Algorithms and Lab (CSE130)

Asymtotic Notations and Complexity Analysis

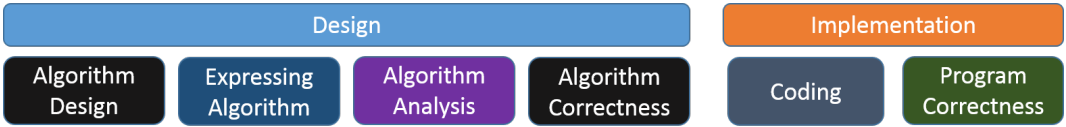
Muhammad Tariq Mahmood

tariq@koreatech.ac.kr
School of Computer Science and Engineering

Note: These notes are prepared from the following resources.

- (main text)Foundations of Algorithms, by Richard Neapolitan and Kumarss Naimipour
- Python Algorithm (파이썬 알고리즘) by Y.K. Choi (2021) (Korean)
- Introduction to the Design and Analysis of Algorithms by Anany Levitin
- Introduction to Algorithms, by By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
- https://www.geeksforgeeks.org

Design and Implementation Phases



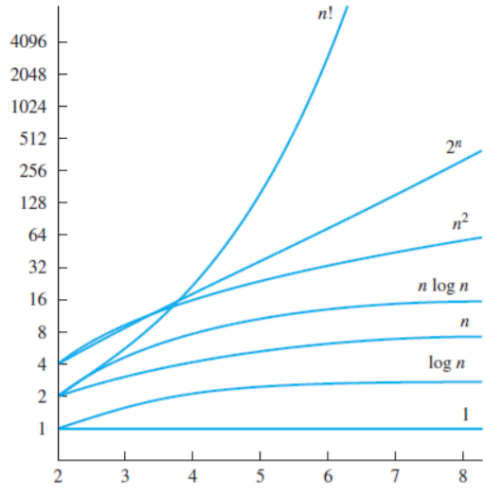
Algorithm Analysis

- The analysis of algorithms is to determine the computational complexity in terms of time and storage.
- Two approaches: Empirical/Experimental analysis and theoretical/formal analysis
- Theoretical/Formal analysis:
  - It involves determining a function (the time complexity function) that relates the algorithm's **input size** to the number of steps it takes or the number of storage locations it uses (in case of space complexity function).
  - It is common to estimate algorithm complexity in the asymptotic sense i.e., using order notations, Big-O notation, Big-omega(Ω) notation and Big-theta(Θ) notation
  - It dose not depend on
    - Actual number of CPU cycles (Computer)
    - Number of instructions
    - Programming language
    - Programmer

Contents

Order of Growth

- The growth rate for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.
- Upon increasing the input size, the complexity (time) of an efficient algorithm grows slowly, whereas the complexity (time) of an inefficient algorithm grows rapidly.
- Order (rate) of growth of functions provides a simple characterization of efficiency and also allows to compare the relative performance of alternative algorithms.
- A variety of growth rates that are representative of typical algorithms are shown.



Order of Growth (cont...)

- Observations
  - Any linear-time algorithm is more efficient than any quadratic-time algorithm.
  - Any quadratic-time algorithm is more efficient than any exponential-time algorithm.
  - Algorithm with logarithmic function is more efficient than Linear function and the algorithm having complexity in cubic function is faster than exponential function.
  - Algorithms that require an exponential number of operations are applicable for solving only problems of very small sizes.
  - Using rate of growth as a measure to compare different functions implies comparing them asymptotically.

- Consider the three functions $T_1(n)$, $T_2(n)$, $T_3(n)$

$T_1(n) = n$	$T_2(n) = n^2$	$T_3(n) = n^2 + n + 10$	$T_2(n)/T_1(n)$	$T_3(n)/T_2(n)$
1	1	12	1	12.00000000
5	25	40	5	1.60000000
10	100	120	10	1.20000000
30	900	940	30	1.04444444
50	2500	2560	50	1.02400000
100	10000	10110	100	1.01100000
1000	1000000	1001010	1000	1.00101000
10000	100000000	100010010	10000	1.00010010
100000	10000000000	10000100010	100000	1.00001000

Order of Growth (cont...)

- The complexity of an algorithms can be analyzed by comparing its complexity functions $T(n)$ with one of the special cases.

- By dropping the less significant terms and the constant coefficients, complexity of an algorithm can be expressed based on the most significant term

Function Name	Special Cases	Examples
Constant	$f(n) = 1$	$T(n) = 10$
Logarithmic	$f(n) = \log n$	$T(n) = 3 + 6 \log n$
Linear	$f(n) = n$	$T(n) = 5n + 2$
Quadratic	$f(n) = n^2$	$T(n) = 7n^2 + 3n + 2$
Cubic	$f(n) = n^3$	$T(n) = 2n^3 + n^2 + 3n + 1$
Exponential	$f(n) = 2^n$	$T(n) = 82^n + 6$
Factorial	$f(n) = n!$	$T(n) = n! + n^2$

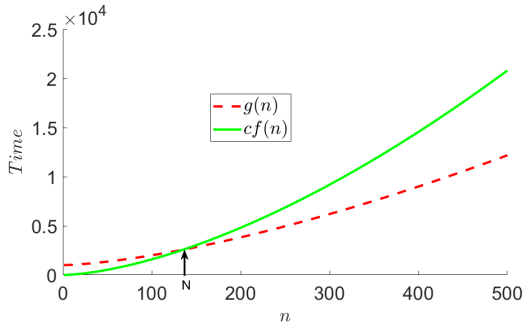
- The asymptotic analysis of an algorithm expresses the running time in term of asymptotic notations, Big O Notation, Big Ω Notation, Big Θ Notation.

