## 1. Understanding and analyzing algorithms


1.2 Time complexity, time as a function of the input size \( n \), Big \( O \), \( o \), Big \( \Omega \), \( \omega \), Big \( \Theta \)

1.3 Running time analysis for algorithms (Upper and Lower Bounds)

1.4 Recursive algorithms: Correctness (Induction), Time analysis (Guess and Check then Induction, Unraveling the Recurrence)

1.5 Divide-and-Conquer

## 2. Using graphs and graph algorithms

## 3. Using combinatorial reasoning to quantitatively analyze algorithms and systems

## 4. Using probability to algorithm analysis
Time Complexity and Big O
An **Algorithm** is a problem solving strategy as a sequence of **Steps**.

- **Steps** can be: - Comparing list elements (which is larger?)
  - Accessing a position in a list (probe for value)
  - Arithmetic operation (+, -, *, ...)
  - etc.

  “**Single step**” depends on the context.

**How long does a “Single step” take?**

- Hardware (CPU, climate, cache ...)
- Software (programming language, compiler)

The time our program takes will depend on

- Input size
- Number of steps the algorithm requires
- Time for each of these steps on our system
Goal: Estimate time as a function of the size of the input $n$

Ignore what we can't control

Focus on how time scales for large inputs

Which of these functions do you think has the "same" rate of growth?

A. All of them
B. $2^n$ and $n^2$
C. $n^2$ and $3n^2$
D. They're all different
Definition of Big O

For functions
\[ f(n) : \mathbb{N} \to \mathbb{R}, \ g(n) : \mathbb{N} \to \mathbb{R} \]
we say
\[ f(n) \in O(g(n)) \]
to mean there are constants, \( C \) and \( k \) such that
\[ |f(n)| \leq C|g(n)| \]
for all \( n \geq k \).

Example: \( f(n) = 3n^2 + 2n \)
\[ g(n) = n^2 \]
3\(n^2\) + 2\(n\) is big \( O \) of \( n^2 \)

What constants can we use to prove that?
A. \( C = 1/3, \ k = 2 \)
B. \( C = 5, \ k = 1 \)
C. \( C = 10, \ k = 2 \)
D. None: \( f(n) \) isn't big \( O \) of \( g(n) \)
Big O is just an upper bound

O is an upper bound, not the exact amount of time.

Sometimes, this bound is not tight, i.e., there are smaller upper bounds that might be also true.

We will come back to this later in the class.
Is \( f(n) \) is big \( O \) of \( g(n) \)? i.e. Is \( f(n) \in O(g(n)) \)?

**Approach 1:** Look for constants \( C \) and \( k \).

**Approach 2:** Use properties

- **Domination**
  
  If \( f(n) \leq g(n) \) for all \( n \) then \( f(n) \) is big-\( O \) of \( g(n) \).

- **Transitivity**
  
  If \( f(n) \) is big-\( O \) of \( g(n) \), and \( g(n) \) is big-\( O \) of \( h(n) \), then \( f(n) \) is big-\( O \) of \( h(n) \).

- **Additivity/Multiplicativity**
  
  If \( f(n) \) is big-\( O \) of \( g(n) \), and if \( h(n) \) is nonnegative, then \( f(n) \cdot h(n) \) is big-\( O \) of \( g(n) \cdot h(n) \) ... where \( * \) is either addition or multiplication.

- **Sum is maximum**
  
  \( f(n) + g(n) \) is big-\( O \) of the \( \max(f(n), g(n)) \)

- **Ignoring constants**
  
  For any constant \( c \), \( cf(n) \) is big-\( O \) of \( f(n) \).

**Approach 3:** The limit method. Consider the limit

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)}
\]

If this limit exists and is 0: then \( f(n) \) grows strictly slower than \( g(n) \).

If this limit exists and is a constant \( c > 0 \): then \( f(n) \), \( g(n) \), grow at the same rate.

If the limit tends to infinity: then \( f(n) \) grows strictly faster than \( g(n) \).

If the limit doesn't exist for a different reason ... use another approach.
Is $f(n)$ big $O$ of $g(n)$? i.e. Is $f(n) \in O(g(n))$?

**Approach 1:** Look for constants $C$ and $k$.

**Approach 2:**
- Domination: If $f(n) \leq C \cdot g(n)$, then $f(n)$ is big-$O$ of $g(n)$.
- Transitivity: If $f(n)$ is big-$O$ of $g(n)$, and $g(n)$ is big-$O$ of $h(n)$, then $f(n)$ is big-$O$ of $h(n)$.

