Module Title: Informatics 2 – Introduction to Algorithms and Data Structures
Exam Diet: August 2022
Brief notes on answers:
PART A:

1. (a) (reasoning, 5 marks)

   (i). $\lg(e^n) = \omega(n)$ is **false**

   (ii). $n^3 + 16n^2 - 100 = \Omega(n^2)$ is **true**

   (iii). $\lg(n^2 + n^5) = O(n)$ is **true**

   (iv). $2^n = \Omega(n!)$ is **false**

   (v). $2^n + n^{100} = \Theta(n^{100})$ is **false**

   marking: 1 mark for each correct answer, no need for explanations.

   (b) (reasoning, 5 marks)

   Since the head pointer is initially pointing to position 3, the three dequeue operations will cause this to shunt forwards to position 4, 5, 6 ... so after the dequeues we have the head at position 6 and the tail at position 7. Then we have the 6 enqueue operations which get executed after the tail pointer to position 7, so taking positions 8, 9, 0, 1, 2 and 3. So after all operations head will point to position 6, and tail to position 3.

   marking: 1 mark each for head and tail locations, 3 marks for the reasoning.

2. (a) $\Theta(n)$.

   [1 mark for n, 1 mark for using Θ. No justification required.]

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

   [Roughly 1 mark for base case, 3 marks for recurrence clause.]

   (c) We have $a = 2, b = 2, k = 1$. Solution is $T(n) = \Theta(n \lg n)$.

   [Roughly 1 mark each for a, b, k, 1 mark for solution. Partial marks will be given for students who have written out th Master theorem with some values, but failed to label the constants accurately.]

   Notes: The pseudocode will take a bit of time to assimilate — this is compensated by the very small amount of writing required. The example is exactly parallel to Mergesort which is covered in detail in lectures — but if they're able to recognize that, they deserve the help that that might give.

3. (a) (reasoning, 3 marks) The sorting algorithm of the three which has asymptotic worst-case most similar to asymptotic best case is Mergesort. For this algorithm best-case, average-case and worst-case are all $\Theta(n \lg(n))$. The reason for this is mainly due to the even splits throughout, the split always being half/almost half regardless of the key values in the subarray. It is also the case that the "work done" to combine two subarrays into the merged one is somewhat independent of the keys ... at least, it is always linear in the size of the merged subarray.

   marking: 1 mark for saying MergeSort, 2 marks for explaining.

(b) (reasoning, 3 marks)

The sorting algorithm of the three which has asymptotic worst-case farthest from asymptotic best case is Insertsort, which has $O(n)$ best-case and $\Theta(n^2)$ worst-case. This is a bigger gap than all others. The reason for the disparity is the "insert one each time" structure of the algorithm, with nothing at all being done (if the key is in sorted order), but a long sweep of the prefix array when the new key is smaller than the others.

marking: 1 mark for saying Insertsort, 2 marks for explaining.

(c) When we have no access to a sorting algorithm, we can consider employing a Red-Black tree and its constituent operations in order to achieve the desired sort. We can insert all keys into the R-B in $O(n.\lg(n))$ time, because we have the $O(\lg(n))$ bound on a single insertion. After doing this, we know that a R-B will always have the minimum key at the leftmost leaf. Hence we can add a second phase, of deletions this time, where we walk (the $O(\lg(n))$ distance) to the leftmost leaf at each step, and then extract/delete it in $O(\lg(n))$ time. Again the whole phase will be $O(n.\lg(n))$.

marking: 2 marks for explaining a method to achieve the sorting, and 2 marks for justifying the running time.

4. (a) (worked example, 5 marks)

$$d = \begin{array}{c} \\ L \\ O \\ R \\ D \end{array} \begin{array}{c} \begin{array}{cccccc} B & O & A & R & D & \end{array} \\ \left[\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 1 & 2 & 3 & 4 & 5 \\ 2 & 2 & 1 & 2 & 3 & 4 \\ 3 & 3 & 2 & 2 & 2 & 3 \\ 4 & 4 & 3 & 3 & 3 & 2 \end{array}\right] \end{array},$$

That gives an edit distance 2, and the alignment of strings $\begin{array}{ccccc} B & O & A & R & D \\ L & O & - & R & D \end{array}$

marking: 1 mark for showing the alignment of strings, 1 mark for the 2, and the other 3 for the details of the matrix.

