COMP3821 Assignment 1

1. **You're given an array A of n integers, and must answer a series of m questions, each of the form: "Given two integers, X_i and Y_i, how many elements a of the array A satisfy $a \times X_i = Y_i$".**
The algorithm for this question will be outlined in a series of steps.

   i. Create an empty Hash Table, where the keys will be the integer values of the elements of A, and the values the count of appearances of that integer value in the array A.

   ii. Take the array A of n integers, and iterate through each element.

   iii. For each element in A, add the element to the Hash Table, using its value as the hash key.

   iv. If there is not a key, for the current element in A, in the Hash Table, then create a new key for the integer value, and initialise the count, which is the value, to 1.

   v. If there is a key, for the current element in A, in the Hash Table, then increment the count stored.

   vi. For each query where we are given X_i and Y_i, we are asked to find the number of elements of A, a, such that $a \times X_i = Y_i \ldots (1)$.

   vii. Compute the value of Y_i/X_i, and use this result to access the Hash Table previously set up.

   viii. If the key exists for Y_i/X_i, then the result returned will be the number of elements of A that satisfy equation (1).

   ix. If the key does not exist for Y_i/X_i, then there are no such elements in A that satisfy (1), for the integers X_i and Y_i.

   Traversing the n element array A and creating the Hash Table will be completed in $\mathcal{O}(n)$ time. The Hash Table Lookup is completed in $\mathcal{O}(1)$ time, and with m such queries, completes in $\mathcal{O}(m)$ time. Overall, the algorithm completes in $\mathcal{O}(n + m)$ time.

   It is fairly trivial to see that the algorithm is correct and produces the results required.

2. **You are given an array S of n integers and another integer x.**

   a) **Describe an $\mathcal{O}(n \log n)$ algorithm that determines whether or not there exist two elements in S whose absolute difference is exactly x.**
   The algorithm for this question will be outlined in a series of steps.

   i. Sort the array S of n integers using MergeSort.

   ii. For each element, $S[i]$, binary search for the values $S[i] + x$ and $S[i] - x$, in $S[i + 1 ... n - 1]$.

   iii. If a value is found in either search, there is such a pair that has the difference x.

   iv. If neither search returns a value, there does not exist such a pair including $S[i]$, so progess to the next element, that is, continue from step **ii**.

   MergeSort completes in $\mathcal{O}(n \log n)$ time. Binary Search completes in $\mathcal{O}(\log n)$ time, and so running on each of the n elements of S takes $\mathcal{O}(n \log n)$ time to complete. Overall, the algorithm completes in $\mathcal{O}(n \log n)$ time.

   It is fairly trivial to see that the algorithm is correct and produces the results required.

b) **Describe an algorithm that acomplishes the same task, but runs in expected $\mathcal{O}(n)$ time.**

The algorithm for this question will be outlined in a series of steps.

i. Create an empty Hash Table, where the keys will be the integer values of the elements of S. The values are not required for this Hash Table, as all that matters is whether or not the keys exist.

ii. Traverse the array S, and add each element to the Hash Table, using the integer value as the key.

iii. If the key already exists, do nothing, and progress to the next element of S.

iv. If the key does not exist, create the new key, before progressing to the next element of S.

v. Traverse S, and for each element $S[i]$, lookup $S[i] + x$ and $S[i] - x$ in the Hash Table.

vi. If either key exists, then there is such a pair of integers with absolute difference of x.

vii. If neither of the keys exists, then there does not exist a pair of integers using that specific array element $S[i]$, so we progress to the next element, that is, continue from step **v**.

Traversing the n element array S and creating the Hash Table is completed in $\mathcal{O}(n)$ time. The Hash Table lookups are completed in $\mathcal{O}(1)$ time. As there are n elements in S, performing 2 Hash Tables lookups for each element completes in $\mathcal{O}(n)$ time. Overall, the algorithm runs in $\mathcal{O}(n)$ time.

It is fairly trivial to see that the algorithm is correct and produces the results required.

3. You are an assistant news broadcaster reporting on a cycling race containing n cyclists. You have been given the task of computing the excitement factor of a given race, which is calculated as follows:

- **The excitement factor starts at zero.**
- **Any time one of the first $\frac{n}{2}$ cyclists is overtaken by any other cyclist, the excitement factor increases by 1.**

Unfortunately, you have only been given the starting and finishing order of the cyclists. From this data, you need to calculate the minimum possible excitement factor for the whole race. You may assume than n is a power of 2 and is strictly greater than 1.

