UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

# INFR08026 INFORMATICS 2: INTRODUCTION TO ALGORITHMS AND DATA STRUCTURES

**Wednesday 3rd May 2023**

**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. **This is a Closed Book examination.**
3. **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. [*3 marks*]

   (b) Which of the following three statements are true and which are false? You need not justify your answers.

$$n+1 \;=\; o(n+1000) \qquad n \;=\; o(1000n) \qquad n \;=\; o(n^{1000})$$

[*3 marks*]

   (c) Could there be functions $f, g$ such that both $f = o(g)$ and $g = o(f)$? Rigorously justify your answer with reference to the formal definition of $o$. [*4 marks*]

2. Consider the following pseudocode for a variant of InsertSort. The difference from the version in lectures is that the search for the correct insertion position j here starts from the *left* of the already sorted portion. Having found the correct j, we then shift any remaining elements of the sorted portion to the right, and finally insert the new element at position j.

```
NewInsertSort(A):
    for i = 1 to |A|−1
        x = A[i]
        j=0
        while j < i and A[j] < x
            j = j+1
        for k = i downto j+1
            A[k] = A[k−1]
        A[j] = x
```

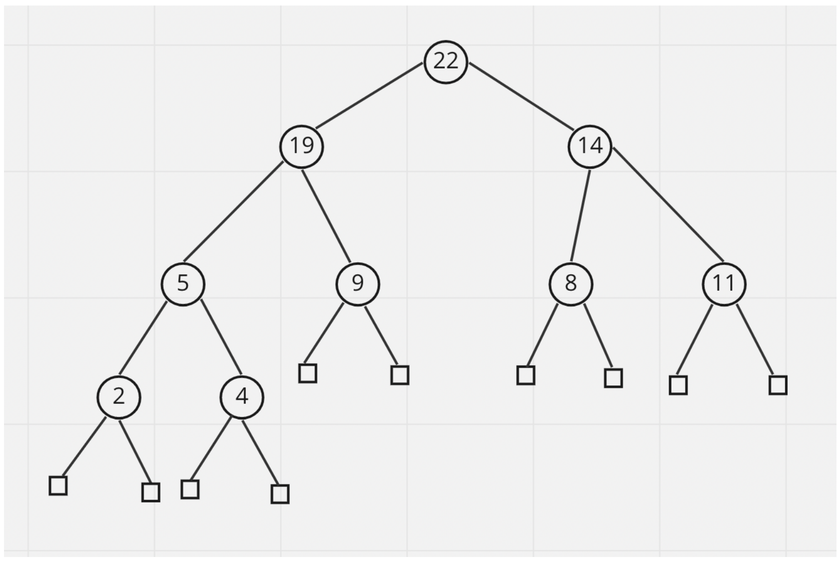
   (a) For an arbitrary $n$, describe an array A of size $n$ that results in the *best case* performance of this algorithm, if performance is measured by *number of comparisons* between items of A. Justify your answer, giving an exact formula for the number of comparisons performed on this input. [*3 marks*]

   (b) For the input array you have described in part (a), how many item comparisons would be performed by the standard version of InsertSort, where the search for the insertion position proceeds from the right? Give both an exact formula and a tight asymptotic estimate (using $\Theta$). Justify your answer. [*4 marks*]

   (c) For the same input array, what is the runtime of NewInsertSort as measured by *number of line executions*? An asymptotic estimate will suffice here. Informally justify your answer. [*3 marks*]

3. In this question we consider the (Max) Heap data structure and its representation as an array.

(a) Starting with the following (Max) Heap, compute the resulting Heap (either in tree or array format) after the following operations are performed in sequence, showing the result after each operation: Heap-Extract-Max, Max-Heap-Insert(6), Max-Heap-Insert(12), Heap-Extract-Max, Max-Heap-Insert(18): [*7 marks*]



| 22 | 19 | 14 | 5 | 9 | 8 | 11 | 2 | 4 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

(b) When working with an array realisation of a Heap, we can usually avoid moving between tree/array indices, and just work on the array. Assuming the array is indexed from 0, and considering the item indexed by $x$, give simple formulas for the left child of $x$ (Left(x)), the right child of $x$ (Right(x)), and the parent of $x$ (Parent(x)). [*3 marks*]

4. In this question we consider Dynamic Programming algorithms for the *Knapsack problem.* We are given $n$ items of sizes $w_1, \ldots, w_n \in \mathbb{N}$ 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 consider a collection of smaller subproblems, and formally, $kp(k, C')$ is defined as the greatest total weight $\leq C'$ that can be achieved using items $w_1, \ldots, w_k$.

