2420 CS2040 Finals

MCQ: 40 marks, 10 Questions

Q1. [4 marks] A graph **G(V,E)** is represented as an "adjacency list" with each array entry **Adj[u]** being a hash table containing the vertices *v* for which *(u, v)* Î E. If all edge lookups are equally likely, what is the **expected/average time** to determine whether an edge is in the graph?

a. O(1)
b. O(V)
c. O(E)
d. $O(V^2)$

Using a HashMap in an Adjacency List will reduce an edge look up to O(1) on average. However, the worst-case time complexity is still O(V).

Q2. [4 marks] Given a complete undirected graph G(V, E) with V vertices and E edges. What is the most efficient time complexity to traverse the graph breadth-first?

a. O(V)
b. O(E)
c. O(V + E)
d. $O(V^2)$

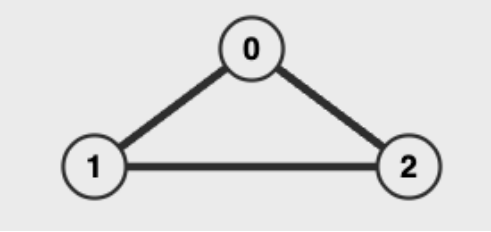
In a complete graph, one can visit one node first then visit all the other nodes as there are edges from one node to any other node.

Q3. [4 marks] Given an adjacency matrix as follows. How many minimum spanning trees do/does the graph have?

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

a. 1
b. 2
c. 3
d. 0

The equivalent graph is below. There are 3 minimum spanning trees in this graph.



Q4. [4 marks] You are given a maximum binary heap **A** of **N** distinct elements, as well as a maximum binary heap **B** of **M** distinct elements.
You want to create a new maximum **binary heap C** containing the **N+M** distinct elements from **A** and **B**. State the time complexity of the most efficient algorithm to do so, in terms of **N** and/or **M** only.

All 3 binary heaps are array-based, as described in lectures.

a. O(log (**N+M**))
b. O(**N+M**)
c. O((**N+M**) log (**N+M**))
d. O(**NM**)

Ans: b
Copy array A and B into the underlying array of C, the order of elements do not matter.
Then linear-time heapify the array used in C in O(N+M) time.

Q5. [4 marks] You are given a binary search tree **A** of **N** distinct elements, as well as a binary search tree **B** of **M** distinct elements. It is possible that **A** and/or **B** is NOT balanced.
You want to create a new **AVL tree C** containing the **N+M** distinct elements from **A** and **B**.
State the time complexity of the most efficient algorithm to do so, in terms of **N** and/or **M** only.
a. O(log (**N+M**))
b. O(**N+M**)
c. O((**N+M**) log (**N+M**))
d. O(**NM**)

Ans: b
In-order traversal **A** and **B** to get 2 sorted sequences in O(N) and O(M) time respectively, even if the trees are unbalanced.
Merge the two sorted sequences into another array containing the elements in **C** in order.
Recursively create an AVL tree by pulling up the middle of the array to be the root node, then recursing on the left subarray to be the left subtree and recursing on the right subarray to be the right subtree.

Q6. [4 marks] A 0-based UFDS of 8 elements, as implemented in lectures using both union-by-rank and path compression, was initialized as usual into 8 disjoint sets. After some operations, the underlying parents array currently has the following contents:

[1, 1, 2, 2, 1, 2, 2, 4]

How many of the following statements are correct:

i. The UFDS currently has 3 disjoint sets.
ii. From initialization to the current state, among all unionSet(x, y) operations performed, exactly 6 of them joined **different sets** together
iii. Currently, there exists at least 2 disjoint sets having the same size (number of elements in the set)

a. 0
b. 1
c. 2
d. 3

MCQ – UFDS

Ans: c
Only ii and iii are correct
i) The UFDS has only 2 disjoint sets {0, 1, 4, 7} and {2, 3, 5, 6}
ii) There are initially 8 sets but currently 2 sets, so there are 6 unions actually joining 2 sets. Any other unions would be of 2 elements that are already in the same set
iv) Both sets have 4 elements, hence this is true

