Homework 1

CSE 6140
Computational Science and Engineering

Problem 1

Algorithm	$n = 10$	$n = 100$	$n = 1000$
n^2	$10^2 = 10^2$	$100^2 = 10^4$	$1000^2 = 10^6$
n^3	$10^3 = 10^3$	$100^3 = 10^6$	$1000^3 = 10^9$
$100n^2$	$100 \times 10^2 = 10^4$	$100 \times 10^4 = 10^6$	$100 \times 10^6 = 10^8$
$n \log n$	$10 \log 10 \approx 10 \times 1 = 10$	$100 \log 100 \approx 100 \times 2 = 2 \times 10^2$	$1000 \log 1000 \approx 1000 \times 3 = 3 \times 10^3$
2^n	$2^{10} \approx 1.02 \times 10^3$	$2^{100} \approx 1.27 \times 10^{30}$	$2^{1000} \approx 1.07 \times 10^{301}$

(a)

(b)
- $n = 10$: $10! \approx 3.63 \times 10^6$
- $n = 50$: $50! \approx 3.04 \times 10^{64}$
- $n = 100$: $100! \approx 9.33 \times 10^{157}$

Problem 2

- $O(n^2)$
- $O(n \log n)$
- $O(n)$

Problem 3

(a) $O(100^n)$

(b) $O(\sqrt{2n})$

(c) Arranged in ascending order of running time:

$$O(\sqrt{2n}) \to O(n + 10) \to O(n^{2.5}) \to O(10^n) \to O(100^n)$$

(d) An algorithm with running time $O(n!)$ would be placed after $O(2^n)$

(e) An algorithm with running time $O(n^2 \log n)$ would be placed between $O(n^2)$ and $O(n^{2.5})$

Problem 4

(a) $2^{n+1} = O(2^n)$: True. The formal definition of O-notation states that $f(n) = O(g(n))$ if there exist constants c and n_0 such that for all $n > n_0$, $f(n) \leq c \cdot g(n)$. In this case, $2^{n+1} = 2 \times 2^n$, so we can choose $c = 2$ and $n_0 = 1$ to satisfy the condition.

(b) $2^{2n} = O(2^n)$: False. $2^{2n} = (2^n)^2$, which grows exponentially faster than 2^n.

Problem 5

(a) If we examine this in the context of the Gale-Shapley algorithm, we can see that strong instability does not exist for any perfect matching that the algorithm produces. The Gale-Shapley algorithm ensures that each man is paired with his best possible partner under the condition that the pairing is stable.

1. Men propose in order of their preferences. If a woman prefers a man who proposes to her, she will accept the proposal, but only tentatively, as she may receive a better proposal later.
2. No man is paired with a woman he prefers less than another available option: If a man is rejected by a woman, it means she prefers her current match or will receive a better one in the future.
3. No woman prefers a man who prefers her over his current match: If a woman ends up with a partner, it is guaranteed that no other man who prefers her is left unmatched, ensuring the absence of strong instability.

Thus, the Gale-Shapley algorithm guarantees a perfect matching with no strong instability, even with ties in preferences.

(b) For weak instability, the situation is more complex. While the Gale-Shapley algorithm guarantees no strong instability, it does not guarantee the absence of weak instability when preferences include ties. Here is why:

The Gale-Shapley algorithm resolves ties arbitrarily during the proposal phase. A tie might result in a man m being paired with a woman w', while another woman w might be indifferent between m and her current match m'. This would result in a weak instability since m prefers w, and w is indifferent between m and her current partner m'.

As an example, consider the following preference lists for 2 men and 2 women, where ties are allowed:

- Men's preferences:
  - m_1: $w_1 > w_2$
  - m_2: $w_1 > w_2$
- Women's preferences:
  - w_1: $m_1 > m_2$
  - w_2: $m_1 > m_2$

Now, let's say that the Gale-Shapley algorithm produces the following matching:

- m_1 is paired with w_1
- m_2 is paired with w_2

This matching is weakly unstable because:

- m_2 prefers w_1 over his current partner w_1, and
- w_2 prefers m_1 over her current partner m_2.

Thus, there can exist weak instability even after running the Gale-Shapley algorithm. As a result, weak instability can be unavoidable in some cases with ties in preferences.

Problem 6

```
for i = 1 to n-1:
    min_index = i
    for j = i+1 to n:
        if A[j] < A[min_index]:
            min_index = j
    swap A[i] with A[min_index]
```

The algorithm runs for only the first $n-1$ elements because after $n-1$ iterations, the largest element will already be in the correct position, so no further swaps are necessary.

- **Worst-case time complexity:** $O(n^2)$, since we have two nested loops
- **Best-case time complexity:** Still $O(n^2)$, as we must compare every element even in the best case where the array is already sorted

Problem 7

The running time of Breadth-First Search (BFS) using an adjacency matrix is $O(n^2)$ because for each node, we must examine all n possible connections to other nodes.

Problem 8

Pseudocode for Virus Spread Algorithm:

```
def virus_spread(grid):
    m = number of rows in grid
    n = number of columns in grid

    days = [-1 for _ in range(m * n)]
    queue = empty queue
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    for i = 0 to m-1:
        for j = 0 to n-1:
            if grid[i][j] == 2:
                index = i * n + j
                queue.push((i, j, 0))
                days[index] = 0

    while queue is not empty:
        (x, y, current_day) = queue.pop()

        for direction in directions:
            new_x = x + direction[0]
            new_y = y + direction[1]

            if 0 <= new_x < m and 0 <= new_y < n and grid[new_x][new_y] == 1:
                grid[new_x][new_y] = 2
                new_index = new_x * n + new_y
                days[new_index] = current_day + 1
                queue.push((new_x, new_y, current_day + 1))

    return days
```

The correctness of the algorithm relies on using Breadth-First Search to simulate the virus spreading process across the grid. BFS is the appropriate choice here because it explores all neighboring towns in layers, ensuring that towns are infected in the shortest possible time. Starting from all initially infected towns (represented by 2s), the algorithm processes each infected town day by day, spreading the infection to adjacent uninfected towns (1s). Each time an uninfected town is infected, it is marked with the current day count plus one, ensuring that it accurately tracks the number of days until infection. The use of a queue ensures that all closer towns are processed before more distant ones, so that each town is infected at the earliest possible time. The grid is traversed once, and the number of days for each town is recorded in a 1D array. If a town never gets infected, its corresponding index remains -1, ensuring all cases are handled correctly. Thus, the algorithm efficiently and correctly computes the days until each town becomes infected or remains uninfected.