Recitation 1: Asymptotic Analysis of Algorithms

1 Asymptotic Notation

1.1 Overview

- Asymptotic notation provides a language for analyzing the performance and resource usage of algorithms.

- Typically we analyze the running time of an algorithm as a function of its input size, though the running time may sometimes depend on other factors like the actual input.

- We usually analyze the running time in the worst-case.
  \[ T(n) = \text{max time on any input of size } n. \]

- Sometimes we analyze the running time in an average case.
  \[ T(n) = \text{expected time over all inputs of size } n. \]

- In particular, we ignore machine-dependent constants and look at the growth of \( T(n) \) as \( n \to \infty \).

1.2 Big O (upper bound)

- For nonnegative functions \( f \) and \( g \), \( f = O(g) \) iff there exists a constant \( c > 0 \) and a \( n_0 \) such that, for all \( n \geq n_0 \), \( f(n) \leq cg(n) \).

- In English: for large \( n \), \( f(n) = O(g(n)) \) means that \( f(n) \) is less than or equal to \( g(n) \) to within a constant factor.

- Example: \( 2n^2 = O(n^2) \), \( 2n^2 = O(n^3) \), etc.

1.3 Little o (strict upper bound)

- \( f = o(g) \) iff, for every constant \( c > 0 \), there exists a constant \( n_0 \) such that, for all \( n \geq n_0 \), \( f(n) < cg(n) \).
• Little o is a stronger bound than Big O because it means that, asymptotically, \( f \) grows strictly slower than \( g \). A Big O bound implies that, asymptotically, \( f \) does not grow faster than \( g \).

• Example: \( 2n^2 = o(n^3) \). Note that \( 2n^2 \neq o(n^2) \).

1.4 Big Omega (lower bound)

• \( f = \Omega(g) \) iff there exists a constant \( c > 0 \) and a \( n_0 \) such that, for all \( n \geq n_0 \), \( f(n) \geq cg(n) \).

• In English: for large \( n \), \( f(n) = \Omega(g(n)) \) means that \( f(n) \) is greater than or equal to \( g(n) \) to within a constant factor. Note that \( f = \Omega(g) \) iff \( g = O(f) \).

• Example: \( 2n^2 = \Omega(n^2) \), \( 2n^2 = \Omega(n) \), etc.

1.5 Little omega (strict lower bound)

• \( f = \omega(g) \) iff, for every constant \( c > 0 \), there exists a constant \( n_0 \) such that, for all \( n \geq n_0 \), \( f(n) > cg(n) \).

• This is analogous to Little o. It is a stronger bound than Big Omega because it means that, asymptotically, \( f \) grows strictly faster than \( g \).

• Example: \( 2n^2 = \omega(n) \). Note that \( 2n^2 \neq \omega(n^2) \).

1.6 Big Theta

• \( f = \Theta(g) \) iff \( f = O(g) \) and \( f = \Omega(g) \).

• That is, \( f = \Theta(g) \) iff both functions are equal to within a constant factor.

• Example: \( 2n^2 = \Theta(n^2) \).

2 Recurrences

2.1 Writing a recurrence

• Recurrences describe a function in terms of itself. We usually use them to describe the performance of recursive algorithms.
• Example: **Binary Search.** Find an element \( x \) in a sorted array \( A \):

1. If \( x \) is equal to the middle element of \( A \), we found \( x \).
2. If \( x \) is less than the middle element, run **Binary Search** on the smaller half of \( A \).
3. If \( x \) is greater than the middle element, run **Binary Search** on the larger half of \( A \).

Let \( T(n) \) be the running time of the algorithm when the input array has \( n \) elements. If \( A \) is down to one element, the algorithm runs in \( T(n) = \Theta(1) \) time. Otherwise, the algorithm takes \( \Theta(1) \) time to compare \( x \) with the middle element and \( T(n/2) \) time to perform its recursive call for a total time of \( T(n) = T(n/2) + \Theta(1) \). Thus,

\[
T(n) = \begin{cases} 
\Theta(1), & \text{if } n = 1 \\
T(n/2) + \Theta(1), & \text{if } n > 1 
\end{cases}
\]

We usually omit the base case and assume \( T(n) = \Theta(1) \) for sufficiently small \( n \). Therefore, overall, \( T(n) = T(n/2) + \Theta(1) \).

