Final Examination
(3 hours)

Write your roll number in the space provided on the top of each page. *Write your solutions clearly in the space provided after each problem. If the space provided is insufficient, please write your solution on additional sheets, and* ***clearly state*** *in the main paper exactly where your solution appears. You may also use additional sheets for working out your solutions; attach all additional sheets at the end of the question paper.* **Attempt all problems.**

Name and Roll Number: _____

Problem	Points	Score
1	20	
2	20	
3	20	
4	20	
5	20	
Total:	100	

1. (a) (BFS and shortest paths) Consider a directed acylic graph G with a source vertex s, where the edges have unit costs. We say that a tree T with root s is a shortest path tree for G if for every vertex v, the unique path from s to v in T is a shortest s–v path in G. Show a directed acylic graph G in which s is a source, and a shortest path tree T for G with root s, such that no matter how the adjacency lists of the vertices of G are ordered, T will NOT be the BFS tree if a *breadth-first search* is started at s. 4

(b) (DFS) Recall that the height of a rooted tree is the length of the longest path from the root to a leaf. Let G be a connected undirected graph with a special vertex s. State if the following statement is *true* or *false.* 4

> If a *depth-first search* is started at s, then the tree that is produced is the spanning tree in G with greatest possible height, that is, of all spanning trees of G with s as root, this tree has the greatest height.

If you answer *true,* provide a short proof; if you answer *false,* show a graph and (separately) a depth-first search tree for it that violates the statement.

(c) (Nested loops) Consider the following code. Assume that the list `A` initially contains a permutation of $\{0, 1, 2, \ldots, n-1\}$. [6]

```
def rearrange(A):
    n = len(A)     # length of A
    for i in range(n):          # range(n) = [0, 1, ..., n-1]
        j = i
        while A[j] != j:
            # the following two lines swap A[j] and A[A[j]]
            X, Y = A[j], A[A[j]]
            (A[j], A[X]) = Y, X
```

Suppose `A` $= [1, 2, 0, 3, 5, 4]$ initially. What are the contents of `A` after it is processed by calling `rearrange(A)`?

Answer: _____

If `rearrange` is called on an n-element list `A` containing a permutation of $\{0, 1, \ldots, n-1\}$, how long does it take? Ignore constant factors, but write the tightest bound you can in terms of n; write ∞ if the algorithm does not stop for certain inputs.

Answer: $O($ _____ $)$

(d) (Search) Suppose $A[0], A[1], \ldots, A[2k-1]$ is a list of $2k$ distinct integers, initially sorted in increasing order. Let P be a list of k disjoint pairs of the form (i, j), where $0 \le i < j \le 2k-1$; that is, we may regard P as a perfect matching of the elements of $\{0, 1, 2, \ldots, 2k-1\}$. Suppose the following action is performed on the array A: [6]

```
for (i,j) in P:
    (A[i], A[j]) = (A[j], A[i])   \\ i.e., swap A[i] and A[j].
```

Now, given an integer x, describe how can one efficiently determine if x is in this resulting list A, and determine where x it is stored. (**Note:** We no longer have access to the pairs in P.)

2. (Divide and conquer) Let $A = A[0], A[1], \ldots, A[n-1]$ be a list of n integers (assume $n \geq 1$). We wish to process A and produce another list B with n integers such that

$$B[i] = A[0] \star A[1] \star \cdots \star A[i-1] \star A[i+1] \star \cdots \star A[n-1],$$

that is, $B[i]$ is the product of all elements of A except $A[i]$. We call the list B, the list of *partial products*.

