Big-Oh

Definition:

\[ f(n) = O(g(n)) \text{ means } \exists n_0, c > 0 \quad \forall n \in \mathbb{N} \; n \geq n_0, \quad f(n) \leq c \cdot g(n) \]

Meaning: \( f \) grows no faster than \( g \), up to constant factors
Big-Oh

Definition:

\( f(n) = O(g(n)) \) means

\[ \exists \ n_0, \ c > 0 \quad \forall n \in \mathbb{N} \ n \geq n_0, \quad f(n) \leq c \cdot g(n) \]

Example 1:

\( 10n = O(n \log n) \) ?

\( c = ? , \ n_0 = ? \) such that \( \forall n \geq n_0, \ 10n \leq c \cdot n \log n. \)
Big-Oh

Definition:
\[ f(n) = O(g(n)) \text{ means } \exists n_0, c > 0 \quad \forall n \in \mathbb{N} \ n \geq n_0, \quad f(n) \leq c \cdot g(n) \]

Example 1:
\[ 10n = O(n \log n) \]
\[ c = 10, \ n_0 = 2 \text{ such that } \forall n \geq 2, \ 10n \leq 10n \log n. \]

Are there other options for \( c, n_0 \)?
Big-Oh

Definition:

\[ f(n) = O(g(n)) \text{ means } \exists \, n_0, \, c > 0 \quad \forall n \in \mathbb{N} \, n \geq n_0, \quad f(n) \leq c \cdot g(n) \]

Example 1:

\[ 10n = O(n \log n). \]

\( c = 10, \, n_0 = 2 \) such that \( \forall n \geq 2, \, 10n \leq 10n \log n. \)

\( c = 1, \, n_0 = 2^{10} \) such that \( \forall n \geq 2^{10}, \, 10n \leq n \log n. \)
Example 2:

$100n^2 = O(2^n)$?

c = ?, $n_0 = ?$ such that $\forall n \geq n_0$, $100 \ n^2 \leq c \cdot 2^n$. 
Example 2:

\[ 100n^2 = O(2^n). \]

\[ c = 100, \ n_0 = 4 \] such that \( \forall n \geq 4, \ 100n^2 \leq 100 \cdot 2^n. \)

Are there other options for \( c, \ n_0? \)
Example 2:

$100n^2 = O(2^n)$.

c = 100, $n_0 = 4$ such that $\forall n \geq 4$, $100n^2 \leq 100 \cdot 2^n$.

c = 25, $n_0 = 8$ such that $\forall n \geq 8$, $100n^2 \leq 25 \cdot 2^n$. 
Example 2:

\[ 100n^2 = O(2^n). \]

\[ c = 100, \ n_0 = 4 \text{ such that } \forall n \geq 4, \ 100n^2 \leq 100 \cdot 2^n. \]

\[ c = 25, \ n_0 = 8 \text{ such that } \forall n \geq 8, \ 100n^2 \leq 25 \cdot 2^n. \]

Example 3:

\[ n^2 \log n = O(100 \ n^2) \ ? \]

\[ c = \ ?, \ n_0 = \ ? \text{ such that } \forall n \geq n_0, \ n^2 \log n \leq c \cdot 100 \ n^2. \]
Example 2:

\[ 100n^2 = O(2^n) \]

c = 100, \( n_0 = 4 \) such that \( \forall n \geq 4, \ 100n^2 \leq 100 \cdot 2^n \).

Example 3:

\[ n^2 \log n \neq O(100 \ n^2) \]

\( \forall c, \ n_0 \exists \ n \geq n_0 \) such that \( n^2 \log_2 n > c \cdot 100 \ n^2 \).

\[ n > 2^{100c} \]

\[ \Rightarrow \ n^2 \log_2 n > c \cdot 100n^2. \]
Example 4:

$2^n = O(2^{n/2})$?

c = ?, $n_0 = ?$ such that $\forall n \geq n_0$, $2^n \leq c \cdot 2^{n/2}$. 
Example 4:

$2^n \neq O(2^{n/2})$.

$\forall c, n_0 \exists n \geq n_0$ such that $2^n > c \cdot 2^{n/2}$.