   The following recurrence for maximum knapsack is used, together with a $(n+1) \cdot (C+1)$ sized array $kp$, to achieve the DP algorithm below:

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

   **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 ($i \leftarrow 1$ to $n$) do
4.     for ($C' \leftarrow 1$ to $C$) do
5.         if ($w_i > C'$) then x
6.             $kp[i, C'] \leftarrow kp[i-1, C']$
7.         else
8.             $kp[i, C'] \leftarrow \max\{kp[i-1, C'], kp[i-1, C' - w_i] + w_i\}$
9. return $kp[n, C]$
```

   (a) Give a $\Theta(\cdot)$ 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 $w_1 = 2, w_2 = 3, w_3 = 4$ and the capacity $C = 6$. Draw the $(4 \times 7)$-dimensional table $kp$ that would be built by maxKnapsack. [*6 marks*]

5. In this question we consider the proof of NP-completeness of the Independent Set problem on general undirected graphs $G = (V, E)$, which involves a reduction from 3-Sat.

   (a) The first step in demonstrating that a problem is NP-complete is to show that it actually *belongs* to NP. For Independent Set, this means that we must show that given a proposed independent set $I \subseteq V$, we can check whether $I$ *is* an independent set, and whether $|I| \geq k$, in time polynomial in the parameters $n = |V|, m = |E|$. [*5 marks*]

   Describe an algorithm which runs in $\Theta(n + m)$ time to verify whether set $I$ satisfies these two conditions with respect to $G$. You may assume that $V = \{0, ..., n-1\}$ and that $E$ is stored in adjacency list format, and that $I$ is presented as a sequence of nodes.

   (b) The second step to showing Independent Set is NP-complete is the *reduction* from 3-Sat given in our slides/lectures. Draw the "Independent set" graph of that reduction for the $\Phi$ below, and also state the total number of edges in the constructed graph, as well as the "$k$" (required size of the Independent set) for the instance. [*5 marks*]

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

**PART B**

1. In this question, we consider implementations of dictionaries with a *fixed* set of key-value pairs, so that insert and delete operations are not required — only lookup. We wish to represent some finite collection $I$ of *items* $x$, each with an associated key $x.\mathsf{key}$ and value $x.\mathsf{value}$, where no two items in $I$ have the same key. We assume our keys are *linearly ordered* by a relation $<$, so that for any keys $a$ and $b$, exactly one of $a < b$, $a = b$, $b < a$ is true.

   (a) One obvious approach is to store our items in an array $A$, sorted according to their keys, then use binary search to perform lookup. Give pseudocode for a recursive function $\mathsf{Lookup}(A, a, i, j)$ which returns the value (if any) associated with key $a$ by an item stored in the portion of $A$ from $A[i]$ to $A[j]$ *inclusive.* An error should be raised if the key is not present. When comparing keys $a$ and $b$, you should use the following construct.