Big O Notation

- O-notation, pronounced ''big-oh notation'', is used to describe the asymptotic upper bound of an algorithm.
- Formally, it is defined as: For a given complexity function $f(n)$, $O(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$,

$$g(n) \leq c \times f(n)$$

- Examples:
  - $g(n) = 3n + 5$, $f(n) = n$ then $g(n) \in O(n)$.
  - $g(n) = n^3 + 1$, $f(n) = n^2$, then $g(n) \notin O(n^2)$.
  - $g(n) = n^2 + n + 5$, $f(n) = n^3$ then $g(n) \in O(n^3)$.



- Less formally, this means that for all sufficiently big n, the running time of the algorithm is less than $f(n)$ multiplied by some constant c.

- **Example-1:** We will show that $g(n) = T(n) = n^2 + 10n \in O\left(n^2\right)$

$$T(n) = n^2 + 10n$$
$$f(n) = n^2$$
$$n^2 + 10n \leq 2n^2$$
$$T(n) = n^2 + 10n \in O\left(n^2\right)$$

Show that $T(n) \leq c \times f(n)$ for $n \geq N$

for $c = 2$, $N = 10$ holds

$T(n) \in O(f(n))$

- **Example-2:** we will show that $g(n) = T(n) = \frac{n(n-1)}{2} \in O\left(n^2\right)$

$$T(n) = \frac{n(n-1)}{2}$$
$$f(n) = n^2$$
$$\frac{n(n-1)}{2} \leq \frac{1}{2}n^2$$
$$T(n) = \frac{n(n-1)}{2} \in O\left(n^2\right)$$

Show that $T(n) \leq c \times f(n)$ for $n \geq N$

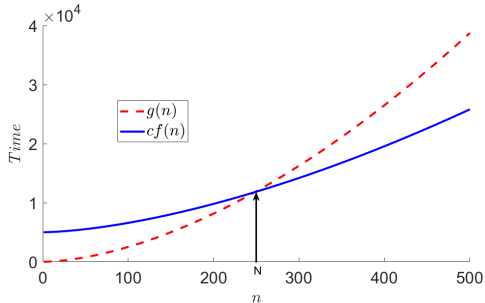
for *for* $c = \frac{1}{2}$, $N = 0$ holds

$T(n) \in O(f(n))$

Big Ω Notation

- Ω-notation, pronounced ''big-omega notation'', is used to describe the asymptotic lower bound of an algorithm.
- Formally, it is defined as: For a given complexity function $f(n)$, $O(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$,

$$g(n) \geq c \times f(n)$$



- Examples:
  - $g(n) = 3n + 5$, $f(n) = n$, then $g(n) \in \Omega(n)$.
  - $g(n) = n^2 + 5$, $f(n) = n^3$, then $g(n) \notin \Omega(n^3)$.
  - $g(n) = n^4 + n^2$, $f(n) = n^3$, then $g(n) \in \Omega(n^3)$.
- Less formally, this means that for all sufficiently big n, the running time of the algorithm is greater than $f(n)$ multiplied by some constant c.

- **Example-1:** we will show that $T(n) = g(n) = n^2 + 10n \in \Omega\left(n^2\right)$

$$
\begin{aligned}
T(n) &= n^2 + 10n \\
f(n) &= n^2 && \text{need to show } T(n) \geq c \times f(n) \text{ for } n \geq N \\
n^2 + 10n &\geq n^2 && \text{for } c = 1,\ N = 0 \text{ holds} \\
T(n) = n^2 + 10n &\in \Omega\left(n^2\right) && \text{as } T(n) \in \Omega\left(f(n)\right)
\end{aligned}
$$

- **Example-2:** we will show $T(n) = g(n) = \frac{n(n-1)}{2} \in \Omega\left(n^2\right)$

$$
\begin{aligned}
T(n) &= \frac{n(n-1)}{2} \\
f(n) &= n^2 && \text{need to show } T(n) \geq c \times f(n) \text{ for } n \geq N \\
\frac{n(n-1)}{2} &\geq \frac{1}{4}n^2 && \text{for } c = \frac{1}{4},\ N = 2 \text{ holds} \\
T(n) = \frac{n(n-1)}{2} &\in \Omega\left(n^2\right) && \text{as } T(n) \in \Omega\left(f(n)\right)
\end{aligned}
$$