Assume for the minimum excitement factor that only the minimum number of swaps to achieve the final ranking from the initial ranking are made. The algorithm for this question will be outlined in a series of steps.

i. If the starting order and finishing order data sets are not presented in array form, transform them first into arrays, and label them S for the starting order array, and F for the finishing order array.

ii. Assume that the cyclists are identified by their position in S. That is, the cyclists will have identifiers in the set of integers $[0 \ldots n-1]$.

iii. Create an empty Hash Table, where the keys will be the cyclist's index in S, and the value stored will be the cyclist's position in F.

iv. Traverse F and insert each cyclist into the Hash Table, using the unique identifier for the key, and their index (position) in F as the value inserted for their key.

v. Initialise a counter for excitement factor to be 0.

vi. Now, traverse S, and for each cyclist, lookup their position in S as their key in the Hash Table, to get their final ranking, which is their position in F.

vii. Compare this final ranking, which is their index in F, to their intial ranking, which is their index in S.

viii. If their final ranking is not less than their initial ranking, progress to the next cyclist in S, that is, continue from step **vi**.

ix. If their final ranking is less than their initial ranking, then a number of possibilities need to be checked. The following possibilities will be the set of indented steps. Treat each possiblity like a branch in an if-else block. Only one of the steps below will apply for each of the cyclists.

x. Note that the top $\frac{n}{2}$ cyclists occupy positions $\left[0 \ldots \frac{n}{2} - 1\right]$.

- α) If the cyclist's final position is equal to position $\frac{n}{2}$, or below, this cyclist has not overtaken any cyclists in the top $\frac{n}{2}$ cyclists.
- β) If the cyclist's final positon and starting position are both less than $\frac{n}{2}$, subtract their final position from their initial position. Add this to the excitement factor, as this difference in rankings is the number of top $\frac{n}{2}$ cyclists that they overtook.
- γ) If the cyclists final positon is less than $\frac{n}{2}$, but their starting position is not, subtract their final position from $\frac{n}{2}$. Add this to the excitement factor, as this difference is the number of top $\frac{n}{2}$ cyclists that they overtook.

xi. Progress to the next cyclist, that is, continue from step **vi**.

Creating both S and F can be completed in $\mathcal{O}(n)$ time. Traversing F to create the Hash Table is completed in $\mathcal{O}(n)$. Inserting into the Hash Table, and Hash Table lookups, are both achieved in $\mathcal{O}(1)$ time. Traversing S to lookup the finishing rank is completed in $\mathcal{O}(n)$ time. All rank difference calculations are done whilst traversing S, and are all completed in $\mathcal{O}(1)$ time. Overall, the algorithm completes in $\mathcal{O}(n)$ time.

In step **viii**, we only consider the case where a cyclist position has increased. Considering as well, all of the cyclists who's positions had decreased would simply be double counting, as in an overtake, one cyclist increases in position, whilst the other decreases in position, by the same amount of positions, which is 1 position. Any cyclists who remain in the same position have not performed any overtakes, nor been overtaken, in the minimum excitement factor scenario, so do not need to be considered at all.

In step **x**, and its sub-steps, the reason differences from position $\frac{n}{2}$ upwards are only counted, is because any gain in rankings below that position are not overtakes of cyclists in the top $\frac{n}{2}$ positions, and so add nothing to the excitement factor. Although $\frac{n}{2}$ is the first position outisde of the top $\frac{n}{2}$, any movement upwards from this position puts the cyclist into the top $\frac{n}{2}$ cyclists, and so they must have overtaken at least one of them, thus affecting the excitement factor.

The rest of the algorithm is fairly trivial to understand, and see that it produces the correct results.

4. Determine if $f(n) = \mathcal{O}(g(n))$, $f(n) = \Omega(g(n))$, or $f(n) = \Theta(g(n))$.

a) We are given $f(n) = n!$ and $g(n) = n^n$. Select $c = 1$, and $n_o = 0$. We can rewrite $f(n) = n! = n \times (n-1) \times \cdots \times 1$, and $g(n) = n^n = n \times n \times \cdots \times n$. It is clear from these forms, that $\forall n > n_0$, $n! \leq 1 \cdot n^n$. Therefore $f(n) = \mathcal{O}(n^n)$.

b) We are given $f(n) = (\log_3 n)^2$ and $g(n) = \log_3\left(n^2 \times n^{\log_3 n}\right)$. Consider $g(n)$, and manipulate to get the following expression.

$$\begin{aligned} g(n) &= \log_3\left(n^2 \times n^{\log_3 n}\right) \\ &= \log_3\left(n^2\right) + \log_3\left(n^{\log_3 n}\right) \\ &= 2\log_3(n) + (\log_3 n)\log_3(n) \\ &= 2\log_3(n) + (\log_3 n)^2 \\ &= 2\log_3(n) + f(n) \end{aligned}$$

Select $c = 1$ and $n_0 = 1$. Thus, clearly $\forall n > n_0$, $f(n) \leq 1 \cdot (f(n) + 2\log_3 n)$. Therefore, $f(n) = \mathcal{O}(g(n))$. Now consider $\log_3 n$, where $n_0 = 1$ still. We get the following result.

$$\begin{aligned} (\log_3 n)^2 &\geq \log_3 n \\ 2(\log_3 n)^2 &\geq 2\log_3 n \end{aligned}$$

Therefore, $\forall n > n_0$, $g(n) = (\log_3 n)^2 + 2\log_3 n \leq 3 \cdot (\log_3 n)^2$. Thus, $g(n) = \mathcal{O}(f(n))$. As $f(n) = \mathcal{O}(g(n))$, and $g(n) = \mathcal{O}(f(n))$, then clearly $f(n) = \Theta(g(n))$.

