University of Glasgow

**Wednesday 4 May 2022**
**14:00-15:30 BST**
**Duration: 1 hour 30 minutes**
**Additional time: 30 minutes**
**Timed exam – fixed start time**

**DEGREES OF MSc, MSci, MEng, BEng, BSc, MA and MA (Social Sciences)**

# ALGORITHMICS II (H)
# COMPSCI4003

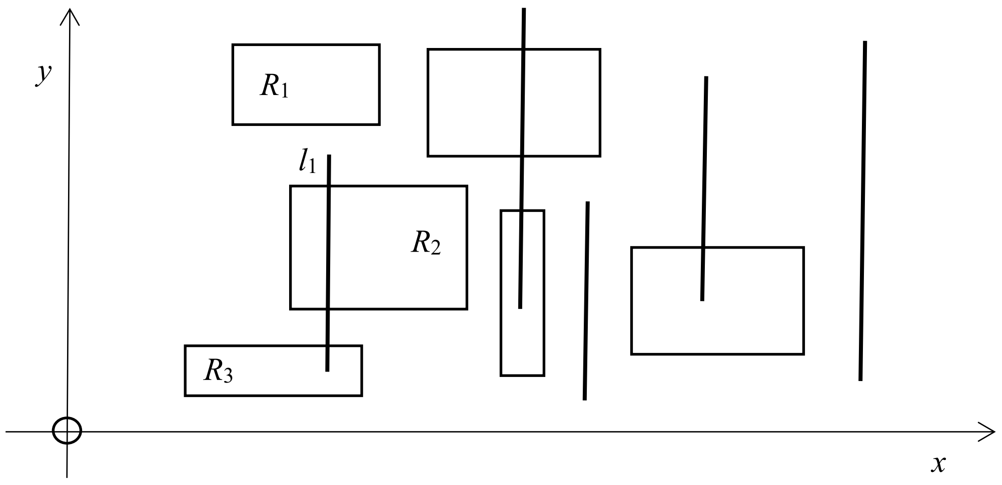
**(Answer all 4 questions)**

**This examination paper is an open book, online assessment and is worth a total of 60 marks**

## 1. Geometric algorithms [15 marks]

Let $R$ be a set of rectangles in the plane, whose sides are parallel to the $x$- and $y$-axes. Let $r=|R|$. Assume that no two rectangles in $R$ have a non-empty area of intersection and no two rectangles in $R$ coincide along a boundary. Also let $V$ be a set of vertical line segments in the plane (each parallel to the $y$-axis). Let $v=|V|$. Again assume that no two line segments in $V$ intersect one another, and no two line segments in $V$ share a common endpoint.

A vertical line segment in $V$ with endpoints $(x,y_1)$, $(x,y_2)$, where $y_1 \leq y_2$, is said to *fully intersect* a rectangle in $R$ with lower-left-hand corner coordinates $(x_3,y_3)$ and upper-right-hand corner coordinates $(x_4,y_4)$ if $y_1 \leq y_3 \leq y_4 \leq y_2$. For example, in the diagram below, the leftmost vertical line segment labelled $l_1$ fully intersects $R_2$, but $l_1$ does not fully intersect $R_1$ or $R_3$.



Let $n = r + v$ be the total number of rectangles and vertical line segments. Give an $O(n \log n + p)$ algorithm to compute the total number of rectangles in $R$ that fully intersect the vertical line segments in $V$, where $p$ is the total number of intersections between the vertical line segments and the horizontal line segments that make up the rectangles. You need not justify the complexity of your algorithm. Assume that the input is given as follows:

```
r // number of rectangles
v // number of vertical line segments
(x1,y1), (x2,y2) // lower-left hand and upper-right-hand corner coordinates of rectangle 1
(x1,y1), (x2,y2) // lower-left hand and upper-right-hand corner coordinates of rectangle 2
…
(x1,y1), (x2,y2) // lower-left hand and upper-right-hand corner coordinates of rectangle r
(x,y1), (x,y2) // lower and upper endpoint coordinates of vertical line segment 1
(x,y1), (x,y2) // lower and upper endpoint coordinates of vertical line segment 2
…
(x,y1), (x,y2) // lower and upper endpoint coordinates of vertical line segment v
```

Note: you need not describe in detail any standard algorithms on tree structures that you might use.