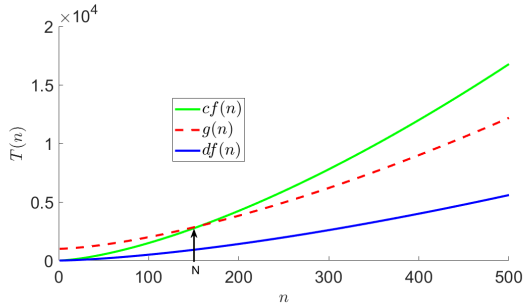
Big Θ Notation

- It is defined as: For given complexity functions $f(n)$ and $g(n)$, $\Theta(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constants c and d and some nonnegative integer N such that, for all $n \geq N$,

$$d \times f(n) \leq g(n) \leq c \times f(n)$$

- It means that if a given complexity function $g(n)$ $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$ then $g(n) \in \Theta(f(n))$. It can be represented as

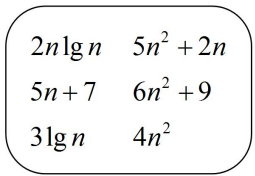
$$\Theta(f(n)) = \mathrm{O}(f(n)) \bigcap \Omega(f(n))$$



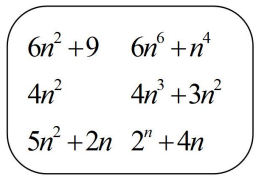
- Θ-notation, pronounced ''big-theta notation'', is used to describe the both asymptotic lower and upper bounds of an algorithm.

Asymptotic Notations (cont...)

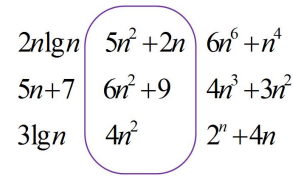
- **Examples**



$O\left(n^2\right)$



$\Omega\left(n^2\right)$



$\Theta\left(n^2\right)$

- A complexity function need not have a quadratic term to be in $O\left(n^2\right)$.
- It need only eventually lie beneath some pure quadratic function on a graph. Therefore, any logarithmic or linear complexity function is in $O\left(n^2\right)$.
- Similarly, any logarithmic, linear, or quadratic complexity function is in $O\left(n^3\right)$.
- If a function is in $\Omega\left(n^2\right)$, then eventually the function lies above some pure quadratic function on a graph. This means that eventually it is least as bad as a pure quadratic function.

Limit Method

Growth of two functions using limit

- The growth of two functions f and g can be found by computing the limit $\lim_{n\to\infty} \frac{g(n)}{f(n)}$.
- Using the definition of O, Ω, Θ it can be shown that :

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \begin{cases} 0, & g(n) \in O(f(n)) \\ \infty, & g(n) \in \Omega(f(n)) \\ c, c > 0 & g(n) \in \Theta(f(n)) \end{cases}$$

- It means:
  - if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$: then intuitively $g(n) < f(n) \implies g(n) = O(f(n))$ or $g(n) \in O(f(n))$ and $g(n) \neq \Theta(f(n))$.
  - if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$: then intuitively $g(n) > f(n) \implies g(n) \in \Omega(f(n))$ and $g(n) \neq \Theta(f(n))$.
  - if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c, c > 0$: then intuitively $g(n) = c \cdot f(n) \implies g(n) \in \Theta(f(n))$.

Limit Method (cont...)

- Examples-1: Let $g(n) = \frac{1}{2}n(n-1)$, $f(n) = n^2$

$$\begin{aligned}\lim_{n\to\infty}\frac{g(n)}{f(n)} &= \lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2}\\ &= \frac{1}{2}\lim_{n\to\infty}\frac{n(n-1)}{n^2}\\ &= \frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right)\\ &= \frac{1}{2}\end{aligned}$$

Hence $g(n) = \frac{1}{2}n(n-1) \in \Theta(n^2)$

- Example-2 Let $T(n) = n^2 + 3n + 4$, $f(n) = n^3$

$$\begin{aligned}\lim_{n\to\infty}\frac{T(n)}{f(n)} &= \lim_{n\to\infty}\frac{n^2+3n+4}{n)}\\ &= \lim_{n\to\infty}\frac{n^2+3n+4}{n^3}\\ &= \lim_{n\to\infty}\left(\frac{1}{n}+\frac{3}{n^2}+\frac{4}{n^3}\right)\\ &= 0\end{aligned}$$

Hence $T(n) = n^2 + 3n + 4 \in O(n^3)$

Properties of log functions:

- $\log(ab) = \log(a) + \log(b)$
- $\lg^k n = (\lg n)^k$
- $\lg\lg n = \lg(\lg n)$
- $a^{\log_b c} = c^{\log_b a}$
- $a^{\log_a b} = b$
- $\log_a n = \frac{\log_b n}{\log_b a}$
- $\lg b^n = n \lg b$
- $\lg xy = \lg x + \lg y$
- $\log_a b = \frac{1}{\log_b a}$

Limit Method (cont...)

