This chapter reviews the fundamentals of algorithmic complexity which will be later used to study the various algorithms for networks. The chapter presents the notation for expressing complexity and then presents the theory of languages and machines so that complexity can be stated in terms of Turing machines.

2.1 Complexity Notation

Complexity of an algorithm is analyzed in terms of time complexity and space complexity. Time complexity is obtained in terms of number of computations required, and space in terms of how much memory is required. We are interested not in the exact computation cost, but only the asymptotic order of complexity. It will be assessed as the size of the problem goes to infinity and we want to find out the dominating term of the complexity.

Definition 2.1. Big-O: The function \( f : \mathbb{N} \rightarrow \mathbb{R} \) has order at most \( g \), i.e. \( f(n) \in O(g(n)) \) if \( \exists M > 0 \) such that \( \exists N > 0, N \in \mathbb{N} \) so that \( \forall n > N, |f(n)| \leq M|g(n)| \).

Definition 2.2. Big-Ω: The function \( f : \mathbb{N} \rightarrow \mathbb{R} \) has order at least \( g \), i.e. \( f(n) \in \Omega(g(n)) \) if \( \exists M > 0 \) such that \( \exists N > 0, N \in \mathbb{N} \) so that \( \forall n > N, |f(n)| \geq M|g(n)| \).
CHAPTER 2. COMPLEXITY ANALYSIS

Definition 2.3. Big-$\Theta$: The function $f : \mathbb{N} \to \mathbb{R}$ has order the same as $g$, i.e. $f(n) \in \Theta(g(n))$ if $\exists M_1 > 0, M_2 > 0$ such that $\exists N > 0, N \in \mathbb{N}$ so that $\forall n > N, M_1 |g(n)| \leq |f(n)| \leq M_2 |g(n)|$.

Definition 2.4. Little-O: The function $f : \mathbb{N} \to \mathbb{R}$ has order much smaller than $g$, i.e. $f(n) \in o(g(n))$ if $\forall \epsilon > 0, \exists N > 0, N \in \mathbb{N}$ such that $\forall n > N, |f(n)| \leq \epsilon |g(n)|$.

The definition for Big-O implies that asymptotically the ratio of the two functions is finite. More precisely,

$$f(n) \in O(g(n)) \Rightarrow \limsup_{n \to \infty} \left| \frac{g(n)}{f(n)} \right| < \infty \quad (2.1)$$

The definition for Little O implies that asymptotically the ratio of the two functions is zero. More precisely, if $\exists N > 0$, such that $\forall n > Ng(n) \neq 0$, then

$$f(n) \in o(g(n)) \Rightarrow \lim_{n \to \infty} \left| \frac{g(n)}{f(n)} \right| = 0 \quad (2.2)$$

The technique to manipulate the complexity orders is to only consider the highest order terms in the polynomials, and also to ignore the coefficients. For instance we can show from the definitions that the following are true.

Example 2.5.

$$O(n^3 + 3n^2 - n + 10) = O(n^3)$$

$$O(5n^3) = O(n^3)$$

$$O(n \log n + 5n) = O(n \log n)$$

△

Let us look at some details of this analysis. We will consider the function $f(n) = 5n^3 + 3n^2 - n + 10$. In order to compare its order with that of $g(n) = n^3$, we take the ratio and the limit and obtain

$$\lim_{n \to \infty} \frac{5n^3 + 3n^2 - n + 10}{n^3} = 5 < \infty \quad (2.3)$$

Now if we take function $h(n) = n^4$, we obtain for the order of $f(n)$

$$\lim_{n \to \infty} \frac{5n^3 + 3n^2 - n + 10}{n^4} = 0 < \infty \quad (2.4)$$
This shows that \( f(n) \in O(g(n)) \), \( f(n) \in O(h(n)) \), and we also have \( f(n) \in o(h(n)) \), but \( f(n) \notin o(g(n)) \). In fact what we see is that \( \forall f, g, f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n)) \), but not the other way around necessarily because for the Little-o, the constant must be zero for the function also to belong to Big-O.

**Example 2.6.** We are given the following functions.

\[
\begin{align*}
  f_1(n) & = 5n^3 + 3n^2 - n + 10 \\
  f_2(n) & = 5n^4 + n \\
  f_3(n) & = 10n^3 + 5n^2
\end{align*}
\]

For these functions we have

\[
\begin{align*}
  f_1(n) & \in o(f_2(n)) \\
  f_1(n) & \in O(f_2(n)) \\
  f_1(n) & \in O(f_3(n)) \\
  f_2(n) & \in \Omega(f_1(n)) \\
  f_1(n) & \in \Theta(f_3(n))
\end{align*}
\]

\[\square\]

**Big-O Algebra**

The Big-O algebra behaves different than the algebra of real numbers as is clear in the following examples.

**Example 2.7.** We are given the following functions.

\[
\begin{align*}
  f_1(n) & = 2n^3 - n + 1 \\
  f_2(n) & = 5n^2 + 5
\end{align*}
\]

We have for multiplication

\[
O(f_1(n)f_2(n)) = O((2n^3 - n + 1)(5n^2 + 5)) \\
= O(10n^5 + \cdots) = O(n^5)
\]

For addition, we have

\[
O(f_1(n) + f_2(n)) = O((2n^3 - n + 1) + (5n^2 + 5)) \\
= O(2n^3 + \cdots) = O(n^3) = O(f_1(n))
\]
CHAPTER 2. COMPLEXITY ANALYSIS

For scaling, we have

\[
O(5 \cdot f_1(n)) = O(5 \cdot (2n^3 - n + 1)) = O(10n^3 + \cdots) = O(n^3) = O(f_1(n))
\]

△

The Big-O algebra has the following properties.