(b) (reasoning, 3 marks) The key points to notice for this part are (*) that the tables used by "seam carving" have exactly the same $n \times m$ (and/or $(n+1)(m+1)$) dimensions that seam carving does. (*) The second thing to notice is that in the *recurrence* for "seam carving" the calculation to decide the value for cell $[i, j]$ is dependent on taking the minimum from 3 other cells - the $[i-1, j]$, the $[i, j-1]$ and the $[i-1, j-1]$. It is exactly those three cells that edit distance calculations checks each step. (*) The "base cases" (top row and top column) in each algorithm use $O(1)$ work to complete one cell.

Essentially we have the same number of $\Theta(mn)$ cells getting filled (in the exact same order) with (1) work at each step. So asymptotic running-time will be the same.

marking: Up to 3 marks depending on details and quality of argument.

(c) (reasoning, 2 marks)

The actual input to "seam carving" is a large array of size $m \times n$, which means the seam carving running time is actually *linear* in the input size, but for edit

distance (which has just $O(m+n)$ input size) it's as bad as quadratic. So the seam carving running-time is better wrt the input size.

marking: Couple of marks if they can observe/explain this.

5. First we eliminate the ternary rules: this yields e.g.

$$S \to \epsilon \mid aX \mid bY \qquad X \to Sa \qquad Y \to Sb$$

Next we eliminate the ϵ-rule. Since only S is nullary, this yields

$$S \to aX \mid bY \qquad X \to Sa \mid a \qquad Y \to Sb \mid b$$

There are no unit productions to be dealt with, so it only remains to eliminate the terminals from the binary rules. We introduce non-terminals A, B with rules $A \to a$, $B \to b$, and replace the above by

$$S \to AX \mid BY \qquad X \to SA \mid a \qquad Y \to SB \mid b$$

[Three stages, respectively worth 3,4,3 marks. Minor variations acceptable, e.g. in the order of the stages.]

PART B:

1. (a) Any list of keys $k_1, \ldots, k_n$ that are distinct modulo m will do: e.g. $1, \ldots, n$. Each goes into the table at the first attempt, so the total number of attempts is simply n.

   [Very easy. 2 marks for an example, 1 for total number of attempts.]

   (b) Any list of distinct keys $k_1, \ldots, k_n$ that are *equivalent* mod m will do: e.g. $m, 2m, \ldots, nm$. Insertion of k_j will then take j attempts and will go in at cell $(k_j + j) \bmod m$, since the cells $(k_j + i) \bmod m$ for $i < j$ will already be filled. So the total number of attempts will be $\sum_{j=1}^{n} j = n(n+1)/2$.

   Clearly this *is* the worst case, because when k_j is inserted, only $j - 1$ cells are filled, so we cannot have more than $j - 1$ failed attempts. The growth rate is clearly $\Theta(n^2)$.

   [2 marks for a correct example. 2 marks for the formula $n(n+1)/2$, 1 mark for the growth rate. 2 marks for explaining why the formula is correct for the example. 1 mark for saying why this is indeed the worst case.]

   (c) No. There are infinitely many possible keys and only finitely many permutations of $[m]$. So for *any* choice of $\#'$, there will be some permutation that arises as $\#'(k, 0), \#'(k, 1), \ldots, \#'(k, m-1)$ for infinitely many keys k.

   Taking $k_1, \ldots, k_n$ to be any selection of such keys, we find ourselves in the same situation as above: when we come to insert k_j, the cells $\#'(k_j, 0), \ldots, \#'(k_j, j-1)$ will be already full and we will insert in cell $\#'(k_j, j)$ at the jth attempt.

   [Harder, but the hint is fairly heavy. 2 marks for 'no'. 2 marks for noting that the 'pigeonhole principle' applies. 1 mark for arriving at the example. 2 marks for justification.]

   (d) Suppose e.g. $\#'$ is such that $\#'(1,0) = \#'(2,0) = \#'(3,0)$ and $\#'(1,1) = \#'(2,1) \neq \#'(3,1)$. (Clearly such functions $\#'$ exist.)

   If we insert the keys 1,2,3 in that order, then 1 takes 1 attempt and 2,3 each take two attempts: total 5 attempts.

   However, if we insert in the order 3,2,1, then 3 takes 1 attempt, and 1 takes 2 attempts, filling cell $\#'(1,1)$, so 2 takes 3 attempts: total 6 attempts.

   [Medium difficulty, involving some fiddly detail. Roughly 5 marks for a suitable example, 2 marks for analysis of the numbers of attempts.]

