Algorithm Analysis

- Algorithms that are equally correct can vary in their utilization of computational resources
  - time and memory
  - a slow program it is likely not to be used
  - a program that demands too much memory may not be executable on the machines that are available
Memory - Time

- It is important to be able to predict how an algorithm behaves under a range of possible conditions
  - usually different inputs
- The emphasis is on the time rather than on the space efficiency
  - memory is cheap!
Running Time

- The time to solve a problem increases with the size of the input
  - measure the rate at which the time increases
  - e.g., linearly, quadratically, exponentially
- This measure focuses on intrinsic characteristics of the algorithm rather than incidental factors
  - e.g., computer speed, coding tricks, optimizations, quality of compiler
Additional Consideration

- Sometimes, simple but less efficient algorithms are preferred over more efficient ones
  - the more efficient algorithms are not always easy to implement
  - the program is to be changed soon
  - time may not be critical
  - the program will run only few times (e.g., once a year)
  - the size of the input is always very small
Bubble sort

- Count the number of steps executed by the algorithm
  - step: basic operation
  - ignore operations that are executed a constant number of times

```latex
\text{for (} i = 1; \ i < (n - 1); \ i ++) \\
\text{for (} j = n; \ j < (i + 1); \ j --) \\
\text{if (} A[j - 1] > A[j] \text{)} \\
\text{swap}(A[j], A[j - 1]); //step}
```

\[ T(n) = c \ n^2 \]
Asymptotic Algorithm Analysis

- Measures the efficiency of a program as the size of the input becomes large
  - focuses on “growth rate”: the rate at which the running time of an algorithm grows as the size of the input becomes “big”
  - Complexity $\Rightarrow$ Growth rate
  - focuses on the upper and lower bounds of the growth rates
  - ignores all constant factors
Growth Rate

- Depending on the order of \( n \) an algorithm can be of:
  - Linear time Complexity: \( T(n) = c \, n \)
  - Quadratic Complexity: \( T(n) = c \, n^2 \)
  - Exponential Complexity: \( T(n) = c \, n^k \)

- The difference between an algorithm whose running time is \( T(n) = 10n \) and another with running time is \( T(n) = 2n^2 \) is tremendous:
  - for \( n > 5 \) the linear algorithm is much faster despite the fact that it has larger constant
  - it is slower only for small \( n \) but, for such \( n \), both algorithms are very fast
Upper - Lower Bounds

- Several terms are used to describe the running time of an algorithm:
  - The upper bound to its growth rate which is expressed by the Big-Oh notation
    - $f(n)$ is upper bound of $T(n) \Leftrightarrow T(n)$ is in $O(f(n))$
  - The lower bound to its growth rate which is expressed by the Omega notation
    - $g(n)$ is lower bound of $T(n) \Leftrightarrow T(n)$ is in $\Omega(g(n))$
  - If the upper and lower bounds are the same $\Leftrightarrow T(n)$ is in $\Theta(g(n))$
Big O (upper bound)

- **Definition:** \( T(n) \in O(g(n)) \) if \( \exists \ n \geq n_0, c \geq 1 \) \( T(n) \leq cg(n) \ \forall \ n \geq n_0 \)

- **Example:**
  
  \[
  T(n) = n^3 \Rightarrow T(n) \in O(n^3)
  
  T(n) = 6n^2 - 2n + 7 \Rightarrow T(n) \in O(n^2)
  
  T(n) = n^3 + 10^6 n^2 \Rightarrow T(n) \in O(n^3)
  \]
Properties of $O$

- **$O$ is transitive:** if $\begin{cases} f \in O(g) \\ g \in O(h) \end{cases} \implies f \in O(h)$

- **If** $f \in O(g) \implies f + g \in O(g)$

- **If** $\begin{cases} f \in O(f') \\ g \in O(g') \end{cases} \implies f \cdot g \in O(f' \cdot g')$

- **If** $\begin{cases} f \in O(f') \\ g \in O(g') \end{cases} \implies f + g \in O(f' + g')$
More on Upper Bounds

