Assignment (part 1)

Question 1:

Gale and Shapley published their paper on the Stable Matching Problem in 1962; but a version of their algorithm had already been in use for ten years by the National Resident Matching Program, for the problem of assigning medical residents to hospitals.

Basically, the situation was the following. There were m hospitals, each with a certain number of available positions for hiring residents. There were n medical students graduating in a given year, each interested in joining one of the hospitals. Each hospital had a ranking of the students in order of preference, and each student had a ranking of the hospitals in order of preference. We will assume that there were more students graduating than there were slots available in the m hospitals.

The interest, naturally, was in finding a way of assigning each student to at most one hospital, in such a way that all available positions in all hospitals were filled. (Since we are assuming a surplus of students, there would be some students who do not get assigned to any hospital.)

We say that the assignment of students to hospitals is **stable** if neither of the following situations arises.

First type of instability: There are students s and s', and a hospital h, so that:

- s is assigned to h; s' is assigned to no hospital; h prefers s' to s.

Second type of instability: There are students s and s', and hospitals h and h', so that:

- s is assigned to h; s' is assigned to h' hospital; h prefers s' to s; s' prefers h to h'.

Based on the above description of the stable matching problem, devise a similar problem with a Singapore context. The problem must have a practical usage in Singapore. Clearly describe the background of the problem and the justification why it is an important problem to be solved.

The following are some specifications of the problem:

- The problem must involve at least 2 different parties (e.g. hospital and student).
- Each party has at least 5 instances (e.g. 5 or more hospitals and 5 or more students)
- Each party has their preference list based on certain criteria (e.g. the lowest salary, the closest distance).

Note: Do not use the same problem we have from last cohort (i.e. matching of staff to different work sites).

Design and implement an algorithm to solve this problem. Explain in detail the algorithm and how it is used to fulfil the objective of both parties. Test the algorithm with at least 10 test cases. Report any constraint or limitation of the algorithm. Analyse the efficiency of your algorithm using the Big-oh notation.

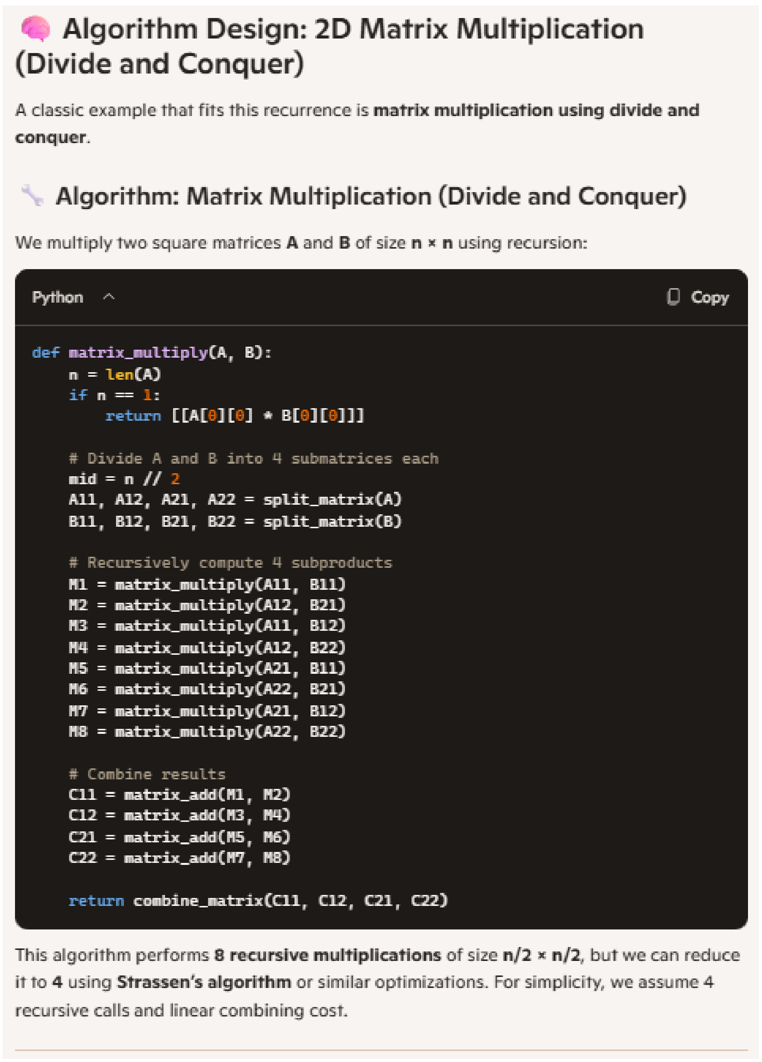
Assignment (part 1)