$n > 2 \log_2 c$

$2^n = 2^{n/2} \cdot 2^{n/2} > c \cdot 2^{n/2}$. 
• $n \log n = O(n^2)$ ?
• $n^2 = O(n^{1.5} \log 10n)$ ?
• $2^n = O(n^{1000000})$ ?
• $\sqrt{2^{\log n}} = O(n^{1/3})$ ?
• $n^{\log \log n} = O((\log n)^{\log n})$ ?
• $2^n = O(4^{\log n})$ ?
• $n! = O(2^n)$ ?
• $n! = O(n^n)$ ?
• $n2^n = O(2^n \log n)$ ?
• $n \log n = O(n^2)$.
• $n^2 = O(n^{1.5} \log 10n)$?
• $2^n = O(n^{1000000})$?
• $\sqrt{2} \log n = O(n^{1/3})$?
• $n^{\log \log n} = O((\log n)^{\log n})$?
• $2^n = O(4^{\log n})$?
• $n! = O(2^n)$?
• $n! = O(n^n)$?
• $n2^n = O(2^n \log n)$?
• $n \log n = O(n^2)$.
• $n^2 \neq O(n^{1.5} \log 10n)$.
• $2^n = O(n^{1000000})$?
• $\sqrt{2} \log n = O(n^{1/3})$?
• $n^{\log \log n} = O((\log n)^{\log n})$?
• $2^n = O(4^{\log n})$?
• $n! = O(2^n)$?
• $n! = O(n^n)$?
• $n2^n = O(2^{n \log n})$?
• \( n \log n = O(n^2) \).
• \( n^2 \neq O(n^{1.5} \log 10n) \).
• \( 2^n \neq O(n^{1000000}) \) ?
• \( \sqrt{2} \log n = O(n^{1/3}) \) ?
• \( n^{\log \log n} = O((\log n)^{\log n}) \) ?
• \( 2^n = O(4^{\log n}) \) ?
• \( n! = O(2^n) \) ?
• \( n! = O(n^n) \) ?
• \( n2^n = O(2^n \log n) \) ?
• $n \log n = O(n^2)$.

• $n^2 \neq O(n^{1.5} \log 10n)$.

• $2^n \neq O(n^{1000000})$.

• $\sqrt{2} \log n = O(n^{1/3})$  \(\neq \)

• $n^{\log \log n} = O((\log n)^{\log n})$  \(\neq \)

• $2^n = O(4^{\log n})$  \(\neq \)

• $n! = O(2^n)$  \(\neq \)

• $n! = O(n^n)$  \(\neq \)

• $n2^n = O(2^n \log n)$  \(\neq \)
\begin{itemize}
  
  \item $n \log n = O(n^2)$.
  
  \item $n^2 \neq O(n^{1.5 \log 10 n})$.
  
  \item $2^n \neq O(n^{1000000})$.
  
  \item $\sqrt{2} \log n \neq O(n^{1/3})$.
  
  \item $n^{\log \log n} = O((\log n)^{\log n})$ ?
  
  \item $2^n = O(4^{\log n})$ ?
  
  \item $n! = O(2^n)$ ?
  
  \item $n! = O(n^n)$ ?
  
  \item $n2^n = O(2^n \log n)$ ?
  