c) We are given $f(n) = \log_2 \sqrt{n}$ and $g(n) = \sqrt{\log_2 n}$. Rewrite $f(n) = \log_2 \sqrt{n} = \frac{1}{2}\log_2 n$. Select $n_0 = 4$, and $c = \frac{1}{2}$. Clearly, $\forall n > n_0$, $\log_2 n \geq \sqrt{\log_2 n}$, and so $\frac{1}{2}\log_2 n \geq \frac{1}{2}\sqrt{\log_2 n}$. Therefore, $f(n) = \Omega(g(n))$.

d) We are given $f(n) = n\Big(2n + \sin\left(\frac{\pi n}{4}\right)\Big)$ and $g(n) = n^2$. Rewrite $f(n) = n\Big(2n + \sin\left(\frac{\pi n}{4}\right)\Big) = 2n^2 + n\sin\left(\frac{\pi n}{4}\right)$. As $\sin\left(\frac{\pi n}{4}\right) \leq 1$, then $n\sin\left(\frac{\pi n}{4}\right) \leq n$, $\forall n > 0$. Select $n_0 = 0$. Clearly, $\forall n > n_0$, $f(n) \leq 2n^2 + n \leq 2n^2 + n^2 = 3n^2$. Thus, $f(n) = \mathcal{O}(n^2) = \mathcal{O}(g(n))$. Now select $n_0 = 1$. We have $f(n) \geq 2n^2 - n$, which $\forall n > n_0$, $f(n) \geq 2n^2 - n \geq n^2$. Therefore $f(n) = \Omega(g(n))$, so $g(n) = \mathcal{O}(f(n))$. Thus, as $f(n) = \mathcal{O}(g(n))$, and $g(n) = \mathcal{O}(f(n))$, then $f(n) = \Theta(g(n))$.

5. **Determine the asymptotic growth rate of the solutions to the following recurrences. If possible, you can use the Master Theorem, if not, find another way of solving.**

   a) We are given $T(n) = 3T\left(\frac{n}{2}\right) + n(2 + \cos n)$. Therefore, $a = 3$, $b = 2$, and $f(n) = n(2 + \cos n)$. Clearly, $\log_b a = \log_2 3$. Select $\epsilon = \log_2 3 - 1 > 0$. As $f(n) = \mathcal{O}(n)$, then $f(n) = \mathcal{O}(n^{\log_2 3 - \epsilon})$. This is case 1 of the Master Theorem. Thus, $T(n) = \Theta(n^{\log_2 3})$.

   b) We are given $T(n) = 3T\left(\frac{n}{4}\right) + n\sqrt[3]{n}$. Therefore, $a = 3$, $b = 4$, and $f(n) = n\sqrt[3]{n} = n^{\frac{4}{3}}$. Clearly, $\log_b a = \log_4 3$. Futhermore, $f(n) = \Omega(n)$, so selecting $\epsilon = 1 - \log_4 3 > 0$, $f(n) = \Omega(n^{\log_4 3 + \epsilon})$. Now, consider $af\left(\frac{n}{b}\right) = \frac{3}{4^{\frac{4}{3}}} \cdot n^{\frac{4}{3}}$. Select $c = \frac{1}{2}$, and $n_0 = 1$. Clearly, $\forall n > n_0$, $\frac{3}{4^{\frac{4}{3}}} \cdot n^{\frac{4}{3}} \leq c \cdot n^{\frac{4}{3}}$. This is case 3 of the Master Theorem. Thus, $T(n) = \Omega(n^{\frac{4}{3}})$.

   c) We are given $T(n) = 5T\left(\frac{n}{2}\right) + n^{\log_2 5}\left(1 + \sin\left(\frac{2\pi n}{3}\right)\right)$. Therefore, $a = 5$, $b = 2$, and $f(n) = n^{\log_2 5}\left(1 + \sin\left(\frac{2\pi n}{3}\right)\right)$. Clearly, $\log_b a = \log_2 5$. As $\sin\left(\frac{2\pi n}{3}\right) \leq 1$, then obviously $f(n) = \Theta(n^{\log_2 5})$. This is case 2 of the Master Theorem. Thus, $T(n) = \Theta(n^{\log_2 5} \log_2 n)$.

   d) We are given $T(n) = T(n-1) + \log(n)$. Following the recurrence through, we get the following result.

$$\begin{aligned}
T(n) &= T(n-1) + \log n \\
&= T(n-2) + \log(n-1) + \log n \\
&= T(n-3) + \log(n-2)\log(n-1) + \log n \\
&\dots \\
&= T(1) + \log 2 + \log 3 + \cdots + \log(n-1) + \log n \\
&= T(1) + \log(2 \times 3 \times \cdots \times (n-1) \times n) \\
&= T(1) + \log(1 \times 2 \times 3 \times \cdots \times (n-1) \times n) \\
&= T(1) + \log(n!)
\end{aligned}$$

   Clearly, $T(n) = \Theta(\log(n!))$.