Due: Thursday, 4th April, 18:00 (AEDT)

Submission is through inspera. Your assignment will be automatically submitted at the above due date. If you manually submit before this time, you can reopen your submission and continue until the deadline.

If you need to make a submission after the deadline, please use this link to request an extension: https://www.cse.unsw.edu.au/ cs9020/extension_request.html. Unless you are granted Special Consideration, a lateness penalty of 5% of raw mark per 24 hours or part thereof for a maximum of 5 days will apply. You can request an extension up to 5 days after the deadline.

Answers are expected to be provided either:

- In the text box provided using plain text, including unicode characters and/or the built-in formula editor (diagrams can be drawn using the built-in drawing tool); or
- as a pdf (e.g. using LaTeX) – each question should be submitted on its own pdf, with at most one pdf per question.

Handwritten solutions will be accepted if unavoidable, but we don't recommend this approach as the assessments are designed to familiarise you with typesetting mathematics in preparation for the final exam and for future courses.

Discussion of assignment material with others is permitted, but the work submitted *must* be your own in line with the University's plagiarism policy.

Objectives and Outcomes

The aim of this assignment is to build your understand of recurrence, induction and the asymptotic analysis of algorithms. Most questions are presented at a highly abstract level so that the consequences are very general, and can be applied in a variety of situations: not just later on in the course, but also beyond. The specific motivation for each problem can be summarised as follows:

Problem 1: This question looks at the complexity of three algorithms for various matrix operations – two iterative algorithms for matrix addition and multiplication and one recursive algorithm for matrix multiplication. This recursive algorithm takes the first steps towards the Strassen Algorithm for matrix multiplication. In that algorithm, the recursive submatrices ($SW + TY$, etc) are cleverly computed via a different set of intermediate matrix calculations (e.g. one step is to calculate $(S + V)(W + Z)$) rather than the more direct approach we take here. The Strassen Algorithm is also closely related to the Karatsuba algorithm for integer multiplication – an elementary form of this is taught in primary school as the Box method.

Problem 2: In this question you are asked to work on a series of identities of gradually increasing complexity. The aim is to practice applying basic induction on algebraic expressions to prove properties of the natural numbers.

Problem 3: This question introduces a commonly used recurrent structure used in computer science, binary trees. The question will guide you through how you define a recursively defined mathematical structure along with functions on that structure, and how to use structural induction to reason about such structures and functions.

Problem 4: In this question you will be tested on your ability to solve a recurrence relation asymptotically. The goal is to test your ability to reason about asymptotic growths beyond the master theorem, and use the idea of asymptotics to solve relations that might be difficult to solve exactly.

After completing this assignment, you will:

- Be able to make rigorous arguments about the foundational structures used in discrete mathematics [All problems, particularly 1 and 3]
- Understand the fundamental Computer Science concepts of recursion and induction [All problems]
- Analyze the correctness and efficiency of algorithms [Problem 1 and 4]

Advice on how to do assignments

Collaboration is encouraged, but all submitted work must be done individually without consulting someone else's solutions in accordance with the University's "Academic Dishonesty and Plagiarism" policies.

- Assignments are to be submitted in inspera.
- When giving answers to questions, we always would like you to prove/explain/motivate your answers. You are being assessed on your understanding and ability.
- Be careful with giving multiple or alternative answers. If you give multiple answers, then we will give you marks only for your worst answer, as this indicates how well you understood the question.
- Some of the questions are very easy (with the help of external resources). You may make use of external material provided it is properly referenced[1] – however, answers that depend too heavily on external resources may not receive full marks if you have not adequately demonstrated ability/understanding.
- In the assignment specification, questions have been given an indicative difficulty level:

  This should be taken as a *guide* only. Partial marks are available in all questions, and achievable by students of all abilities.

[1] Proper referencing means sufficient information for a marker to access the material. Results from the lectures or textbook can be used without proof, but should still be referenced.