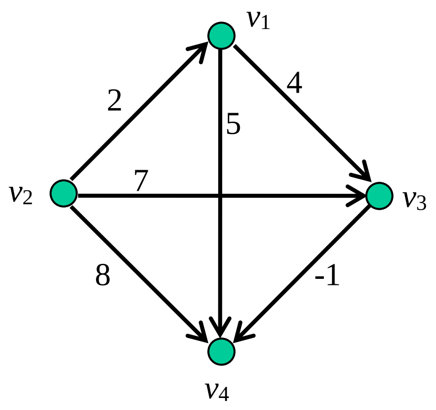
[15]

## 2. Graph and matching algorithms [16 marks]

Let $G=(V,E)$ be a weighted directed graph, where $V=\{v_1,v_2,\ldots,v_n\}$ and the weight of the edge $(v_i,v_j)$ is an integer denoted by $wt(v_i,v_j)$. We assume that the edge weights could be negative, but that $G$ does not contain a negative weight cycle.

For $k \geq 0$, let $D_k$ be an $n\times n$ distance matrix, where $D_k(i,j)$ contains the shortest path distance from $v_i$ to $v_j$ considering only those paths whose intermediate vertices (if any) belong to the set $\{v_1, v_2, \ldots, v_k\}$.

**(a)** Let $G$ be the weighted directed graph shown below, where the weight of each edge is as shown. Use the Floyd-Warshall algorithm to compute $D_0, D_1, \ldots, D_n$ for $G$. Ensure that your answer shows explicitly the contents of each of these matrices.



[8]

The Floyd-Warshall algorithm can also be used to construct an actual shortest path between each pair of vertices in $G$. This can be done by computing, for $k \geq 0$, an $n\times n$ predecessor matrix $\Pi_k$, where $\Pi_k(i,j)$ contains the predecessor vertex of $v_j$ on a shortest path from $v_i$ to $v_j$ considering only those paths whose intermediate vertices (if any) belong to the set $\{v_1, v_2, \ldots, v_k\}$. The predecessor matrices are defined as follows:

For each $i$ and $j$ ($1 \leq i \leq n$, $1 \leq j \leq n$), $\Pi_0(i,j)=i$ if $i\neq j$ and $(v_i,v_j)\in E$, otherwise $\Pi_0(i,j)=$**null**.

For each $i$, $j$ and $k$ ($1 \leq i \leq n$, $1 \leq j \leq n$, $1 \leq k \leq n$), $\Pi_k(i,j)= \Pi_{k-1}(i,j)$ if $D_k(i,j)=D_{k-1}(i,j)$, otherwise $\Pi_k(i,j)=\Pi_{k-1}(k,j)$.

**(b)** Explain, using pseudocode or otherwise, how to interpret the $\Pi_k$ matrices in order to compute an actual shortest path from $v_i$ to $v_j$ or report that no path from $v_i$ to $v_j$ exists, for any two vertices $v_i$, $v_j$ in $V$.

[3]

**(c)** Relative to the weighted directed graph $G$ shown above, compute $\Pi_0, \Pi_1, \ldots, \Pi_n$ for $G$. Ensure that your answer shows explicitly the contents of each of these matrices.

[5]

## 3. String and text algorithms [16 marks]

Let $S$ be a string of length $n$ over an alphabet $\Sigma$. The last character of $S$ may possibly appear elsewhere in $S$.

**(a)** Why, in general, is it necessary to append a unique termination symbol (usually denoted '\$') to $S$ before constructing the suffix tree for $S$ (which will store all suffixes of $S\$$)?

[2]

Let $T$ be the suffix tree for $S$ (which will store all suffixes of $S\$$). It is known that $T$ can be constructed in $O(n)$ time.