2. (a) (worked example, 8 marks)

From our starting point with $S = \{0\}$, we have two outgoing edges $0 \to 3$ and $0 \to 4$. This gives a potential route with value $d[0] + w(0,3) = 0 + 4 = 4$ for vertex 3 and gives the potential value $d[0] + w(0,4) = 2$ for vertex 4. Hence we add 4 to S and update d and π with our improved fringe options:

$S = \{0, 4\}$ $\quad d$

0	∞	∞	4	**2**

$\quad \pi$

–	NIL	NIL	0	0

Next we need to consider the edges outgoing from 4, $4 \to 1$, $4 \to 2$ and $4 \to 3$. Those give us $d[4]+w(4,.)$ values of $2+2 = 4$, $2+4 = 6$ and $2+3 = 5$ respectively. For vertices 1 and 2 this is an improvement on what was available fo far (∞), so we update the arrays. We could either add 1 or 3 to the set S, let's break ties for the lower indexed vertex 1. So:

$S = \{0, 1, 4\}$ $\quad d$

0	**4**	6	4	**2**

$\quad \pi$

–	4	4	0	0

The outgoing edges from 1 are $1 \to 0$ (ignore, as 0 is in S already) and $1 \to 3$, whose new option is $d[1] + w(1,3)$ values of $4 + 2 = 6$ and this is worse/longer than the prior option, so we don't update the arrays.

Next we add 3 to S, and explore its outgoing edges. The only outgoing edge is $3 \to 2$, with value 1 ... this would give $d[3] + 1 = 4 + 1$ as an alternative route for vertex 2 ... and this *is* an improvement. Our result is

$S = \{0, 1, 3, 4\}$ $\quad d$

0	**4**	5	**4**	**2**

$\quad \pi$

–	4	3	0	0

Finally we add 2 to S, to have $S = \{0, 1, 3, 4\}$ and we are finished. No changes to the arrays because there are no fringe edges to consider now.

marking: 2 marks for each of the 4 stages, and those 2 are split with 1 for specifying which vertex is added next, and 1 for the details of the arrays.

(b) (worked example, 10 marks) For APSP, we consider the "paths via i" at each step, in increasing order of i. First we consider allow the vertex 0 as intermediate, and this only is relevant for $0 \to 1 \to .$ given the many ∞ in column 0. The only $D^{<0}[1,0] + D^{<0}[0,k]$ to improve what we already have is for $k = 5$, giving value 3. Here is $D^{<1}$

$$\begin{bmatrix} 0 & \infty & \infty & 4 & 2 \\ 1 & 0 & 4 & 2 & \mathbf{3} \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & 1 & 0 & \infty \\ \infty & 2 & 4 & 3 & 0 \end{bmatrix}$$

Next we allow "paths via vertex 1" and again there is just one relevant row, which is row 4, we get the sequence $3, -, 6, 4, 5$ as alternative $4 \to 0 \to k$ options. The only one which improves $D^{<1}[4,k]$ is the first one, for cell $[4,0]$. Here is $D^{<2}$:

$$\begin{bmatrix} 0 & \infty & \infty & 4 & 2 \\ 1 & 0 & 4 & 2 & 3 \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & 1 & 0 & \infty \\ \mathbf{3} & 2 & 4 & 3 & 0 \end{bmatrix}$$

Next we should consider "paths via vertex 2" but there are none as 2 only has incoming edges. So $D^{<3}$ is exactly $D^{<2}$.

Next we should consider "paths via vertex 3", and the only options for improvement are those of the form $D^{<3}[\cdot, 3] + D^{<3}[3, 2]$, given the ∞ values in row 3. This improves the option for cells $[0, 2]$ and $[1, 2]$, giving $D^{<4}$:

$$\begin{bmatrix} 0 & \infty & \mathbf{5} & 4 & 2 \\ 1 & 0 & \mathbf{3} & 2 & 3 \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & 1 & 0 & \infty \\ 3 & 2 & 4 & 3 & 0 \end{bmatrix}$$

Finally we consider "paths via vertex 4", and just one cell is updated for $D^{<5}$:

$$\begin{bmatrix} 0 & \mathbf{4} & 5 & 4 & 2 \\ 1 & 0 & \mathbf{3} & 2 & 3 \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & 1 & 0 & \infty \\ 3 & 2 & 4 & 3 & 0 \end{bmatrix}$$

marking: 2 marks each for $D^{<1}, \ldots, D^{<5}$, with 1 mark for showing the details, 1 mark for the short explanation.