Specific advice on how to do *this* assignment

Problem 1

- For Question 1, you are asked for asymptotic upper bounds. You should aim to give the best upper bound you can derive as this indicates your level of ability and understanding. Sub-optimal upper bounds may still be eligible for full marks if optimisations fall outside the expected knowledge of the course; but sub-optimal bounds where there is a reasonable expectation that students can find improvements will only be eligible for partial marks.
- As an example, recall the following algorithm outline for `insertionSort` from lectures:

```
sort(L) :
  if L.isEmpty() :
    return  L
  else :
    L2 := sort(L.next)
    insert  L.data into  L2
    return L2
```

  In particular, the line:

```
    insert  L.data into  L2
```

  If you were asked to provide an asymptotic upper bound for the running time of this line, then an HD-level answer eligible for full marks might be:

> To insert `L.data` into `L2` we could iterate through `L2`, comparing each element of the list with `L.data`, until we find the correct place to insert it. In the worst case this will require checking every element of `L2`, which means at most $n - 1 = O(n)$ comparisons. Therefore an upper bound on the running time for this line would be $O(n)$.

  Note that since `L2` is sorted, it is possible to use binary search to find the correct place to insert `L.data`, and this gives a solution that will only require $O(\log n)$ comparisons. So the upper bound of $O(n)$ is sub-optimal; however as we don't assume knowledge of binary search, the well-reasoned solution given above would be eligible for full marks.

- As always, it is recommended that you justify your answer. For parts a) and b) a sensible way to show how you obtain the running time would be to use the process used in lectures – calculating the running time of each individual line and combining them all together. Textual descriptions for non-trivial running times are a good idea. It is also a good idea to annotate the algorithm to assist with referencing. Again, taking the example above, here is an HD-level answer calculating the recurrence relation for the running time of that algorithm:

```
sort(L) :
  if L.isEmpty() :                 O(1)
    return  L                      O(1)
  else :
    L2 := sort(L.next)             T(n − 1)
    insert  L.data into  L2        O(n)
    return L2                      O(1)
```

> Let the running time of the above algorithm on a list of length n be $T(n)$. Then:
>
> – Lines 1, 2 and 6 consist of a constant number of elementary operations, so have

running time $O(1)$ (as indicated)

- Line 4 involves a recursive call to `sort` with a list of length $n-1$. So it has running time $T(n-1)$.
- Line 5 has running time $O(n)$ as discussed above.
- The "if" part is only executed when `L` is the empty list (i.e. when $n = 0$); for all other n only the "else" part is executed.

Putting this all together gives the following recurrence relation for the running time:

$$T(0) = O(1) \qquad T(n) = O(1) + O(n) + T(n-1) = T(n-1) + O(n).$$

In inspera, the algorithms in the question are presented in tables. Copying and pasting the algorithms from the question to your solution will give you the algorithm in a table form making annotation easier

- For part c) you are only provided with a description of the algorithm idea. To provide a justification for your recurrence it is sufficient to identify what parts of the algorithm are contributing to the running time and what their algorithmic cost is. When the algorithm description is informal, you may have to work out a way to fill in some missing steps (e.g. the insert example above) but if this is necessary, a simple/obvious approach is acceptable.

Remark

The recursive algorithm takes the first steps towards the Strassen Algorithm for matrix multiplication. In that algorithm, the recursive submatrices ($SW + TY$, etc) are cleverly computed via a different set of intermediate matrix calculations (e.g. one step is to calculate $(S+V)(W+Z)$) rather than the more direct approach we take here.