Question 2:

a) The complexity of an algorithm can be defined using asymptotic notations. The order of growth for an algorithm can be classified into constant, logarithmic, linear, linearithmic, quadratic, exponential and factorial. Explain in detail the characteristics of each type. Further elaborate logarithmic/lineararithmic and exponential/factorial with 2 algorithms to demonstrate its behavior. One algorithm showing logarithmic or lineararithmic order of growth. The other algorithm shows exponential or factorial order of growth.

Note: This must be a new algorithm which is not in our lectures or tutorials.

b) Recursion is a frequently used technique in solving complex problems. The derivation of its complexity is also more complicated than normal algorithm. For solving recurrence problem we can use the substitution, recursion-tree or the master method. Design and implement an algorithm which satisfies the time complexity T(n) = 4. T(n/2) + n. Explain your algorithm in detail and demonstrate how it satisfies the complexity with test data, substitution method, recursion-tree method and master method. Below is a sample output produced by copilot. You can use this as a reference to devise another problem with Singapore context. Clearly discuss the background of the problem and explain the practically of the problem.



Algorithm Design: 2D Matrix Multiplication (Divide and Conquer)

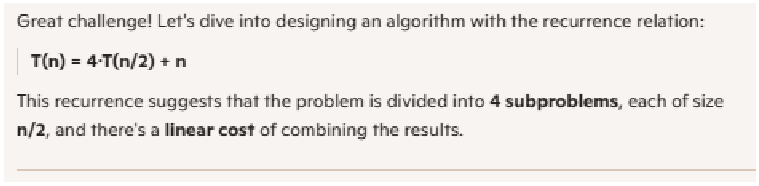
A classic example that fits this recurrence is **matrix multiplication using divide and conquer**.

Algorithm: Matrix Multiplication (Divide and Conquer)

We multiply two square matrices **A** and **B** of size **n × n** using recursion:

```python
def matrix_multiply(A, B):
    n = len(A)
    if n == 1:
        return [[A[0][0] * B[0][0]]]

    # Divide A and B into 4 submatrices each
    mid = n // 2
    A11, A12, A21, A22 = split_matrix(A)
    B11, B12, B21, B22 = split_matrix(B)

    # Recursively compute 4 subproducts
    M1 = matrix_multiply(A11, B11)
    M2 = matrix_multiply(A12, B21)
    M3 = matrix_multiply(A11, B12)
    M4 = matrix_multiply(A12, B22)
    M5 = matrix_multiply(A21, B11)
    M6 = matrix_multiply(A22, B21)
    M7 = matrix_multiply(A21, B12)
    M8 = matrix_multiply(A22, B22)

    # Combine results
    C11 = matrix_add(M1, M2)
    C12 = matrix_add(M3, M4)
    C21 = matrix_add(M5, M6)
    C22 = matrix_add(M7, M8)

    return combine_matrix(C11, C12, C21, C22)
```