**Additivity/Multiplicativity:** If $f(n)$ is big-$O$ of $g(n)$, and if $h(n)$ is nonnegative, then $f(n) \cdot h(n)$ is big-$O$ of $g(n) \cdot h(n)$ ... where $\cdot$ is either addition or multiplication.

**Sum is Maximum:** $f(n) + g(n)$ is big-$O$ of the max($f(n)$, $g(n)$)

**Ignoring Constants:** For any constant $c$, $cf(n)$ is big-$O$ of $f(n)$.

**Approach 3:** The limit method. Consider the limit

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

If this limit exists and is 0: then $f(n)$ grows strictly slower than $g(n)$.
If this limit exists and is a constant $c > 0$: then $f(n)$, $g(n)$, grow at the same rate.
If the limit tends to infinity: then $f(n)$ grows strictly faster than $g(n)$.
if the limit doesn't exist for a different reason ... use another approach.

Rosen pp. 210-213
Big O: How to compute?

Look at terms one-by-one and drop constants. Then only keep maximum.

It is important to distinguish constants from variables. Both are numbers. What makes a number a “constant”? Just as we discussed when we listed “single step" operations, a constant is a number that doesn't change as the input grows. So 7 is a constant, as is “the time taken by a compare operation”. “The time taken to run one pass of BubbleSort" is NOT a constant, because it will increase as the input array grows in size.
Big O: How to compute?

One useful fact is that the leading term of a polynomial determines its order. For example, if \( f(x) = 3x^5 + x^4 + 17x^3 + 2 \), then \( f(x) \) is of order \( x^5 \). This is stated in Theorem 4, whose proof is left as Exercise 50.

Let \( f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \), where \( a_0, a_1, \ldots, a_n \) are real numbers with \( a_n \neq 0 \). Then \( f(x) \) is of order \( x^n \).

The polynomials \( 3x^8 + 10x^7 + 221x^2 + 1444, x^{19} - 18x^4 - 10,112, \) and \( -x^{99} + 40,001x^{98} + 100,003x \) are of orders \( x^8, x^{19}, \) and \( x^{99} \), respectively.
Big O: Examples (1/5)

Let $f(n) = n^2 + 2n + 1$. Find a simpler function of the same order. $f(n) = n^2 + 2n + 1$ is big $O$ of $g(n) = n^2$.

\[
\begin{align*}
    n^2 + 2n + 1 & \leq n^2 + 2n^2 + n^2 \quad \text{(for } n \geq 1) \\
    & \leq 4n^2
\end{align*}
\]

**Witnesses:**
- $k = 1$
- $C = 4$

**Other Witnesses:**
- $k = 100$
- $C = 100$
Big O: Examples (2/5)

\(f(n) = 7n^2\) is \(O(n^3)\)

\[
7n^2 \\
\leq \quad n^3 \quad \text{(for } n \geq 7) 
\]

**Witnesses:**
\(k = 7\)
\(C = 1\)

**Other Witnesses:**
\(k = 100\)
\(C = 100\)

\[|f(n)| \leq C|g(n)|\]
for all \(n \geq k\).
Big O: Examples (3/5)

\[ f(n) = n^2 \log n + 3n^2 + 3n \text{ is } O(n^2 \log n) \]

\[
\begin{align*}
& n^2 \log n + 3n^2 + 3n \\
\leq & n^2 \log n + 3n^2 \log n + 3n^2 \log n \quad \text{(for } n \geq 2) \\
\leq & 7n^2 \log n
\end{align*}
\]

**Witnesses:**

\[ k = 2 \]
\[ C = 7 \]

**Other Witnesses:**

\[ k = 100 \]
\[ C = 100 \]
Big O: Examples (4/5)

\[ f(n) = n! \text{ is } O(n^n) \]

\[
\begin{align*}
&n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 \\
\leq &\quad n \cdot n \cdot n \cdot \ldots \cdot n \cdot n \quad \text{(for } n \geq 1) \\
\leq &\quad n^n
\end{align*}
\]

**Witnesses:**

\[ k = 1 \]

\[ C = 1 \]

------------------------------------------------------------------------------------------------------------------

This implies that

\[ f(n) = \log n! \text{ is big } O(n \log n). \]

------------------------------------------------------------------------------------------------------------------
Big O: Examples (5/5)

Number of comparisons of list elements

\[ 1 + 2 + \ldots + n-1 + n = \frac{n(n-1)}{2} \]

Rewrite this formula in order notation:

A. \( O(n) \)
B. \( O(n(n-1)) \)
C. \( O(n^2) \)
D. \( O(1/2) \)
E. None of the above

\[ |f(n)| \leq C|g(n)| \]
for all \( n \geq k \).