- Example-3 Let $T(n) = g(n) = log_2 n$, $f(n) = \sqrt{n}$

$$\begin{aligned}\lim_{n\to\infty} \frac{g(n)}{f(n)} &= \lim_{n\to\infty} \frac{log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(log_2 n)'}{(\sqrt{n})'} \\ &= \lim_{n\to\infty} \frac{\frac{log_2 e}{n}}{\frac{1}{2\sqrt{n}}} = 2log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} \\ &= 0\end{aligned}$$

Hence $T(n) = g(n) = log_2 n \in O(\sqrt{n})$

- Example-4: Let $T(n) = g(n) = n!$, $f(n) = 2^n$

$$\begin{aligned}\lim_{n\to\infty} \frac{T(n)}{f(n)} &= \lim_{n\to\infty} \frac{n!}{2^n} \\ &= \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n} \\ &= \lim_{n\to\infty} \sqrt{2\pi n}\left(\frac{n}{2e}\right)^n \\ &= \infty\end{aligned}$$

Hence $T(n) = g(n) = n! \in \Omega(2^n)$

- L'H'opital's rule $\lim_{n\to\infty} \frac{T(n)}{f(n)} = \lim_{n\to\infty} \frac{T'(n)}{f'(n)}$ Stirling's formula $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$
- Properties of Exponential functions:
  - $a^x \cdot a^y = a^{x+y}$
  - $(a^x)^y = a^{xy}$
  - $a^{-x} = \frac{1}{a^x}$
  - $a^1 = a$
  - $a^0 = 1$

Properties of Asymptotic Notations

Properties of Asymptotic Notations

- Symmetric Properties
  - $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
  - $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$
- General Properties: For a constant a.
  - If $g(n) \in O(f(n))$ then $a \times g(n) \in a \times O(f(n))$
  - If $g(n) \in \Omega(f(n))$ then $a \times g(n) \in a \times \Omega(f(n))$
  - If $g(n) \in \Theta(f(n))$ then $a \times g(n) \in a \times \Theta(f(n))$
- Transitive Properties:
  - If $g(n) \in O(f(n))$ *and* $f(n) \in O(h(n))$ then $g(n) \in O(h(n))$
  - If $g(n) \in \Omega(f(n))$ *and* $f(n) \in \Omega(h(n))$ then $g(n) \in \Omega(h(n))$
  - If $g(n) \in \Theta(f(n))$ *and* $f(n) \in \Theta(h(n))$ then $g(n) \in \Theta(h(n))$
- Reflexive Properties : For a given complexity function $g(n)$
  - $g(n) \in O(g(n))$
  - $g(n) \in \Omega(g(n))$
  - $g(n) \in \Theta(g(n))$
- if $g(n) \in O(h(n))$ *and* $f(n) \in O(h(n))$ then $g(n) + f(n) \in O(h(n))$
- if $g_1(n) \in O(f_1(n))$ *and* $g_2(n) \in O(f_2(n))$, then $g_1(n) + g_2(n) \in O(max\{f_1(n), f_2(n)\})$
- Let $p(n)$ be a polynomial of degree a and $q(n)$ be a polynomial of degree b. Then
  - $p(n) \in O(q(n))$ if and only if $a \leq b$
  - $p(n) \in \Omega(q(n))$ if and only if $a \geq b$
  - $p(n) \in \Theta(q(n))$ if and only if $a = b$

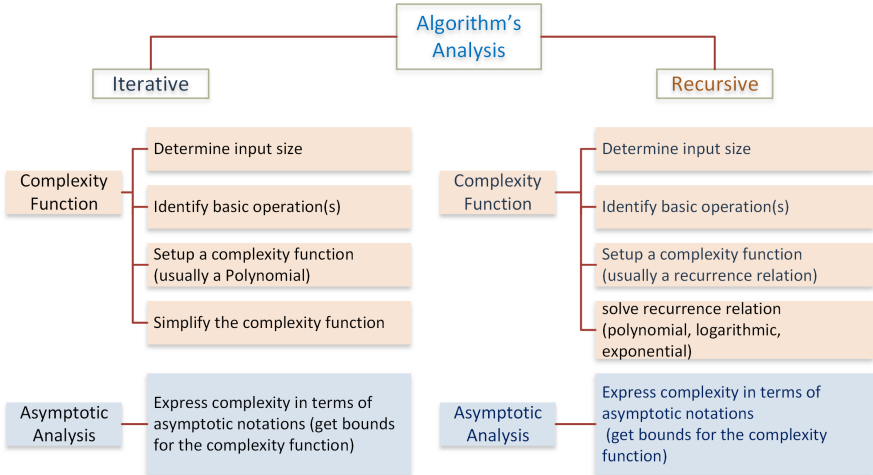
Properties of Asymptotic Notations (cont...)

Summary : Let $f(n)$ and $g(n)$ be functions from the set of nonnegative integers to the set of nonnegative real numbers.

- Big-O: $g(n) \in \mathrm{O}(f(n))$ iff there exist constants $c > 0$ and $k \geq 1$ such that $g(n) \leq cf(n)$ for every $n \geq k$.
- Big-Ω: $g(n) \in \Omega(f(n))$ iff there exist constants $c > 0$ and $k \geq 1$ such that $g(n) \geq cf(n)$ for every $n \geq k$.
- Big-Θ: $g(n) \in \Theta(f(n))$ iff $g(n) = \mathrm{O}(f(n))$ and $g(n) = \Omega(f(n))$.
- little-o: $g(n) \in \mathrm{o}(f(n))$ iff $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.
  If $g(n) \in o(f(n))$ then $g(n) \in (O(f(n)) - \Omega(f(n)))$, it means $g(n) \in O(f(n))$ but $g(n) \notin \Omega(f(n))$ That is, g(n) is belongs to $O(f(n))$ but does not belong to $\Omega(f(n))$.
- little-ω: $g(n) \in \omega(f(n))$ iff $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$. If $g(n) \in \omega(f(n))$ then $g(n) \in (\Omega(f(n)) - O(f(n)))$, it means $g(n) \in \omega(f(n))$ but $g(n) \notin O(f(n))$ That is, g(n) is belongs to $\Omega(f(n))$ but does not belong to $O(f(n))$.