```
case compare (a, b) :
      Lt => ...    # what happens if $a < b$
    | Eq => ...    # what happens if $a = b$
    | Gt => ...    # what happens if $b < a$
```

   [*8 marks*]

   (b) Write down an asymptotic recurrence relation for the worst-case runtime of $\mathsf{lookup}$ as a function of $n = j+1-i$ (the length of the relevant portion of $A$). Use this to obtain a $\Theta$-estimate for the worst-case runtime, showing your reasoning. What is the asymptotic *best-case* runtime of your operation? [*5 marks*]

   (c) A different approach is to use a bucket-style hash table, but one in which the individual buckets are not linked lists but *arrays* of items sorted by their key. That is, we have a hash table $H$ of some size $N$, each key $a$ will hash to some code $c = \#a < N$, and each $H[c]$ will be either $\mathsf{null}$ or a reference to a sorted array of items. Given this setup, write pseudocode for a function $\mathsf{HashLookup}(H, a)$ which returns the value (if any) associated with key $a$. Your pseudocode may call your $\mathsf{Lookup}$ operation from part (a). [*4 marks*]

   (d) We now suppose that we have chosen $N$ so that it is exceedingly unlikely that more than 15 items in our set $I$ will hash to the same key. (If they do, we can try again with a different hash function, reducing the probability of this scenario still further.) So let us now assume that all of the arrays $H[c]$ have length $\leq 15$.

   Assuming this, and assuming a constant-time hash operation, what is the asymptotic worst-case runtime for $\mathsf{HashLookup}$? More concretely, how many hashings and how many key comparisons may a lookup involve in the worst case? Justify your answers. [*3 marks*]

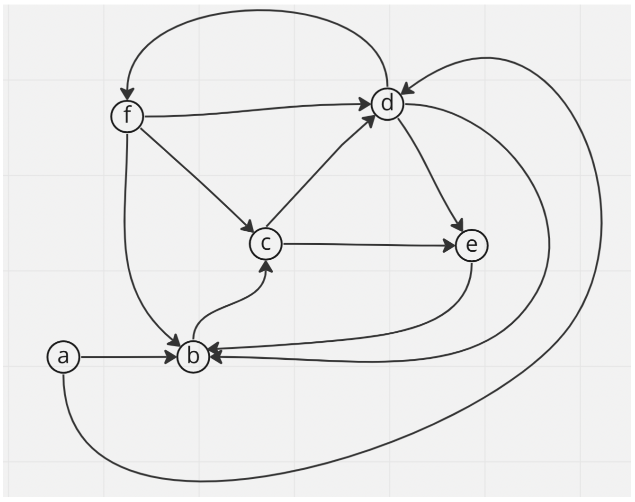
   (e) Suppose a hashing takes 7 times as long as a key comparison, and that all other steps take negligible time relative to these operations. How large must our set $I$ be for worst-case HashLookup to be faster than worst-case Lookup, according to your analysis? Explain your reasoning. [*5 marks*]

2. In this question we consider graph exploration algorithms and their application to solving other graph problems. We will assume that we are working with a (directed or undirected) graph $G = (V, E)$ and that $n = |V|, m = |E|$.

   (a) Explain the method of breadth-first search, by describing the methods bfs and bfsFromVertex. Include details of the data structure used to deliver the desired order of exploration, and the operations on this data structure during execution of bfs. [*5 marks*]

   (b) Justify the $\Theta(n+m)$ running-time of bfs, making sure to cover the $\Omega(n+m)$ as well as the $O(n+m)$. [*5 marks*]

   (c) Execute bfs($a$) on the graph below, drawing the breadth-first tree as well as showing the linear order of the explored vertices. The vertices should be explored in lexicographic order, and this should also be the order of considering outgoing nodes from the current vertex $v$. [*7 marks*]



   (d) Suppose now that $G$ is *undirected* and consider the question of determining whether the graph is *bipartite* - that is, whether $V$ can be partitioned into two subsets $V = V_1 \uplus V_2$ such that every edge $e = (u, v)$ has one endpoint in $V_1$ and one endpoint in $V_2$.

      i. Give an algorithm that runs in $\Theta(n + m)$ time to solve this problem. [*4 marks*]

      ii. Justify the correctness of your proposed algorithm. [*4 marks*]

   [**Hint:** You may use the fact that a graph is bipartite if and only if it has no odd-length cycles (without proof).]

3. Suppose $\mathcal{L}$ is a context-free grammar containing *no $\epsilon$-rules.*

(a) Briefly explain, in order, the three stages one would go through to convert $\mathcal{L}$ to an equivalent grammar in Chomsky Normal Form. (Note that in the absence of $\epsilon$-rules, two of the usual five stages may be skipped.) [*6 marks*]

(b) Now consider the following grammar, with the four terminals $\mathsf{a}$, :, (, ), two non-terminals $\mathsf{S}$, $\mathsf{X}$, and start symbol $\mathsf{S}$.

$$\mathsf{S} \longrightarrow \mathsf{X} \mid \mathsf{X} : \mathsf{S} \qquad \mathsf{X} \longrightarrow \mathsf{a} \mid ( \mathsf{X} )$$

Apply your procedure to convert this to Chomsky Normal Form, explicitly showing the grammar obtained at each of the three stages. [*6 marks*]

(c) Using your Chomsky Normal Form grammar, construct a CYK parse chart for the phrase:

$$( \mathsf{a} )$$

For each non-diagonal entry in the chart, add backtrace pointers to show how that entry was obtained. [*4 marks*]

(d) Let us define the *size* of a production $X \to \alpha$ to be the length of its right-hand side $\alpha$ (e.g. $\mathsf{X} \to \mathsf{Y} = \mathsf{Y}$ would have size 3). We then define the size of an entire grammar to be the sum of the sizes of all its productions.

For each of the stages $s = 1, 2, 3$ you described in part (a), give a $\Theta$-estimate for the worst-case runtime of stage $s$ as a function of $n$, the size of the original grammar $\mathcal{L}$. Justify your answers. [*9 marks*]