Underlying Concepts

Lecture 2
Asymptotic Analysis
RAM Model of computation
Reduction
Analyzing Algorithms

- **Analyzing** algorithms means determining the runtime of the algorithms. Doing so, will allow us to compare different algorithms.
- How to measure runtime of an algorithm?
  - Running the program? No
- We are looking for a way to measure the runtime of algorithm without implementing it. We want it to be independent of the underlying machine
- **Time Complexity**
Analyzing Algorithms

● **Analyzing** algorithms means determining the runtime of the algorithms.
● How to measure runtime of an algorithm?
  ○ Running the program? No
● We are looking for a way to measure the runtime of algorithm without implementing it. We want it to be independent of the underlying machine
● **Time Complexity**
Analyzing Algorithms: Time complexity

- Factors affecting the runtime of an algorithm
  - Number of steps of algorithm
  - input size

- The runtime of an algorithm, Time Complexity, is the number of steps performed on an input of size n

- $T(n)$: Time Complexity
Analyzing Algorithms: Time complexity

- **Model of computation**
  - Specify the steps in an algorithm
  - Specify measure of time, space, input size
  - The model should reflect reality
word-RAM (Random Access Machines) Model of Computation

- A simple computer with a CPU and memory
  - The algorithm is executed sequentially, one instruction at a time. No concurrent execution
  - Each “simple” operation like addition, subtraction, multiplication and division of fixed-size integer or floating point numbers, array indexing, read/write from/to memory etc. takes 1 step
word-RAM Model of Computation

- Random access means CPU can access memory location \( i \) in 1 time step.
- Each memory cell stores a \( w \)-bit integer.
  - Assume that \( w \geq \log n \). \( w \): the number of bits required to store the input size and \( n = \) input size.
  - That is why the basic operations on numbers take one step.
  - Hardware is changing to fit the input.
- **Runtime**: Number of constant-time operations.
- **Space**: Number of memory cells used.
- Is \( w \) large enough to hold any integer?
  - No, the large numbers are stored in array of words.
  - But, it is large enough to hold an address of a data structure if the data structure fits into RAM.
  - If the input is an array \( A[1..n] \), we want an index \( i \) in \( [1..n] \) to fit in a word.
A note about input size

- What is the runtime of the following **Fibonacci** code?
  - Exactly n iterations, so runtime is $O(n)$.
  - Is this a polynomial runtime?

```python
function Fibonacci(n)
    i = 0
    j = 1
    for k = 1 to n do
        j = i + j
        i = j
    return j
```

- Size of input here is actually $\log n$
- If the runtime is calculated to be $O(n)$ then it is not polynomial in size of input, but polynomial in $n$
Analyzing Algorithms: types of analysis

- Time complexity, $T(n)$, is a function of input size
- For a given size $n$, there are various inputs
- How do we combine runtime to one number?
  - **Worst Case Time Complexity**: max # steps algorithm takes on any input of size $n$
  - **Average Case Complexity**: avg # steps algorithm takes on inputs of size $n$
  - **Best Case Complexity**: min # steps algorithm takes on any input of size $n$
- Usually, we seek the worst case runtime. $T(n)$ mean the worst case time unless otherwise specified
  - it provides an upper bound on the runtime, and everybody likes a guarantee
  - average-case analysis is more difficult: needs assumption of statistical distribution of data (uniform)?
Analyzing time complexity of an algorithm

- Problem: Sorting
- Algorithm: Insertion sort
- find the worst-case time complexity of insertion sort
Insertion Sort Algorithm

Pass 1

Pass 2

Pass 3

Pass 4
Time Complexity of Insertion sort

\[ T(n) = n + \sum_{j=2}^{n} (n-1) + \sum_{j=2}^{n} (t_j - 1) + (n-1) \]

**Insertion-Sort(A)**

1. for \( j = 2 \) to \( A.length \)
2. \( key = A[j] \)
4. \( i = j - 1 \)
5. while \( i > 0 \) and \( A[i] > key \)
7. \( i = i - 1 \)
8. \( A[i+1] = key \)
Time Complexity of Insertion sort

\[ T(n) = 2n - 3 + 3 \sum_{j=2}^{n} t_j \]