(a) The function `partial_products` below is meant to take an array `A` and returns a pair (`B`, `p`), where `B` is the list of partial products and `p` is the product of all elements in `A`. Provide the missing parts in the code below (at the *five* places marked by ???). [10]

```
def partial_products(A):

    if len(A) == 1:           # base case
        return(1, ______)        # ???
    else:
        ell = len(A)//2       # ell = integer part of len(A)/2
        A1 = A[0:ell]         # A1 = A[0]...A[ell-1]
        A2 = A[ell:]          # A2 = A[ell]...A[len-1]

        (B1, p1) = _____________          # ???

        (B2, p2) = _____________          # ???

        for i in range(len(B1)):

            B1[i] *= ___________________   # ???

        for i in range(len(B2)):

            B2[i] *= ___________________   # ???

        return (B1 + B2, p1*p2)
                              # B1 + B2 is the concatenation of B1 and B2
```

(b) Write and solve a recurrence for the running time of the algorithm. Assume that the concatenation `B1` + `B2` takes time proportional to the sum of the lengths of `B1` and `B2`. [10]

3. (Dynamic programming) Given a set of n closed intervals, $I_1, I_2, \ldots, I_n$ in $\mathbb{R}$, with weights $w_1, w_2, \ldots, w_n$, where the interval I_j has the form $[a_j, b_j]$ $(a_j < b_j)$. Assume that the $2n$ end points $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n$ are distinct; furthermore, assume that the intervals have been provided in the increasing order of their right end points, that is, $b_1 < b_2 < \cdots < b_n$. For the following solutions using dynamic programming, do not worry much about the base cases (but state which values constitute the base cases, and which values are to be computed using the recurrence); it will be sufficient if you write the recurrences accurately. Explain what your doing; don't just write some code.

   (a) We wish to find a subset of intervals with maximum total weight, such that every point in $\mathbb{R}$ is in at most one interval in the subset. Provide an efficient solution for this problem. [8]

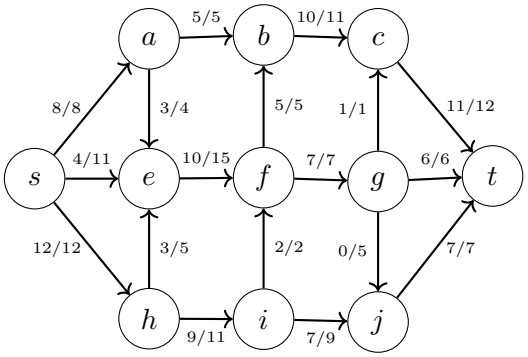
   Hint: For $j = 1, 2, \ldots$, successively compute the maximum weight subset of disjoint intervals among $I_1, I_2, \ldots, I_j$; let $M[j]$ denote the weight of the maximum weight subset of $I_1, I_2, \ldots, I_j$, and show how you will compute $M[j]$ if $M[1], \ldots, M[j-1]$ are known. Note that your algorithm must not only compute the optimum weight but also list the elements of the subset of disjoint intervals that provides that weight. For full credit, the algorithm should run in time $O(n \log n)$. You will need to find an efficient way to determine the largest j such that $b_j < a$ for a given $a \in \mathbb{R}$.

(b) Now, we wish to find a subset of intervals with maximum total weight, such that every point in $\mathbb{R}$ is in at most TWO intervals in the subset. 12

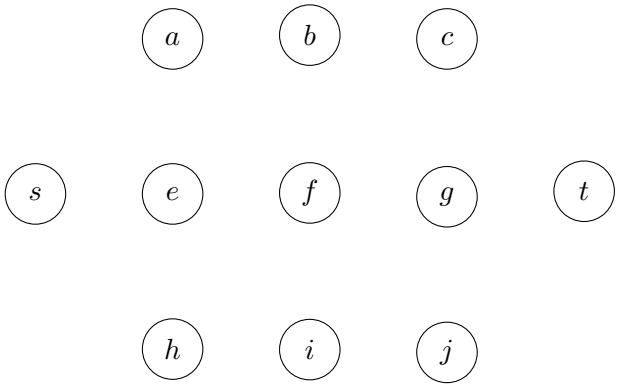
Hint: Define $M_2[i, j]$ $(i \leq j)$ to be the maximum total weight of a subset of intervals among $I_1, I_2, \ldots, I_j$, where all points in (b_i, ∞) are covered at most once and all points in $(-\infty, b_i]$ are covered at most twice. Consider two cases based on whether or not $a_j < b_i$. In each case, state which previously computed values will be used to determine $M_2[i, j]$. For full credit, present an algorithm with running time $O(n^2 \log n)$.

