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? The readings for this practical are taken from Chapter 2 of the prescribed textbook: *Data Structures and Algorithms in `C++`*, Fourth Edition, Adam Drozdek.

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 by counting the number of elementary operations (in this case assignments) performed. This complexity will take the form of a polynomial expression. Then give asymptotic complexity using the general theorem on 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: **Finding Average Case Complexity**

Consider sequentially searching an unordered array to find a number. The best case is when the number is in the first cell: only one comparison is required, and $\Theta(1)$ in the best case. The worst case is when the number is in the last cell, or is not in the array at all: n comparisons are then required, and $\Theta(\mathrm{n})$ in the worst case. The average case when there is an equal chance for the number to be in any cell (i.e., with probability $\frac{1}{n}$) is $\frac{n+1}{2}$. This is the *expectation* of the total number of comparisons required to find the number. Notice that this is approximately half the number of comparisons required in the worst case.

Now find the average case when the probability of the number being in the *last* cell is $\frac{1}{2}$, the probability of the number being in the next-to-last cell is $\frac{1}{4}$, and the probability of the number being in any of the remaining cells is $\frac{1}{4(n-2)}$ (i.e., the total probability must sum to one).

Question 4: **Finding Amortized Complexity**

Amortized complexity involves analysing complexity over a sequence of operations instead of only single operations. This is important, as one operation may perform modifications that change the run time of the next operation in a sequence. In this question, consider the process of incrementing a binary n-bit counter for $n \leq 4$. An increment causes some bits to be flipped: some 0s are changed to 1s, and some 1s to 0s. Sometimes, only one bit is flipped: for example, when 000 is incremented to 001. Other times, all bits are flipped: for example, when 011 is incremented to 100. A table for $n = 3$ is as follows:

Number	Flipped Bits
000	
001	1
010	2
011	1
100	3
101	1
110	2
111	1

In the worst case, the number of bits needing to be flipped to increment a counter through $m = 2^n - 1$ numbers would not be more than mn, leading to a complexity of $\Theta(mn)$ in the worst case. For the 3-bit counter worst-case would equate to $7 \times 3 = 21$, while the real case is 11, as shown in the table. Show that the amortized complexity of incrementing a counter through m numbers is $\Theta(m)$.

Solutions

Question 1:

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

$$\begin{aligned} 2x^2 &\leq 2x^2 + 100x + 10 && \text{for } x > 0, \\ \text{and } 2|x^2| &\leq |2x^2 + 100x + 10| && \text{for } x > 0. \end{aligned}$$

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

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

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:

$$\begin{aligned} 2x^2 + 100x + 10 &\leq 2x^2 + 100x^2 + 10x^2 && \text{for } x > 1, \\ \text{so } |2x^2 + 100x + 10| &\leq |2x^2 + 100x^2 + 10x^2| && \text{for } x > 1, \end{aligned}$$

giving,

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

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

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

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:

$$A|x^2| \leq |2x^2 + 100x + 10| \leq B|x^2| \qquad \text{for } x > k.$$

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 number of assignments can be counted as follows. Firstly, the two variables `sum` and `i` are initialized, giving two assignments. The `for` loop then executes n times, updating `sum` and `i` each time, giving `2n` assignments. Therefore, there are exactly $2n + 2$ assignments in this example. Using the general theorem on polynomial orders (i.e., we don't have to explicitly show this as we did in Question 1) gives asymptotic complexity $\Theta(n)$.

(b) In this example, the loops are independent, so there are n^3 assignments. 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. Each of those times the inner loop is executed from $i+1$ to $n-1$ and three assignments are made in the inner loop (only the inner loop assignments contribute to asymptotic complexity). Therefore, there are $3(n-1)$ assignments for $i = 0$, $3(n-2)$ assignments for $i = 1$,..., and 3 assignments for $i = n-2$. The total number of assignments can therefore be written as $\sum_{j=1}^{n-1} 3j = 3\sum_{j=1}^{n-1} j$. Finding that $\sum_{j=1}^{n-1} j = \frac{n}{2}(n-1)$ from summing the arithmetic sequence, we have the total number of assignments as $3\frac{n}{2}(n-1) = \frac{3}{2}n^2 - \frac{3}{2}n$. 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:

The expectation of the total number of comparisons is:

$$E = \sum_{i=1}^{n} \text{comparisons}(i) \times \text{probability}(i).$$

Since there is one comparison to test the first element, two comparisons to test the first two elements, and i comparisons to test the first i elements: comparisons$(i) = i$. Also, probability$(n) = 1/2$, probability$(n-1) = 1/4$, and probability$(i) = 1/(4(n-2))$ for any other value of i. Therefore,

$$\begin{aligned} E &= \frac{n}{2} + \frac{n-1}{4} + \frac{n-2+n-3+...+2+1}{4(n-2)}, \\ &= \frac{n}{2} + \frac{n-1}{4} + \frac{(n-1)(n-2)}{8(n-2)}, \\ &= \frac{n}{2} + \frac{n-1}{4} + \frac{(n-1)}{8}, \\ &= \frac{7n-3}{8}. \end{aligned}$$

Question 4:

To help answer this question we construct the following table for $n = 3$:

Number	Flipped Bits(Cost)	Amortized Cost	Units Left
000			
001	1	1	0
010	2	1	-1
011	1	1	-1
100	3	1	-3
101	1	1	-3
110	2	1	-4
111	1	1	-4

An amortized cost of 1 flip per increment quickly gets “Units Left” into negative values. How about an amortized cost of 2 flips per increment? Now the table becomes:

Number	Flipped Bits(Cost)	Amortized Cost	Units Left
000			
001	1	2	1
010	2	2	1
011	1	2	2
100	3	2	1
101	1	2	2
110	2	2	2
111	1	2	3

From this table we can see that an amortized cost of 2 flips per increment keeps "Units Left" in positive values for the 3-bit counter. A similar analysis for the 4-bit counter is show below:

Number	Flipped Bits(Cost)	Amortized Cost	Units Left
0000			
0001	1	2	1
0010	2	2	1
0011	1	2	2
0100	3	2	1
0101	1	2	2
0110	2	2	2
0111	1	2	3
1000	4	2	1
1001	1	2	2
1010	2	2	2
1011	1	2	3
1100	3	2	2
1101	1	2	3
1110	2	2	3
1111	1	2	4

In each case "Units Left" gets down to 1, but never becomes negative. It also never goes above n, which would be wasteful. These results suggest that we have an amortized complexity of $2m$, or $\Theta(m)$, for m increments of a binary n-bit counter with $n \leq 4$. For the 3-bit counter this equates to $2 \times 7 = 14$, which over-counts the actual case of 11 but is a closer estimate than the worst case of 21.