\end{itemize}
• \( n \log n = O(n^2) \).
• \( n^2 \neq O(n^{1.5} \log 10n) \).
• \( 2^n \neq O(n^{1000000}) \).
• \( \sqrt{2} \log n \neq O(n^{1/3}) \).
• \( n^{\log \log n} = O((\log n)^{\log n}) \)?
• \( 2^n = O(4^{\log n}) \)?
• \( n! = O(2^n) \)?
• \( n! = O(n^n) \)?
• \( n2^n = O(2^n \log n) \)?
\begin{itemize}
\item $n \log n = O(n^2)$.
\item $n^2 \neq O(n^{1.5} \log 10n)$.
\item $2^n \neq O(n^{1000000})$.
\item $\sqrt{2} \log n \neq O(n^{1/3})$.
\item $n^{\log \log n} = O((\log n)^{\log n})$.
\item $2^n = O(4^{\log n})$ ?
\item $n! = O(2^n)$ ?
\item $n! = O(n^n)$ ?
\item $n2^n = O(2^n \log n)$ ?
\end{itemize}
• $n \log n = O(n^2)$.
• $n^2 \neq O(n^{1.5} \log 10n)$.
• $2^n \neq O(n^{1000000})$.
• $\sqrt{2} \log n \neq O(n^{1/3})$.
• $n^{\log \log n} = O((\log n)^{\log n})$.
• $2^n = O(4^{\log n}) \ ? \ 4^{\log n} = 2^{2\log n} \quad 2^n = 2^{\log n}$.
• $n! = O(2^n)$ ?
• $n! = O(n^n)$ ?
• $n2^n = O(2^n \log n)$ ?
• $n \log n = O(n^2)$.
• $n^2 \neq O(n^{1.5} \log 10n)$.
• $2^n \neq O(n^{1000000})$.
• $\sqrt{2} \log n \neq O(n^{1/3})$.
• $n^{\log \log n} = O((\log n)^{\log n})$.
• $2^n \neq O(4^{\log n})$.
• $n! = O(2^n)$?
• $n! = O(n^n)$?
• $n2^n = O(2^n \log n)$?
• $n \log n = O(n^2)$.
• $n^2 \neq O(n^{1.5} \log 10n)$.
• $2^n \neq O(n^{1000000})$.
• $\sqrt{2} \log n \neq O(n^{1/3})$.
• $n^{\log \log n} = O((\log n)^{\log n})$.
• $2^n \neq O(4^{\log n})$.
• $n! \neq O(2^n)$.  
  \[2.5 \sqrt{n} (n/e)^n \leq n! \leq 2.8 \sqrt{n} (n/e)^n\]
• $n! = O(n^n)$ ?
• $n2^n = O(2^n \log n)$ ?
• \( n \log n = O(n^2). \)
• \( n^2 \neq O(n^{1.5} \log 10n). \)
• \( 2^n \neq O(n^{1000000}). \)
• \( \sqrt{2^n} \log n \neq O(n^{1/3}). \)
• \( n^\log \log n = O(((\log n)^{\log n}). \)
• \( 2^n \neq O(4^{\log n}). \)
• \( n! \neq O(2^n). \)
• \( n! = O(n^n). \)
• \( n2^n = O(2^n \log n)? \)
- $n \log n = O(n^2)$.
- $n^2 \neq O(n^{1.5} \log 10n)$.
- $2^n \neq O(n^{1000000})$.
- $\sqrt{2} \log n \neq O(n^{1/3})$.
- $n^{\log \log n} = O((\log n)^{\log n})$.
- $2^n \neq O(4^{\log n})$.
- $n! \neq O(2^n)$.
- $n! = O(n^n)$.
- $n2^n = O(2^{n \log n})$ ? $n2^n = 2^{\log n+n}$.
• $n \log n = O(n^2)$.
• $n^2 \neq O(n^{1.5} \log 10 n)$.
• $2^n \neq O(n^{1000000})$.
• $\sqrt{2} \log n \neq O(n^{1/3})$.
• $n^{\log \log n} = O((\log n)^{\log n})$.
• $2^n \neq O(4^{\log n})$.
• $n! \neq O(2^n)$.
• $n! = O(n^n)$.
• $n2^n = O(2^n \log n)$.
Big-omega

Definition:

\( f(n) = \Omega(g(n)) \) means

\[ \exists n_0, c > 0 \quad \forall n \in \mathbb{N} \ n \geq n_0, \quad f(n) \geq c \cdot g(n). \]
Big-omega

Definition:

\[ f(n) = \Omega \left( g(n) \right) \text{ means} \]

\[ \exists \ n_0, \ c > 0 \quad \forall n \in \mathbb{N} \ n \geq n_0, \quad f(n) \geq c \cdot g(n). \]

Example 1:

\[ n = \Omega \left( 10 \log n \right) ? \]

\[ c = ?, \ n_0 = ? \text{ such that } \forall n \geq n_0, \ n \geq c \cdot 10 \log n. \]
Big-omega

Definition:
f(n) = Ω (g(n)) means

\[ \exists n_0, c > 0 \quad \forall n \in \mathbb{N} \quad n \geq n_0, \quad f(n) \geq c \cdot g(n). \]

Example 1:
n = Ω (10 \log n)

c = 1, n_0 = 64 \text{ such that } \forall n \geq 64, \quad n \geq 10\log n.

