UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08026 INFORMATICS 2: INTRODUCTION TO ALGORITHMS AND DATA STRUCTURES

Monday 9 th August 2021

13:00 to 15:00

INSTRUCTIONS TO CANDIDATES

1. **Answer all five questions in Part A, and two out of three questions in Part B. Each question in Part A is worth 10% of the total exam mark; each question in Part B is worth 25%.**
2. **Calculators may be used in this exam.**

Convener: D.K.Arvind
External Examiner: J.Gibbons

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

PART A

1. (a) Which of the following statements are true and which are false? You need not justify your answers. All functions are assumed to be from the positive integers to the positive reals.

   i. $n \; = \; o(3n)$
   ii. $n + \sqrt{n} \; = \; \Theta(n)$
   iii. $n^{100} \; = \; O(100^n)$
   iv. $2^n \; = \; \Omega(n!)$
   v. $n^2 + O(n^2) \; = \; \Theta(n^2)$

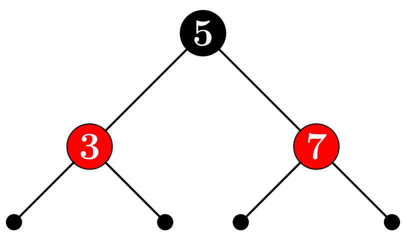
   [5 marks]

   (b) Give a tight asymptotic estimate (using Θ) for the value of

$$f(n) \;=\; \sum_{i=n}^{n^2} \lg i$$

   Mathematically justify your answer. [Hint: You do not need an exact formula for this summation, but there is an easy way to give explicit lower and upper bounds.] *[5 marks]*

2. (a) Consider the following red-black tree representing the set $\{3, 5, 7\}$. (The nodes labelled 3 and 7 are red, the rest are black.)



   Explain in three steps what happens when the operation 'insert(4)' is performed, including diagrams of the two intermediate tree states and the final state. Your diagrams should include the trivial nodes. You may indicate the colours of nodes in any way you wish. *[7 marks]*

   (b) Suppose we have a red-black tree in which every root-to-leaf path has 5 black nodes (including the root and the trivial leaf node). Suppose now that we insert one new item into this tree. What is the maximum possible number of applications of the *red-uncle rule* that this might involve? Give a short informal explanation. *[3 marks]*

3. Below is the LL(1) parse table for a simple grammar for arithmetic expressions. The start symbol is Exp. We think of the terminal n as representing a lexical class of numerals.

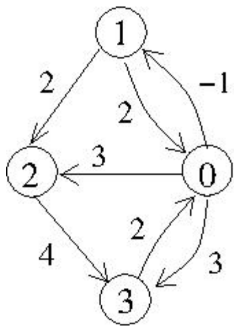
	n	(	)	+	$
Exp	Exp $\to$ n Ops	Exp $\to$ (Exp) Ops			
Ops			Ops $\to \epsilon$	Ops $\to$ + Exp	Ops $\to \epsilon$

Show how the LL(1) parsing algorithm executes on the input string

(n)

showing at each stage the operation performed, remaining input and stack state. Use the table format from Lecture 22. [*10 marks*]

4. Consider the *all pairs shortest paths (APSP)* problem on (directed) weighted graphs, and the dynamic programming algorithm for solving APSP. Run the dynamic programming algorithm on the directed graph below, constructing each of the matrices $D^{<1}$, $D^{<2}$, $D^{<3}$ and $D^{<4}$. Please justify your updates (or lack of updates) with a sentence of two about each of these $D^{<k}$. [*10 marks*]



As a starting point, the initial matrix $D^{<0}$ of directed edges is

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

(we index the rows as $0, \ldots, 3$ to match the vertices.)

5. In this question we consider polynomial-time reductions between NP-complete problems, specifically reductions from 3-SAT.

We will work in relation to the example

$$\begin{aligned}\Phi &= (x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee x_5) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \\ &\quad \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_5)\end{aligned}$$

(a) Consider the polynomial-time reduction from 3-SAT to INDEPENDENT SET given in our slides/lectures. Draw the "Independent set" graph of that reduction for the Φ above, and also state the total number of edges in the constructed graph, plus the "k" (required size of the Independent set) for the instance. *[6 marks]*

(b) Describe how we could carry out a polynomial-time reduction from 3-SAT to the VERTEX COVER problem instead. *[2 marks]*

(c) There is a very quick way to justify that the example formula Φ does have a satisfying assignment. What is it? *[2 marks]*

[**Hint:** Think about the expected number of clauses of a 3-SAT formula satisfied by a uniform random assignment to the logical variables.]

PART B

1. In this question we consider a variation of mergesort which, at each level of the recursion, splits the subarray in question into three roughly equal parts rather than two. We assume we are already given a function Merge3(B,C,D) that takes three already sorted arrays and merges them to produce a new sorted array, which it returns. You need not provide code for Merge3.