Q7. [4 marks] Consider an unweighted DAG with V vertices and E edges. While we know that we can use BFS or a one-pass Bellman-Ford to solve for SSSP in an unweighted DAG, we sometimes need to compute the longest path from a single source instead.
Let's try augmenting algorithms that you have learnt in this course.
Which of the following approaches is the best in terms of runtime for finding the longest path in an unweighted DAG?
A. Make use of an O(V+E) algorithm — adapt traversal and/or topological sort and/or one-pass Bellman-Ford in some way.
B. Make use of an O(E log V) algorithm — adapt MST algorithms in some way.
C. Make use of an O((V+E) log V) algorithm — adapt O. or M. Dijkstra's algorithm in some way.
D. Make use of an O(VE) algorithm — adapt Bellman-Ford in some way.
E. Make use of an $O(V^2)$ algorithm — scan the adjacency-matrix to compute chain lengths.
F. Make use of an $O(V^3)$ algorithm — adapt Floyd-Warshall for longest paths in some way.
Answer: A
In a DAG we can exploit acyclicity to compute longest path in linear time. We can make use of an O(V+E) algorithm via topological traversal.

- Produce a topological ordering of all V vertices in O(V+E) using Khan's algorithm or DFS-based algorithm
- Initialise a distance array D[v] = -INF for all v, except D[src] = 0
- Visit vertices in topological order, and relax all outgoing edges (u -> v) using
    - D[v] = max (D[v], D[u] + 1)
    - Since all edges is considered exactly once, this is O(E).
- Total cost is hence O(V+E)

OR;
Treat each edge as weight –1 (instead of +1), then run one pass of Bellman–Ford using a topological ordering to compute shortest paths, and finally negate those distance values to recover the longest-path lengths.

- Total cost is O(V+E) since we have to produce the topological ordering, then run one-pass Bellman-Ford.

Option B is incorrect as MST algorithms are designed for minimum spanning trees.
Option C – F is incorrect as even though it is possible, it is just slower than option A.

- Option C: Augment M. Dijkstra's algorithm with the relaxation rule

Q8. [4 marks] Let G be a directed graph composed of only two strongly connected components, A and B.
There is only one edge from a vertex in one SCC to a vertex in the other, which is a->b where a ∈ A and b ∈ B.
Now construct an augmented graph G' by adding three new vertices p, q, and r and the following additional **directed** edges:

- p -> a' for some a' ∈ A
- b' -> q for some b' ∈ B
- p -> r, r -> q and q -> p
- b'' -> p for some b'' ∈ B

How many SCC(s) do/does G' have?

A. 1
B. 2
C. 3
D. 4

Answer: A

The additional edges create cycles connecting the cycle formed by p, q, and r with both A and B, thereby making every vertex reachable from every other vertex, which merges all components into a single SCC.

Q9. [4 marks]

Suppose G = (V, E) is a connected, undirected, weighted graph. There exists T, a shortest-path spanning tree of G, rooted at source vertex s.
Pick an edge (u, v) in G, and decrease the weight by a constant C > 0. The edge (u, v) can now have a negative weight.
Which of the following **must be true** for the new shortest-path spanning tree T'?

A. **The tree structure must remain the same as T;** only the weight of edge (u, v) is smaller.
B. **The tree structure must remain the same as T;** only the distance to v decreases by C; all other distances stay the same.
C. **A shortest-path spanning tree of G rooted at s must still exists;** every shortest-path distance for a vertex reached from s via v decreases.
D. **A shortest-path spanning tree of G rooted at s must still exists;** only vertices not reached via v can now find shorter paths (by rerouting through (u, v)); distances to vertices reached via v remain unchanged.
E. None of the above are required to be true; **T' may not exist.**

Answer: E

The decrease may cause a negative cycle, this will result in T' not exist at all, as shortest path may not be defined on G.
Option A to D are assuming that there still exist a shortest-path spanning tree.

Q10. [4 marks] Referring to the implementation of Floyd-Warshall algorithm in the lecture notes, after completing iteration k=3 (i.e. after using the first 3 vertices as intermediates) for a graph with > 4 vertices, the state of the distance matrix is best described by which option?

A. The distance matrix contains correct values only for pairs where both endpoints are among the first 3 vertices.
B. Only direct edge weights are used; no intermediate improvements are applied.
C. All shortest paths are finalized; further iterations won't change the matrix.
D. Only paths with no intermediates or with one intermediate from the first 3 are computed correctly.
E. The matrix shows, for each pair of vertices (x, y) among the first 3 vertices, whether there exists a path from x to y involving vertices from among the first 3 vertices only.
F. Shortest paths using only the first 3 vertices as intermediates are optimal.