Are there other options for c, n_0?
Big-omega

Definition:
\[ f(n) = \Omega (g(n)) \text{ means} \]
\[ \exists n_0, c > 0 \quad \forall n \in \mathbb{N} \ n \geq n_0, \quad f(n) \geq c \cdot g(n). \]

Example 1:
n = \Omega (10 \log n)
\[ c = 1, \ n_0 = 64 \ \text{such that} \ \forall n \geq 64, \ n \geq 10 \log n. \]
\[ c = \frac{1}{10}, \ n_0 = 2 \ \text{such that} \ \forall n \geq 2, \ n \geq \left(\frac{1}{10}\right)10 \log n. \]
Example 2:

\[ \frac{n^2}{100} = \Omega (n \log n) ? \]

\[ c = ?, n_0 = ? \] such that \( \forall n \geq n_0, \frac{n^2}{100} \geq c \cdot n \log n. \]
Example 2:

\[ \frac{n^2}{100} = \Omega(n \log n). \]

\[ c = 1, \ n_0 = 2^{10} \] such that \( \forall n \geq 2^{10}, \ n^2/100 \geq n \log n. \)

Are there other options for \( c, \ n_0? \)
Example 2:

\[ \frac{n^2}{100} = \Omega (n \log n). \]

\[ c = 1, \ n_0 = 2^{10} \] such that \( \forall n \geq 2^{10}, \ \frac{n^2}{100} \geq n \log n. \]

\[ c = \frac{1}{100}, \ n_0 = 2 \] such that \( \forall n \geq 2, \ \frac{n^2}{100} \geq n \log \frac{n}{100}. \]
Example 2:
\(n^2/100 = \Omega(n \log n)\).

c = 1, \(n_0 = 2^{10}\) such that \(\forall n \geq 2^{10}, n^2/100 \geq n \log n\).

\(c = 1/100, n_0 = 2\) such that \(\forall n \geq 2, n^2/100 \geq n \log n/100\).

Example 3:
\(\sqrt{n} = \Omega(n/100)\) ?

\(c = \? , n_0 = \? \) such that \(\forall n \geq n_0, \sqrt{n} \geq c \cdot n/100\).
Example 2:
\[ \frac{n^2}{100} = \Omega(n \log n). \]
\[ c = 1, \ n_0 = 2^{10} \] such that \( \forall n \geq 2^{10}, \ \frac{n^2}{100} \geq n \log n. \)
\[ c = \frac{1}{100}, \ n_0 = 2 \] such that \( \forall n \geq 2, \ \frac{n^2}{100} \geq n \log n/100. \)

Example 3:
\[ \sqrt{n} \neq \Omega(\frac{n}{100}) \]
\[ \forall c, \ n_0 \ \exists \ n \geq n_0 \text{ such that }, \sqrt{n} < c \cdot \frac{n}{100}. \]
Example 4:

\[2^{n/2} = \Omega(2^n)\]?

c = ?, \ n_0 = ? such that \(\forall \ n \geq n_0, \ 2^{n/2} \geq c \cdot 2^n\).
Example 4:

\[ 2^{n/2} \neq \Omega(2^n) \]

\( \forall c, n_0 \exists n \geq n_0 \text{ such that } 2^{n/2} < c \cdot 2^n. \)
Big-omega, Big-Oh

Note: \( f(n) = \Omega \left(g(n)\right) \iff g(n) = O \left(f(n)\right) \)
\[
 f(n) = O \left(g(n)\right) \iff g(n) = \Omega \left(f(n)\right).
\]

Example:
\[
n = \Omega \left(10 \log n\right).
\]
\[
c = 1, \; n_0 = 6 \text{ such that } \forall n \geq 6, \; n \geq 10 \log n.
\]
\[
10 \log n = O \left(n\right).
\]
\[
c = 1, \; n_0 = 6 \text{ such that } \forall n \geq 6, \; 10 \log n \leq n
\]
Theta

Definition:

\( f(n) = \Theta(g(n)) \) means

\[ \exists n_0, c_1, c_2 > 0 \quad \forall n \in \mathbb{N} \quad n \geq n_0, \]

\( f(n) \leq c_1 \cdot g(n) \) and \( g(n) \leq c_2 \cdot f(n). \)
Theta

Definition:

\[ f(n) = \Theta (g(n)) \] means

\[ \exists n_0, c_1, c_2 > 0 \quad \forall n \in \mathbb{N} \quad n \geq n_0, \]

\[ f(n) \leq c_1 \cdot g(n) \text{ and } g(n) \leq c_2 \cdot f(n).\]

Example:

\[ n = \Theta (n + \log n). \]

\[ c_1 = ?, \quad c_2 = ? \quad n_0 = ? \quad \text{such that } \forall n \geq n_0, \]

\[ n \leq c_1 (n + \log n) \text{ and } n + \log n \leq c_2 n. \]
Theta

Definition:

\( f(n) = \Theta (g(n)) \) means

\[ \exists n_0, c_1, c_2 > 0 \quad \forall n \in \mathbb{N} \quad n \geq n_0, \]

\( f(n) \leq c_1 \cdot g(n) \) and \( g(n) \leq c_2 \cdot f(n). \)

Example:

\( n = \Theta (n + \log n). \)

\( c_1 = 1, \; c_2 = 2 \quad n_0 = 2 \) such that \( \forall n \geq 2, \)
\[ n \leq 1 \cdot (n + \log n) \] and \( n + \log n \leq 2 \cdot n. \)
Theta

Definition:

\[ f(n) = \Theta (g(n)) \text{ means} \]

\[ \exists n_0, c_1, c_2 > 0 \quad \forall n \in \mathbb{N} n \geq n_0, \]

\[ f(n) \leq c_1 \cdot g(n) \text{ and } g(n) \leq c_2 \cdot f(n). \]

Note:

\[ f(n) = \Theta (g(n)) \iff f(n) = \Omega (g(n)) \text{ and } f(n) = O(g(n)) \]

\[ f(n) = \Theta (g(n)) \iff g(n) = \Theta (f(n)). \]
Mixing things up

We also write things like \( n + O(\log n) = O(n) \)

This means \( \forall c \exists c', n_0 : \forall n > n_0 \ n + c \log n < c' n \)

Similarly, \( n^3 \log(n) = n^{O(1)} \)

\[
1/ \Omega(\log n) = O(1/\log n)
\]

\( 2^n + n^{O(1)} = \Theta(2^n) \)

...
Algorithms
Sorting problem:

- Input:
  A sequence (or array) of n numbers \((a[1], a[2], \ldots, a[n])\).

- Desired output:
  A sequence \((b[1], b[2], \ldots, b[n])\) of sorted numbers (in increasing order).

Example:
Input = \((5, 17, -9, 76, 87, -57, 0)\).
Output = ?
Sorting problem:

- Input:
  A sequence (or array) of n numbers \((a[1], a[2], \ldots, a[n])\).

- Desired output:
  A sequence \((b[1], b[2], \ldots, b[n])\) of sorted numbers (in increasing order).

Example:

Input = \((5, 17, -9, 76, 87, -57, 0)\).
Output = \((-57, -9, 0, 5, 17, 76, 87)\).
Sorting problem:

• Input:
  A sequence (or array) of n numbers \((a[1], a[2], \ldots, a[n])\).

• Desired output:
  A sequence \((b[1], b[2], \ldots, b[n])\) of sorted numbers (in increasing order).

Who cares about sorting?

• Sorting is a very basic operation that shows up in countless other algorithms
• Often when you look at data you want it sorted
• There is one more reason why sorting is central
**3SAT problem:** Given a 3CNF formula such as
\[ \varphi := (x \lor y \lor z) \land (\neg x \lor \neg y \lor z) \land (x \lor y \lor \neg z) \]
can we set variables True/False to make \( \varphi \) True? Such \( \varphi \) is called *satisfiable*.