Problem 2

- The problem encourages you to follow the typical layout for an inductive proof. You are asked to prove $P(n)$, $Q(n)$ and $R(n)$ for all $n \in \mathbb{N}$. To do so follow find an appropriate base case, for which proving the identity should be fairly simple, and then show that $P(n) \implies P(n+1)$.
- You will probably want to do the questions in order, as the answers found in the previous questions allow you to simplify the sums you will be working with in the subsequent ones. This is however not strictly necessary.
- There are multiple ways you can rearrange the sums you will obtain. In particular, for question $b)$, you may use the answer obtained in question $a)$ but it is also possible that you can take another route. It might be of use (in particular for question $c)$ to use the following identity (derived in the lectures starting from 0): $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- The goal of the question is to familiarise yourself with basic induction. Although you do not strictly need to use induction, and will get full marks for a non-inductive proof, you are encouraged to think about how you would use induction to work on such a problem, and how the proofs might differ whether you use induction or not. Often induction makes it simpler to find a formal proof, however you might find that using $a)$ to prove $b)$ without using induction is somewhat simpler.

- If you approach a question inductively, but find that you did not make use of your inductive hypothesis in your inductive step, then you are not actually using induction, so you might want to reframe your proof as a non-inductive proof.
- The question requires you to rearrange sums of terms. Remember that if you are given an expression of the form $\sum_{i=a}^{c}(f(i)+g(i))$ you can decompose it as $\sum_{i=a}^{b}(f(i)+g(i))+\sum_{i=b+1}^{c}(f(i)+g(i))$ for $a \leq b < c$ but also as $\sum_{i=a}^{c} f(i) + \sum_{i=a}^{c} g(i)$.

Remark

The identity you are asked to show in c) is actually quite famous and is known as Nicomachus's theorem.

As an example of how your inductive proof should look, lets rederive $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

- Let $P(n)$ be the proposition that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. We will show that $P(n)$ holds for all $n \in \mathbb{N}$ by induction.
- **Base case:** $n = 1$
$$P(1) = \sum_{i=1}^{1} i = 1 = \frac{(1)(1+1)}{2}$$
- **Inductive case:** $P(k) \implies P(k+1)$ Assume $P(k)$ holds. In other words, our inductive hypothesis is:
$$\left(\sum_{i=0}^{k} i\right) = \frac{k(k+1)}{2} + (k+1)$$

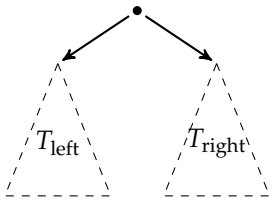
  To prove $P(k+1)$ we rearrange the sum:

$$\begin{aligned}\left(\sum_{i=1}^{k+1} i\right) &= \left(\sum_{i=1}^{k} i\right) + (k+1) \\ &= \frac{k(k+1)}{2} + (k+1) \text{ (By inductive hypothesis)} \\ &= \frac{k(k+1)+2(k+1)}{2} \\ &= \frac{(k+1)(k+2)}{2}\end{aligned}$$

  Proving $P(k+1)$.

Problem 3

- The abstract definition of a binary tree is similar to the abstract definition of a linked list – except we have two "tails", and no data item in the head. Visually, it may be useful to think of the recursive case as:



- An implementation of the functions count, leaves and internal in the programming language of your choice, rather than as an abstract mathematical function, is acceptable and will not incur any penalties. However, such an answer for (b) and (c) will make it almost impossible to provide a suitable answer for part (d) (see comments below).
- Here are two examples of recursively defined functions for computing the length of a linked list, and concatenating two linked lists (compare to the definition given in lectures of similar functions for words):

$$\mathsf{length}(L) = \begin{cases} 0 & \text{if } L \text{ is empty} \quad (\mathsf{length.B}) \\ 1 + \mathsf{length}(L') & \text{if } L = (d, L') \quad (\mathsf{length.R}) \end{cases}$$

$$\mathsf{concat}(L_1, L_2) = \begin{cases} L_2 & \text{if } L_1 \text{ is empty} \quad (\mathsf{concat.B}) \\ (d, \mathsf{concat}(L', L_2)) & \text{if } L_1 = (d, L') \quad (\mathsf{concat.R}) \end{cases}$$

- For a distinction level answer or higher, a short (one or two sentence) justification for correctness is expected. Here is an example of an HD-level answer giving a recursive definition for computing the **height** of a binary tree: the length of the longest path from the root to a leaf (and defined to be -1 for the empty tree):