(c) (reasoning, 4 marks) This question asks about the impact on run-time if Floyd-Warshall were to instead operate with respect to an adjacency list representation of the input. Makes no difference to the asymptotics. This is something of a trick question in fact. It is true that looking through the adjacency list for vertex u when wanting to look up the weight of $u \to w$ is awkward (and not "direct lookup"). However, the very first step of Floyd-Warshall is to create matrix $D^{<0}$ from the weights in the input graph. This can we done in $O(m)$ time by considering each list in turn, and using the edges/weights there to fill the appropriate row of $D^{<0}$. After that we use the $D^{<k}$ matrices to get $D^{<k}$ so we don't need to consider the input representation.

marking: Up to 4 marks for a well-explained answer. Even a wrong answer which shows some understanding will get some marks.

(d) (reasoning, 3 marks) This question asks about the impact on run-time if Dijkstra were to instead operate with respect to an adjacency matrix representation of the input. The main effect would be at the point where we consider "fringe edges" after committing the most recent fringe vertex v - we would need to examine $\Theta(n)$ cells of the adjacency matrix, regardless of how many outgoing edges v actually had. This would change the m inside the running-time to n^2, which may be considerably worse if the graph was sparse.

marking: Up to 3 marks for a well-explained answer.

3. (a) (reasoning, 5 marks) This Greedy algorithm will work on the graph representation, most likely Adjacency list structure. We maintain a set I of the IS vertices, but the main work involves deleting vertices and edges from V' and E' at each step. This could potentially take $O(n^2)$ at a single step perhaps, but there are at most n steps.

   It is necessary to have access to the degree counts, and ideally those would be initialised in an array at the start of the algorithm (taking $O(m)$ time) and then reduced in value (or crossed out) as the algorithm persists. The update time would be no greater than the deletions for Vertices and Edges.

   marking: Up to 5 marks for a complete explanation.

   (b) (worked example, 5 marks) First of all we will note that for this particular graph there is an independent set with three vertices, containing $\{1, 3, 5\}$.

   If we apply GreedyIS, then we must select the lowest-indexed degree-2 node, which is 0. This will require us to delete all adjacent edges, and the vertices 1 and 5 (as well as 0) . This results in a "triangle" on 2,3,4 being left, and as it's a triangle, we can only take 1 of those three vertices into an IS. So we get the independent set $\{0, 2\}$ which only has 2 vertices (when 3 is possible). **marking:** Up to 5 marks for good details.

   (c) (new reasoning/algorithm, 10 marks) For this part of the question, we will define $Out(v)$ to be the child nodes of v for every $v \in V(T)$. We will refer to the leaves of T as $L(T)$. The recurrences asked for are the following ones:

$$\kappa_{v,0} = \begin{cases} 0 & v \in L(T) \\ \sum_{u \in Out(v)} \max\{\kappa_{u,0}, \kappa_{u,1}\} & \text{otherwise} \end{cases}$$

$$\kappa_{v,1} = \begin{cases} 1 & v \in L(T) \\ 1 + \sum_{u \in Out(v)} \kappa_{u,0} & \text{otherwise} \end{cases}$$

   If we are interested in the "max size independent set" or T, we don't care whether the root r is in there or not. Therefore we will compute both these $\kappa_{v,\cdot}$ quantities for each vertex (bottom-up) and then we will take the max of $\kappa_{r,0}$ and $\kappa_{r,1}$.

   It will be easy to *store* the computed values in the tree itself (by expanding the nodes to also have two count parameters), but we could alternatively have two linear arrays to store these.

   It will be very straightforward to apply these recurrences for a specific node, as we can easily check which are the child nodes, and then do the lookups. However, we need to make sure that all results have been computed for the lower nodes, before applying the recurrence for v. One way to do that is to have a "postorder traversal" wrapper as the outer structure of the final algorithm.

   marking: 5 marks for correct recurrences with all details, and 5 marks for the details to how to achieve it algorithmically.

   (d) The reason that the polynomial-time algorithm of (c) is not in conflict with our NP-completeness result for I.S is because we only showed that the *general* case of I.S. is NP-complete, not this special case. In fact, the specific details of the "reduction" which showed NP-completeness has many small triangles in it. It is quite far from being a tree.

marking: 5 marks for the general/trees distinction, with some reference to explain how the construction differs.