UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08026 INFORMATICS 2: INTRODUCTION TO ALGORITHMS AND DATA STRUCTURES

Friday 9th May 2025

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. **If all three questions in Part B are attempted, only questions 1 and 2 will be marked.**
3. **This is a NOTES NOT PERMITTED, CALCULATORS PERMITTED examination. Notes and other written or printed material MAY NOT BE CONSULTED during the examination. CALCULATORS MAY BE USED IN THIS EXAMINATION.**

Convener: D.K.Arvind
External Examiner: B.Konev

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

PART A

1. (a) Give the formal definition of the relation $f = O(g)$, where f, g are functions from $\mathbb{N}$ to the positive reals. *[2 marks]*

   (b) Consider the following four functions:

   $$f_1(n) = n^3, \quad f_2(n) = 3^n, \quad f_3(n) = 3^{n+3}, \quad f_4(n) = 1^2+2^2+\cdots+n^2.$$

   Draw a 4x4 square grid with rows and columns labelled 1,2,3,4, then tick the cell at row i, column j for exactly those i, j such that $f_i = O(f_j)$. *[4 marks]*

   (c) Prove rigorously from the definition of O that if $f(n) = O(g(n))$ then $2f(n/3) = O(g(n/3))$. (To reduce confusion, you should first translate these two statements into logical formulae using different variable names.) *[4 marks]*

2. Suppose we are implementing an open-address hash table for storing a set of positive integers, using an array A with cells indexed by $0, \ldots, n-1$, with the value 0 indicating an empty cell. We assume we are given a hash-probe function $\mathsf{h} : \mathbb{Z}^+ \times \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$.

   (a) Give pseudocode for a function insert(x) which attempts to insert a positive integer x into the table, returning True if this succeeds, False otherwise. *[3 marks]*

   (b) Give pseudocode for a function contains(x) which tests whether x is present in the table, returning True or False. Avoid unnecessary probing. *[4 marks]*

   (c) Write down simple Θ-estimates in terms of n for both the best- and worst-case runtimes for insert(x) in each of the following situations (you need not justify your answers):

      i. When the table is empty.

      ii. When the table is half full.

      iii. When the table is full. *[3 marks]*

3. The (Max) Heap data structure offers a number of valuable operations for an algorithm designer needing to store items in a way that allows for ready access to (and removal of) the "Max" element. These operations are Heap-Extract-Max, Max-Heap-Insert(k, e), Heap-Maximum and isEmpty. These operations have fast running-time on the binary Heap, with Heap-Extract-Max and Max-Heap-Insert(k, e) running in $O(\lg(n))$ time, and the other two running in $O(1)$ time.

   In this question we consider the quandary of an algorithm designer who would like to be able to *also* update the key value of an item e already in the Max Heap. In this setting, all cells are pair values of the form (k, e), k being the key relevant to the Heap property.

   We will consider two different cases.

   (a) In the first case, we are asked to design a method called increaseKey(k', o) which will *increase* the key value of the Heap cell referenced by o to k', assuming k' is greater than the current key value of that cell. The method must ensure that the (Max) Heap property is maintained. [*5 marks*]

   Note o is *an object reference* to some existing Heap cell (k, e).

   Give details of how increaseKey(o, k') can be implemented in $O(\lg(n))$ time on a Heap, with reference to known Heap methods if this is helpful.

   (b) In the second case, we are asked to design a method called increaseKey(k', e) which will *increase* the key value of item e to become k' (maintaining the Heap property), assuming $(\cdot, e)$ is a cell in the Heap, and k' is greater than the current key for e. [*5 marks*]

   In this case, e is an item rather than a reference, and we do not start with a direct reference to the cell of interest.

   Explain why increaseKey(k', e) will take $\Omega(n)$ time in the worst-case in this scenario, giving details of some Heap instances where this may occur.

4. In this question we consider a recurrence for a mysterious function which takes two parameters $i, j \in \mathbb{N}$

$$F[i,j] = \begin{cases} i+j & i=0, i=n \\ i+j & j=n \\ g(F[i-1,j], F[i-1,j+1], F[i,j+1], F[i+1,j+1]) & \text{otherwise} \end{cases}$$

where g is some given 4-parameter function.

Note that the dependence between $F[i,j]$ and other $F[\cdot,\cdot]$ cells is different to what we have previously seen in this course.

(a) Write pseudocode for a dynamic programming algorithm DP-Mystery to compute all entries of the table F of dimensions $n \times n$, taking care of base/boundary cases first, and then applying the recurrence to use the *already computed* cell-values to evaluate $F[i,j]$ for all $0 \leq i, j \leq n$. You may assume that you have some method/oracle to hand to evaluate g. *[5 marks]*

(b) Suppose that $g(x,y,z,w)$ can be evaluated in $\Theta(\max\{x,y,z,w\})$ time for all $x,y,z,w \in \mathbb{N}$, and that the value $g(x,y,z,w)$ will be at most $\max\{x,y,z,w\}$. Show that the worst-case running-time of DP-Mystery is $O(n^3)$, justifying your reasons. *[5 marks]*

Is it possible to also show a $\Omega(n^3)$ bound? If so, why? If not, why not?

