Midterm Examination

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. You may use additional sheets for working out your solutions; attach such sheets at the end of the question paper.* **Attempt all problems.**

Name and Roll Number: ______________________________

Problem	Points	Score
1	7	
2	4	
3	4	
4	15	
5	15	
6	15	
7	15	
Total:	75	

1. Consider the following code.

```
def brian(n):
    count = 0
    while ( n != 0 )
        n = n & ( n-1 )
        count = count + 1

    return count
```

Here `n` is meant to be an unsigned integer. The operator `&` considers its arguments in binary and computes their bit wise AND. For example, 22 `&` 15 gives 6, because the binary (say 8-bit) representation of 22 is 00010110 and the binary representation of 15 is 00001111, and the bit-wise AND of these binary strings is 00000110, which is the representation of 6.

(a) What does brian(22) return? $\boxed{3}$

Answer: 3

(b) Suppose $n = \sum_{i=0}^{k} b_i 2^i$, where each $b_i \in \{0,1\}$. What does brian(n) return? $\boxed{4}$

Answer: $\sum_{i=0}^{k} b_i$

2. Assume that two sorted lists are given: $\boxed{4}$

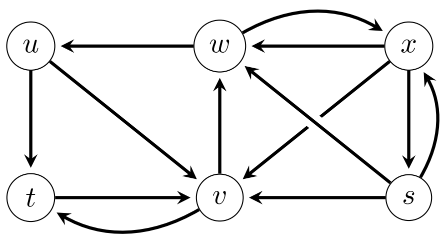
$$\begin{aligned} L_1 &= a_1 < a_2 < \cdots < a_{17} \\ L_2 &= b_1 < b_2 < \cdots < b_{15} \end{aligned}$$

What algorithm will you use to determine if the two lists have an element in common, if you are only allowed comparisons between elements of the two lists? (We wish to perform the minimum number of comparisons in the worst case, where each comparison has the form x <= y or x == y. In a couple of sentences sketch an idea that requires fewer than 65 comparisons.)

Solution: Merge the two lists by comparing the elements of the two lists using the comparisons of the form $a_i \leq b_j$. This requires at most $17 + 15 - 1 = 31$ comparisons. Now, we compare each b_j with any a_i that appears adjacent to it in the merged list. This way, we will be able to find all equal pairs, with an addition 30 comparisons, that is, 61 comparisons in all. [In fact, we may notice that whenever a b_j immediately precedes an a_i in the merged list, then $b_j < a_i$. So we need only check for equality between a b_j to an a_i that immediately precedes it. So, we need no more than $31 + 15 = 46$ comparisons in all.]

The number of comparisons performed in the worst case is 46.

3. Perform a breadth-first search on the directed graph given below, starting from vertex s, and exploring the list of neighbours of vertices in the alphabetical order: s, t, u, v, w, x. Draw the BFS tree, with the root on top, and the children of each vertex written from left to right in the order in which they were visited. $\boxed{4}$



Solution: In the notation, tree = (root, list of subtrees), the BFS tree is

$$(s, [(v, [(t, [])]), (w, [(u, [])]), (x, [])]).$$

4. Suppose A is an $n \times n$ array of integers, indexed by $\{0, 1 \ldots, n-1\} \times \{0, 1, \ldots, n-1\}$. We would like to find the minimum of these n^2 integers. We are told that the array has the following property. $\boxed{15}$

   Every column has a unique minimum. Let $r(j)$ be the row number of the minimum element in column j. Then, for $i = 1, \ldots, n-1$, we have $r(j) \geq r(j-1)$. E.g.,

$$\begin{bmatrix} \boxed{0} & \boxed{14} & 19 & 25 & 25 \\ 13 & 19 & 24 & 18 & 18 \\ 12 & 15 & 17 & 12 & 14 \\ 20 & 18 & \boxed{16} & 16 & 18 \\ 5 & 27 & 18 & \boxed{11} & \boxed{10} \end{bmatrix}$$

   The boxed entries are the minima, and as we scan the columns from left to right, the box in the next column is not higher than the box in the previous column.

   Describe an efficient algorithm to find the minimum element using binary comparisons between elements of the array. Using a recurrence (or otherwise) derive an upper bound on the number of comparisons made by your algorithm.

