COSC230 Practical 2: Computational Complexity of Algorithms (Practical for Week 3)

The purpose of this practical is to apply techniques for finding the computational complexity of algorithms. Asymptotic complexity is used to estimate time efficiency, that is: how does the time taken for an algorithm to execute vary with the amount of input data it processes?

Question 1: **Asymptotic Behaviour of Functions**

(a) Use Θ notation to show that for real x,

$$2x^2 + 100x + 10,$$

is $\Theta(x^2)$.

(b) If the time taken for an algorithm to execute is proportional to $2n^2 + 100n + 10$; where n is the number of input data points, compare how much the execution time increases in going from one data point to two data points. How about in going from 5000 data points to 10000 data points? Explain why the two comparisons are different.

Question 2: **Finding Asymptotic Complexity of `C++` Code**

In the following questions determine the complexity of each algorithm implementation, and give asymptotic complexity using the results presented in lectures for polynomial orders. Finally, test your predictions by timing each implementation in `C++` code for different-sized input data. Timing can be done using the following code:

```
#include <ctime>
int main()
{
    clock_t start = clock();

    //Code to be timed goes here...

    clock_t end = clock();
    double time_sec = (end - start)/(double) CLOCKS_PER_SEC;

    return 0;
}
```

(a) Given a double precision array `a[n]` of size `n`, sum the numbers in the array:

```
double sum = 0.0;
for (int i = 0; i != n; ++i) {
    sum += a[i];
}
```

(b) Given two double precision square matrices `a[n][n]` and `b[n][n]` of size `n`×`n`, multiply them together:

```
double c[n][n];
for (int i = 0; i != n; ++i) {
    for (int j = 0; j != n; ++j) {
        for (int k = 0; k != n; ++k) {
            c[i][j] += a[i][k]*b[k][j];
        }
    }
}
```

(c) Given a double precision square matrix `a[n][n]` of size `n`×`n`, find its transpose:

```
double temp;
for (int i = 0; i != n-1; ++i) {
    for (int j = i+1; j != n; ++j) {
        temp = a[i][j];
        a[i][j] = a[j][i];
        a[j][i] = temp;
    }
}
```

Question 3: **Average-Case Analysis**

Perform an average-case analysis of `sequential search` where the key is always found at either end of the array with equal probability. How could you modify `sequential search` to take advantage of this fact, and what result would the new average-case analysis lead to? (Hint: Use indicator random variables as in the lectures)

Solutions

Question 1:

(a) Show Θ by showing both Ω and O. Starting with Ω, we have that:

$$2x^2 \leq 2x^2 + 100x + 10 \qquad \text{for } x > 0.$$

For $\Omega(x^2)$, we need to show:

$$Ax^2 \leq 2x^2 + 100x + 10 \qquad \text{for } x > a.$$

Therefore, choosing $A = 2$ and $a = 0$ shows that $2x^2 + 100x + 10$ is $\Omega(x^2)$.

Now moving to O: For $x > 1$, we have that $x^2 > x$ (by multiplying both sides of $x > 1$ by x), and so $x^2 > x > 1$. These inequalities allow us to write:

$$2x^2 + 100x + 10 \leq 2x^2 + 100x^2 + 10x^2 \qquad \text{for } x > 1,$$

giving,

$$2x^2 + 100x + 10 \leq 112x^2 \qquad \text{for } x > 1.$$

For $O(x^2)$, we need to show:

$$2x^2 + 100x + 10 \leq Bx^2 \qquad \text{for } x > b.$$

Therefore, choosing $B = 112$ and $b = 1$ shows that $2x^2 + 100x + 10$ is $O(x^2)$.

For $\Theta(x^2)$, we need to show:

$$Ax^2 \leq 2x^2 + 100x + 10 \leq Bx^2 \qquad \text{for } x > k.$$

Therefore, from Ω and O, choosing $A = 2$, $B = 112$, and $k = 1$ shows that $2x^2+100x+10$ is $\Theta(x^2)$.