**Theorem [3SAT is NP-complete]**
Let \( M : \{0,1\}^n \rightarrow \{0,1\} \) be an algorithm running in time \( T \)
Given \( x \in \{0,1\}^n \) we can *efficiently* compute 3CNF \( \varphi : \)
\[ M(x) = 1 \iff \varphi \text{ satisfiable} \]

**How efficient?**
Theorem [3SAT is NP-complete]

Let $M : \{0,1\}^n \rightarrow \{0,1\}$ be an algorithm running in time $T$

Given $x \in \{0,1\}^n$ we can efficiently compute 3CNF $\varphi :$

$$M(x) = 1 \iff \varphi \text{ satisfiable}$$

Standard proof: $\varphi$ has $\Theta(T^2)$ variables (and size), $x_{i,j}$

$$
\begin{array}{cccc}
  x_1, 1 & x_1, 2 & \cdots & x_1, T \\
  \vdots \\
  \end{array}
$$

row $i = \text{memory, state at time } i=1..T$

$$
\begin{array}{cccc}
  x_i, 1 & x_i, 2 & \cdots & x_i, T \\
  \vdots \\
  \end{array}
$$

$\varphi$ ensures that memory and state evolve according to $M$
Theorem [3SAT is NP-complete]

Let $M : \{0,1\}^n \rightarrow \{0,1\}$ be an algorithm running in time $T$

Given $x \in \{0,1\}^n$ we can efficiently compute 3CNF $\phi$:

$M(x) = 1 \iff \phi$ satisfiable

Better proof: $\phi$ has $O(T \log^{O(1)} T)$ variables (and size),

$C_i := x_i, 1, x_i, 2, \ldots, x_i, \log T$ = state and what algorithm
reads, writes at time $i = 1.. T$

Note only 1 memory location is represented per time step.

How do you check $C_i$ correct? What does $\phi$ do?
Theorem [3SAT is NP-complete]

Let \( M : \{0,1\}^n \rightarrow \{0,1\} \) be an algorithm running in time \( T \).

Given \( x \in \{0,1\}^n \) we can efficiently compute \( 3\text{CNF } \varphi : M(x) = 1 \iff \varphi \text{ satisfiable} \)

Better proof: \( \varphi \) has \( O(T \log^{O(1)} T) \) variables (and size),

\[
C_i := x_{i,1} x_{i,2} \ldots x_{i,\log T} = \text{state and what algorithm reads, writes at time } i = 1.. T
\]

\( \varphi : \text{Check } C_{i+1} \text{ follows from } C_i \text{ assuming read correct} \)

Compute \( C'_i := C_i \text{ sorted on memory location accessed} \)

Check \( C'_{i+1} \text{ follows from } C'_i \text{ assuming state correct} \)
Theorem [3SAT is NP-complete]

Let $M: \{0,1\}^n \rightarrow \{0,1\}$ be an algorithm running in time $T$.

Given $x \in \{0,1\}^n$, we can efficiently compute 3CNF $\varphi$:

$$M(x) = 1 \iff \varphi \text{ satisfiable}$$

Better proof: $\varphi$ has $O(T \log^{O(1)} T)$ variables (and size), $C_i := x_1^i, x_2^i, \ldots, x_{\log T}^i$ state and what algorithm reads, writes at time $i = 1..T$

$\varphi$: Check $C_{i+1}$ follows from $C_i$ assuming read correct

Let $C'_i$ be $C_i$ sorted on memory location accessed

Check $C'_{i+1}$ follows from $C'_i$ assuming state

THAT'S WHY SORTING MATTERS!
Bubble sort:

Input (a[1], a[2], …, a[n]).

for (i=n; i > 1; i - -)
    for (j=1; j < i; j++)
        if (a[j] > a[j+1])
            swap a[j] and a[j+1];
Bubble sort:

Input \((a[1], a[2], \ldots, a[n])\).

\[
\text{for } (i=n; i > 1; i - -) \\
\text{for } (j=1; j < i; j++) \\
\quad \text{if } (a[j] > a[j+1]) \\
\quad \quad \text{swap } a[j] \text{ and } a[j+1];
\]
Bubble sort:

Input \((a[1], a[2], \ldots, a[n])\).

\[
\text{for (i=n; i > 1; i - -)} \\
\quad \text{for (j=1; j < i; j++)} \\
\quad \quad \text{if (a[j] > a[j+1])} \\
\quad \quad \quad \text{swap a[j] and a[j+1];}
\]