Framework for Algorithm Analysis

- A general framework for complexity analysis of iterative and recursive algorithms



- Example-1: Add Array Members

```
1: procedure sumArray(number S[])
2:     integer i
3:     number sum
4:     sum = 0
5:     for (i = 1; i <= n; i++) do
6:         sum = sum + S[i]
7:     end for
8:     return sum
9: end procedure
```

- Input size: n, number of elements in the array S.
- Basic operation(s): comparison $i <= n$ and \ or $sum = sum + S[i]$ inside loop

- Complexity Function

$$T(n) = 1 + \sum_{i=1}^{n} +1 = 1 + n + 1 = n + 2$$

- Complexity in terms of asymptomatic notations

$$n + 2 \leq c_1 n \qquad \text{for } n \geq 1, c_1 = 3 \text{ holds}$$
$$n + 2 \in \mathrm{O}(n)$$

$$c_2 n \leq n + 1 \qquad \text{for } n \geq 1,\ c_2 = 1$$
$$n + 2 \in \Omega(n)$$

$$c_2 n \leq n + 2 \leq c_1 n \quad \text{for } n \geq 1,\ c_1 = 3,\ c_2 = 1$$
$$n + 2 \in \Theta(n)$$

- Example-2: Exchange Sort

```
1: procedure exchangeSort(S[])
2:     integer i, j
3:     for (i = 1; i <= n − 1; i++) do
4:         for (j = i + 1; j <= n; j++) do
5:             if (S[j] < S[i]) then
6:                 exchange S[i] and S[j]
7:             end if
8:         end for
9:     end for
10: end procedure
```

- Input size: n, the size of the input array S.
- Basic operation(s): comparisons inside the inner most loop

- Complexity function and its simplification

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 \\
&= \sum_{i=1}^{n-1} (n - (i+1) + 1) \\
&= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i \\
&= n(n-1) - \frac{(n-1)(n-1+1)}{2} \\
&= n^2 - n - \frac{n^2 - n}{2} \\
&= \frac{2n^2 - 2n - n^2 + n}{2} = \frac{n(n-1)}{2}
\end{aligned}
$$

- Complexity in terms of asymptomatic notations

$$\frac{n(n-1)}{2} \leq c_1 n^2$$

$$\because n \geq 2, c_1 = 1 \therefore \frac{n(n-1)}{2} \in \mathrm{O}\left(n^2\right)$$

$$c_2 n^2 \leq \frac{n(n-1)}{2}$$

$$\because n \geq 2, c_2 = \frac{1}{4} \therefore \frac{n(n-1)}{2} \in \Omega\left(n^2\right)$$

$$c_2 n^2 \leq \frac{n(n-1)}{2} \leq c_1 n^2$$

$$\because n \geq 2,\ c_1 = 1,\quad c_2 = \frac{1}{4}$$

$$\therefore \frac{n(n-1)}{2} \in \Theta\left(n^2\right)$$

- Using Limit Method

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \begin{cases} c & \textit{implies}\ \ g(n) \in \Theta(f(n)) \\ 0 & \textit{implies}\ \ g(n) \in O(f(n)) \\ \infty & \textit{implies}\ \ g(n) \in \Omega(f(n)) \end{cases}$$

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = \lim_{n\to\infty} \frac{\frac{n(n-1)}{2}}{n^2}$$

$$= \lim_{n\to\infty} \frac{1}{2} - \frac{1}{2n}$$

$$= \frac{1}{2}\ \ (\textit{constant})$$

$$\therefore \frac{n(n-1)}{2} \in \Theta(n^2)$$

- Example-3: Matrix Multiplication :

```
1:  procedure matrixMultuply(A[][],B[][])
2:      integer i, j, k
3:      number C[][]
4:      for (i = 1; i <= n; i++) do
5:          for (j = 1; j <= n; j++) do
6:              C[i][j] = 0
7:              for (k = 1; k <= n; k++) do
8:                  C[i][j] = C[i][j] + A[i][k] * B[k][j]
9:              end for
10:         end for
11:     end for
12:     return C
13: end procedure
```

- Complexity function and its simplification

$$
\begin{aligned}
T(n) &= \left(\sum_{i=1}^{n}\left(\sum_{j=1}^{n}\left(1+\sum_{k=1}^{n}1\right)\right)\right)+1 \\
&= \sum_{i=1}^{n}\left(\sum_{j=1}^{n}(1+n)\right)+1 \\
&= \sum_{i=1}^{n}\left(\sum_{j=1}^{n}1\right)+\sum_{i=1}^{n}\left(\sum_{j=1}^{n}n\right)+1 \\
&= \sum_{i=1}^{n}n+\sum_{i=1}^{n}n^2+1 \\
&= n^2+n^3+1 \\
&= n^3+n^2+1
\end{aligned}
$$