- $T(n) \in O(g(n)) \iff g(n)$ : upper bound for $T(n)$
- There exist many upper bounds
  - e.g., $T(n) = n^2 \in O(n^2), O(n^3), \ldots O(n^{100}) \ldots$
- Find the smallest upper bound!!
- $O$ notation “hides” the constants
  - e.g., $cn^2, c^2n^2, c!n^2, \ldots c^kn^2 \in O(n^2)$
Among different algorithms solving the same problem, prefer the algorithm with the smaller “rate of growth” $O$

- it will be faster for large $n$
- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < ... < O(2^n) < O(n^n)$
- $O(n), O(n \log n), O(n^2) < O(n^3)$: polynomial
- $O(\log n)$: logarithmic
- $O(n^k), O(2^n), (n!), O(n^n)$: exponential!!
<table>
<thead>
<tr>
<th>Complexity</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000n</td>
<td>.02</td>
<td>.5</td>
<td>.1</td>
<td>.2</td>
<td>.5</td>
<td>0</td>
</tr>
<tr>
<td>1000nlogn</td>
<td>.9</td>
<td>.3</td>
<td>.6</td>
<td>1.5</td>
<td>4.5</td>
<td>10</td>
</tr>
<tr>
<td>100n²</td>
<td>.04</td>
<td>.25</td>
<td>1.0</td>
<td>4.0</td>
<td>25</td>
<td>2min</td>
</tr>
<tr>
<td>100n³</td>
<td>.08</td>
<td>1.0</td>
<td>10</td>
<td>1.0min</td>
<td>21min</td>
<td>27hrs</td>
</tr>
<tr>
<td>nlogn</td>
<td>.4</td>
<td>1.1</td>
<td>220da</td>
<td>125cent</td>
<td>5.10⁶cent</td>
<td></td>
</tr>
<tr>
<td>2ⁿ/³</td>
<td>.0001</td>
<td>0.1</td>
<td>2.7hrs</td>
<td>3.10⁴cent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2ⁿ</td>
<td>1.0</td>
<td>35yrs</td>
<td>3.10⁴cent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nⁿ</td>
<td>58min</td>
<td>2.10⁹cent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Omega Ω (lower bound)

- Definition: \( f(n) \in \Omega(g(n)) \) if \( \exists \ c \geq 1 \)
  \[ f(n) \geq cg(n) \]

- \( g(n) \) is a lower bound of the rate of growth of \( f(n) \)
- \( f(n) \) increases faster than \( g(n) \) as \( n \) increases
- e.g: \( T(n) = 6n^2 - 2n + 7 \Rightarrow T(n) \) in \( \Omega(n^2) \)
More on Lower Bounds

- $\Omega$ provides lower bound of the growth rate of $T(n)$
  - if there exist many lower bounds
  - $T(n) = n^4 \in \Omega(n), \in \Omega(n^2), \in \Omega(n^3), \in \Omega(n^4)$
  - find the larger one $\Rightarrow T(n) \in \Omega(n^4)$
If the lower and upper bounds are the same:

\[
\begin{align*}
f \in O(g) \quad \left\{ \quad f \in \Omega(g) \right\} & \iff f \in \Theta(g) \\
f \in \Theta(g) \\
\end{align*}
\]

- e.g.: \( T(n) = n^3 + 135n \in \Theta(n^3) \)
Theorem: \( T(n) = \sum a_i n^i \in \Theta(n^m) \)

- \( T(n) = a_n n^m + a_{n-1} n^{m-1} + \ldots + a_1 n + a_0 \in O(n^m) \)

Proof:

\[
T(n) \leq |T(n)| \leq |a_n n^m| + |a_{m-1} n^{m-1}| + \ldots + |a_1 n| + |a_0| \\
\leq n^m (|a_m| + 1/n |a_{m-1}| + \ldots + 1/n^{m-1} |a_1| + 1/n^m |a_0|) \\
\leq n^m (|a_m| + |a_{m-1}| + \ldots + |a_1| + |a_0|) \\
\leq n^m (|a_m| + |a_{m-1}| + \ldots + |a_1| + |a_0|) \Rightarrow \\
T(n) \in O(n^m)
Theorem (cont.)