- **Best case:** The array is sorted
  - We don’t enter the while loop and \( t_j = 1 \), \( T(n) = an + b \)
- **Average case:** the while loop breaks at halfway into the sorted list
  - \( t_j = j/2 \), \( T(n) = an^2 + bn + c \)
- **Worst case:** Input is reverse sorted
  - the loop continues for the entire sorted array and \( t_j = j \)
  - \( T(n) = dn^2 + en + f \)
Analyzing Algorithms: growth rate

\[ T_1(n) = n^2 \]
\[ T_2(n) = 1000n \]
Analyzing Algorithms: Asymptotic Analysis

- Asymptotic Analysis
  - $T(n)$: worst case time complexity of an algorithm as a function of input size

- We want $T(n)$ to be
  - Simple to express
  - Ignoring constant factors and focus on the behavior of the algorithm as the size of the input goes to infinity.
  - Look at growth of $T(n)$ as $n \to \infty$
Asymptotic Analysis of Algorithms: Big-Oh

- **Definition:**
  - Let \( f(n) \) and \( g(n) \) be two positive functions from \( \mathbb{N} \) to \( \mathbb{R}_{>0} \).
  - \( f(n) \) is \( O(g(n)) \) if there exists constants \( c > 0 \) and \( n_0 \geq 0 \) s.t., \( 0 \leq f(n) \leq c \cdot g(n) \) for all \( n \geq n_0 \).

  \( f \) is bounded by a constant times \( g \) for \( n \) sufficiently large – this is what asymptotic means.

- **Notations:**
  - \( O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \} \)
  - \( f(n) \in O(g(n)) \)
  - \( f(n) = O(g(n)) \) You should read the equal sign as “Is” not equal sign.
  - Big-O gives an upper bound, corresponds roughly to \( \leq \)
    - \( g(n) \) is an **asymptotic upper bound** for \( f(n) \).
Big-O: Example

$T(n) = 3n^2 + 2n + 1$ and $f(n) = n^2$. Is this true or not: $T(n) = O(f(n))$?

Proof: We need to show that there exist $n_0$ and $c$ so that $T(n) \leq c \cdot f(n)$ for $n \geq n_0$.

$3n^2 + 2n + 1 \leq c \cdot n^2$

Letting $n_0 = 1$, we conclude that $1 \leq n \leq n^2$

$3n^2 + 2n + 1 \leq 3n^2 + 2n^2 + n^2 = 6n^2$

$\leq 6(n^2)$

If we take $n_0 = 1$ and $c = 6$, the above inequality holds.
Example

Suppose $T(n) = 3n^2 + 2n + 1$ and $f(n) = n^{100}$. Can we say that $T(n) = O(f(n))$?

Proof: Take $n_0 = 1$ and $c = 6$. Then for any $n \geq n_0$, we have

$$3n^2 + 2n + 1 \leq 3n^{100} + 2n^{100} + n^{100} = 6n^{100}$$

$$\leq 6(n^{100})$$

Big-O does not guarantee a tight bound
Let \( f(n) = 3n^2 + 17n \log_2 n + 1000 \). Which of the following are true?

A. \( f(n) \) is \( O(n^2) \)

B. \( f(n) \) is \( O(n^3) \)

C. Both A and B.

D. Neither A nor B.
Asymptotic Analysis: Further Definition

- **Big Omega**: $f(n)$ is $\Omega(g(n))$ iff there are positive constants $n_0 > 0$ and $c > 0$ s.t.,

  $$f(n) \geq c \cdot g(n) \text{ for all } n \geq n_0$$

  - Alternatively: $\Omega(g(n)) = \{ f(n) : \text{there exist constants } c > 0, n_0 > 0 \text{ such that } 0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0 \}$
  - $g(n)$ is the **asymptotic lower bound** for $f(n)$
  - $\Omega(g(n))$ is the set of all functions that have the same or higher rate of asymptotic growth than $g(n)$

- **Notations**:
  - $f(n) \in \Omega(g(n))$
  - $f(n) = \Omega(g(n))$
Big-Omega: Example

Given \( f(n) \), which one is true?