4. We are given a directed network $G = (V, E)$ ($|V| = n$, $|E| = m$) with capacities $(c_e : e \in E)$ and flows $(f_e : e \in E)$.

   (a) In the graph shown below, the label on edge e has the form f_e/c_e. [12]



   (i) Draw the residual network for G wrt the given flow.



   (ii) State why the given flow is optimal?

   (iii) List *all* the edges e of the above graph such that if we increase c_e (keeping the other capacities untouched), the max-flow for the network will increase. (Only list the edges; you don't have to show the flows with greater value explicitly.)

(b) For a general network $G(V, E)$ with a given s-t maximum flow, describe an algorithm that in time $O(m + n)$ lists all edges $e \in E$, such that if we increase c_e, the max-flow for the network will increase. [8]

5. (a) Consider the 3-colouring problem for undirected graphs. Suppose there is a decision algorithm A to determine if a graph is 3-colourable, that is, $A(G)$ is 1 if G is 3-colorable, and 0 otherwise. We wish to use A to design a method to *find* the 3-colouring for G if one exists. Consider the following method; assume that G is 3-colourable. [10]

Step 1: Let v_R, v_G, v_B be three new vertices. Let H_0 be the graph obtained from G by adding a triangle on these vertices. That is, let $V(H_0) = V(H) \cup \{v_R, v_G, v_B\}$, and $E(H_0) = E(G) \cup \{\{v_R, v_G\}, \{v_G, v_B\}, \{v_R, v_B\}\}$.

Step 2: *We will add edges between the three new vertices and each vertex v of G, and use the algorithm A to determine the colour $\chi[v]$ for the vertex v.* Initially, set $k = 0$; k denotes the number of vertices of G that have been processed.

Step 3: **(*)** For $v \in V(G)$:

(i) Let $H_{\text{tmp}} = (V(H_0), E(H_k \cup \{\{v, v_G\}, \{v, v_B\}\})$;
if $A(H_{\text{tmp}}) = 1$, then set $\chi[v] =$ `Red`, $H_{k+1} = H_{\text{tmp}}$, and go to **(*)**;

(ii) Let $H_{\text{tmp}} = (V(H_0), E(H_k \cup \{\{v, v_R\}, \{v, v_B\}\})$;
if $A(H_{\text{tmp}}) = 1$, then set $\chi[v] =$ `Green`, $H_{k+1} = H_{\text{tmp}}$, and go to **(*)**;

(iii) Set $\chi[v] =$ `Blue`, $H_{k+1} = H_{\text{tmp}}$, and go to **(*)**;

Continued on the next page ...

Formulate a suitable induction hypothesis about the 3-colourability of the graph H_k, for $k = 0, 1, ..., n$, and show that the algorithm is correct. Assume that the A takes time $t_A(m, n)$ in the worst case for graphs with n vertices and m edges. Show an upper bound on the running time of the above algorithm to find a 3-colouring for a graph with n vertices and m edges in terms of $t_A(\cdot, \cdot)$, m and n.

(b) Show directly (without appealing to the Cook-Levin theorem) a polynomial time reduction from the graph 3-colouring problem (3COL) to the problem 3SAT. That is, describe a polynomial time transformation: 10

$$T : G \mapsto \Phi,$$

that maps undirected graphs to 3CNF Boolean expressions. State the following clearly: (i) the variables that appear in $T(G)$, and what they stand for; (ii) the clauses of $T(G)$; (iii) how $T(G)$ is obtained efficiently from G; (iv) why $T(G)$ is satisfiable if G is 3-colourable; (v) why G is 3-colourable if $T(G)$ is satisfiable.

Hint: For each vertex v, define three Boolean variables R_v, G_v, B_v, which are supposed to indicate if the vertex v is assigned *red*, *green* or *blue*.