Due 23:59 Jan 28 (Sunday). There are 100 points in this assignment. Submit your answers (**must be typed**) in pdf file to CourSys
`https://coursys.sfu.ca/2024sp-cmpt-307-d1/`.
Submissions received after 23:59 will get penalty of reducing points: 20 and 50 points deductions for submissions received at $[00:00, 00:10]$ and $(00:10, 00:30]$ of Jan 29, respectively; no points will be given to submissions after 00 : 30 of Jan 29.

In the assignment, $\log n = \log_2 n$.

1. 10 points (Ex. 1.2-2, 1.2-3 of text book)

   (a) For inputs of size n, assume insertion sort runs in $8n^2$ steps and merge sort runs in $64n \log n$ steps. For which value n does insertion sort run faster than merge sort on a same machine? (b) What is the smallest value of n such that an algorithm of running time $100n^2$ runs faster than an algorithm of running time 2^n on a same machine?

---

a) Insertion sort runs faster than merge sort for small n values:
- for n values 1-43 on a machine processing 1 million (10^6) instructions per second insertion sort runs faster

This test was conducted using the following equations to find running speed in seconds
- Insertion Sort: $\frac{8n^2}{10^6}$ seconds
- Merge Sort: $\frac{64n \log n}{10^6}$ seconds

b) The smallest n value for which $100n^2$ runs faster than 2^n is when n=15
- this is assuming a machine processing 1 million (10^6) instructions per second

This test was conducted using the following equations to find running speed in seconds
- Algorithm 1 ($100n^2$): $\frac{100n^2}{10^6}$ seconds
- Algorithm 2 (2^n): $\frac{2^n}{10^6}$ seconds

2. 10 points

   Suppose you have algorithms with the six running times listed below. (Assume these are the exact number of operations performed as a function of the input size n.) Suppose you have a computer that can perform 10^{10} operations per second, and you need to compute a result in at most an hour of computation. For each of the algorithms, what is the largest input size n for which you would be able to get the result within an hour?

   (1) n^2, (2) n^3, (3) $100n^2$, (4) $n \log n$, (5) 2^n, (6) 2^{2^n}.

The following are the maximum n values for which the algorithms run in at most an hour with a machine performing 10^{10} operations per second; using the equation $\frac{algorithm\ run\ time}{10^{10}}$ seconds

1. n^2: $n = 6.0 \cdot 10^6$ (six million)
2. n^3: $n = 32019$ (thirty-two thousand nineteen)
3. $100n^2$: $n = 6.0 \cdot 10^5$ (six-hundred thousand)

4. $n \log n$: $n = 9.06316 \cdot 10^{11}$ ($\approx$ nine-hundred billion)
5. 2^n: $n = 45$ (forty-five)
6. 2^{2^n}: $n = 5$ (five)

3. 20 points (Ex 2.1-4 of text book)

Consider the searching problem:

Input: A sequence of n numbers $A = (a_1, a_2, .., a_n)$ and a value v.
Output: An index i such that $v = A[i]$ or the special value nil if v is not in A.

Write pseudocode for linear search which scans through the sequence, looking for v. Using a loop invariant, prove your algorithm is correct. Make sure your loop invariant fulfills the three properties (initialization, maintenance, termination).

Pseudocode

```
Algorithm LinearSearch(A, v)
    for i from 1 to length(A)
        if A[i] == v
            return i
    return nil
```

Proof of Correctness using Loop Invariant
- Loop Invariant: at the start of each iteration, the value of v has not been found in previous elements of the sequence A[1...i-1]
- **Initialization**: Before the loop starts (when i=1), the sub-sequence A[1...i-1] is empty. Since there are no elements, v has not been found, so the loop invariant is trivially true.
- **Maintenance**: During the i^{th} iteration, the algorithm checks if A[i] is equal to v. If A[i] is not equal to v, the loop moves to the next iteration, and the invariant is maintained because v has still not been found in A[1...i]. If A[i] is equal to v, the loop terminates by returning i
- **Termination**: The loop terminates in one of two cases:
. 1. the value v is found at index i for which $v == A[i]$
. 2. the value v is not found at any index returning nil

4. 20 points (Ex 2.3-6 of text book)

Referring back to the searching problem (Ex 2.1-4), assume the sequence A is sorted, we can check the midpoint of A and eliminate half of the elements of A from further consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of A each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\log n)$.

Pseudocode

```
Algorithm BinarySearch(A, v)
    low = 1
    high = length(A)
    while low <= high
        mid = (low + high) / 2
        if A[mid] == v
            return mid
        else if A[mid] < v
            low = mid + 1
        else
            high = mid - 1
    return nil
```

Worst-case running time of binary search is $\Theta(\log_2 n)$
- Worst-case for Binary search is when element v is not in sequence A
- Since the problem size is reduced by half with each iteration this implies that the maximum number of operations is $\frac{n}{2}$; we represent this as $\log_2 n$
- Therefore, since each iteration takes $\theta(1)$ (constant time complexity for primitive operations) and the number of iterations is $\log_2 n$ the worse case running time is $\log_2 n$

5. 10 points (Problem 3-2 of text book)

Relative asymptotic growths: Indicate, for each pair of expressions $f(n)$ and $g(n)$ in the table below, whether $f(n)$ is O, o, Ω, ω, or Θ of $g(n)$. Assume that $k \geq 1$, $\epsilon > 0$, and $c > 1$ are constants. Your answer should be in the form of the table with "yes" or "no" written in each box. The answers for $f(n) = \log^k n$ and $g(n) = n^\epsilon$ are given in the 1st row of table as an example.