Assign complexity class (complexity in asymptotic notations)

- Upper bound - O notation

$$f(n) = n^3, \quad T(n) = n^3 + n^2 + 3$$
$$T(n) \leq c_1 f(n)$$
$$n^3 + n^2 + 3 \leq c_1 n^3$$
$$\because c_1 = 5, \ \ n \geq 1 n^3 + n^2 + 3 \in O(n^3)$$

- Lower bound - Ω notation

$$f(n) = n, \quad T(n) = n^3 + n^2 + 3$$
$$c_2 f(n) \leq T(n)$$
$$c_2 n^3 \leq n^3 + n^2 + 3$$
$$\because c_2 = \frac{1}{5} \ \ n \geq 1$$
$$n^3 + n^2 + 3 \in \Omega(n^3)$$

- Both upper and lower bounds - Θ notation

$$f(n) = n^3, \quad T(n) = n^3 + n^2 + 3$$
$$c_2 f(n) \leq T(n) \leq c_1 f(n)$$
$$c_2 n^3 \leq n^3 + n^2 + 3 \leq c_1 n^3$$
$$\because c_2 = \frac{1}{5}, \ \ c_1 = 5, \ \ n \geq 1$$
$$n^3 + n^2 + 3 \in \Theta(n^3)$$

- Limit Method

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \begin{cases} c & \textit{implies} \quad g(n) \in \Theta(f(n)) \\ 0 & \textit{implies} \quad g(n) \in O(f(n)) \\ \infty & \textit{implies} \quad g(n) \in \Omega(f(n)) \end{cases}$$

- Applying above Theorem

$$\begin{aligned} \lim_{n\to\infty} \frac{T(n)}{f(n)} &= \lim_{n\to\infty} \frac{n^3 + n^2 + 3}{n^3} \\ &= \lim_{n\to\infty} \frac{1 + \frac{1}{n} + \frac{3}{n^3}}{1} \\ &= 1 \quad (\textit{constant}) \end{aligned}$$

$$n^3 + n^2 + 3 \in \Theta(n^3)$$

- Example-4: Tower of Hanoi puzzle

```
1:  procedure HANOI(n , S, A,T)
2:      if (n == 1) then
3:          move n from source to target.
4:          return
5:      else
6:          Hanoi(n - 1, S, T, A)
7:          move n from source to target
8:          Hanoi(n - 1, A, S, T)
9:      end if
10: end procedure
```

- Complexity Function:

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n-1) + 1, & n > 1 \end{cases}$$

- Simplified Complexity Function

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2[2T(n-2) + 1] + 1 \\ &= 2[2[2T(n-3) + 1] + 1] + 1 \\ &\ldots\ldots \\ &= 2^i T(n-i) + 2^i - 1 \\ &\ldots\ldots \\ &= 2^{n-1} T(n-(n-1)) + 2^{n-1} - 1 \\ &= 2^{n-1} T(1) + 2^{n-1} - 1 \\ &= 2^{n-1} + 2^{n-1} - 1 \\ &= \frac{2^n}{2} + \frac{2^n}{2} - 1 \\ &= 2^n - 1 \end{aligned}$$

Assign complexity class (complexity in asymptotic notations)

- Upper bound - O notation

$$T(n) = 2^n - 1, \quad f(n) = 2^n$$
$$T(n) \leq c_1 f(n)$$
$$2^n - 1 \leq c_1 2^n$$
$$\because c_1 = 3, \ \ n \geq 1$$
$$2^n - 1 \in O(2^n)$$

- Lower bound - Ω notation

$$T(n) = 2^n - 1, \quad f(n) = 2^n$$
$$T(n) \geq c_2 f(n)$$
$$2^n - 1 \geq c_2 2^n$$
$$\because c_2 = \frac{1}{3}, \ \ n \geq 1$$
$$2^n - 1 \in \Omega(2^n)$$

- Both upper and lower bounds - Θ notation

$$T(n) = 2^n - 1, \quad f(n) = 2^n$$
$$c_2 f(n) \leq T(n) \leq c_1 f(n)$$
$$c_2 2^n \leq 2^n - 1 \leq c_1 2^n$$
$$\because c_2 = \frac{1}{3}, \ \ c_1 = 3, \ \ n \geq 1$$
$$2^n - 1 \in \Theta(2^n)$$

- Limit Method

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \begin{cases} c & \textit{implies} \quad g(n) \in \Theta(f(n)) \\ 0 & \textit{implies} \quad g(n) \in O(f(n)) \\ \infty & \textit{implies} \quad g(n) \in \Omega(f(n)) \end{cases}$$

- Applying above Theorem

$$\begin{aligned} \lim_{n\to\infty} \frac{T(n)}{f(n)} &= \lim_{n\to\infty} \frac{2^n - 1}{2^n} \\ &= \lim_{n\to\infty} \frac{1 + \frac{1}{2^n}}{1} \\ &= 1 \quad (\textit{constant}) \\ 2^n - 1 &\in \Theta(2^n) \end{aligned}$$

Master Theorem

- Master Method is a direct way to get solutions for particular types of recurrence relations.
- The master method works only for following type of recurrences or for recurrences that can be transformed to following type.
- Suppose a complexity function, where $b \geq 2$ and $k \geq 0$ are integers while $a > 0$, $c > 0$, $d > 0$ are real constants,

$$\begin{cases} T(1) = d, & n = 1 \\ T(n) = aT\left(\frac{n}{b}\right) + cn^k, & n > 1 \end{cases}$$

- The solution of the complexity function is given as

$$\begin{cases} T(n) \in \Theta\left(n^k\right) & a < b^k \quad (case - 1) \\ T(n) \in \Theta\left(n^k \lg n\right) & a = b^k \quad (case - 2) \\ T(n) \in \Theta\left(n^{\log_b a}\right) & a > b^k \quad (case - 3) \end{cases}$$

Master Theorem (cont...)