**Multiplication:**

- **Action** \( f_1(n) \in O(g_1(n)), f_2(n) \in O(g_2(n)) \Rightarrow f_1 f_2(n) \in O(g_1 g_2) \)

- **Absorption** \( f \cdot O(g(n)) \subset O(fg) \)

Here \( f \cdot O(g(n)) = \{fh | h \in O(g(n))\} \).

**Addition:**

- **Action** \( f_1(n) \in O(g_1(n)), f_2(n) \in O(g_2(n)) \Rightarrow f_1 f_2(n) \in O(|g_1| + |g_2|) \)

- **Absorption** \( O(f_1(n)) + O(f_2(n)) = \begin{cases} O(f_1(n)), & \text{if } f_2 \in O(f_1) \\ O(f_2(n)), & \text{if } f_1 \in O(f_2) \end{cases} \)

\( f > 0, g > 0 \Rightarrow f + O(g(n)) \subset O(f + g) \)

Here \( f + O(g(n)) = \{f+h | h \in O(g(n))\} \).

**Scaling:**

- **Action** \( \forall k \in \mathbb{R}, f(n) \in O(g(n)) \Rightarrow kf(n) \in O(g(n)) \)

- **Absorption** \( O(kf(n)) = O(f(n)) \)

Some example subset relationships for various orders are \( O(1) \supset O(\log n) \supset O(n) \supset O(n^2) \supset O(n^k) \supset O(2^n) \supset O(n!) \).

### 2.2 Complexity Examples

In analyzing complexity for various algorithms, we can perform the analysis for the best case, the worst case, or the average case. In these examples we will be generally performing the worst case analysis. For instance if one is searching for a number in a list, the number could be the first one on the list where the search would stop at the first item in the best case scenario, at the last item in the worst case scenario, and in the middle item for the average case scenario.
2.2. COMPLEXITY EXAMPLES

$O(1)$ Example

Take an algorithm that takes an input containing $N$ numbers. The algorithm adds the first two numbers. The number of computations (additions) involved in the algorithm are independent of the problem size $N$, and hence the algorithm has an order $O(1)$. An example Python listing is shown in Program 2.1.

Program 2.1 $O(1)$ Example Code

```python
L = [3, 2, 6, 12, 25]
result = L[0] + L[1]
print(result)
```

$O(\log(n))$ Example: Binary Sort

An algorithm that has $O(\log_2(n))$ complexity is the binary sort algorithm. The binary sort algorithm that we will show here takes a list of numbers in an ascending order and also a number. If the number is in the list, the function returns the numerical index of the number in the list. If the number is not in the list, the function returns a zero.

The algorithm works as follows. It compares the number with the mid value of the list. If they are equal, then the algorithm returns the index, otherwise it applies the algorithm again to the upper list if the number was greater than the middle value, or else the lower list. Python listing for this code is shown in Program 2.2.

$O(n)$ Example: Linear Search

An algorithm that has $O(n)$ complexity is the simple linear search for a given number in a given list of unordered numbers.

The algorithm works as follows. It compares the number with the first value in the list. If they are equal, it returns the position of the number, one in this case. If they are not equal it starts again on the rest of the list. If the number is not found, it returns a zero. Python listing for this code is shown in Program 2.3.

$O(n^2)$ Example: Bubble Sort

An algorithm that has $O(n^2)$ complexity is the bubble sort. Bubble sort is the algorithm used to sort an unordered list. The input to the algorithm is an unsorted list of numbers, and the output is the list with the numbers sorted.
**Program 2.2** $O(\log n)$ Example Code: Binary Search

```python
def BinarySort(L, n):
    size = len(L)
    if size <= 1:
        if n == L[0]:
            return 1
        else:
            return 0
    mid = size // 2
    if n == L[mid]:
        return mid + 1
    if n < L[mid]:
        return BinarySort(L[0:mid], n)
    if n > L[mid]:
        temp = BinarySort(L[mid+1:size], n)
        if temp == 0:
            return 0
        else:
            return mid + 1 + temp
    return 0
```

**Program 2.3** $O(n)$ Example Code: Linear Search

```python
def Find(L, n):
    size = len(L)
    if size == 0:
        return 0
    if n == L[0]:
        return 1
    else:
        temp = Find(L[1:size], n)
        if temp == 0:
            return 0
        else:
            return 1 + temp
    return 0
```
The algorithm works as follows. It compares numbers sequentially in pairs till the lowest number is shifted all the way to the highest index. Then the process is repeated to the list without the lowest number. Python listing for this code is shown in Program 2.4.

Number of comparisons to obtain the lowest number will be \( n - 1 \), the number of comparisons to obtain the next lowest number will be \( n - 2 \), and so on. Hence the total number of comparisons will be

\[
\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in O(n^2)
\]

---

**Program 2.4** \( O(n^2) \) Example Code: Bubble Sort

```python
def BubbleSort(L):
    size = len(L)
    for nlist in range(size-1, 0, -1):
        for i in range(nlist):
            if L[i] < L[i+1]:
                temp = L[i+1]
                L[i+1] = L[i]
                L[i] = temp
```

---