5. Below is the LL(1) parse table for a simple grammar of mathematical expressions built from numerals via function application. The start symbol is Exp. We think of the terminal n as representing a lexical class of numerals, and f as representing a class of function symbols.

	n	f	(	)	,	$
Exp	n	f (Exp More)				
More				ϵ	, Exp More	ϵ

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

f (n , n)

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

PART B

1. In this question we consider two divide-conquer-combine algorithms for multiplying decimal numbers. In both algorithms, given numbers a, b each of n digits, where n is a power of 2, we split a into two halves a_0, a_1 (so that $a = 10^{n/2}a_0 + a_1$) and do the same for b. In the first algorithm, we recursively compute $a_0b_0, a_0b_1, a_1b_0, a_1b_1$, then use these to calculate ab as in two-digit long multiplication. In the second algorithm (known as *Karatsuba multiplication*), we achieve the same effect with a clever optimization. The simplified version we give here will not work on all inputs, but it illustrates the main idea.

   We represent numbers as Python-style lists of digits, most significant first. We are given functions add(A,B) and sub(A,B) which do addition and subtraction for lists of any length; shift(r,A) which left-shifts by r places (e.g. shift(2,[3,4]) = [3,4,0,0]); and digitMult(d,e) which returns d*e as a two-digit list.

   The following pseudocode is common to both algorithms:

```
        mult(A,B):
1          r = length(A)
2          if r = 1 then return digitMult(A[0],B[0])
3          else
4             A0 = A[0:r/2],   A1 = A[r/2:r]
5             B0 = B[0:r/2],   B1 = B[r/2:r]
6             C0 = mult(A0,B0),   C2 = mult(A1,B1)
              # compute C1 = (A0*B1)+(A1*B0)
11            G0 = shift(r,C0),   G1 = shift(r/2,C1)
12            return add(G0,add(G1,C2))
```

   In **Algorithms A** and **B**, the commented line is replaced respectively by:

```
A7    D0 = mult(A0,B1)          B7    E0 = add(A0,A1)
A8    D1 = mult(A1,B0)          B8    E1 = add(B0,B1)
A9    C1 = add(D0,D1)           B9    F = mult(E0,E1)
                                B10   C1 = sub(sub(F,C0),C2)
```

   (a) Step through the execution of Algorithm B with A = [5,4], B = [2,3]. It will suffice simply to list the values assigned to every variable in the course of the calculation (e.g. A0 = [5]), then to state the final result returned. For calls to mult on one-digit lists, you may immediately note the results without giving details of how these calls are executed. [*7 marks*]

   Our Algorithm B may hit trouble if E0 or E1 has r/2+1 digits rather than r/2. From here on, we simply assume the inputs are such that this never happens.

QUESTION CONTINUES ON NEXT PAGE

(b) For each labelled line of code above that does *not* perform a recursive call to mult, give a Θ-estimate for its worst-case runtime in terms of r. Use the line labels 1, A7, B8, ... rather than copying out the code. You should assume add, sub, shift, digitMult have the expected efficiency. For lines that perform more than one command, the total execution time will suffice. [*7 marks*]

(c) Let $T_A(n), T_B(n)$ be the worst-case runtime for Algorithms A and B respectively on n-digit inputs, where n is a power of 2. Write down asymptotic recurrence relations satisfied by T_A, T_B, with a brief explanation. [*4 marks*]

(d) Solve these relations to find Θ-estimates for T_A and T_B. Show your working. Logarithms whose value is not an integer may be left unevaluated. [*4 marks*]

(e) Which of the following are true? [*3 marks*]

$$T_B = o(T_A)\ , \qquad T_B = O(T_A)\ , \qquad T_B = \Theta(T_A)\ .$$

2. We consider Dijkstra's Algorithm for Single-Source Shortest Paths, and how we can achieve an efficient worst-case running time by using a (min) Priority Queue (augmented with the reduceKey operation).