\[ f(n) = 32n^2 + 17n + 1 \]

\[ f(n) = \Omega(n^2) \]

\[ f(n) = \Omega(n) \]

\[ f(n) = \Omega(n^3) \]
Asymptotic Analysis: Further Definition

- **Big Theta**: \( f(n) \) is \( \Theta(g(n)) \) iff there are positive constants \( c_1, c_2, n_0 > 0 \) such that eventually always \( c_1 g(n) \leq f(n) \leq c_2 g(n) \) for all \( n \geq n_0 \)
  - \( g(n) \) is a **tight** asymptotic bound for \( f(n) \)
  - \( f(n) \) is \( \Theta(g(n)) \) if \( f(n) \) is \( O(g(n)) \) and \( \Omega(g(n)) \)

- **Notation**:
  - \( f(n) \in \Theta(g(n)) \)
  - \( f(n) = \Theta(g(n)) \)

![Diagram depicting the relationship between f(n), c1g(n), and c2g(n) for n ≥ n0.](image)
Asymptotic Analysis: Various Notations

- **O**: Big-O notation
- **Ω**: Big-Omega notation
- **Θ**: Big-theta notation
Little-o

- \( f(n) \) is \( o(g(n)) \) if for any constant \( c > 0 \), there is a constant \( n_0 > 0 \) such that
  \[
  0 \leq f(n) < cg(n) \text{ for all } n \geq n_0
  \]

- Alternatively: \( o(g(n)) = \{ f(n) : \text{for any constant } c > 0, \text{ there is a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0 \} \)

- \( O \)-notation is like \( \leq \) and \( o \)-notation is like \( < \)

- Intuitively, \( f(n) = o(g(n)) \) means \( f(n) \) becomes insignificant compared to \( g(n) \) as \( n \) approaches infinity
  - \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \)
  - Using limit, we can compare the growth rate of functions
**ω-notation**

- \( \omega(g(n)) = \{ f(n) : \text{for any constant } c > 0, \text{ there is a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0 \} \)
- \( \Omega \)-notation is like \( \geq \) and \( \omega \)-notation are like \( > \)
- \( \lim_{n \to \infty} f(n)/g(n) = \infty \)
Little-oh and $\omega$-notation

- **Little-oh: Example**
  - $2n^2 = o(n^3)$ $(n_0 = 2/c +1)$
  - $2n^2 \neq o(n^2)$
  - $2n^2 = O(n^2)$

- **Little-$\omega$: Example**
  - $2n^2 = \omega(n)$
  - $2n^2 \neq \omega(n^2)$. 
  - $2n^2 = \Omega(n^2)$
Asymptotic notations properties

To find that some function $T(n) = O(f(n))$, we usually do not apply the definition formally but instead use a repository of known results:

**Adding functions**

$O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$

$Ω(f(n)) + Ω(g(n)) = Ω(\max\{f(n), g(n)\})$

$θ(f(n)) + θ(g(n)) = θ(\max\{f(n), g(n)\})$

**Multiplying two functions**

$O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$

$Ω(f(n)) \times Ω(g(n)) = Ω(f(n) \times g(n))$

$θ(f(n)) \times θ(g(n)) = θ(f(n) \times g(n))$
Proving A Big-Oh property

- **Max rule:**
  - If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, then $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$
    - If $T(n)$ is a sum of several terms, the one with the largest growth rate is kept and all others omitted.

- **Proof:**
  - if $T_1(n) = O(f(n))$ then, there are constants $c_1$ and $n_1$ such that $T_1(n) \leq c_1 f(n)$ when $n \geq n_1$.
  - if $T_2(n) = O(g(n))$ then, there are constants $c_2$ and $n_2$ such that $T_2(n) \leq c_2 g(n)$ when $n \geq n_2$.
  - Now, let $n_0 = \max(n_1, n_2)$. Therefore:
    - $T_1(n) \leq c_1 f(n)$ when $n \geq n_0$
    - $T_2(n) \leq c_2 g(n)$ when $n \geq n_0$
    - $T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n)$ when $n \geq n_0$. Now, let $c_3 = \max(c_1, c_2)$. Therefore:
      - $T_1(n) + T_2(n) \leq c_3 f(n) + c_3 g(n)$ when $n \geq n_0$
      - $T_1(n) + T_2(n) \leq c_3 (f(n) + g(n))$ when $n \geq n_0$
      - $T_1(n) + T_2(n) \leq 2 c_3 \max(f(n), g(n))$ when $n \geq n_0$
      - $T_1(n) + T_2(n) \leq c \max(f(n), g(n))$ when $n \geq n_0$ and $c = 2 c_3$
Big-O properties: Example