$S$ has a *repeated substring with non-overlapping embeddings* if there is a substring $X$ of $S$ such that $X = S(i \,..\, i + k - 1)$ and $X = S(j \,..\, j + k - 1)$ for some $i$, $j$ and $k$ where $k=|X|$, $1 \le i < j \le n$ and $i + k - 1 < j$.

$X$ is a *longest repeated substring of S with non-overlapping embeddings* if (i) $X$ is a longest repeated substring of $S$, and (ii) subject to (i), $X$ is a repeated substring of $S$ with non-overlapping embeddings.

For example, `abc` is a longest repeated substring of `abcabc` with non-overlapping embeddings. Such a substring may not exist: for example `ababa` has a unique longest repeated substring, namely `aba`, but it does not have non-overlapping embeddings in `ababa`.

**(b)** **(i)** Give an $O(n)$ algorithm to find a longest repeated substring of $S$ with non-overlapping embeddings or report that none exists.

[6]

**(ii)** Justify briefly the time complexity of your algorithm from Part (b)(i).

[2]

$X$ is a *repeated substring of S with non-overlapping embeddings of maximum length* if (i) $X$ is a repeated substring of $S$ with non-overlapping embeddings, and (ii) subject to (i), $|X|$ is maximum.

For example, `ba` is a repeated substring of `ababa` with non-overlapping embeddings of maximum length, whilst `aaaa` is a repeated substring of `aaaaaaaa` with non-overlapping embeddings of maximum length. Such a substring will always exist as long as $S$ has at least one repeated character, otherwise $S$ will not have a repeated substring with non-overlapping embeddings (as there is no repeated substring).

**(c)** Describe an $O(n)$ algorithm to find a repeated substring of $S$ with non-overlapping embeddings of maximum length or report that none exists. You need not justify the time complexity of your algorithm.

*Hint*: calculate, for each node $v$ in $T$, the minimum and maximum suffix numbers of a descendant leaf node of $v$, denoted $\min_v$ and $\max_v$, respectively.

[6]

**4. Algorithms for hard problems** **[13 marks]**

**(a)** The Knapsack Problem (KP) can be defined as follows:

*Instance*: $n$ items, where item $i$ has a weight $w_i$ and a profit $p_i$ ($1 \leq i \leq n$); knapsack capacity $C$.

*Solution*: maximum $k$ such that there exists a subset $S$ of $\{1,\ldots,n\}$ for which $\Sigma_{i \in S}\, w_i \leq C$ and $\Sigma_{i \in S}\, p_i = k$.

There is a dynamic programming algorithm for KP that runs in O($nC$) time. Now let $I$ be the following instance of KP with 4 items, whose weights and profits are as follows:

| | item | | | |
|---|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| weight | 3 | 1 | 5 | 2 |
| profit | 10 | 6 | 18 | 9 |

The knapsack capacity is 6.

**(i)** Illustrate the dynamic programming algorithm for KP by computing the dynamic programming table that would be constructed during the algorithm's execution as applied to the above example KP instance.

[5]

**(ii)** Explain why the O($nC$) dynamic programming algorithm is not a polynomial-time algorithm for KP in general.

[2]

**(b)** The Minimum Bin Packing problem (MBP) can be defined as follows:

*Instance*: $n$ items, where item $i$ has a size $x_i$; bin capacity $B$.

*Solution*: minimum $k$ such that there exists a packing of all items into $k$ bins such that no bin capacity is exceeded.

The following algorithm, called First Fit Decreasing (FFD) is a 3/2-approximation algorithm for MBP:

```
sort items into non-increasing order of size;
for (item x : items) // in sorted order, largest first
   if (no existing bin can accommodate x)
      place x in a new bin;
   else
      place x in the first bin that can accommodate it;
```

Give an example instance of MBP and an execution of FFD on this instance to show that FFD may not produce an optimal solution.

*Hint*: consider some quantities of items of sizes 2 and 3, one item of size 5, and a bin capacity of 9 (note that you are not obliged to use these values).

[6]