### 2.2 Solving a recurrence

#### 2.2.1 Recursion trees

- In a recursion tree, each node represents a recursive call; the tree allows us to visualize the recursive calls. Each node is labeled by the amount of work done at that call. Thus, we can add up the work done at all each call to find the total running time.

- Example: \( T(n) = 2T(n/2) + cn \) for some constant \( c > 0 \).
At each row (level) of the tree, $cn$ work is done. Because there are $\Theta(\log n)$ rows, the total work done is $\Theta(n \log n)$.

- We could apply a recursion tree to the Binary Search recurrence. In the tree, each node would be labeled with $\Theta(1)$ since each call does $\Theta(1)$ work, and there would be $\Theta(\log n)$ nodes. The total running time is thus $\Theta(\log n)$.

2.2.2 Substitution method

- To solve a recurrence using the substitution method, we first guess the form of the solution and then verify it by induction. In the process, we solve for constants.

- To establish an upper bound on $T(n) = 2T(n/2) + n$, we first guess that $T(n) = O(n \log n)$. We then verify that

$$T(n) \leq cn \log n$$

for some constant $c > 0$.

- By the inductive hypothesis, $T(n/2) \leq c(n/2) \log (n/2)$. Thus,

$$T(n) = 2T(n/2) + n$$
$$\leq 2[c(n/2) \log (n/2)] + n$$
$$= cn \log (n/2) + n$$
$$= cn \log n - cn \log 2 + n$$
$$= cn \log n - (c - 1)n$$
$$\leq cn \log n$$

as long as $c \geq 1$.

- We must also establish base cases for the induction. The base cases of these inductive proofs do not necessarily need to set $n = 1$ because, according to the definition of Big O, we are only concerned with $n \geq n_0$, where $n_0$ is a constant we choose. For this proof, we can use $n = 2$ and $n = 3$ as base cases. ($n = 1$ is difficult to work with because setting $T(1) = 1$ conflicts with $T(1) \leq c1 \log 1 = 0$.) We can set $T(2) = 4$ and $T(3) = 5$ as the base cases. This must satisfy $4 \leq c2 \log 2$ and $5 \leq c3 \log 3$, which are true for any $c \geq 2$.

- In an analogous way, the substitution method can be used to verify a lower bound.
3 The Master Theorem

The Master Theorem provides cookbook solutions to recurrences of the form

\[ T(n) = aT(n/b) + f(n) \]

where \( a \geq 1, b > 1, \) and \( f \) is asymptotically positive. There are three cases that compare \( f(n) \) with \( n^{\log_b a} \).

3.1 Case 1: \( f(n) = O(n^{\log_b a - \epsilon}) \) for some constant \( \epsilon > 0 \)

- \( f(n) \) grows polynomially slower than \( n^{\log_b a} \).

- If we imagine a recursion tree for the recurrence, at each level the work done increases geometrically from the root to the leaves. The leaves have a constant fraction of the total work.

- **Solution:** \( T(n) = \Theta(n^{\log_b a}) \).

- Example: \( T(n) = 4T(n/2) + n \) has the solution \( T(n) = \Theta(n^2) \).

3.2 Case 2: \( f(n) = \Theta(n^{\log_b a \log^k n}) \) for some constant \( k \geq 0 \)

- \( f(n) \) grows at about the same rate as \( n^{\log_b a} \).

- If \( k = 0 \), the work done at each level of the recursion tree is the same.

- **Solution:** \( T(n) = \Theta(n^{\log_b a \log^{k+1} n}) \).

- Example: \( T(n) = 4T(n/2) + n^2 \) has the solution \( T(n) = \Theta(n^2 \log n) \). The Binary Search recurrence above can also be solved by Case 2 with \( a = 1, b = 2, \) and \( k = 0 \).

3.3 Case 3: \( f(n) = \Omega(n^{\log_b a + \epsilon}) \) for some constant \( \epsilon > 0 \)

- \( * \) and \( af(n/b) \leq cf(n) \) for some constant \( c < 1 \).

- \( f(n) \) grows polynomially faster than \( n^{\log_b a} \).

- At each level in the recursion tree, the work decreases geometrically. The root has a constant fraction of the total work.
• **Solution:** \( T(n) = \Theta(f(n)) \).

• Example: \( T(n) = 4T(n/2) + n^3 \) has the solution \( T(n) = \Theta(n^3) \).