- $T_1(n) * T_2(n) = O(f(n) * g(n))$
  - If $T(n)$ is a product of several factors, all constants are omitted.
    - Example: $T(n) = O(2n^2)$ or $T(n) = O(n^2 + n)$ are not in correct form. The correct form is $T(n) = O(n^2)$. $n^2 + n$ is the sum of two terms, one is $O(n^2)$ and the other is $O(n)$. Therefore, the larger one is chosen. Also, $2n^2$ is the multiplication of two terms that are $O(1)$ and $O(n^2)$.
  - This is why the running time of two nested loops is $O(n^2)$
    - If $T(n)$ is a polynomial of degree $n$, $a_0 + a_1n + \cdots + a_kn^k$, then $T(n) = O(n^k)$
- If $T(n)$ is constant, then we say that $T(n)$ is $O(1)$
Relationship Between Asymptotic Notations

- Many of the relational properties of real numbers apply to asymptotic comparisons as well. Assume that $f(n)$ and $g(n)$ are asymptotically positive:

**Transitivity:**

- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$,
- $f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$,
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$,
- $f(n) = o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$,
- $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$.

As an exercise try to prove these
Relationship Between Asymptotic Notations

Reflexivity:

\[
\begin{align*}
f(n) &= \Theta(f(n)), \\
f(n) &= O(f(n)), \\
f(n) &= \Omega(f(n)).
\end{align*}
\]

Symmetry:

\[
\begin{align*}
f(n) &= \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)).
\end{align*}
\]

Transpose symmetry:

\[
\begin{align*}
f(n) &= O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)), \\
f(n) &= o(g(n)) \text{ if and only if } g(n) = \omega(f(n)).
\end{align*}
\]

Note that the reflexivity property does not hold for the $o$ and $\omega$ functions.
Comparing functions

There is an analogy between the asymptotic comparison of two functions $f$ and $g$ and the comparison of two real numbers $a$ and $b$:

- $f(n) = O(g(n))$ is like $a \leq b$,
- $f(n) = \Omega(g(n))$ is like $a \geq b$,
- $f(n) = \Theta(g(n))$ is like $a = b$,
- $f(n) = o(g(n))$ is like $a < b$,
- $f(n) = \omega(g(n))$ is like $a > b$.

For any two real numbers $a$ and $b$, exactly one of the following must hold: $a < b$, $a = b$, or $a > b$. Are any two functions asymptotically comparable?
Summation Rules

Assume $I$ is a finite set:

$$O\left(\sum_{i \in I}\right) = \sum_{i \in I} O(f(i))$$

$$\Omega\left(\sum_{i \in I}\right) = \sum_{i \in I} \Omega(f(i))$$

$$\Theta\left(\sum_{i \in I}\right) = \sum_{i \in I} \Theta(f(i))$$

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

Prove that $f(n) \in \Theta(g(n))$ implies $g(n) \in \Theta(f(n))$
Typical run-times and how they compare

- Common growth functions
  - $c$ (constant)
  - $\log n$ - binary search
  - $n$ - find max
  - $n \log n$ - sorting
  - $n^2$ - insertion sort
  - $n^3$ - multiplying two $n \times n$ matrices
  - $2^n$ - try all subsets
  - $n!$ - try all ordering of a set (e.g., travelling salesman)
  - More: $\log \log n$, $(\log n)^2$, $\sqrt{n}$

$\ll$ Is used for ordering: where $f(n) \ll g(n)$ means $f(n)$ has lower growth rate compared to $g(n)$

$c \ll \log \log n \ll \log n \ll (\log n)^2 \ll \sqrt{n} \ll n \ll n \log n \ll n^2 \ll n^3 \ll 2^n n!$
Time Complexity $T(n)$ vs. $n$

- $O(n!)$
- $O(2^n)$
- $O(n^3)$
- $O(n^2)$
- $O(n)$
- $O(vn)$
- $O(\log n)$
- $O(1)$
## Why runtime is important?