- Similarly for the complexity function

$$\begin{cases} T(1) = d, & n = 1 \\ T(n) \geq aT\left(\frac{n}{b}\right) + cn^k, & n > 1 \end{cases}$$

- Complexity function

$$\begin{cases} T(1) = d, & n = 1 \\ T(n) \leq aT\left(\frac{n}{b}\right) + cn^k, & n > 1 \end{cases}$$

- The solution

$$\begin{cases} T(n) \in \Omega\left(n^k\right) & a < b^k \quad (case - 1) \\ T(n) \in \Omega\left(n^k \lg n\right) & a = b^k \quad (case - 2) \\ T(n) \in \Omega\left(n^{\log_b a}\right) & a > b^k \quad (case - 3) \end{cases}$$

- The solution

$$\begin{cases} T(n) \in O\left(n^k\right) & a < b^k \quad (case - 1) \\ T(n) \in O\left(n^k \lg n\right) & a = b^k \quad (case - 2) \\ T(n) \in O\left(n^{\log_b a}\right) & a > b^k \quad (case - 3) \end{cases}$$

Master Theorem (cont...)

- Examples

Recurrence Relation	Solution Using Master Theorem
$\begin{cases} T(1) = 3 \\ T(n) = 8T\left(\frac{n}{4}\right) + 5n^2, n > 1 \end{cases}$	Since, $a = 8,\ b = 4,\ k = 2$ $\Rightarrow 8 < 4^2 \Rightarrow\ a < b^k \Rightarrow (case - 1)$ $\Rightarrow T(n) \in \Theta\left(n^2\right)$
$\begin{cases} T(1) = 7 \\ T(n) = 9T\left(\frac{n}{3}\right) + 5n^1, n > 1 \end{cases}$	Since, $a = 9,\ b = 3,\ k = 1$ $\Rightarrow 9 > 3^1 \Rightarrow\ a > b^k \Rightarrow (case - 3)$ $T(n) \in \Theta\left(n^{\log_3(9)}\right) \Rightarrow \Theta\left(n^2\right)$
$\begin{cases} T(64) = 200 \\ T(n) = 8T\left(\frac{n}{2}\right) + 5n^3, n > 64 \end{cases}$	Since, $a = 8,\ b = 2,\ k = 3$ $\Rightarrow 8 = 2^3 \Rightarrow\ a = b^k \Rightarrow (case - 2)$ $T(n) \in \Theta\left(n^3 \lg n\right)$

Instances (case)based Analysis Types

- There are three types of analysis:
  - Best case: Defines the input for which the algorithm takes minimum time. Input is the one for which the algorithm runs the fastest. Note that the best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest. The analysis of the best-case efficiency is not nearly as important as that of the worst-case or the average-case efficiency
  - Worst case: Defines the input for which the algorithm takes maximum time. Input is the one for which the algorithm runs the slowest. The way to determine the worst-case efficiency of an algorithm is, in principle, quite straightforward: analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count among all possible inputs of size n
  - Average case: Provides a prediction about the running time of the algorithm by assuming that the input is random. There are many important algorithms for which the average-case efficiency is much better than the worst-case efficiency. So, without the average-case analysis, computer scientists could have missed many important algorithms.

- Sequential Search

```
1: procedure SEQSEARCH(S[], x)
2:     location = 1
3:     while location <= n do
4:         if (S[location] == x)) then
5:             return
6:         end if
7:         location = location + 1
8:     end while
9:     location = −1
10:    return location
11: end procedure
```

- Best Case Analysis : $T_b(n) = 1 \in \Theta(1)$
- Worst Case Analysis $T_w(n) = \sum_{i=1}^{n} 1 = n \in \Theta(n)$

- lets run the algorithm to find 5 from the different inputs of same sizes and count the comparison operation

$$S_1 = [5, 4, 2, 6, 8, 7, 3]$$
$$S_2 = [6, 4, 2, 5, 8, 7, 3]$$
$$S_3 = [3, 4, 2, 6, 8, 7, 5]$$
$$S_4 = [9, 4, 2, 6, 8, 7, 3]$$

- Average Case Analysis-1 : We first analyze the case in which it is known that x is in S, where the items in S are all distinct, Based on this information, for $1\ k\ n$, the probability that x is in the kth array slot is 1/n. If x is in the kth array slot, the number of times the basic operation is done to locate x is k. This means that the average time complexity is given by

$$T_a(n) = \sum_{k=1}^{n} \left(k \times \frac{1}{n} \right) = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2} \in \Theta(n)$$