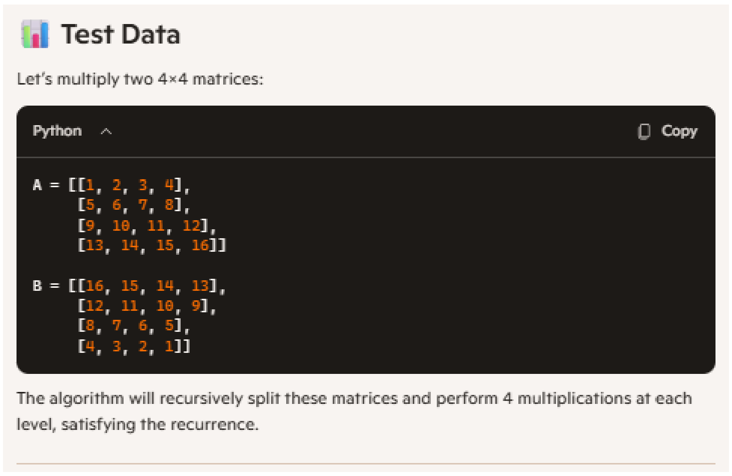
This algorithm performs **8 recursive multiplications** of size **n/2 × n/2**, but we can reduce it to **4** using **Strassen's algorithm** or similar optimizations. For simplicity, we assume 4 recursive calls and linear combining cost.



Great challenge! Let's dive into designing an algorithm with the recurrence relation:

T(n) = 4·T(n/2) + n

This recurrence suggests that the problem is divided into **4 subproblems**, each of size **n/2**, and there's a **linear cost** of combining the results.



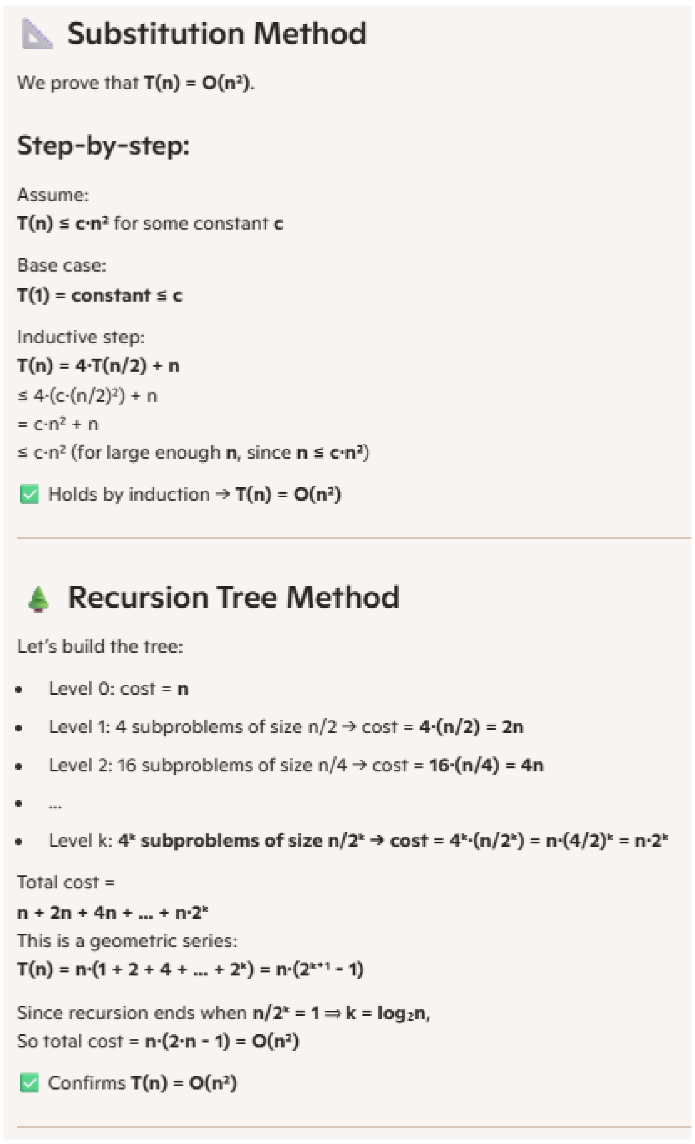
Test Data

Let's multiply two 4×4 matrices:

```python
A = [[1, 2, 3, 4],
     [5, 6, 7, 8],
     [9, 10, 11, 12],
     [13, 14, 15, 16]]

B = [[16, 15, 14, 13],
     [12, 11, 10, 9],
     [8, 7, 6, 5],
     [4, 3, 2, 1]]
```

The algorithm will recursively split these matrices and perform 4 multiplications at each level, satisfying the recurrence.