On a computer executing 1 instruction per nanosecond, running time for varying input size:

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>30</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>50</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>100</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>1,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>10,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>100,000</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>
Efficient runtime = Polynomial Time

- An algorithm runs in polynomial time if $T(n) = O(n^k)$ for some constant $k$ independent of the input size $n$.
- Why Polynomial time?
  - If problem size grows by at most a constant factor then so does the running time
    - Example: Suppose $T(n) = O(n^k)$, if $n$ is doubled
      - $T(n) \leq c \cdot n^k \rightarrow T(2n) \leq c(2n)^k \leq c2^k(n^k) = C(n^k)$
  - **Polynomial-time** is the set of running times that have this property
  - Typical running times are small degree polynomials, mostly less than $N^3$, at worst $N^6$, not $N^{100}$
  - **Exponential time**: If problem size increases by one, the runtime doubles
Comparing growth rate of functions using limit

Limit technique can be used for comparing growth rate of runtime functions

If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = c \) where \( 0 < c < \infty \) then \( f(n) \) is \( \Theta(g(n)) \)

If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \) then \( f(n) \) is \( o(g(n)) \)

If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \) then \( f(n) \) is \( \omega(g(n)) \)
**THEOREM 1  Basic Limit Laws**  
Assume that \( \lim_{x \to c} f(x) \) and \( \lim_{x \to c} g(x) \) exist. Then:

(i) **Sum Law:**
\[
\lim_{x \to c} \left( f(x) + g(x) \right) = \lim_{x \to c} f(x) + \lim_{x \to c} g(x)
\]

(ii) **Constant Multiple Law:** For any number \( k \),
\[
\lim_{x \to c} kf(x) = k \lim_{x \to c} f(x)
\]

(iii) **Product Law:**
\[
\lim_{x \to c} f(x)g(x) = \left( \lim_{x \to c} f(x) \right) \left( \lim_{x \to c} g(x) \right)
\]

(iv) **Quotient Law:** If \( \lim_{x \to c} g(x) \neq 0 \), then
\[
\lim_{x \to c} \frac{f(x)}{g(x)} = \frac{\lim_{x \to c} f(x)}{\lim_{x \to c} g(x)}
\]
Limit Laws

Limit of an Exponential Function:

\[ \lim_{x \to a} b^{f(x)} = b^{\lim_{x \to a} f(x)} \]

Limit of a Logarithm of a Function

\[ \lim_{x \to a} \log_b f(x) = \log_b \lim_{x \to a} f(x) \]
What Is and When To Use L'Hospital's Rule?

A method use to find the limit of a function of a \( \frac{0}{0} \) or \( \frac{\infty}{\infty} \) condition.

if \( \lim_{x\to a} f(x) = 0 \) and \( \lim_{x\to a} g(x) = 0 \)

or if \( \lim_{x\to a} f(x) = \pm \frac{\infty}{\infty} \) and \( \lim_{x\to a} g(x) = \pm \frac{\infty}{\infty} \)

then \( \lim_{x\to a} \frac{f(x)}{g(x)} = \lim_{x\to a} \frac{f'(x)}{g'(x)} \) (if the limit exists)

https://lifethroughamathematicianseyes.wordpress.com/2016/06/18/lhopital-rule-interesting-history/
Example

\[ f(n) = n^2 + 17n + 20 \]

\[ g(n) = n^2 \]
Example

\[ f(n) = (\ln n)^2 \]

\[ g(n) = n^{\frac{1}{2}} \]
Asymptotic bounds for polynomial function

Let \( f(n) = a_0 + a_1 n + \ldots + a_d n^d \) with \( a_d > 0 \)

Then \( f(n) \) is \( \Theta(n^d) \)

Proof.

\[
\lim_{n \to \infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0
\]
Logarithm Rules
Properties of Logarithms

All logarithm bases are assumed to be greater than 1 in the formulas below; \( \lg x \) denotes the logarithm base 2, \( \ln x \) denotes the logarithm base \( e = 2.71828 \ldots \); \( x, y \) are arbitrary positive numbers.