  - For a non-empty tree $T = (T_{\text{left}}, T_{\text{right}})$, where T_{left} and T_{right} are not both τ, the longest path from the root, r, to a leaf will either consist of an edge from r to the longest path from root to leaf in T_{left}, or from r to the longest path from root to leaf in T_{right}. Therefore, $\mathsf{height}(T) = 1 + \max\{\mathsf{height}(T_{\text{left}}), \mathsf{height}(T_{\text{right}})\}$.
  - If $T = (\tau, \tau)$ then the root of T is also a leaf, so T has height 0, which, because $\mathsf{height}(\tau) = -1$, can also be expressed as $1 + \max\{\mathsf{height}(T_{\text{left}}), \mathsf{height}(T_{\text{right}})\}$.

  Therefore, the function height can be recursively defined as follows:

$$\mathsf{height}(T) = \begin{cases} -1 & \text{if } T = \tau \quad (\mathsf{height.B}) \\ 1 + \max\{\mathsf{height}(T_{\text{left}}), \mathsf{height}(T_{\text{left}})\} & \text{if } T = (T_{\text{left}}, T_{\text{right}}) \quad (\mathsf{height.R}) \end{cases}$$

- Instead of providing justification as above, it may be worth checking your answer against the example tree given in the assignment description. Indeed, such a check would demonstrate a level of understanding expected of a credit-level student, so partial marks are available for a check like the following:

  The longest path from root to leaf in the example tree has length 2, so the tree has height 2. We can see the correctness of the previously defined height function for this tree as

follows:

$$
\begin{array}{lrcll}
 & \mathsf{height}(\tau) & = & -1 & (\mathsf{height.B}) \\
\text{So,} \quad \mathsf{height}(T_3) = \mathsf{height}(T_4) = \mathsf{height}(T_5) & & = & 1 + \max\{\mathsf{height}(\tau), \mathsf{height}(\tau)\} & (\mathsf{height.R}) \\
 & & = & 1 + \max\{-1, -1\} & \\
 & & = & 0. & \\
\text{So,} & \mathsf{height}(T_2) & = & 1 + \max\{\mathsf{height}(T_5), \mathsf{height}(\tau)\} & (\mathsf{height.R}) \\
 & & = & 1 + \max\{0, -1\} & \\
 & & = & 1, & \\
\text{and} & \mathsf{height}(T_1) & = & 1 + \max\{\mathsf{height}(T_3), \mathsf{height}(T_4)\} & (\mathsf{height.R}) \\
 & & = & 1 + \max\{0, 0\} & \\
 & & = & 1. & \\
\text{So,} & \mathsf{height}(T) & = & 1 + \max\{\mathsf{height}(T_1), \mathsf{height}(T_2)\} & (\mathsf{height.R}) \\
 & & = & 1 + \max\{1, 1\} & \\
 & & = & 2. &
\end{array}
$$

- Even though the recursive definition of binary trees consist of one base case and one recursive case, some of the functions in parts (a)–(c) involve other cases: in particular additional base cases/special cases of the recursive case (it is not important which category it falls under).
- Part (d) is about proving the specified relationship between **the functions you have defined**, it is not about showing a particular property of binary trees (i.e. that the number of leaves is one more than the number of fully-internal nodes). In particular, your proof should be relying on the definitions of leaves and internal provided in parts (b) and (c) [and the recursive definition of binary trees on which these are based]. An induction proof based on "the number of nodes in the tree" is unlikely to be proving the correct result. As we are proving a result for an infinite set of structured objects, proof by **structural** induction is a recommended approach. Here is an example of an HD-level structural induction proof involving the functions length and concat defined above:

For a linked list L, let $P(L)$ be the proposition that:

$$\text{For any linked list } L': \quad \mathsf{length}(\mathsf{concat}(L, L')) = \mathsf{length}(L) + \mathsf{length}(L').$$

We will prove that $P(L)$ holds for all linked lists L by structural induction on L.