Answer: F

After iteration k=3, the algorithm has updated shortest path estimates only for paths that use vertices from the first 3 as intermediates. Thus, any path that solely relies on these vertices is finalized, while others could potentially be shortened in later iterations.

A - Not correct because there do exist pairs with endpoints outside the first 3 whose distances are already final after k=3.
B - Not correct because after iteration k=3, intermediate improvements using those 3 vertices have been applied.
C - Not correct because paths that could benefit from vertices beyond the first 3 haven't been considered yet.
D - Not correct because paths may involve multiple intermediates from among the first 3, and other paths may also be partially optimized even if not finalized.
E - Not correct because transitive closure is about reachability, not optimal path costs, and the matrix isn't limited to just the first 3 vertices' connectivity.

Analysis: 21 marks, 3 Questions – 7 marks each

Q11. [7 marks] It is impossible to have a binary tree containing unique integers with more than 1 node that is both an AVL tree and a valid binary max heap.

a) True / False?
b) Explain your answer.

Answer: False

```
 10
 /
9
```

We can come up with a counterexample binary tree with 2 nodes. Notice that the tree still fits the property of AVL tree:

- All nodes in the left subtree must be less than the root node
- All nodes are balanced based on the balance factor

The tree also fits the property of binary max heap:

- It is a complete tree, where all levels are fully filled except the last, of which all nodes are as far left as possible
- The parent node is greater than all children nodes

Q12. [7 marks] A UFDS, as implemented in lectures using both union-by-rank and path compression, has 5,000 elements. Within the UFDS, 2 of the disjoint sets are **A** and **B**:

Set **A** has 1,000 elements with the same representative
Set **B** has only 10 elements, and its representative has a rank of 3

Claim: It is possible that unionSet(**A**, **B**) places the representative of set **A** under the representative of set **B**.

a) Is the claim True/False?
b) Give your rationale for your answer to (a).

Analysis T/F Question – UFDS
Ans: (a) True

(b) Set **A** can have a height and rank of 1, e.g. by unionSet(i, 0) for i = 1 to 999 inclusive. Union on sets **A** and **B** with union-by-rank will then cause set **A** of rank 1 to be under set **B** of rank 3 (which makes sense, so that the rank will not increase in an aim to keep the tree short)

Q13. [7 marks] A weighted connected undirected graph **G** is made up of 2 trees **A** and **B**, as well as 4 additional edges. This means that, if **A** and **B** contain **N** and **M** vertices respectively, then **G** contains **N**+**M** vertices and exactly **N**+**M**+2 edges.

Each of the 4 additional edges (**u**, **v**) is such that **u** is a vertex in **A**, while **v** is a vertex in **B**. Trees **A** and **B** each contains at least 4 vertices, and do not share any vertices between them. The weights of ALL the **N**+**M**+2 edges in **G** are distinct.

Claim: The Minimum Spanning Tree of **G** will include exactly one of the 4 additional edges mentioned above.

a) Is the claim True/False?
b) Give your rationale for your answer to (a).

Analysis T/F Question – MST
Ans: (a) False

(b) It is possible for the MST to pick more than one of the additional edges, but leaving an edge (or edges) within **A** (and/or **B**) of higher weight out instead (can prove by cycle property)
One such counter example:
A is a path graph of 4 vertices 3 edges, of weights 21, 22, 23
B is a path graph of 4 vertices 3 edges, of weights 1, 2, 3
(path graph looks like a linked list)
Additional edges are (A1, B1, 12), (A2, B2, 13), (A3, B3, 14), (A4, B4, 15)
By cycle property, these 3 edges would NOT be in the MST:
(A1, A2), (A2, A3), (A3, A4)
All 4 additional edges will be taken instead
(The MST could include 1, 2, 3, or 4 of the additional edges, but not 0)

Structured: 39 marks, 3 Questions – 13 marks each.

14) [13 marks] Range of orange

Ivan has **N** (which can be very large) orange slices, each having a volume (positive floating point number with no fixed precision). Ivan takes out a sharp knife and makes **C** cuts on the oranges.
When Ivan makes a cut, he takes the slice with the largest volume and splits it into two equal volumes (which add up to the original slice's volume). After each cut, you are to output the range of the oranges' volume, i.e. $V_{largest\_slice} - V_{smallest\_slice}$.