Assignment (part 1)



📐 Substitution Method

We prove that **$T(n) = O(n^2)$**.

Step-by-step:

Assume:
$T(n) \le c \cdot n^2$ for some constant **c**

Base case:
$T(1) = \text{constant} \le c$

Inductive step:
$T(n) = 4 \cdot T(n/2) + n$
$\le 4 \cdot (c \cdot (n/2)^2) + n$
$= c \cdot n^2 + n$
$\le c \cdot n^2$ (for large enough **n**, since **$n \le c \cdot n^2$**)

✅ Holds by induction → **$T(n) = O(n^2)$**

🌲 Recursion Tree Method

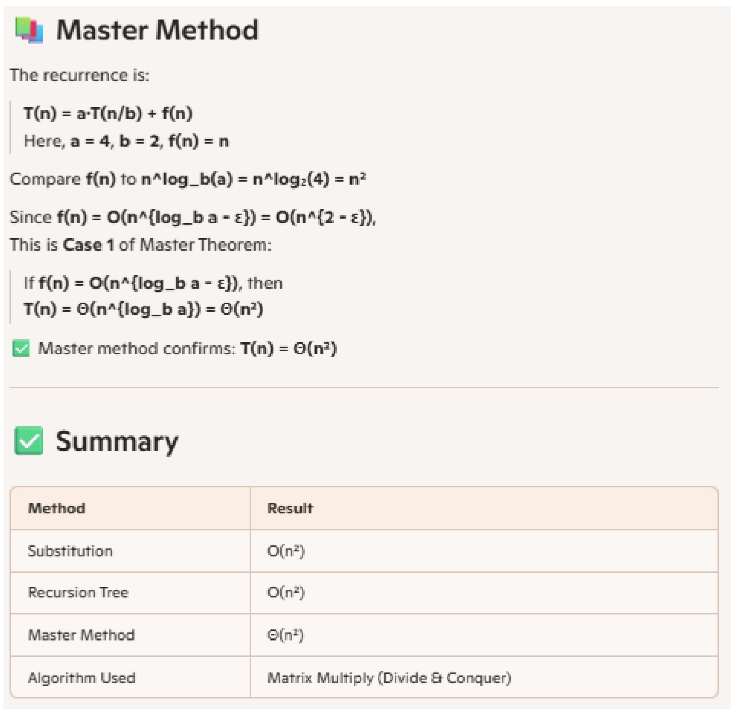
Let's build the tree:

- Level 0: cost = **n**
- Level 1: 4 subproblems of size $n/2$ → cost = **$4 \cdot (n/2) = 2n$**
- Level 2: 16 subproblems of size $n/4$ → cost = **$16 \cdot (n/4) = 4n$**
- ...
- Level k: **4^k subproblems of size $n/2^k$ → cost = $4^k \cdot (n/2^k) = n \cdot (4/2)^k = n \cdot 2^k$**

Total cost =
$n + 2n + 4n + \ldots + n \cdot 2^k$
This is a geometric series:
$T(n) = n \cdot (1 + 2 + 4 + \ldots + 2^k) = n \cdot (2^{k+1} - 1)$

Since recursion ends when **$n/2^k = 1 \Rightarrow k = \log_2 n$**,
So total cost = **$n \cdot (2 \cdot n - 1) = O(n^2)$**

✅ Confirms **$T(n) = O(n^2)$**



📚 Master Method

The recurrence is:

> **$T(n) = a \cdot T(n/b) + f(n)$**
> Here, **$a = 4$**, **$b = 2$**, **$f(n) = n$**

Compare **$f(n)$** to **n^log_b(a) = n^log₂(4) = n²**

Since **f(n) = O(n^{log_b a - ε}) = O(n^{2 - ε})**,
This is **Case 1** of Master Theorem:

> If **f(n) = O(n^{log_b a - ε})**, then
> **T(n) = Θ(n^{log_b a}) = Θ(n²)**

✅ Master method confirms: **$T(n) = \Theta(n^2)$**

✅ Summary

Method	Result
Substitution	$O(n^2)$
Recursion Tree	$O(n^2)$
Master Method	$\Theta(n^2)$
Algorithm Used	Matrix Multiply (Divide & Conquer)