$f(n)$	$g(n)$	O	o	Ω	ω	Θ
$\log^k n$	n^ϵ	yes	yes	no	no	no
n^k	c^n	**yes**	**yes**	**no**	**no**	**no**
2^n	$2^{n/2}$	**no**	**no**	**yes**	**yes**	**no**
$n^{\log c}$	$c^{\log n}$	**yes**	**no**	**yes**	**yes**	**no**
$\log(n!)$	$\log(n^n)$	**yes**	**yes**	**no**	**no**	**no**

6. 10 points

Implement the conventional matrix multiplication and the fast matrix multiplication by Strassen. For $n = 2^i$, $i = 4, 5, 6, 7, 8, 9, 10$, report the running times of your programs for computing the product of two $n \times n$ matrices by the two methods. You can

use any programming language, but the same computing platform for both methods. You only need to report the running times, no need to submit the programs.

---

1. i=4:
- Conventional multiplication (n=16): 2.7505e-05 seconds.
- Strassen's multiplication (n=16): 0.00160062 seconds.

2. i=5:
- Conventional multiplication (n=32): 0.000202412 seconds.
- Strassen's multiplication (n=32): 0.011371 seconds.

3. i=6:
- Conventional multiplication (n=64): 0.00162077 seconds.
- Strassen's multiplication (n=64): 0.0783561 seconds.

4. i=7:
- Conventional multiplication (n=128): 0.0130656 seconds.
- Strassen's multiplication (n=128): 0.542735 seconds.

5. i=8:
- Conventional multiplication (n=256): 0.106665 seconds.
- Strassen's multiplication (n=256): 3.70416 seconds.

6. i=9:
- Conventional multiplication (n=512): 0.788366 seconds.
- Strassen's multiplication (n=512): 25.7389 seconds.

7. i=10:
- Conventional multiplication (n=1024): 8.57843 seconds.
- Strassen's multiplication (n=1024): 179.742 seconds.

7. 10 points

How would you modify Strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2? Show that the resulting algorithm runs in time $O(n^{\log 7})$.

Modifying Strassen's algorithm to multiply n×n matrices where n is not a power of 2 involves padding the matrices to the nearest size that is a power of 2

Modifying the Algorithm

1. Determine the next power of 2:
- for a given matrix since n, find the smallest integer m such that $m \geq n$ and m is a power of 2
- this can be done using the formula $m = 2^{log\_2n}$

2. Pad the matrices:
- extend both matrices to $m \times m$ by adding extra rows and columns filled with zeroes
- this will not change result of the multiplication but makes the matrices compatible for Strassen's algorithm

3. Apply Strassen's as normal
- apply Strassen's to the newly padded matrices

4. Extract result
- extract the top-left $n \times n$ sub-matrix as the final result
- this sub-matrix contains the product of the original matrices

Time Complexity Analysis
- The time complexity of Strassen's algorithm is $O(n^{\log_2 7})$
- when modifying the algorithm for matrices of size not a power of 2, the size becomes the next power of 2 (say m)
- Then the complexity is then $O(m^{\log_2 7})$

- since m is the smallest power of 2 greater than or equal to n, we have $m = O(n)$
- therefore, the time complexity of the modified algorithm is $O(n^{\log_2 7})$

8. 10 points (Ex 4.5-1 of text book)

Use the master method to give tight asymptotic bounds for the following recurrences.
(a) $T(n) = 2T(n/4) + 1$.
(b) $T(n) = 2T(n/4) + \sqrt{n}$.
(c) $T(n) = 2T(n/4) + n$.
(d) $T(n) = 2T(n/4) + n^2$.

---

a) a=2, b=4, c=1, k=0; since $a > b^k \rightarrow 2 > 4$ then $\theta(n^{\log_4 2}) = \theta(n^{1/2})$
b) a=2, b=4, c=1, k=1/2; since $a = b^k \rightarrow 2 = 4^{1/2}$ then $\theta(n^{1/2} \log_n)$
c) a=2, b=4, c=1, k=1; since $a < b^k \rightarrow 2 < 4$ then $\theta(n)$
d) a=2, b=4, c=1, k=2; since $a < b^k \rightarrow 2 < 4^2$ then $\theta(n^2)$

.

The above problem (Q8) was solved using the following:
The recurrence
- $T(n) = aT(n/b) + cn^k$ (where n = driving function)
- $T(1) = c$

where a, b, c, and k are all constants, solves to:
- $T(n) \in \theta(n^k)\ if\ a < b^k$
- $T(n) \in \theta(n^k \log n)\ if\ a = b^k$
- $T(n) \in \theta(n^{\log_b a})\ if\ a > b^k$