Solution: Let us denote the number of comparisons needed for an array with r rows and c columns by $T(r, c)$, assuming that the property assumption the minima holds for array. Compute ℓ be the index of the middle column and find its minimum (using $(r-1)$ comparisons)—say it is row i. The minima to the left of column ℓ appear in rows $1, \ldots, i$ and those to its right appear in rows $i, \ldots, r$. Solve the two subproblems separately and combine their answer with two more comparisons. We get

$$T(r, c) \leq \max_i T(i, c/2) + T(r-i+1, c/2) + r - 1 + 2.$$

Expanding the recurrence a few times, allows us to conclude that the following bound holds:

$$T(r, c) \leq (r+2) + (r+4) + (r+8) + \cdots + (r + 2^{1+\log c}).$$

Thus, $T(r, c) = O(r \log c + c)$.

5. Let G be an undirected graph, which is represented using a vertex list V and an adjacency list Adj. Suppose a dfs has been performed on G, and the following values have been recorded for each vertex v: $\mathsf{v} \cdot \mathsf{dfslevel}$, which records the dfs level of the vertex (each root has level 0), $\mathsf{v} \cdot \mathsf{parent}$, which gives the parent of vertex v, and $\mathsf{v} \cdot \mathsf{lownumber}$, which records the smallest dfs level of a vertex reachable from v by taking some (maybe zero) *tree edges* (directed from a parent to the child) followed by at most one *back edge*. Now, recall that an edge of G is called a cut edge if its deletion increases the number of connected components.

   (a) State *true* or *false*: every *cut edge* of the graph is a *tree edge*. $\boxed{1}$

   Answer: true

   (b) Assume that the above dfs information is available already. What algorithm would you use to determine all the *cut edges* in G? $\boxed{10}$

   Solution: We consider vertices $\mathsf{v} \in \mathsf{V}$, and whenever the condition $\mathsf{v} \cdot \mathsf{parent} \neq \mathsf{Nil}$ AND $\mathsf{v} \cdot \mathsf{lownumber} > \mathsf{v} \cdot \mathsf{parent} \cdot \mathsf{dfsnumber}$ holds, we declare the edge $(\mathsf{v} \cdot \mathsf{parent}, \mathsf{v})$ a cut edge. This can be implemented as follows.

```
list_of_cut_edges = []
for v in V:
    if v.parent != None:
        if v.parent.dfslevel < v.lownumber:
           list_of_cut_edges.append((v.parent,v))
```

   (c) How long does your algorithm take? Assume that the graph has n vertices and m edges. Briefly justify your estimate. $\boxed{4}$

   Solution: The above method goes through the list of vertices once and for each vertex in the list does a constant amount of work. So the running time is $O(n)$.

6. Suppose you are given a directed graph G (again as $(\mathsf{V}, \mathsf{Adj})$), where vertices represent locations in Bengaluru and the directed edges represent connectivity by buses. We wish to buy ice for a post midterm party.[1] We are given a list P, indexed by vertices, where $P[v]$ gives the price/kg of ice at vertex v; if ice is not available at v, then $P[v] = \infty$.

   (a) Design an efficient algorithm to determine for each location in Bangalore, the place from where we can buy ice at the lowest price in order to hold the party at v. (Note that we should be able to go to the shop and also return to the location where the party will be held!) Your algorithm should produce a list L indexed by vertices of G, where $L[v]$ gives the location from where we can buy ice to hold the party at location v. You do not have to supply the code. State informally, but precisely, the steps you would perform. $\boxed{10}$

   (b) What is the running time of your algorithm, if the graph has n vertices and m edges? Briefly justify your estimate. $\boxed{5}$

[1] We are aware that a successful party requires more than just ice, but a midterm problem must be kept simple (and legal).

7. Recall Dijkstra's algorithm for finding shortest paths from a vertex s. Assume that G has no edges with length zero. $\boxed{8}$

   (a) Fill in the blanks in the code below so that at the end, for each vertex v, the number of shortest paths from s to v is recorded in v · pathcount.

```
def single_source(s):
    global H
    for v in vertex_list:         # initialization
        (v.dist, v.prev, v.pathcount) = (INFINITY, None, 0)
    (s.dist, s.pathcount) = (0, 1)
    H = []
    makeheap(H, vertex_list)      # add all vertices to the heap
    while H:
        u = deletemin(H)
        for (v, ell) in u.out_nbrs:   # ell is the length of (u,v)
            if v.dist > u.dist + ell:
                (v.dist, v.prev)  = (u.dist + ell, u)
                bubble_up(H,v)
                v.pathcount = ___________________
#
            elif v.dist == ______________________:
#
                v.pathcount = ___________________
```

   (b) Why is your algorithm correct? (E.g., state the invariant condition concerning the pathcount attributes of the various vertices that holds at the beginning of each iteration. You don't necessarily have to formally establish that the invariant holds.) $\boxed{4}$

   (c) Will your algorithm (as stated) work correctly if there are edges with length 0? Provide a justification or a counter-example. $\boxed{3}$