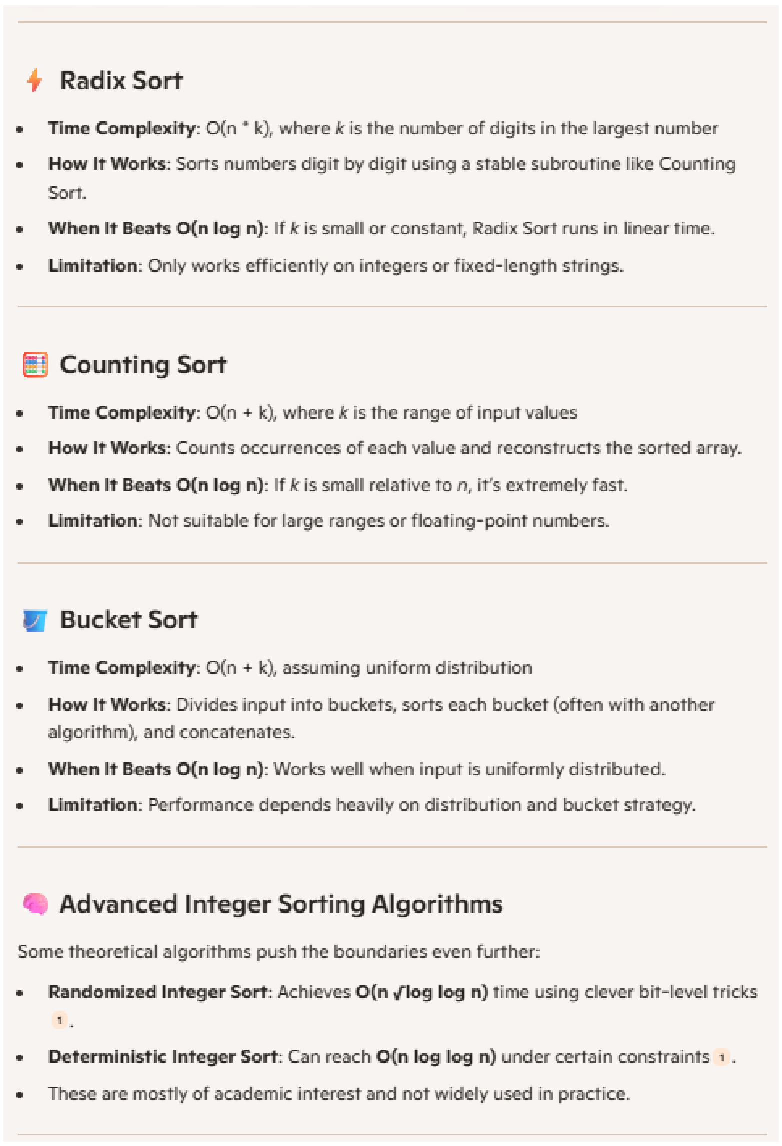
Assignment (part 1)

Question 3:

a) Explain what it means by stability in a sorting algorithm. Explain 2 situations why stability in sorting is desired. Implement and discuss 1 stable and 1 unstable sorting algorithm for a large data set of 1 million records. Proof your selection with test cases and a detailed discussion of the algorithm. Analyze the efficiency of your algorithm using the Big-oh notation.

b) Most of the sorting algorithms have a O(n^2) complexity. The best algorithm has achieved a O(n lg n) complexity. It is shown that sorting algorithm can never be better than O(n lg n). Research and discuss this statement in detail. Proof that this statement is true and under what condition this is true.

Despite the claim, we do have sorting algorithms which can achieve a better complexity than O(n lg n) as shown below via copilot. Implement one of these algorithms with test data to illustrate how the algorithm works and proof that it is better than O(n lg n).



Radix Sort

- **Time Complexity**: O(n * k), where *k* is the number of digits in the largest number
- **How It Works**: Sorts numbers digit by digit using a stable subroutine like Counting Sort.
- **When It Beats O(n log n)**: If *k* is small or constant, Radix Sort runs in linear time.
- **Limitation**: Only works efficiently on integers or fixed-length strings.