Number of comparisons of list elements

\[ 1 + 2 + \ldots + n-1 + n = \frac{n(n-1)}{2} \]

\[ \leq n + n + \ldots + n + n \quad \text{(for } n \geq 1\text{)} \]

\[ \leq n^2 \]

**Witnesses:**
\[ k = 1 \]
\[ C = 1 \]

\[ \frac{n(n-1)}{2} \]

\[ \leq n(n-1) \text{ (for } n \geq 1\text{)} \]

**Witnesses:**
\[ k = 1 \]
\[ C = 1 \]
Big O classes

- $O(n!)$: factorial
- $O(2^n)$: exponential
- $O(n^2)$: quadratic
- $O(n \log n)$: “linearithmic”
- $O(n)$: linear
- $O(\log n)$: logarithmic
- $O(1)$: constant
Other asymptotic classes

\[ f(n) \in O(g(n)) \]
means there are constants \( C \) and \( k \) such that
\[ |f(n)| \leq C|g(n)| \]
for all \( n > k \).

\[ f(n) \in \Omega(g(n)) \]
means \( g(n) \in O(f(n)) \)

\[ f(n) \in \Theta(g(n)) \]
means \( f(n) \in O(g(n)) \) and \( g(n) \in O(f(n)) \)

Big \( O \) is an upper bound.

Big \( \Omega \) is a lower bound.

Big \( \Theta \) is a tight bound.
# Content

<table>
<thead>
<tr>
<th>1</th>
<th>Understanding and analyzing algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td>Time complexity, time as a function of the input size ( n ), Big ( O ), ( o ), Big ( \Omega ), ( \omega ), Big ( \Theta )</td>
</tr>
<tr>
<td>1.3</td>
<td><strong>Running time analysis for algorithms (Upper and Lower Bounds)</strong></td>
</tr>
<tr>
<td>1.4</td>
<td>Recursive algorithms: Correctness (Induction), Time analysis (Guess and Check then Induction, Unraveling the Recurrence)</td>
</tr>
<tr>
<td>1.5</td>
<td>Divide-and-Conquer</td>
</tr>
</tbody>
</table>

| 2 | Using graphs and graph algorithms |

| 3 | Using combinatorial reasoning to quantitatively analyze algorithms and systems |

| 4 | Using probability to algorithm analysis |
Running Time Analysis for Algorithms
Running time analysis for algorithms

LinearSearch(A[1, ..., n] an integer array, x an integer)
   i ← 1
   While i ≤ n and x does not equal A[i]
      i ← i + 1
   If i ≤ n, Then location ← i
   Else location ← 0
   Return location

BinarySearch(A[1, ..., n] a sorted integer array, x an integer)
   i ← 1
   j ← n
   While i ≤ j
      m ← (i + j)/2
      If x = A[m], Then Return m
      If x > A[m], Then i ← m + 1
      If x < A[m], Then j ← m - 1
   Return 0

MinSort(a_1, a_2, ..., a_n: real numbers with n >=2)
   for i := 1 to n-1
      m := i
      for j := i+1 to n
         if (a_j < a_m) then m := j
      interchange a_i and a_m

   { a_1, ..., a_n is in increasing order}
1. Basic operations: operation whose time doesn't depend on input size

Examples:

```
i ← 1
j ← n

m ← (i + j) / 2
If x = A[m], Then Return m
If x > A[m], Then i ← m + 1
If x < A[m], Then j ← m - 1

if (a_j < a_m) then m := j
```
Running time analysis for algorithms

**LinearSearch**($A[1, ... , n]$ an integer array, $x$ an integer)

\[
\begin{align*}
1 - & 1 \\
\textbf{While} & i \leq n \text{ and } x \text{ does not equal } A[i] \\
& i \leftarrow i + 1 \\
\textbf{If} & i \leq n, \text{ Then location } \leftarrow i \\
\textbf{Else} & \text{location } \leftarrow 0 \\
\textbf{Return} & \text{location}
\end{align*}
\]

**BinarySearch**($A[1, ... , n]$ a sorted integer array, $x$ an integer)

\[
\begin{align*}
i & \leftarrow 1 \\
j & \leftarrow n \\
\textbf{While} & i \leq j \\
& m \leftarrow (i + j)/2 \\
\textbf{If} & x = A[m], \text{ Then Return } m \\
\textbf{If} & x > A[m], \text{ Then } i \leftarrow m + 1 \\
\textbf{If} & x < A[m], \text{ Then } j \leftarrow m - 1 \\
\textbf{Return} & 0
\end{align*}
\]