   (a) By adapting the MergeSort pseudocode from lectures, give pseudocode for a recursive function MergeSort3(A,p,q) that uses 3-way mergesort to sort the portion of the array A from A[p] to A[q−1] inclusive, returning the result in a new array of size q−p. *[7 marks]*

   (b) A natural implementation of Merge3 will have worst-case runtime $\Theta(n)$, where n is the combined size of the three arrays supplied. Using this information, write down an *asymptotic recurrence relation* satisfied by $T(n)$, defined as the worst-case time for MergeSort3 on subarrays of size n.

   Use the Master Theorem to obtain an asymptotic solution to this recurrence relation, indicating briefly how your result is obtained. *[6 marks]*

   (c) We now look more closely at some of the constants hidden within this asymptotic estimate, comparing them with those for ordinary MergeSort. We first consider the *number of recursive calls* required by the two versions.

   How many calls to MergeSort3 (including the top-level call) would be required to sort a subarray of size 9? Generalize this observation to give an exact closed formula for the number of calls required for a subarray of size n, where n is an exact power of 3. Give an analogous formula for ordinary MergeSort, assuming n is a power of 2. You need not give proofs. *[6 marks]*

   (d) Next, we investigate *number of comparisons* between array elements. A good implementation of Merge3 will have worst-case number of comparisons $5n/3 - O(1)$, where n is the combined array size. Use this information to give an informative estimate for the worst-case number of comparisons performed by MergeSort3 on a subarray of size n. Your estimate should include a specific value for the coefficient on the dominant term. [Hint: how many comparisons overall are performed at each level of the recursion, and how many levels are there?]

   Give an analogous estimate for ordinary MergeSort. Your two estimates should make clear which algorithm is the better in this regard.

   Again, you need not give proofs. Informal reasoning is acceptable. *[6 marks]*

2. In this question we consider Dynamic Programming algorithms for *Knapsack problems*. We consider the scenario where we are given n items of sizes $w_1, \ldots, w_n \in \mathbb{N}_0$ respectively, as well as some capacity $C \in \mathbb{N}$. We are in the *binary knapsack* setting, where we may choose to omit item w_i or alternatively add a single copy of w_i to the knapsack, for every $i, 1 \leq i \leq n$ (subject to the total being below C).

   We will consider two variants of the problem - the *maximum knapsack* problem, and the *knapsack counting* problem.

   We first present the dynamic programming algorithm for the maximum knapsack, which operates on a $(n+1)\cdot(C+1)$ sized array kp, and which relies on the following recurrence:

$$kp(k+1, C') \;=\; \begin{cases} kp(k, C') & w_{k+1} > C' \\ \max\{kp(k, C'),\; w_{k+1} + kp(k, C' - w_{k+1})\} & \text{otherwise} \end{cases}$$

   The base cases are specified in lines 1., 2. of maxKnapsack below.

   Formally, $kp(k, C')$ is the greatest total weight $\leq C'$ that can be achieved using items $w_1, \ldots, w_k$.

   In the Algorithm below, we can see the use of the recurrence in lines 5-8.

   Algorithm maxKnapsack($w_1, \ldots, w_n$, C)

```
1. initialise row 0 of kp to "all-0s"
2. initialise column 0 of kp to "all-0s"
3. for (i ← 1 to n) do
4.     for (C' ← 1 to C) do
5.         if (w_i > C') then
6.             kp[i, C'] ← kp[i − 1, C']
7.         else
8.             kp[i, C'] ← max{kp[i − 1, C'], kp[i − 1, C' − w_i] + w_i}
9. return kp[n, C]
```

   (a) Give an $\Theta(\cdot)$ bound for the worst-case running-time of maxKnapsack in terms of n and C. *[4 marks]*

   (b) Suppose the input to maxKnapsack is the list $w_1 = 3, w_2 = 4, w_3 = 5$ and the capacity $C = 7$. Draw the 4×8-dimensional table kp that would be built by maxKnapsack. *[8 marks]*

   (c) For the example of (b), suppose we now consider the *knapsack counting* problem - how many subsets of weights (there are 2^3 options) are feasible with respect to capacity 7 (a subset is feasible as long as the total weight is no greater than 7). How many feasible knapsacks are there for this example? *[4 marks]*

(d) Suppose we define $feas(k, C')$ to mean the *number of feasible knapsacks*; the number of subsets of $\{w_1, \ldots, w_i\}$ whose sum is $\leq C'$.