b) \( T(n) = a_m n^m + a_{m-1} n^{m-1} + ... a_1 n + a_0 \geq cn^m + cn^{m-1} + ... + cn + c \geq cn^m \)

where \( c = \min\{a_m, a_{m-1}, ..., a_1, a_0\} \Rightarrow T(n) \in \Omega(n^m) \)

\((a), (b) \Rightarrow T(n) \in \Theta(n^m)\)
Theorem: \[ T(n) = \sum_{i=1}^{n} i^k \in \Theta(n^{k+1}), \forall k \geq 0 \]

(a) \[ T(n) = \sum_{i=1}^{n} i^k \leq \sum_{i=1}^{n} n^k = n \cdot n^k \Rightarrow T(n) \in O(n^{k+1}) \]

(b) \[ 2 \cdot T(n) = \sum_{i=1}^{n} i^k + \sum_{i=1}^{n} (n-i+1)^k = \]

\[ \sum_{i=1}^{n} (i^k + (n-i+1)^k) \geq \sum_{i=1}^{n} \left( \frac{n}{2} \right)^k \Rightarrow \]

(either \( i \geq \frac{n}{2} \) or \( (n-i+1) \geq \frac{n}{2} \))

\[ T(n) \geq \frac{1}{2^{k+1}} n^{k+1} \Rightarrow T(n) \in \Omega(n^{k+1}) \]
Recursive Relations (1)

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
T(n-1) + n & \text{if } n > 1 
\end{cases} \]

\[ T(n) = T(n-1) + n = \]
\[ = T(n-2) + (n-1) + n = \]
\[ = \ldots \]
\[ = T(1) + 2 + \ldots + (n-1) + n = \]
\[ = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \Rightarrow T(n) \in \Theta(n^2) \]
Fibonacci Relation

\[ F(0) = 0 \]
\[ F(1) = 1 \]
\[ F(n) = F(n - 1) + F(n - 2) \text{ if } n \geq 2 \]

\( (a) \ F(n) = F(n - 1) + F(n - 2) \leq \)
\[ \leq 2F(n - 1) \leq 4F(n - 2) \ldots \]
\[ \leq 2^{n-1} F(1) = 2^{n-1} \quad \Rightarrow \quad F(n) \in O(2^n) \]

\( (b) \ F(n) = F(n - 1) + F(n - 2) \geq 2F(n - 2) \geq 4F(n - 4) \ldots \Rightarrow \)
\[
\begin{cases}
  F(n) \geq 2^{(n-1)/2} & \text{if } n \text{ odd} \\
  F(n) \geq 2^{(n-2)/2} & \text{if } n \text{ even}
\end{cases}
\]
\[
\Rightarrow \quad F(n) \in \Omega(2^{n/2}) = \Omega(\sqrt{2^n})
\]
Fibonacci relation (cont.)

- From (a), (b): \( F(n) \in \Omega(\sqrt{2^n}) \cap O(2^n) \)
- It can be proven that \( F(n) \in \Theta(\phi^n) \)

where
\[
\phi = \frac{1 + \sqrt{5}}{2} = 1.6180
\]

golden ratio
Binary Search

int BSearch(int table[a..b], key, n)
{
    if (a > b) return (-1);
    int middle = (a + b)/2;
    if (T[middle] == key) return (middle);
    else if (key < T[middle])
        return BSearch(T[a..middle - 1], key);
    else return BSearch(T[middle + 1..b], key);
}
Analysis of Binary Search

- Let $n = b - a + 1$: size of array and $n = 2^k - 1$ for some $k$ (e.g., 1, 3, 5 etc)
- The size of each sub-array is $2^{k-1} - 1$

$$T(n) = T(2^k - 1) = \begin{cases} 
  c & k = 0 \\
  d + T(2^{k-1} - 1) & k > 0 
\end{cases}$$

- $c$: execution time of first command
- $d$: execution time outside recursion
- $T$: execution time
Binary Search (cont.)

