Due Friday 16^{th} of June at 6pm Sydney time (week 3)

In this assignment we review some basic algorithms and data structures, and we apply the divide-and-conquer paradigm. There are *three problems* each worth 20 marks, for a total of 60 marks. Partial credit will be awarded for progress towards a solution. We'll award one mark for a response of "one sympathy mark please" for a whole question, but not for parts of a question.

Any requests for clarification of the assignment questions should be submitted using the Ed forum. We will maintain a FAQ thread for this assignment.

For each question requiring you to design an algorithm, you *must* justify the correctness of your algorithm. If a time bound is specified in the question, you also *must* argue that your algorithm meets this time bound. The required time bound always applies to the *worst case* unless otherwise specified.

You must submit your response to each question as a separate PDF document on Moodle. You can submit as many times as you like. Only the last submission will be marked.

Your solutions must be typed, *not* handwritten. We recommend that you use LaTeX, since:

- as a UNSW student, you have a free Professional account on Overleaf, and
- we will release a LaTeX template for each assignment question.

Other typesetting systems that support mathematical notation (such as Microsoft Word) are also acceptable.

Your assignment submissions must be your own work.

- You may make reference to published course material (e.g. lecture slides, tutorial solutions) without providing a formal citation. The same applies to material from COMP2521/9024.
- You may make reference to either of the recommended textbooks with a citation in any format.
- You may reproduce general material from external sources in your own words, along with a citation in any format. 'General' here excludes material directly concerning the assignment question. For example, you can use material which gives more detail on certain properties of a data structure, but you cannot use material which directly answers the particular question asked in the assignment.
- You may discuss the assignment problems privately with other students. If you do so, you must acknowledge the other students by name and zID in a citation.
- However, you must write your submissions entirely by yourself.
  - Do not share your written work with anyone except COMP3121/9101 staff, and do not store it in a publicly accessible repository.
  - The only exception here is UNSW Smarthinking, which is the university's official writing support service.

Please review the UNSW policy on plagiarism. Academic misconduct carries severe penalties.

Please read the Frequently Asked Questions document, which contains extensive information about these assignments, including:

- how to get help with assignment problems, and what level of help course staff can give you
- extensions, Special Consideration and late submissions
- an overview of our marking procedures and marking guidelines
- how to appeal your mark, should you wish to do so.

Question 1

You are given an array A of n distinct positive integers and a positive integer x.

1.1 **[8 marks]** Design an algorithm which decides if there exist two distinct indices $1 \leq i, j \leq n$ such that $2A[i] - 3A[j] = x$. In the worst-case, your algorithm must run in $O(n \log n)$ time.

We propose a binary search-based algorithm as follows.

Algorithm: Construct another array $B[1 \ldots n]$ such that $B[j] = 3A[j] + x$ where $j = 1, \ldots, n$ and then sort B in ascending order using merge sort. Now we go through A and check for each i in $1, \ldots, n$ if $2A[i]$ belongs to B using binary search. Any successful search indicates a yes to our question else it we return no.

Correctness: If there exists a some valid $\alpha, \beta \in A$ then we must have $2\alpha = 3\beta + x \in B$ which we can binary search for. In the case of no solution, the uniqueness of our solution guarantees that the binary search will return no result.

Time complexity: Merge sorting B takes $O(n \log n)$, then for each step of iteration through A, we use $O(\log n)$ to binary search the value in B resulting in a total complexity of $O(n \log n)$ again. Thus the whole algorithm runs in time $O(n \log n)$.

1.2 **[4 marks]** Solve the same problem as in 1.1 but with an algorithm which runs in the **expected time** of $O(n)$.

Expected time complexity usually suggests that we use a *hash table*.

Algorithm: Go through A for each $i = 1, \ldots, n$ we insert $3A[i] + x$ into our hash table H. Then we go through A again for $j = 1, \ldots, n$ and check if $2A[j]$ can be found in the hash table. Similarly, any successful check indicates yes else we return no.

Correctness: Same as 1.1, the corresponding value 2α must be accessible in H.

Time complexity: For both iterations of A, we either hash or check if a value is in H which takes $O(1)$ *expected time*. Therefore the total complexity is $O(n)$ *expected time*.

1.3 **[8 marks]** Design an algorithm which counts how many distinct pairs of indices (i, j) where $1 \leq i < j \leq n$ satisfy both:

- $A[i] > A[j]$; and
- $A[i] + A[j] = x$.