(b) In going from one data point to two data points the algorithm execution time increases by

$$\frac{2(2)^2 + 100(2) + 10}{2(1)^2 + 100(1) + 10} = 1.9464 < 2.$$

In going from 5000 data points to 10000 data points the algorithm execution time increases by

$$\frac{2(10000)^2 + 100(10000) + 10}{2(5000)^2 + 100(5000) + 10} = 3.9802 \approx 4.$$

In the first case, doubling the input data almost doubles the algorithm execution time. In the second case, doubling the input data almost quadruples the algorithm execution time. The key point is that asymptotic complexity has not been achieved until the input gets to tens of thousands of data points in this case.

Question 2:

(a) In this example the inner-most loop iterates n times, performing updates of `sum` and `i` each time. Using the general theorem on polynomial orders therefore gives an asymptotic complexity of $\Theta(n)$.

(b) In this example the three loops are independent, so the asymptotic complexity is therefore $\Theta(n^3)$.

(c) In this example, the loops are not independent. The outer loop is executed $n-1$ times (the loop counter `i` runs from `0` to `n-2`). Each of those times the inner loop is executed from `i+1` to `n-1`. The total number of iterations of the inner loop is therefore found from the following table:

`i`	`j`	times
`0`	`1` to `n-1`	`n-1`
`1`	`2` to `n-1`	`n-2`
`2`	`3` to `n-1`	`n-3`
...	...	...
`n-3`	`n-2` to `n-1`	`2`
`n-2`	`n-1` to `n-1`	`1`

Summing together the terms in the "times" column gives the total number of iterations of the inner loop as $n(n+1)/2-n = n(n-1)/2$. The asymptotic complexity is therefore $\Theta(n^2)$.

Timing each algorithm implementation in `C++`, and comparing for different array sizes in a similar manner to Question 1 part (b), should approximately agree with each asymptotic complexity result if large enough arrays are declared (but not so large that memory is exhausted: you are stack-allocation-limited for static arrays). Confirm this for yourself.

Question 3:

In lectures, we defined the random variable X to be the number of array elements searched to find the key. This can be written as a sum of indicator random variables: $X = X_1+X_2+...+X_n$; where $X_1 = 1$ is the event that the first array element is searched before the key is found and `sequential search` terminates, $X_2 = 1$ is the event that the second array element is

searched, and more generally $X_i = 1$ is the event that the ith array element is searched. Note that $X_i = 1$ also implies that the first i elements were searched due to the operation of `sequential search`. If the key is always found at either end of the array with equal probability, then $\mathbb{P}(X_1 = 1) = 1$ and $\mathbb{P}(X_i = 1) = 1/2$ for $1 < i \leq n$. The reason is that the operation of `sequential search` means the first array element will always be searched (i.e., with probability 1). Subsequent elements will then only be searched if the key was not found in the first element. The probability of this event (i.e., the key was not found in the first element) is 1/2. This leads to:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}[X_1] + \mathrm{E}[X_2] + ... + \mathrm{E}[X_n], \\
&= 1 + \frac{1}{2} + ... + \frac{1}{2}, \\
&= 1 + \frac{n-1}{2}, \\
&= \frac{n+1}{2}.
\end{aligned}$$

This is the same result we found for a key that had uniform probability of being in any element of the array. The reason can be understood by taking the average of the number of sequential searches required when finding the key in the first element, and finding the key in the nth element,

$$\frac{1+n}{2}.$$

This suggests a better algorithm would be to modify `sequential search` to only focus on the first and last elements. With this modification, our average-case analysis becomes:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}[X_1] + \mathrm{E}[X_2], \\
&= 1 + \frac{1}{2}, \\
&= \frac{3}{2},
\end{aligned}$$

where $X_2 = 1$ is now the event that the second search is the last element of the array. This result can be understood as the average of performing one search, and two searches,

$$\frac{1+2}{2}.$$