You are given the integer **C**, as well as a nonempty array **A** which contains the **N** volumes, one for each orange slice.
Example: **C** = 5, **A** = [8.0, 4.0, 6.0]
the 5 outputs in sequence are: 2.0 1.0 2.0 2.0 1.0
because

after 0 cuts: **8.0** 4.0 6.0	range : 2.0 but not in output
after 1 cuts: 4.0 4.0 4.0 **6.0**	range : 2.0
after 2 cuts: **4.0** 4.0 4.0 3.0 3.0	range : 1.0
after 3 cuts: **4.0** 4.0 3.0 3.0 2.0 2.0	range : 2.0
after 4 cuts: **4.0** 3.0 3.0 2.0 2.0 2.0 2.0	range : 2.0
after 5 cuts: 3.0 3.0 2.0 2.0 2.0 2.0 2.0 2.0	range : 1.0

Design a most efficient algorithm to output the **C** ranges, stating clearly any data structures required where they matter.

A correct O(**N** + **C** log **C**) algorithm will score full marks, while a O(**N** log **N** + **C** log **C**) algorithm will score "high" but not full marks.

Ans:
Each cut involves removal of the maximum volume, 2 insertions of volume, and then finding the maximum and minimum volume to compute the range of oranges' volume.
A max binary heap can be used – This approach also requires manually keeping track of the minimum slice volume in addition to the heap. The binary heap should be created in O(N) time by heapifying all internal nodes from bottom to top, and NOT through repeated insertion which takes O(N log N) time.

```
MaxHeap<Double> volumes = new MaxHeap<>(A) // O(N) fast ver.
smallest = min_element_in(A) // linear pass
repeat C times {
      half = volumes.removeMax() / 2
      smallest = min(smallest, half)
      volumes.add(half)
      volumes.add(half)
      output(volumes.peek() - smallest)
}
```

Alternatively, if only minimum heap is supported, weights can be stored negated (and negated once again when peeked/removed

A less efficient solution uses a balanced BST – This approach requires you to keep track of the frequency (number of orange slices) of each volume, since BST does not allow duplicate keys. i.e. use a TreeMap. The AVL tree creation takes O(N log N) time.

15)
[13 marks] TransferWise, a global money transfer company, needs a system to compute processing fee differences between banks. Each of the N banks charges a fee, but the exact fees are not known. However pairwise information about the differences between some bank's fees are known (e.g., Bank A is $5 cheaper than Bank B). Processing fees may vary across different pairs of banks, i.e., the difference in processing fee may not be constant. Moreover, assume that the processing fee is always fixed for each bank. Consequently, all comparisons between pairs of banks will always lead to a consistent difference in processing fee.

Design a system to answer customer queries about fee differences between any two banks. Assume all queries can be answered based on the given pairwise information.
Sample data, queries and answers are provided below:
Consider 3 banks OSBC, MBS, HCBC along with the following information.

- K1. Transferring through OSBC costs 4 dollars more than transferring through MBS.
- K2. Transferring through OSBC costs 3 dollars less than transferring through HCBC.

Query 1: How much more expensive it is to transfer through MBS than OSBC?
Answer: -4 dollars. (From K1)

Query 2: How much more expensive it is to transfer through MBS than HCBC?
Answer: -7 dollars. (-4-3, where -4 is because of K1 and -3 is because of K2)

Ensure you clearly indicate and answer these three parts:

1. Model the given scenario as a graph.

2. Assume the number of pairwise information available is N-1, design an algorithm to answer Q queries efficiently.
3. Assuming the number of pairwise information is available is M, where M is much higher than (N-1), design an algorithm to answer Q queries efficiently.

1. Model:

This problem is a variation of Q4 of Tutorial Week 12. The problem can be modelled as a weighted directed graph. The vertices are banks and an edge exists if the relative difference in processing charges available.

For the given toy example:

OSBC, MBS and HCBC are vertices. There are two edges: one between OSBC and MBS; one between OSBC and HCBC.

The directed edge from OSBC to MBS would have a weight of 4. The directed edge from OSBC to HCBC would have a weight of -3. We can also think of two additional edges in the opposite direction. The directed edge from MBS to OSBC would have a weight of -4. The directed edge from HCBC to OSBC would have a weight of 3.

2. For the case with (N-1) pairs, the model would be a tree. Hence, we can use DFS to establish the cost of processing with respect to a reference bank and store this information in an array. It can be done O(N) time.