Write a recurrence for $feas(k, C')$. [*6 marks*]

(e) Suppose that we were interested in generating a *random* knapsack solution from the pool of all feasible solutions, and that we take the approach of generating a uniform random subset of the weights, and returning this if it is feasible; otherwise trying again. Is this likely to be a fast method for generating a random feasible solution? [*3 marks*]

3. This question considers a variation on breadth-first search for undirected graphs.

The FIREFIGHTER model simulates a fire which spreads through a graph from nodes to adjacent nodes, with the exception of previously-defended nodes. We are given an undirected graph $G = (V, E)$, a specific root vertex $f \in V$ where the fire starts, and an ordered list of defence vertices $v_1, \ldots, v_k$ for some k. Recall that for any node v of a graph, the *neighbourhood* of v is the set of vertices adjacent to v, denoted $Nbd(v)$. The firefighting process operates as follows, with an evolving set of *burnt vertices* B_t, of *defended vertices* D_t, and a set of *threatened vertices* T_t, for time steps $t = 0, 1, \ldots$:

(a) At time $t = 0$, the set of burnt vertices is $B_0 = \{f\}$, and the set of threatened vertices is $T_0 = Nbd(f)$. D_0 is empty.

(b) At each subsequent time step $t > 0$, the process first:

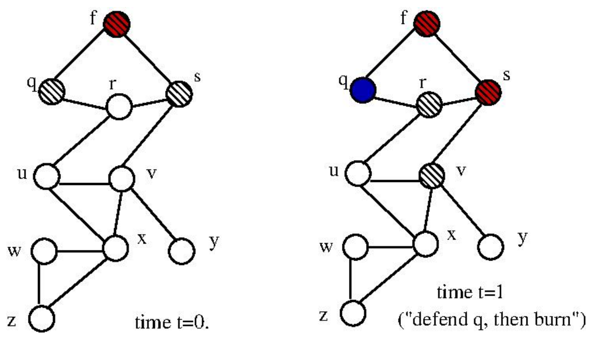
- Carries out the "defence" of v_t, and updates $D_t = D_{t-1} \cup \{v_t\}$. If $v_t \notin B_{t-1}$, it is protected, and will never catch fire.
- Every threatened node $u \in T_{t-1} \setminus \{v_t\}$ will "catch fire", and we update $B_t = B_{t-1} \cup (T_{t-1} \setminus \{v_t\})$.
- The updated set of threatened nodes is now

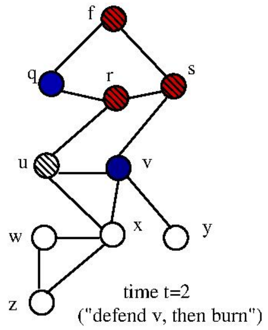
$$T_t = \bigcup_{u \in T_{t-1} \setminus \{v_t\}} Nbd(u) \setminus (B_t \cup D_t).$$

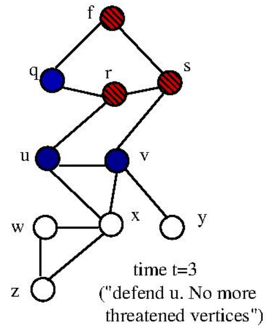
(c) The simulation terminates after step $t = k$ is completed.

The fire is said to have been *contained* by $v_1, \ldots, v_k$ if $T_k = \emptyset$.

Here is an example execution of the Firefighting process with defence list q, v, u:







(a) In the example simulation for the example graph above, we saw that a defence list of length 3 (q, v, u in that order) would "contain" the fire. *[6 marks]*

Is there an alternative (shorter) list of defences which would suffice to contain the fire on our example graph in fewer time steps? Justify your answer with respect to the graph.

(b) The process by which the fire spreads from f through a graph $G = (V, E)$ can be seen as a modified version of the bfsFromVertex(G, f) process.

Give pseudocode for a modified method called fireFighter($G = (V, E)$, f, $v_1, \ldots, v_k$) which simulates the Firefighter model with respect to the defence strategy $v_1, \ldots, v_k$, **return**ing TRUE if the fire has been *contained* after step k, and FALSE otherwise. You should also write a few sentences to justify why your method remains $O(n + m)$. *[13 marks]*

You can assume that the input graph $G = (V, E)$, is represented by an *Adjacency list* data structure, and that the defence list $v_1, \ldots, v_k$ is given as an ordered list.

[**Hint:** It may be helpful to enqueue the vertices tagged by the time step where they were discovered. It may also be helpful to define some new arrays to store details of the status of the vertices.]

(c) In our description above, we simulated the model with a pre-determined strategy (list of vertices to be defended, in that order). In practice, it is common to use a heuristic to determine the "next vertex to be defended", in advance of each step. A common heuristic is to defend the *threatened* vertex with the "highest effective degree" (effective meaning neighbours that are not already burned, not already defended). Give an example of a graph and starting f where this heuristic will fail to minimise the number of steps to contain the fire, and explain why it fails. *[6 marks]*