\[ T(n) = T(2^k - 1) = d + T(2^{k-1} - 1) = 2d + T(2^{k-2} - 1) = \ldots \]
\[ = kd + T(0) = kd + c = d \cdot \log(n+1) + c \]
\[ \Rightarrow T(n) \in O(\log n) \]
Binary Search for Random n

\[
T(n) = \begin{cases} 
  c & k = 0 \\
  d + \max\{T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\), T\left(\left\lceil \frac{n-1}{2} \right\rceil\right)\} & k > 0 
\end{cases}
\]

- \( T(n) \leq T(n+1) \leq \ldots \leq T(u(n)) \) where \( u(n) = 2^k - 1 \)
- For some \( k \): \( n < u(n) < 2n \)
- Then, \( T(n) \leq T(u(n)) = d \log(u(n) + 1) + c \Rightarrow \\
  d \log(2n + 1) + c \Rightarrow T(n) \in O(\log n) \)
Merge sort

list mergesort(list L, int n)
{
    if (n == 1) return (L);
    L₁ = lower half of L;
    L₂ = upper half of L;
    return merge (mergesort(L₁,n/2),
                  mergesort(L₂,n/2));
}

n: size of array L (assume L some power of 2)
merge: merges the sorted L₁, L₂ in a sorted array
Analysis of Merge sort

N steps per merge, \( \log_2 n \) steps/merge \( \Rightarrow \) \( O(n \log_2 n) \)
Analysis of Merge sort

\[ T(n) = \begin{cases} 
2T(n/2) + c_2n & n > 1 \\
 c_1 & n = 1 
\end{cases} \]

- Two ways:
  - recursion analysis
  - induction
Induction

- Let $T(n) = O(n \log n)$ then $T(n)$ can be written as $T(n) \leq a \ n \log n + b$, $a, b > 0$
- **Proof:** (1) for $n = 1$, true if $b \geq c_1$
  - (2) let $T(k) = a \ k \log k + b$ for $k=1,2,..n-1$
  - (3) for $k=n$ prove that $T(n) \leq a \ n \log n + b$

For $k = n/2$:

- $T(n/2) \leq a \ n/2 \log n/2 + b$

Therefore, $T(n)$ can be written as

$T(n) = 2 \{a \ n/2 \log n/2 + b\} + c_2 \ n = a \ n(\log n - 1)+ 2b + c_2 \ n$
Induction (cont.)

- \( T(n) = a \, n \log n - a \, n + 2 \, b + c_2 \, n \)

- Let \( b = c_1 \) and \( a = c_1 + c_2 \) \( \Rightarrow \)
  \( T(n) \leq (c_1 + c_2) \, n \log n + c_1 \Rightarrow \)
  \( T(n) \leq C \, n \log n + B \)
Recursion analysis of mergesort

- $T(n) = 2T(n/2) + c_2 n \leq$
  - $\leq 4T(n/4) + 2c_2 n \leq$
  - $\leq 8T(n/8) + 3c_2 n \leq$

  ……

- $T(n) \leq 2^i T(n/2^i) + ic_2n$ where $n = 2^i$
- $T(n) \leq n T(1) + c_2 n \log n \Rightarrow T(n) \leq n c_1 + c_2 n \log n \Rightarrow$
  - $T(n) \leq (c_1 + c_2) n \log n \Rightarrow$
  - $T(n) \leq C n \log n$
Best, Worst, Average Cases

- For some algorithms, different inputs require different amounts of time
  - e.g., sequential search in an array of size \( n \)

\[
\overline{x} = \sum p_i x_i = (n+1)/2
\]

- worst-case: element not in array \( \Rightarrow T(n) = n \)
- best-case: element is first in array \( \Rightarrow T(n) = 1 \)
- average-case: element between 1, \( n \) \( \Rightarrow T(n) = n/2 \)
Comparison

- The **best-case** is not representative
  - happens only rarely
  - useful when it has high probability
- The **worst-case** is very useful:
  - the algorithm performs at least that well
  - very important for real-time applications
- The **average-case** represents the typical behavior of the algorithm
  - not always possible
  - knowledge about data distribution is necessary