Subsequently each query can be answered in O(1) time and the total time complexity for Q queries would be O(N+Q)

3. For the case of M pairs of information with M greater than (N-1), the model is no longer a tree. However, we can still use BFS/DFS as the processing fees difference between a given pair of banks would be the same regardless of the path traversed in the graph. Hence, we can use DFS to establish the cost of processing with respect to a reference bank and store this information in an array. It can be done O(N+M) time.

Subsequently each query can be answered in O(1) time and the total time complexity for Q queries would be O(N+M+Q).

16) **[13 marks]** Static vs current electricity

Tom has **K** (which can be very large) steel plates numbered 0 to **K**-1 (in that order) in a line from left to right. All plates do not touch each other. Initially, a plate is connected to either one power source or to nothing.

There are only two power sources – a positive source, and a negative source. The positive source always remains positive, and the negative source always remains negative.

Subsequently, wires are secured between 2 plates so that electricity can pass between them:
- **Current electricity flows** along paths from the positive to the negative source
- However, if there is a path to only one source (but not both), then **current does NOT flow**, but the plate becomes positively or negatively charged instead (static electricity)

Hence, each plate takes one of four states:

N Neutral – Plate is not in any way connected to any power source

\+ charged – The only source that the plate is connected to is the positive source

\- charged – The only source that the plate is connected to is the negative source

C Current – There is current flowing through **some part of the circuit this plate is connected** to (to be precise, current may not be flowing through this plate itself)

Design an algorithm for each of the 2 operations:

`Init(int K, int[] P, int[] N)` – There are **K** plates. **P**, **N** are lists of plate numbers that are connected to the positive, negative source respectively. Perform any initialization on data structures to support the `Wire` operations

`Wire(int L, int R)` – Secure a wire between plate **L** and plate **R** and **return the state of the plates** on both sides of the wire after it is secured (Think about it, the states of both plates will always be the same after the wire is secured…)

For full credit, both operations should run correctly, and the `Wire` operation should run as efficiently as possible ("not far from O(1) time" on average) while the `Init` operation should support the given implementation of `Wire` as efficiently as possible.

Example: (visualization?)

Init(6, {1}, {5, 4}) <-- the metal plates have states ['N', '+', 'N', 'N', '-', '-'] in sequence

Wire(3, 2) returns 'N' <-- states are still ['N', '+', 'N', 'N', '-', '-'] though wire secured

Wire(1, 2) returns '+' <-- states are now ['N', '+', '+', '+', '-', '-'] as plates 2,3 are + charged

Wire(3, 4) returns 'C' <-- states are now ['N', 'C', 'C', 'C', 'C', '-'] as plates 1 to 4 are connected to a circuit with current flowing through

Wire(0, 1) returns 'C' <-- states are now ['C', 'C', 'C', 'C', 'C', '-'] as plate 0 is ALSO connected to a circuit with current flowing through, **even though current doesn't take a path through plate 0**

Ans:

Since all plates reachable through wires will share the same state, and since wires can only be added but not removed, a UFDS can help to union wires and query the state of a plate efficiently (ensure union-by-rank and path compression are used as in lectures). Each maximal set of plates that are connected through wires is a UFDS set (individual plate without wire is also a set).

States change according to the following rules:

		R			
	Old State	N	+	-	C
L	N	N	+	-	C
	+	+	+	C	C
	-	-	C	-	C
	C	C	C	C	C

New state is shown in cells with yellow background

```
Init(int K, int[] P, int[] N):
      ufds = UFDS(K)
      states = char[] of K 'N's
      for (elm in P) states[elm] = '+'
      for (elm in N) states[elm] = '-'

FindNewState(char A, char B):
      oldStates = {A, B} // hash set
```

```
      if (oldStates.size() == 1) return A // unchanged
      if (oldStates.contains('C') or oldStates.eq({'-','+'}))
            return 'C' // current flow
      if (oldStates.contains('+')) return '+' // static + chrg.
      return '-' // static - charge

Wire(int L, int R):
      newState = FindNewState(states[ufds.findSet(L)],
                              states[ufds.findSet(R)])
      ufds.unionSet(L, R)
      states[ufds.findSet(L)] = newState
      return newState
```

Init takes O(**K**) time
Wire takes amortized O(inverseAckermann(**K**)) time (i.e. better than O(log **K**) time, close to O(1) time on average)