**MinSort**($a_1, a_2, ..., a_n$: real numbers with $n \geq 2$)

\[
\begin{align*}
\textbf{for} & i := 1 \text{ to } n-1 \\
& m := i \\
\textbf{for} & j := i+1 \text{ to } n \\
& \textbf{if} \ (a_i < a_m) \text{ then } m := j \\
\text{interchange } a_i \text{ and } a_m \\
\{a_1, ..., a_n \text{ is in increasing order}\}
\end{align*}
\]
Running time analysis for algorithms

1. Basic operations: operation whose time doesn't depend on input size

```plaintext
i ← 1
j ← n
m ← (i + j) / 2
If x = A[m], Then Return m
If x > A[m], Then i ← m + 1
If x < A[m], Then j ← m - 1
```

2. Consecutive (non-nested) code: Run Prog1 followed by Prog2

If Prog1 takes $O(f(n))$ time and Prog2 takes $O(g(n))$ time, what's the big-O class of runtime for running them consecutively? $O(f(n) + g(n))$
Running time analysis for algorithms

LinearSearch(A[1, ..., n] an integer array, x an integer)
   i ← 1
   While i ≤ n and x does not equal A[i]
      i ← i + 1
   If i ≤ n, Then location ← i
   Else location ← 0
   Return location

BinarySearch(A[1, ..., n] a sorted integer array, x an integer)
   i ← 1
   j ← n
   While i ≤ j
      m ← (i + j)/2
      If x = A[m], Then Return m
      If x > A[m], Then i ← m + 1
      If x < A[m], Then j ← m - 1
   Return 0

Running time analysis for algorithms

- \( O(n) \)
- \( O(\log n) \)
- \( O(n + \log n) = O(n) \)
Running time analysis for algorithms

1. Basic operations: operation whose time doesn't depend on input size
   \[ i := 1 \]
   \[ j := n \]
   \[ m := (i + j) / 2 \]
   \[
   \text{if } (a_i < a_m) \text{ then } m := j
   \]

2. Consecutive (non-nested) code: Run Prog1 followed by Prog2
   If Prog1 takes \(O(f(n))\) time and Prog2 takes \(O(g(n))\) time, what's the big-O class of runtime for running them consecutively? \(O(f(n)+g(n))\)

3. Subroutine Call method \(S\) on (some part of) the input.
   If sub-routine \(Sub\) has runtime \(O(Sub(n))\) and if we call \(Sub\) at most \(t\) times, then the runtime is \(O(t \cdot Sub(m))\) where \(m\) is the size of the biggest input given to \(Sub\).
   Distinguish between the size of input to subroutine \((m)\) and the size of the original input \((n)\) to main procedure.
Running time analysis for algorithms

LinearSearch(A[1, ..., n] an integer array, x an integer)
   i ← 1
   While i ≤ n and x does not equal A[i]
      i ← i + 1
   If i ≤ n, Then location ← i
   Else location ← 0
   Return location

BinarySearch(A[1, ..., n] a sorted integer array, x an integer)
   i ← 1
   j ← n
   While i ≤ j
      m ← (i + j)/2
      If x = A[m], Then Return m
      If x > A[m], Then i ← m + 1
      If x < A[m], Then j ← m - 1
   Return 0

MinSort(a₁, a₂, ..., aₙ: real numbers with n ≥2)
   for i := 1 to n-1
      m := i
      for j := i+1 to n
         if (aᵢ < aᵦ) then m := j
         interchange aᵢ and aᵦ
   { a₁, ..., aₙ is in increasing order}

\[ \text{Running time} = O((n-1) \cdot (1)) = O(n-1) \]
Running time analysis for algorithms

4. Simple Loops

\[
\text{while (Guard Condition)} \\
\quad \text{Body of the Loop}
\]

If Guard Condition uses basic operations and body of the loop is constant time, then runtime is of the same order as the number of iterations.
Running time analysis for algorithms

LinearSearch(A[1, ..., n] an integer array, x an integer)
\[ \begin{align*}
& i \leftarrow 1 \\
& \text{While } i \leq n \text{ and } x \text{ does not equal } A[i] \\
& \quad i \leftarrow i + 1 \\
& \quad \text{If } i \leq n, \text{ Then location } \leftarrow i \\
& \quad \text{Else location } \leftarrow 0 \\
& \text{Return location}
\end{align*} \]

\[ O(n \cdot (1)) = O(n) \]