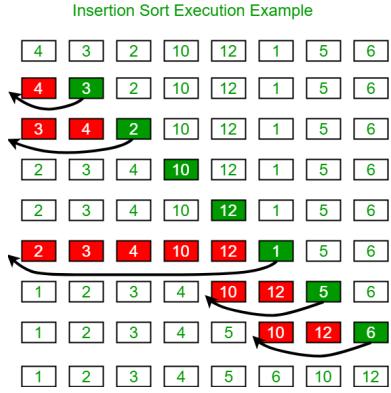
- Average Case Analysis-2 : Consider, the case in which x may not be in the array. To analyze this case we must assign some probability p to the event that x is in the array. If x is in the array, we will again assume that it is equally likely to be in any of the slots from 1 to n. The probability that x is in the kth slot is then p/n, and the probability that it is not in the array is $1 - p$.

$$\begin{aligned} T_a(n) &= \underbrace{\sum_{k=1}^{n} \left(k \times \frac{p}{n} \right)}_{} + \underbrace{(1-p) \sum_{k=1}^{n} 1}_{} \\ &= \frac{p}{n} \times \sum_{k=1}^{n} k + n(1-p) = \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) = \left(1 - \frac{p}{2}\right) n + \frac{p}{2} \in \Theta(n) \end{aligned}$$

Instances (case)based Analysis Types (cont...)

- Insertion Sort

```
1:  procedure insertionSort(S[])
2:      for i = 1; i < n; i++ do
3:          key = S[i]
4:          j = i − 1;
5:          while j >= 0 && S[j] > key do
6:              S[j + 1] = S[j]
7:              j = j − 1
8:          end while
9:          S[j + 1] = key
10:     end for
11: end procedure
```



- Consider the inputs of same sizes

$$S_1 = [1, 2, 3, 4, 5, 6, 10, 12]$$
$$S_2 = [12, 10, 6, 5, 4, 3, 2, 1]$$
$$S_3 = [12, 4, 2, 5, 10, 3, 6, 1]$$

- Best case analysis: $T_b(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$
- Worst case analysis:
  $T_w(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$
- Average case analysis: ?

- A pair $(A[i], A[j])$ is called an inversion if $i < j$ and $A[i] > A[j]$.
  - Worst-case: The largest number of inversions for $A[i]$ $0 \leq i \leq n-1$ is $n-1-i$. This happens if $A[i]$ is greater than all the elements to the right of it.

$$\sum_{i=0}^{n-1}(n-1-i) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} 1 - \sum_{i=1}^{n} i = n^2 - n - \frac{n(n+1)}{2} = \frac{2n^2 - 2n - n^2 - n}{2} = \frac{n(n-1)}{2}$$

  - Best-case: The largest number of inversions for $A[i]$ $0 \leq i \leq n-1$ is 0. This happens if $A[i]$ is smaller than all the elements to the right of it.
  - Average-case: Assuming that all elements are distinct and that inserting $A[i]$ in each of the $i+1$ possible positions among its predecessors is equally likely, we obtain the following for the expected number of key comparisons for i_{th} iteration

$$\frac{1}{i+1}\sum_{j=1}^{i+1} j = \frac{1}{i+1}.\frac{(i+1)(i+2)}{2} = \frac{i+2}{2}$$

for the average number of key comparisons

$$T_a(n) = \sum_{i=1}^{n-1} \frac{i+2}{2} = \frac{1}{2}\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 = \frac{1}{2}.\frac{(n-1)n}{2} + n - 1 = \frac{n^2 - n + 4n - 4}{4} = \frac{n^2 + 3n - 4}{4} \in \Theta(n^2)$$

- For the no-sentinel version, the expected number of key comparisons on the ith iteration is:

$$\frac{1}{i+1}\sum_{j=1}^{i} j + \frac{i}{i+1} = \frac{1}{i+1}\cdot\frac{i(i+1)}{2} + \frac{i}{i+1} = \frac{i}{2} + \frac{i}{i+1}$$

for the average number of key comparisons

$$T_a(n) = \sum_{i=1}^{n-1}\left(\frac{i}{2} + \frac{i}{i+1}\right)$$

Amortized Analysis

- Yet another type of efficiency is called amortized efficiency.
- It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure.
- So we can "amortize" the high cost of such a worst-case occurrence over the entire sequence in a manner similar to the way a business would amortize the cost of an expensive item over the years of the item's productive life.
- Motivation: given a **sequence** of operations, the vast majority of the operations are cheap, but some rare operations within the sequence might be expensive; thus a standard worst-case analysis might be overly pessimistic.
- Objective: to give a tighter bound for a **sequence** of operations.
- Basic idea: when the expensive operations are particularly rare, their costs can be "spread out" (amortized) to all operations. If the artificial amortized costs are still cheap, we will have a tighter bound of the whole sequence of operations.

Instances (case)based Analysis Types (cont...)

- The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.
- Amortized analysis differs from average-case analysis in:
  - Average-case analysis: **average over all input** , e.g., QuickSort algorithm performs well on "average" over all possible input even if it performs very badly on certain input.
  - Amortized analysis: **average over operations** , e.g., TableInsertion algorithm performs well on "average" over all operations even if some operations use a lot of time.