Base case: L is empty.
In this case we have, for any linked list L':

$$
\begin{array}{rcll}
\mathsf{length}(\mathsf{concat}(L, L')) & = & \mathsf{length}(L') & (\mathsf{concat.B}) \\
 & = & 0 + \mathsf{length}(L') & \\
 & = & \mathsf{length}(L) + \mathsf{length}(L') & (\mathsf{length.B})
\end{array}
$$

Therefore $P(L)$ holds when L is empty.

Inductive case: $L = (d, L_1)$ and $P(L_1)$ implies $P(L)$
Assume that $P(L_1)$ holds. That is:

$$\text{For any linked list } L': \quad \mathsf{length}(\mathsf{concat}(L_1, L')) = \mathsf{length}(L_1) + \mathsf{length}(L') \qquad (\mathrm{IH})$$

Then, for any linked list L' we have:

$$\begin{aligned}
\mathsf{length}(\mathsf{concat}(L, L')) &= \mathsf{length}\left((d, \mathsf{concat}(L_1, L'))\right) && (\mathsf{concat.R}) \\
&= 1 + \mathsf{length}(\mathsf{concat}(L_1, L')) && (\mathsf{length.R}) \\
&= 1 + \mathsf{length}(L_1) + \mathsf{length}(L') && (\mathrm{IH}) \\
&= \mathsf{length}\left((d, L_1)\right) + \mathsf{length}(L') && (\mathsf{length.R}) \\
&= \mathsf{length}(L) + \mathsf{length}(L')
\end{aligned}$$

So $P(L)$ holds.

Conclusion .
We have $P(L)$ holds when L is empty, and if $P(L_1)$ holds, then $P\left((d, L_1)\right)$ holds. Therefore, by structural induction, $P(L)$ holds for all linked lists L.

- Note that when using structural induction, the induction hypothesis is that the proposition holds *for all* smaller structures required to construct the complex structure. For example, when trying to prove $P(T)$ when $T = (T_{\text{left}}, T_{\text{right}})$ the standard inductive step would be to show:

  If $P(T_{\text{left}})$ *and* $P(T_{\text{right}})$ both hold, then $P(T)$ holds.

Problem 4

- In this problem you are asked to find the asymptotic solution of a recurrence relation. The master theorem works when you have a single recurrent term on the right-hand side of the equation, however in this relation you have several recurrent terms with different coefficients.
- A common way of solving such problems, is to guess the solution and substitute your guess in the equation and see if it holds.
- Since you are only asked to find asymptotic behaviour, your guess doesn't need to worry about details such as the exact values of coefficients.
- Here is an example of how you might analyse the asymptotic growth of a recurrence relation without using the master theorem:

  Given the recurrence relation $S(n) = S(\frac{n}{2}) + n$, show that it is in $O(n)$

  - To solve this without using the master theorem we will use induction to show that the relation is in $O(n)$.
  - To show this we will prove that for all n, our strategy will be to show that $S(n) \leq 3n + S(1)$. From there it is easy to show that if $c = 4$ and $n_0 = S(1)$ then for all $n > 1$ we have $S(n) \leq 3n + S(1) \leq 4n$ meaning $S(n) \in O(n)$.
  - **Base case:** $S(1) \leq 3n + S(1)$
  - **Inductive Hypothesis:** For all $m < n$, $S(m) \leq 3m + S(1)$.
  - **Inductive Step:** Given $n > 1$, $S(n) = S(\frac{n}{2}) + n$. Therefore we have $S(n) = S(\frac{n}{2}) + S(1) + n \leq \frac{3n}{2} + n + S(1) \leq \frac{5n}{2} + S(1) \leq 3n + S(1)$
  - Using our observation from the beginning we can now claim that $S(n) \in O(n)$.
  - Notice that we are using strong induction in this example.

- This example gives you some indications as to what your proof should look like. In the exercise however you might have to be more careful as some steps might not be as straightforward.

Remark

This recurrence relation is used in the analysis of several fundamental algorithms in computer science centred around the idea of median of medians.