BinarySearch(A[1, ..., n] a sorted integer array, x an integer)
\[ \begin{align*}
& i \leftarrow 1 \\
& j \leftarrow n \\
& \text{While } i \leq j \\
& \quad m \leftarrow (i + j)/2 \\
& \quad \text{If } x = A[m], \text{ Then Return } m \\
& \quad \text{If } x > A[m], \text{ Then } i \leftarrow m + 1 \\
& \quad \text{If } x < A[m], \text{ Then } j \leftarrow m - 1 \\
& \text{Return 0}
\end{align*} \]

\[ O((\log(n)+1) \cdot (1)) = O(\log n) \]

MinSort(a_1, a_2, ..., a_n: real numbers with n >=2 )
\[ \begin{align*}
& \text{for } i := 1 \text{ to } n-1 \\
& \quad m := i \\
& \quad \text{for } j := i+1 \text{ to } n \\
& \quad \quad \text{if } ( a_j < a_m ) \text{ then } m := j \\
& \text{interchange } a_i \text{ and } a_m
\end{align*} \]

\{ a_1, ..., a_n is in increasing order \}

\[ O((n-i) \cdot (1)) = O(n) \]
Running time analysis for algorithms

4. Simple Loops

```
while (Guard Condition)
    Body of the Loop
```

If Guard Condition uses basic operations and body of the loop is constant time, then runtime is of the same order as the number of iterations.

5. Nested Loops

```
while (Guard Condition)
    Body of the Loop,
    May contain other loops, etc.
```

If Guard Condition uses basic operations and Body of the loop has runtime $O(\text{BoL})$ in the worst case, then runtime is $O(t \cdot \text{BoL})$, where $t$ is the bound on the number of iterations through the loop.
Running time analysis for MinSort

MinSort(a_1, a_2, ..., a_n: real numbers with n \geq 2 )

\textbf{for} \ i := 1 \ \textbf{to} \ n-1 \\
\hspace{1em} m := i \\
\hspace{2em} \textbf{for} \ j := i+1 \ \textbf{to} \ n \\
\hspace{3em} \textbf{if} \ ( a_j < a_m ) \ \textbf{then} \ m := j \\
\hspace{2em} \text{interchange} \ a_i \ \text{and} \ a_m \\

\{a_1, ..., a_n \text{ is in increasing order}\}
Running time analysis for MinSort: Upper Bound

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
<th>Big O Analysis</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( i := 1 ) to ( n-1 )</td>
<td>( O(1) ) Basic Operation</td>
<td>( O(1) ) times</td>
</tr>
<tr>
<td>2</td>
<td>( m := i )</td>
<td>( O(1) ) Basic Operation</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>3</td>
<td>( j := i+1 ) to ( n )</td>
<td>( O(n-i) = O(n) ) Simple Loop</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>4</td>
<td>if ( a_j &lt; a_m ) then ( m := j )</td>
<td>( O(n-i) = O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>5</td>
<td>interchange ( a_i ) and ( a_m )</td>
<td>( O(1) ) Basic Operation</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

We work from the inside out, going from the body of the inside loop to the main algorithm.

The inner-most Line 4 is defined in terms of a fixed number of basic operations: a comparison, some logic, some variable writes. It is thus \( O(1) \).

Line 3 is a loop, with constant time line 4 inside. It repeats \( n-i \) times, so the total time is \( O(n-i) \). This ranges from constant time when \( i \) reaches \( n-1 \) to \( O(n) \) when \( i=1 \). So the worst-case is \( O(n) \).

Line 2 and 5 are constant time, so the body of the FOR loop in line 1 takes \( O(1+n+1) = O(n) \) total.

Finally, line 1 is a loop whose body is \( O(n) \) and gets repeated \( n-1<n \) times. So the whole algorithm is \( O(n^2) \).
Running time analysis for MinSort: Lower Bound

MinSort($a_1, a_2, \ldots, a_n$: real numbers with $n \geq 2$)

1. **for** $i := 1$ **to** $n-1$
   
   2. $m := i$  
      
   3. **for** $j := i+1$ **to** $n$
      
   4. **if** ($a_j < a_m$) then $m := j$
      
   5. interchange $a_i$ and $a_m$

---

$O$ is an upper bound, not always tight. We can ask: is the running time also lower bounded by a quadratic, or is there a smaller upper bound? We don't need to find the “worst-case input” or give an exact formula to answer this question, just show that sometimes the algorithm performs at least on the order of $n^2$ operations of some kind.

Look at the first $n/2$ times we run the loop in line 3. Then $i \leq n/2$, so $n-i \geq n/2$.

Thus, we run it at least $n/2 \times n/2 = n^2/4$ times total. This is $\Omega(n^2)$.

Thus, the time is both $O(n^2)$ and $\Omega(n^2)$, so our analysis is tight, and the time is $\Phi(n^2)$.

So in this example, our first analysis is the best possible.