grammar from (d), even just writing out the $\geq 2^{n-2}$ parses for a^n (never mind computing them) will take time $\Omega(2^n)$, which is faster growing than any polynomial in n. [1 mark for 'no', 1 mark for explanation.]

Comment on B2: This is both a Lang Proc question and in some sense a DP one.

4. This question asks about the following "Greedy" approximation algorithm for Vertex Cover:

Algorithm Greedy-VC($G = (V, E)$)

1. $E' \leftarrow E$
2. Initialise array K, $K[v] \leftarrow 0$ for all $v \in V$
3. Initialise array D, $D[v] \leftarrow deg(v)$ for each $v \in V$
4. **while** $(E' \neq \emptyset)$
5. $\quad w \leftarrow \text{argmax}_v\{D[v] : D[v] \neq 0\}$
6. $\quad K[w] \leftarrow 1$
7. $\quad$ Update E', D to reflect deletion of all edges adjacent to w
8. **return** K

(a) **"new" reasoning (10 marks)**

We need to implement Greedy-VC to to achieve a running-time bound of $\Theta(n^2)$ for Greedy-VC, specifying the details of how we initialise D in line 3., how we compute w in 5., how we update D and E' in line 7., and how we perform the test of the while-loop.

initialising D in line 3: We have an *Adjacency Matrix* representation for E. To compute $deg(v)$ for any v, we need to check the number of 1 values in the v-th row of E. This takes $\Theta(n)$ time for any individual v, and $\Theta(n^2)$ time for all of the vertices. Fortunately this initialisation is only done once.

How to compute w in line 5.: We want to find the vertex of maximum residual degree. We assume that the array D has the residual degrees (we show below how this is updated). Therefore it suffices to do a linear pass on D to find the largest value and its index w, achieveable in $\Theta(n)$ time.

how we update D and E' in line 7: We carry out a linear pass on row w of E', checking each entry. For every $E'[w, v]$ with value 1, we *subtract 1 from* $D[v]$, and then reset $E'[w, v] \leftarrow 0$. We then do a linear pass on column w of E', setting every entry to 0.

performing the test of the while loop: We just check to see whether there are some non-0 entries in D (this can be combined with the computation for line 5). $\Theta(n)$

Note that we have shown that the test of the while-loop, and each of the actions within that loop, are $\Theta(n)$. There can be at most n iterations of the loop, so the whole loop concludes in $\Theta(n^2)$ time, and together with the initialisations before the loop, the Algorithm takes $\Theta(n^2)$ time in total.

marking: 2 marks for the initialisation of D, 2 marks for line 5, 3 marks for line 7, 1 mark for the while-loop, 2 marks for bringing it all together.

(b) (i). **"new" reasoning (4 marks)**

For every $v \in L$, and every k we have the restriction that v can only be adjacent to *one* of the vertices in R_k. Therefore the maximum number of

adjacent vertices to v from $R_m \cup \ldots \cup R_2$ can be at most the number of R_is that we see in there - that is, $(m-2)+1 = m-1$.

marking: up to 4 marks for making the right points.

(ii). **"new" reasoning (8 marks)**

We want to show that the vertex cover constructed is all of R. We proceed by induction on the R_i of maximum index that still has vertices which are active in the graph.

We will prove that at each step, we have all of L still participating in the graph, plus all R_k for $k \leq m$, with K containing all nodes from $R_k, k > m$.

base case: this is certainly true at the beginning for $m = n$, before we add *any* vertices to K.

induction step: Suppose that this is true for some m, and consider what happens next. The I.H. says that K contains R_k for every $k > m$. By (a)i. above, this implies that *every* $u \in L$, that the current degree (after the R_k for every $k > m$ are eliminated) is at most $m-1$.

However, every node in R_m has degree m, so we will choose one of these nodes next. Also, the nodes in R_m only have neighbours in L, so removing adjacent edges to $v \in R_m$ cannot affect the degree of any of the other R_m. Hence the algorithm will add *each of the* R_k *vertices* in succession before considering other vertices. Hence K will now contain all of the R_k for $k > (m-1)$, and we now have our proof of the induction step.

The process finished when $m \leftarrow 1$, and then K contains all the nodes of R.

marking: Up to 8 marks depending on the details/quality of the argument.

(iii). For the Spindle graphs it will always be the case that there are more nodes in R than in L. However, since these are bipartite graphs, we can just take L, of size n, and this will form a Vertex Cover.

marking: 3 marks for the correct answer.