1. \( \log_a 1 = 0 \)
2. \( \log_a a = 1 \)
3. \( \log_a x^y = y \log_a x \)
4. \( \log_a xy = \log_a x + \log_a y \)
5. \( \log_a \frac{x}{y} = \log_a x - \log_a y \)
6. \( a^{\log_b x} = x^{\log_b a} \)
7. \( \log_a x = \frac{\log_b x}{\log_b a} = \log_a b \cdot \log_b x \)
Asymptotic Bound for Logarithm function

- Base of logarithm is not important.
  - \( \log_a n = O(\log_b n) \) for all constants \( a, b > 0 \)
  - Proof:
    \[
    \frac{\log_a n}{\log_b n} = \frac{1}{\log_b a}
    \]

- Logarithms: \( \log \) grows slower than every polynomial
  - For all \( n > 0 \), \( \log n = O(n^k) \)
  - Proof:
    \[
    \lim_{n \to \infty} \frac{\log_a n}{n^d} = 0
    \]
Techniques for Algorithm Analysis

- Use $\Theta$-bounds throughout the analysis and obtain a $\Theta$-bound for the complexity of the algorithm
- Prove a $O$-bound and a matching $\Omega$-bound separately to get a $\Theta$-bound.
Asymptotic Analysis of Insertion sort

**INSERTION-SORT**(*A*)

1. for *j* = 2 to *A*.length
2. \hspace{10mm} **key** = *A*[j]
3. \hspace{10mm} // Insert *A*[j] into the sorted sequence *A*[1..*j* - 1].
4. \hspace{10mm} *i* = *j* - 1
5. \hspace{10mm} while *i* > 0 and *A*[i] > **key**
6. \hspace{20mm} *A*[i + 1] = *A*[i]
7. \hspace{20mm} *i* = *i* - 1
8. \hspace{10mm} *A*[i + 1] = **key**
Runtime of Insertion sort: Worst case and Best case

- Worst-case:
  - The outer loop repeats n times. The inner loop repeats at most n-1 times for each iteration of the outer loop Therefore runtime is $O(n^2)$
  - There is actually a case that takes $c$ times $n^2$, therefore the worst-case runtime it is $\Theta(n^2)$

- Best-case:
  - It happens when the inner loop does not execute (the input is sorted), so it will be $\Omega(n)$
  - If you use $O$, it does not exclude a best-case runtime of 1 or $\lg n$
  - The best case actually takes $cn$, therefore it is $\Theta(n)$
Question

Does Big-Oh complexity means “worst case time complexity”

Does Big-Ω complexity means “best case time complexity”?
Big-Oh notation with multiple variables

Sometimes we will analyze an algorithm’s runtime in terms of several parameters.

**Definition.** \( f(m, n) \) is \( O(g(m, n)) \) if there exist constants \( c > 0, m_0 \geq 0, \) and \( n_0 \geq 0, \) such that \( 0 \leq f(m, n) \leq c \cdot g(m, n) \) for all \( n \geq n_0 \) or \( m \geq m_0. \)

**Example:**

- \( f(m, n) = 32mn^2 + 17mn + 32n^3 \)
- \( f(m, n) \) is both \( O(mn^2 + n^3) \) and \( O(mn^3) \)
- \( f(m, n) \) is \( O(n^3) \) if a precondition to the problem implies \( m \leq n \)
- \( f(m, n) \) is neither \( O(n^3) \) nor \( O(mn^2). \)

**Application:** Analyzing algorithms on graphs with \( n \) vertices and \( m \) edges.
Frequently encountered summations while analyzing algorithms

### Important Summation Formulas

1. \[
\sum_{i=l}^{u} 1 = 1 + 1 + \cdots + 1 = u - l + 1 \quad (l, u \text{ are integer limits, } l \leq u); \quad \sum_{i=1}^{n} 1 = n
\]

2. \[
\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n + 1)}{2} \approx \frac{1}{2} n^2
\]

3. \[
\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n + 1)(2n + 1)}{6} \approx \frac{1}{3} n^3
\]

4. \[
\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k + 1} n^{k + 1}
\]

