COMP 3804 — Assignment 2

Due: Thursday February 16, 23:59.

Assignment Policy:

- Your assignment must be submitted as one single PDF file through Brightspace.

> Use the following format to name your file:
>
> `LastName_StudentId_a2.pdf`

- **Late assignments will not be accepted. I will not reply to emails of the type "my internet connection broke down at 23:57" or "my scanner stopped working at 23:58", or "my dog ate my laptop charger".**
- You are encouraged to collaborate on assignments, but at the level of discussion only. When writing your solutions, you must do so in your own words.
- Past experience has shown conclusively that those who do not put adequate effort into the assignments do not learn the material and have a probability near 1 of doing poorly on the exams.
- When writing your solutions, you must follow the guidelines below.
  - You must justify your answers.
  - The answers should be concise, clear and neat.
  - When presenting proofs, every step should be justified.

Question 1: Write your name and student number.

Question 2: You are given k sorted lists $L_1, L_2, \ldots, L_k$ of numbers. Let n denote the total length of all these lists.

Describe an algorithm that returns one list containing all these n numbers in sorted order. The running time of your algorithm must be $O(n \log k)$.

Explain why your algorithm is correct and why the running time is $O(n \log k)$.

Hint: If $k = 2$, this should look familiar.

Question 3: This is a long question. Don't be intimidated! As always, for each part in this question, you must justify your answer.

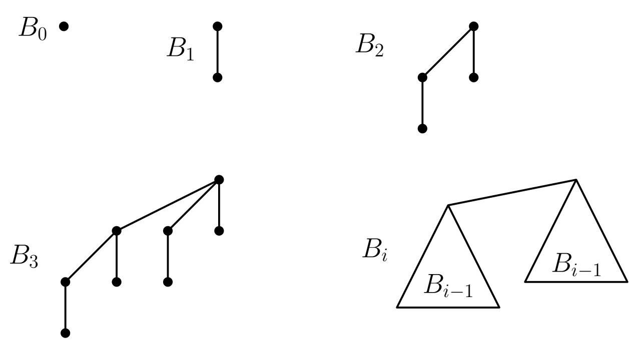
Professor Justin Bieber needs a data structure that maintains a collection $A, B, C, \ldots$ of sets under the following operations:

1. Maximum(X): return the largest element in the set X.
2. Insert(X, y): add the number y to the set X.
3. ExtractMax(X): delete and return the largest element in the set X.
4. Combine(X, Y): take the union $X \cup Y$ of the sets X and Y, and call the resulting set X.

Professor Bieber knows how to support the first three operations: Store each set X in a max-heap. The fourth operation seems to be more problematic, because we have to take two max-heaps and combine them into one max-heap.

To support all four operations, Professor Bieber has invented the following sequence $B_0, B_1, B_2, \ldots$ of trees, which are now universally known as *Bieber trees*:

1. B_0 is a tree with one node.
2. For each $i \geq 1$, the tree B_i is obtained as follows: Take two copies of B_{i-1} and make the root of one copy a child of the root of the other copy.



Question 3.1: Let $i \geq 0$. How many nodes does the tree B_i have?

Question 3.2: Let $i \geq 0$. What is the height of the tree B_i?

Question 3.3: Let $i \geq 1$. Prove that the subtrees of the root of B_i are the Bieber trees $B_0, B_1, \ldots, B_{i-1}$.

Let X be a set of n numbers, assume that $n \geq 1$, and let

$$n = (b_m, b_{m-1}, \ldots, b_1, b_0)$$

be the binary representation of n. Note that $b_m = 1$ and

$$n = \sum_{i=0}^{m} b_i \cdot 2^i.$$

The *Bieber max-heap* for the set X is obtained as follows:

1. Partition the set X, arbitrarily, into subsets such that for each i for which $b_i = 1$, there is exactly one subset of size 2^i.

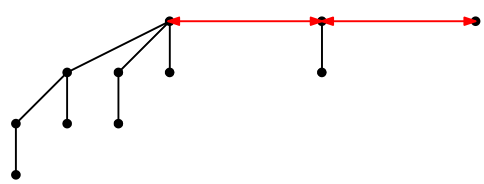
   For example, if $n = 11 = 2^3 + 2^1 + 2^0$, the set X is partitioned into three subsets: one of size 2^3, one of size 2^1, and one of size 2^0.

2. Each subset of size 2^i is stored in a Bieber tree B_i. Each node in B_i stores one element of the subset. Each node in B_i has pointers to its parent and all its children. There is a pointer to the root of B_i.

3. Each Bieber tree has the property that the value stored at a node is larger than the values stored at any of its children.

4. The roots of all these Bieber trees are connected using a doubly-linked list.

The figure below gives an example when $n = 11$.



Question 3.4: Let X be a non-empty set of numbers, and assume that this set is stored in a Bieber max-heap. Describe an algorithm that implements the operation MAXIMUM(X) in $O(\log |X|)$ time.