In the worst-case, your algorithm must run in $O(n(\log n)^2)$ time.

We propose an solution that modifies the stand inversion counting algorithm.

Algorithm: When merging 2 sub-arrays in the standard inversion counting algorithm, let the sub-arrays that are being merged be $A_{\text{lo}}[1 \ldots k]$ and $A_{\text{hi}}[1 \ldots k]$. Then during the process of finding the inversion, for each instance where $A_{\text{hi}}[i] < A_{\text{lo}}[j]$, we binary search and count how many times we find $x - A_{\text{hi}}[i]$ within $A_{\text{lo}}[j \ldots k]$. The occurrence we have counted is then our final solution.

Correctness: As both sub-arrays are sorted, we know that if $A_{\text{hi}}[i] < A_{\text{lo}}[j]$ then all elements in $A_{\text{lo}}[j \ldots k]$ are part of an inversion with $A_{\text{hi}}[i]$. Then we only need to find if $x - A_{\text{hi}}[i]$ is within $A[j \ldots k]$ which we can binary search as $A_{\text{lo}}[1 \ldots k]$ is sorted.

Time complexity: For each pair of inversion, we require an additional $O(\log n)$ for binary search, this changes our recurrence form to $T(n) = 2T(n/2) + \Theta(n \log n)$ with $f(n) = n \log n$ and the critical exponent $c^* = \log 2 = 1$. We see that as

[A] $f(n) \neq O(n^{1-\epsilon})$,

[B] $f(n) \neq \Theta(n)$,

[C] $f(n) \neq \Omega(n^{1+\epsilon})$,

all three cases of the master theorem do not apply. We can expand our recurrence via

$$\begin{aligned} T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n \log n) &= 2\left(T\left(\frac{n}{4}\right) + \Theta\left(\frac{n}{2}\log\frac{n}{2}\right)\right) + \Theta(n \log n) \\ &= 2T\left(\frac{n}{4}\right) + 2\Theta\left(\frac{n}{2}\log\frac{n}{2}\right) + \Theta(n \log n) \end{aligned}$$

The pattern suggests for the depth of recursion being $m = \log(n)$,

$$\begin{aligned} T(n) =& 2^m T\left(\frac{n}{2^m}\right) + \sum_{k=0}^{m-1} 2^k \Theta\left(\frac{n}{2^k}\log\frac{n}{2^k}\right) \\ =& nT(1) + \sum_{k=0}^{m-1} 2^k \Theta\left(\frac{n}{2^k}\log\frac{n}{2^k}\right). \end{aligned}$$

By looking the second term of the previous expression, we get

$$\begin{aligned} \sum_{k=0}^{m-1} 2^k \Theta\left(\frac{n}{2^k}\log\frac{n}{2^k}\right) = \Theta\left(n\sum_{k=0}^{m-1}\log\frac{n}{2^k}\right) &= \Theta\left(n\sum_{k=0}^{m-1}\left(\log n - \log 2^k\right)\right) \\ &= \Theta\left(n(\log n)^2 - \log n \log 2\left(\sum_{k=0}^{m-1} k\right)\right) \\ &= \Theta\left(n(\log n)^2 - \log n\left(\frac{m(m-1)}{2}\right)\right) \\ &= \Theta\left(n(\log n)^2\right). \end{aligned}$$

Alternative time complexity We can also justify our complexity by recognizing that our recurrence forms a tree of height $k = \lfloor \log n \rfloor$ and for each level, it takes $O(n \log n)$ to compute all its merging (and searching) operations as the number of times we have to binary searches will not exceed the total length of $A_{\text{hi}}[1 \ldots k]$. This hence gives us the final complexity of $O(kn \log n) = O\left(n(\log n)^2\right)$.

We can alternatively solve the problem in $O(n \log n)$.

Algorithm

1. Form an array $A' = [(A[1], 1), (A[2], 2), \ldots, (A[n], n)]$, then sorting with MergeSort in ascending order of the first value of each element.
2. For each element i in A', binary search for $x - A[i]$, and if found, check whether it is an inversion using the original index of the value.
3. Return all the distinct pairs found in step 2.

Correctness

By creating and sorting the array A' that stores the index and value of each element in A

we can use binary search to find if an element exists such that $A[i] + A[j] = x$ for any i by searching for $A[j] = x - A[i]$. We can then check if it is an inversion in our original array by checking that original index (the second value stored in A'). Repeating this process for all i in A' then allow us to find all distinct pairs that satisfy the required conditions.