5. \[
\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \quad (a \neq 1); \quad \sum_{i=0}^{n} 2^i = 2^{n+1} - 1
\]

6. \[
\sum_{i=1}^{n} i^2 = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n - 1)2^{n+1} + 2
\]

7. \[
\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n + \gamma, \text{ where } \gamma \approx 0.5772 \ldots \quad \text{(Euler’s constant)}
\]

8. \[
\sum_{i=1}^{n} \log i \approx n \log n
\]
For each of the following pairs of functions \( f(n) \) and \( g(n) \), state whether \( f(n) = O(g(n)) \), \( f(n) = \Omega(g(n)) \), \( f(n) = \Theta(g(n)) \), or none of the above.

1. \( f(n) = n^2 + 3n + 4 \), \( g(n) = 6n + 7 \)
2. \( f(n) = n \sqrt{n} \), \( g(n) = n^2 - n \)
3. \( f(n) = 2^n - n^2 \), \( g(n) = n^4 + n^2 \)
Reduction
Reduction

Algorithm design technique:

Using known algorithms to solve new problem. Don’t reinvent the wheel!

Example: 2SUM and 3SUM
2-Sum Problem

- **Input:**
  - An array of n integers: \( A[1 \ldots n] \)
  - A target integer number \( m \)

- **Output:**
  - \( i, j \) such that \( A[i] + A[j] = m \)

Example: \( A=[5, 12, 11, 2, 3, 22, 20] \) and \( m=23 \)
2-Sum problem: first algorithm

Algorithm

```
for i = 1 to n do
    for j = i to n do
        If A[i] + A[j] == m return (i, j)
    return fail
```

Runtime: $O(n^2)$

Correctness: It tries all possibilities

Can we do better?
2-Sum problem: second algorithm

Algorithm

Sort A
for each i
   Perform a binary search to find m-A[i]

Runtime: $O(n \log n) + n \times O(\log n)$

Can we do better?
2-Sum problem: third algorithm

- **Algorithm.** Instead of multiple search scan the array from both sides
- **A = [2, 3, 5, 11, 12, 20, 22], m=23**

```
Sort A
i=1, j=n
while i <= j do:
    if S > m:
        j = j-1
    elif S < m
        i = i+1
    else
        return i,j
return fail
```

- **Runtime:** $O(n)$ (after sorting)
2-Sum problem: third algorithm

- **Algorithm.** Instead of multiple search scan the array from both sides
- \( A = [2, 3, 5, 11, 12, 20, 22] \), \( m=23 \)

- **Correctness:**
  - If there is no solution, our algorithm will not find it
  - If there is a solution \( i' \) and \( j' \) then \( i' \geq i \) and \( j' \leq j \)
  - **Loop invariant:** At the start of each iteration of the for loop, \( i' \geq i \) and \( j' \leq j \) where \( A[i'] + A[j'] = m \)

Sort \( A \)

\[
\begin{align*}
i &= 1, \quad j = n \\
\text{while } i &\leq j \text{ do:} \\
&\quad \text{if } S > m: \\
&\quad &\quad j = j - 1 \\
&\quad \text{elif } S < m: \\
&\quad &\quad i = i + 1 \\
&\quad \text{else} \\
&\quad &\quad \text{return } i, j \\
\text{return fail}
\end{align*}
\]
The 3-Sum problem

- **Input:**
  - An array of n integers: A[1 \ldots n]
  - A target number m

- **Output:**

**First Algorithm.** Check all triplet (i, j, k)

**Runtime:** $O(n^3)$
The 3-Sum problem

Second Algorithm

- We can reduce 3Sum to 2Sum (Solve 3-Sum by calling 2-Sum)
- So run 2Sum with target m-A[k] for each k

\[
\text{3sum}(A, m) \\
\text{for } k=1 \text{ to } n \\
\quad r=2\text{-sum}(A, m-A[k]) \\
\quad \text{if } r \text{ is not fail} \\
\quad \quad \text{return } r \\
\text{return fail}
\]

- Runtime: \(O(n \cdot n \log n) = O(n^2 \log n)\)
- 2sum was \(O(n \log n) + O(n)\). We only need to sort once. Therefore:
- **Runtime** = \(O(n \log n) + O(n^2) = O(n^2)\)