Question 3.5: Let X and Y be two sets of numbers, and assume that both sets have the same size 2^i. A Bieber max-heap for X consists of one single Bieber tree B_i. Similarly, a Bieber max-heap for Y consists of one single Bieber tree B_i. Describe an algorithm that implements the operation COMBINE(X, Y) in $O(1)$ time.

Question 3.6: Let X and Y be two non-empty sets of numbers, and assume that X is stored in a Bieber max-heap and Y is stored in a Bieber max-heap. Describe an algorithm that implements the operation COMBINE(X, Y) in $O(\log|X| + \log|Y|)$ time.
Hint: This operation computes one Bieber max-heap storing the union $X \cup Y$. If you take the sum of two integers, both given in binary, then you go through the bits from right to left and keep track of a carry bit.

Question 3.7: Let X be a non-empty set of numbers, and assume that this set is stored in a Bieber max-heap. Describe an algorithm that implements the operation INSERT(X, y) in $O(\log|X|)$ time.

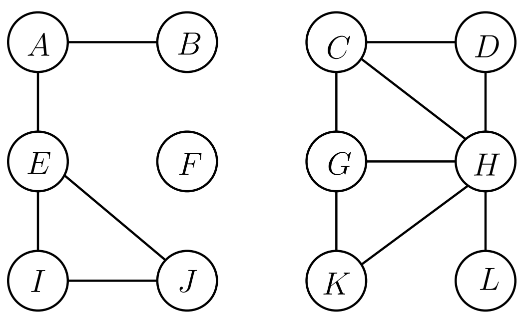
Note that this operation computes a Bieber max-heap for the set $X \cup \{y\}$.

Question 3.8: Let X be a non-empty set of numbers, and assume that this set is stored in a Bieber max-heap. Describe an algorithm that implements the operation EXTRACTMAX(X) in $O(\log|X|)$ time.

Note that this operation computes a Bieber max-heap for the set $X \setminus \{y\}$, where y is the largest number in X.

Question 3.9: Let X be a non-empty set of numbers, and assume that this set is stored in a Bieber max-heap. How would you extend this data structure such that the operation MAXIMUM(X) only takes $O(1)$ time, whereas the running times for the other operations COMBINE, INSERT, and EXTRACTMAX remain as above?

Question 4: Consider the following undirected graph:

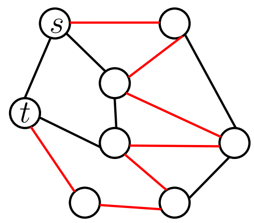


Draw the DFS-forest obtained by running algorithm DFS on this graph. The pseudocode is given at the end of this assignment. Algorithm DFS uses algorithm EXPLORE as a subroutine; the pseudocode for this subroutine is also given at the end of this assignment.

In the forest, draw each tree edge as a solid edge, and draw each back edge as a dotted edge.

Whenever there is a choice of vertices, pick the one that is alphabetically last.

Question 5: Tyler is not only your friendly TA, he is also the inventor of Tyler paths and Tyler cycles in graphs: A *Tyler path* in an undirected graph is a path that contains every vertex exactly once. In the figure below, you see a Tyler path in red. A *Tyler cycle* is a cycle that contains every vertex exactly once. In the figure below, if you add the black edge $\{s, t\}$ to the red Tyler path, then you obtain a Tyler cycle.



If $G = (V, E)$ is an undirected graph, then the graph G^3 is defined as follows:

1. The vertex set of G^3 is equal to V.
2. For any two distinct vertices u and v in V, $\{u, v\}$ is an edge in G^3 if and only if there is a path in G between u and v consisting of at most three edges.

Question 5.1: Describe a *recursive* algorithm TylerPath that has the following specification:

> **Algorithm** TylerPath(T, u, v)**:**
> **Input:** A tree T with at least two vertices; two distinct vertices u and v in T such that $\{u, v\}$ is an edge in T.
> **Output:** A Tyler path in T^3 that starts at vertex u and ends at vertex v.

Hint: You do not have to analyze the running time. The base case is easy. Now assume that T has at least three vertices. If you remove the edge $\{u, v\}$ from T, then you obtain two trees T_u (containing u) and T_v (containing v).

1. One of these two trees, say, T_u, may consist of the single vertex u. How does your recursive algorithm proceed?
2. If each of T_u and T_v has at least two vertices, how does your recursive algorithm proceed?

Question 5.2: Prove the following lemma:

Tuttle's Lemma: For every tree T that has at least three vertices, the graph T^3 contains a Tyler cycle.

Question 5.3: Prove the following theorem:

Tuttle's Theorem: For every connected undirected graph G that has at least three vertices, the graph G^3 contains a Tyler cycle.

```
Algorithm DFS(G):
for each vertex u
do visited(u) = false
endfor;
cc = 0;
for each vertex v
do if visited(v) = false
    then cc = cc + 1
         Explore(v)
    endif
endfor
```

```
Algorithm Explore(v):
visited(v) = true;
ccnumber(v) = cc;
for each edge {v, u}
do if visited(u) = false
    then Explore(u)
    endif
endfor
```