Dijkstra's algorithm takes a (directed or undirected) graph $G = (V, E)$ with weighting $W : E \to \mathbb{R}^+$ and a special node $s \in V$ as input, and computes the "shortest path from s to v" $d[v]$ and "predecessor node to s" $\pi[v]$ for every $v \in V$. The operation of the algorithm is iterative, committing a single *fringe node* v^* to the set S at each step (and fixing the $d[v^*], \pi[v^*]$ values), until there are no more fringe vertices. At each step, Dijkstra's algorithm will choose the fringe node v^* to add to S via the fringe edge

$$(u, v^*) \leftarrow \arg \min_{u \in S, v^* \in V \setminus S} \{d[u] + w(u, v^*)\}.$$

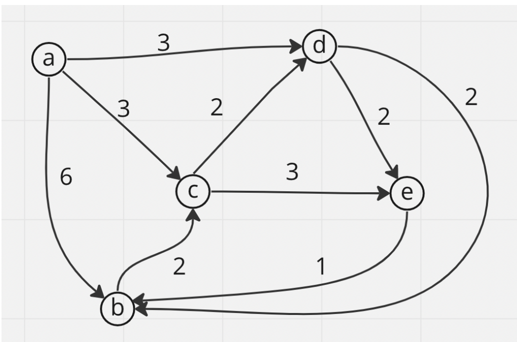
We let $n = |V|$ and $m = |E|$.

We assume that the Graph is stored in an *Adjacency list* throughout.

(a) Execute Dijkstra's Algorithm on the graph from start vertex a to build the arrays d and π. Break ties using lexicographic order. [*10 marks*]

Give details of the initialisation of d, π and S, and show how these are changed after each of the 4 steps after a vertex is added, with reference to new fringe edges after each step.

You do not need to draw the Heap/Priority Queue details.



(b) A simple approach to the implementation of Dijkstra's algorithm, considering all pairs $u \in S, v^* \in V \setminus S$ at each step to evaluate the "arg min", will lead to a running time of $\Theta(n \cdot m)$ in the worst-case. [*5 marks*]

Justify this $\Theta(n \cdot m)$ worst-case by explaining the details of a $\Theta(\cdot)$ bound for a single "arg min" calculation, and then aggregating the running times for these calls.

QUESTION CONTINUES ON NEXT PAGE

(c) The implementation that we covered in class achieves an overall running-time of $O((n+m)\lg(n))$ by exploiting the use of a (Min) Priority Queue which offers the operations $\mathsf{isEmpty}()$ and $\mathsf{minElement}()$ in $\Theta(1)$ time, and the operations $\mathsf{extractMin}()$, $\mathsf{insertItem}(d, v)$ and $\mathsf{reduceKey}(d', v)$ in $O(\lg(n))$ time.

i. Give details of how our better implementation uses the Priority Queue to manage the information about the collection of fringe vertices and their "best so far" route from s. You should explain the information that gets stored in the Priority Queue, and how it is updated at each iteration. *[5 marks]*

ii. Justify the $O((n+m)\lg(n))$ running-time for our better implementation, with reference to the details given in (i). *[5 marks]*

3. We consider the MIN VERTEX COVER optimisation problem:

Input: An undirected graph $G = (V, E)$.

Output: A minimum sized "cover" $C \subseteq V$ such that for every edge $e \in E, e = (u, v)$, at least one of $u \in C$, $v \in C$ holds.

We will also consider the *decision version* of VERTEX COVER, where we ask whether $G = (V, E)$ has a vertex cover C of cardinality $|C| \leq k$? We will initially consider the NP status of this decision problem.

(a) Show that the decision version of VERTEX COVER belongs to the class NP, by giving details of a polynomial-time algorithm to check a proposed solution C against the input graph G and input value k. [*5 marks*]

(b) Show that the decision version of VERTEX COVER is NP-complete, by reducing 3-SAT to VERTEX COVER. [*7 marks*]

(You may either reduce directly, or via some intermediate NP problem. If you use an intermediate problem, please give full details of both reductions.)

(c) We have seen the following Approximation Algorithm for MIN VERTEX COVER in our lectures:

Algorithm Approx-Vertex-Cover($G = (V, E)$)

$C' \leftarrow \emptyset$
$E' \leftarrow E$
while $E' \neq \emptyset$
  do take any edge $(u, v) \in E'$
    $C' \leftarrow C' \cup \{u, v\}$ // add **both** u and v to the cover
    Remove every edge g with u or v endpoint from E'
Print("There is a VC of size ", $|C'|$)

i. Show that Approx-Vertex-Cover can be implemented in $O(n + m)$ time (for $n = |V|$) when the graph is being stored in Adjacency List format, giving specific details of memory management. [*4 marks*]

ii. Argue that the cover C' computed by Approx-Vertex-Cover satisfies the inequality $|C'| \leq 2 \cdot |C|$, where C is an optimal Vertex Cover for G. [*4 marks*]

(d) It is well-known that for any graph $G = (V, E)$, that C is a Vertex Cover of G if and only if $V \setminus C$ is an independent set of G. It is also the case that the complement of a Minimum Vertex Cover of G will be a Maximum Independent set of G. [*5 marks*]

Consider the complement $I' \leftarrow V \setminus C'$ of the Vertex Cover C' computed by Approx-Vertex-Cover. Do we expect I' to satisfy a similar approximation ratio for Maximum Independent set (as C' did for Min Vertex Cover)? Justify your answer.