Counting Sort

- **Time Complexity**: O(n + k), where *k* is the range of input values
- **How It Works**: Counts occurrences of each value and reconstructs the sorted array.
- **When It Beats O(n log n)**: If *k* is small relative to *n*, it's extremely fast.
- **Limitation**: Not suitable for large ranges or floating-point numbers.

Bucket Sort

- **Time Complexity**: O(n + k), assuming uniform distribution
- **How It Works**: Divides input into buckets, sorts each bucket (often with another algorithm), and concatenates.
- **When It Beats O(n log n)**: Works well when input is uniformly distributed.
- **Limitation**: Performance depends heavily on distribution and bucket strategy.

Advanced Integer Sorting Algorithms

Some theoretical algorithms push the boundaries even further:

- **Randomized Integer Sort**: Achieves **O(n √log log n)** time using clever bit-level tricks [1].
- **Deterministic Integer Sort**: Can reach **O(n log log n)** under certain constraints [1].
- These are mostly of academic interest and not widely used in practice.

Assignment (part 1)

Required Documentation:

- For each question
    - Name of the student(s) complete this question.
    - Requirements/Specification: a paragraph giving a detailed account of what the program is supposed to do. State any assumptions or conditions on the format of input and expected output.
    - User Guide: instructions on how to compile, run and use the program.
    - Design and analysis: Outline the design of your program by describing your approach. Give a written description. Use diagrams or pseudo code, if required. Provide a detail space or/and time complexity analysis (big-oh) of the implemented algorithm.
    - Limitations: Describe program shortfalls (if any). For example, the features asked for but not implemented, the situations it cannot handle etc.
    - Testing: Describe your testing strategy (the more systematic, the better) and any errors noticed. Provide a copy of all your results of testing in this section.
    - Listings: Your source code must be included in this section.

Submission instruction:

Name your report as Txx-ID1,ID2,ID3,ID4,ID5,ID6 where x is the team number. Convert to a pdf format and submit it via the report submission tab.

Create a folder and name it as Txx-ID1,ID2,ID3,ID4,ID5,ID6. This folder will contain all your sources for each question. Compress the folder as a .zip file and submit it via the source submission tab.

Each student will need to complete a self and peer evaluation after the submission. More instruction will be given.

Please follows closely what are stated above. Invalid format will result in mark deduction.

Assignment (part 1)

Marking Guidelines:

Items	Poor	Attempted	Good	Excellent	Points
Design (algorithms). For non-programming question, this will be the accuracy of the answer.	Minimal design or inaccurate deign.	Valid design with clear explanation. Minor error.	Valid design with good explanation and justification with no error.	Excellent and efficient design with clear explanation and justification with no error. Consider alternative approaches.	20
Implementation (algorithms). For non-programming question, this will be the explanation and justification.	Minimal implementation or inaccurate result.	Correct result with good programming practice (e.g. well explained and comment)	Correct result with good programming practice. Some testing to ensure correctness of the code	Correct result with good programming practice. Proof of detail testing to ensure the correctness of all cases. Efficient code with extensive effort considering all scenarios.	20
Analysis (e.g. Big-O)	Results are presented. Little analysis is made.	Some analysis and justification with minor error.	Complete analysis and justification with no error.	In depth analysis with no error. Excellent arguments and justification.	20
Documentation	Minimal documentation and didn't follow submission instruction	Basic documentation with most of the required components and follow submission instruction partially.	Good documentation with well-structured report containing all the required components. Follow submission instructions fully.	Excellent and well-structured documentation with clear and accurate content beyond the requirement (e.g. references with comparative study). Follow submission instruction fully.	20
Presentation	Partial presentation with inaccurate explanation.	Full presentation with clear and accurate explanation in most parts.	Full presentation with clear and accurate explanation for all parts.	The Presentation is exceptional. The work is consistent and professional. Full and accurate explanation for all parts beyond the requirement.	15
Peer Evaluation	Absent	Present	Present	Present	5