Time Complexity

In step 1, we generate a new array A' of size n based off of A, which takes $O(n)$ time. Also in step 1, we then use MergeSort on A', which costs $O(n \log n)$ time. In step 2 and 3, we execute a binary search on all n elements, so this takes $O(n) * O(\log n) = O(n \log n$ time. So in total, our algorithm runs in $O(n) + O(n \log n) + O(n \log n) = O(n \log n)$ time as required.

Question 2

2.1 [**6 marks**] Blake and Red each have a collection of n distinct Pokemon cards. Blake creates an array B containing the serial numbers of their cards, and Red creates an array R with the serial numbers of his. Blake wishes to know how many of their cards are *not* also owned by Red (that is, the size of $B \setminus R$).

For example, if Blake has cards with serial numbers $B = [7, 2, 1, 5]$ and Red has cards numbered $R = [2, 8, 7, 3]$, then there are 2 cards owned by Blake but not Red.

Design an algorithm which runs in $O(n \log n)$ time and determines how many cards are owned by Blake but not Red.

Algorithm: We start by sorting R in an ascending order using merge sort. Then for each element $B[i]$ for $i = 1, \ldots, n$, we binary search the value of $B[i]$ in R. The serial numbers of the desired cards are the values of $B[i]$ that does not exist in R.

Correctness: For each element $B[i]$ we iterate, we either have that $B[i] \in B \setminus R$ where the binary search will return false or $B[i] \in B \cap R$ where the binary search will return true. Both cases are covered by our solution.

Time complexity: For each element in B we expend $O(\log n)$ to binary search its value in R, this results in a total complexity of $O(n \log n)$.

2.2 [**4 marks**] Gerald has a class of k students, each of which has a collection of distinct Pokemon cards. One day, he asks each member of his class to bring their collection, sorted by serial number. The ith student has c_i cards, with their **sorted** serial numbers in an array $N_i[1 \ldots c_i]$. The total number of (not necessarily distinct) cards owned by the class is $S = \sum_{i=1}^{k} c_i$.

He wants to find all the distinct cards owned by the class (i.e. $\bigcup_{i=1}^{k} N_i$), but needs to do so in $O(S \log k)$ to finish the lesson in time. Gerald devises the following algorithm to do so:

> We create an array N^* to store the serial numbers of the unique cards owned by the class. We first put all the numbers from N_1 into N^*.
> Then, for each i from 2 to k, we merge N_i with N^*, only including one copy of duplicate cards. Since N_i and N^* are both sorted, we are guaranteed to see duplicate cards one after the other while merging. We then replace N^* with the merged array.
> The final array N^* contains the merged arrays $N_1, N_2, \ldots, N_k$ without duplicate serial numbers, i.e. all the distinct cards owned by the class.

This algorithm is correct, but unfortunately does not meet Gerald's $O(S \log k)$ time complexity requirement. Justify why the worst-case time complexity of Gerald's algorithm is slower than $O(S \log k)$.

Note that an example with a fixed size (i.e., fixed S or k) is **not** sufficient!

Suppose that all students have $c_i = m$ for all $i = 1 \ldots k$ cards and all those cards are unique. Then merging the collection N_i takes im many computations (i.e. $2m, 3m, \ldots, km$) as each merge requires us to sequentially iterate through all cards that are already in N^*.

For the sum of an arithmetic progression, we have

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

Then our total computation is

$$S_m = \sum_{i=1}^{k} mi = m \sum_{i=1}^{k} i = \frac{mk(k+1)}{2} = \Theta(k^2 m).$$

Then from the settings given by our question that $m = S/k$, we yield

$$\Theta(k^2 m) = \Theta\left(\frac{k^2 S}{k}\right) = \Theta\left(kS\right)$$

which is too slow.

2.3 [**10 marks**] Design an algorithm that finds the distinct cards owned by Gerald's class, and runs in $O(S \log k)$ time.

Algorithm:

1. Recursively find all the distinct cards in the first $k/2$ and the last $k/2$ students.
2. Merge the results to determine the distinct cards by checking through each card one by one in both halves.
3. Base case: for $n = 1$ students, the distinct cards are that of the one student

Correctness: The distinct cards of all the students are the distinct cards of the first half of the students and the second half of the students, so merging together our results maintains the validity. For a singular student, the distinct cards can only be the distinct cards of that singular student, so our base case is correct.

Time Complexity: At each layer in merging, each of the remaining cards are checked once, which has an $O(S)$ cost. Further, we half the number of sub-problems to solve at each step, so for k students, this will take $O(\log k)$ time. Thus our algorithm runs in $O(S \log k)$ time as required.

Question 3

3.1 Read about the asymptotic notation in the review material and determine if $f(n) = O(g(n))$ or $g(n) = O(f(n))$ or both (i.e., $f(n) = \Theta(g(n))$) or neither of the two, for the following pairs of functions.

[A] [**5 marks**]

$$f(n) = \log_2(n); \quad g(n) = \sqrt[5]{n}$$

[B] [**5 marks**]

$$f(n) = n^n; \quad g(n) = 2^{n \log_2(n^2)}$$

[C] [**5 marks**]

$$f(n) = n^{1+\cos(\pi n)}; \quad g(n) = n$$

You might find L'Hôpital's rule useful: if $f(x), g(x) \to \infty$ as $x \to \infty$ and they are differentiable, then

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}.$$

[A] We see that both $f(n)$ and $g(n)$ are differentiable. Also, as $n \to \infty$,

$$f(n) = \log_2(n) \to \infty \quad \text{and} \quad g(n) = \sqrt[5]{n} \to \infty.$$

Then we can use L'Hôpital's rule to check the ratio between $f(n)$ and $g(n)$ to see which function dominates as $n \to \infty$. Expanding our expression yields

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log_2 n}{\sqrt[5]{n}} = \lim_{n\to\infty} \frac{\frac{1}{n \ln 2}}{\frac{1}{5n^{4/5}}} = \lim_{n\to\infty} \frac{5}{n^{1/5} \ln 2} = 0$$

Thus, $\log_2 n = O(\sqrt[5]{n})$.

[B] We see that as $g(n)$ is 2 raised to power of multiples of $\log_2(n)$, we can manipulate and get

$$2^{n \log_2(n^2)} = 2^{2n \log_2(n)} = \left(2^{\log n}\right)^{2n} = (n^n)^2 = f(n)^2.$$

Thus it is clear that $n^n = O\left(2^{n \log_2(n^2)}\right)$.

[C] Note that for a periodic function like $\cos(n)$, we always have for all odd values of n we get

$$n^{1+\cos(\pi n)} = n^{1-1} = n^0 = 1$$

and for all even values of n we get

$$n^{1+\cos(\pi n)} = n^{1+1} = n^2.$$

Thus neither $f(n) = O(g(n))$ nor $g(n) = O(f(n))$ as $f(n)$ oscillates infinitely around (above and below) $g(n)$ as $n \to \infty$.

3.2 [**5 marks**] Given n strings, each of length at most m, a divide and conquer approach can be used to find the longest common prefix amongst the strings. For example, the longest common prefix amongst `apple`, `apply` and `apart` is `ap`. Song has provided an algorithm to do so below:

1. Recursively determine the longest common prefixes among the first $n/2$ words and the last $n/2$ words.
2. Merge the results to determine the longest common prefix between the two halves by scanning through character by character.
3. Base case: for $n = 1$, the longest common prefix is the entire string.

However, Song isn't sure about the time complexity of the above algorithm. In Big-Theta notation, explain and justify what the time complexity of the above algorithm is.

Without loss of generality we assume that all n strings have a length of m, then the step of our algorithm will have a time complexity of $\Theta(m)$. This gives us a recurrence form of $T(n) = 2T(n/2) + \Theta(m)$ and a recursion depth of $\lceil \log_2(n) \rceil$. At each level of the recursion, we have 2 times the amount of instances of the previous level with depth $\lceil \log_2(n) \rceil$ having n instances.

From the sum of a geometric progression, we have

$$\sum_{k=1}^{\infty} \frac{1}{2^k} = 1.$$

We can then count the total number of instances of our recursion across all depths as

$$D = 1 + 2 + 4 + \ldots + \frac{n}{4} + \frac{n}{2} = \sum_{k=1}^{\lceil \log_2(n) \rceil} \frac{n}{2^k} = n \left(\sum_{k=1}^{\lceil \log_2(n) \rceil} \frac{1}{2^k} \right) < n.$$

As each instance requires $\Theta(m)$ time to compute the longest common prefix, our final complexity is

$$\Theta(Dm) = \Theta(nm).$$