Claim: Bubble sort sorts correctly

Proof: Fix \(i\); let \(a'[1], \ldots, a'[n]\) be array at start of inner loop. Note at the end of the loop: \(a'[i] = \max_{k \leq i} a'[k]\) and the positions \(k > i\) are not touched. Since the outer loop is from \(n\) down to 1, the array is sorted. ■
Bubble sort

DEMO
**Analysis of running time**

\[ T(n) = \text{number of comparisons} \]

\[ i = n-1 \Rightarrow n -1 \text{ comparisons.} \]

\[ i = n-2 \Rightarrow n -2 \text{ comparisons.} \]

\[ \ldots \]

\[ i = 1 \Rightarrow 1 \text{ comparison.} \]

\[ T(n) = (n-1) + (n-2) + \ldots + 1 < n^2 \]

Is this tight? Is also \( T(n) = \Omega(n^2) \)?

---

**Bubble sort:**

Input \((a[1], a[2], \ldots, a[n])\).

for \((i=n; i > 1; i--)\)

\[ \text{for } (j=1; j < i; j++) \]

\[ \text{if } (a[j] > a[j+1]) \]

\[ \text{swap } a[j] \text{ and } a[j+1]; \]
Analysis of running time

T(n) = number of comparisons

i = n-1 ⇒ n -1 comparisons.
i = n-2 ⇒ n -2 comparisons.
...
i = 1 ⇒ 1 comparison.

T(n) = (n-1) + (n-2) + ... + 1 = n(n-1)/2 = Θ(n²)

Bubble sort:
Input (a[1], a[2], ..., a[n]).
for (i=n; i > 1; i--)
    for (j=1; j < i; j++)
        if (a[j] > a[j+1])
            swap a[j] and a[j+1];
Space (also known as Memory)

We need to keep track of i, j

We need an extra element to swap values of input array a.

Space = O(1)

Bubble sort:
Input (a[1], a[2], …, a[n]).
for (i=n; i > 1; i--)
    for (j=1; j < i; j++)
        if (a[j] > a[j+1])
            swap a[j] and a[j+1];
Bubble sort takes quadratic time

Can we sort faster?

We now see two methods that can sort in linear time, assuming that the numbers to sort are small.
Counting sort:

Assumption: all elements of the input array are integers in the range 0 to k.

The basic idea is to determine, for each $x$, the number of elements in the input array that are smaller than $x$. This way we can put element $x$ directly into its position.
Counting sort:

Countingsort (A[1..n])
for (i=0; k ; i++) C[i] = 0; //Initializes C to 0

for (j=1; n ; j++) C[A[j]] = C[A[j]]+1;
    // C[i] = number of elements equal to i.

for (i=1; k ; i++) C[i] = C[i]+C[i-1] ;
    // C[i] = number of elements ≤ i.

for (j=n; 1 ; j - -) {
    C[A[j]] = C[A[j]]-1; // Decrease for equal elements
}
Analysis of running time

$T(n) = \text{number of operations}$

$= O(k) + O(n) + O(k) + O(n)$

$= \Theta(n + k)$.

If $k = O(n)$ then $T(n) = \Theta(n)$

**Countingsort** $(A[1..n])$

```plaintext
for (i = 0; i < k; i++)
  C[i] = 0;

for (j = 1; j < n; j++)
  C[A[j]] = C[A[j]] + 1;

for (i = 1; i < k; i++)
  C[i] = C[i] + C[i-1];

for (j = n; j > 1; j--)
  B[C[A[j]]] = A[j];
  C[A[j]] = C[A[j]] - 1;
```
Space
O(k) for C
Recall numbers in 0..k.
O(n) for B, where output is
Total space: O(n + k)
If k = O(n) then Θ(n)

Countingsort (A[1..n])
for (i =0; i<k ; i++)
    C[i] = 0;
for (j =1; j<n ; j++)
    C[A[j]] =C[A[j]] +1;
for (i =1; i<k ; i++)
    C[i] = C[i] +C[i-1] ;
for (j =n; j>1 ; j--) {
}
Radix sort

Assumption: all elements of the input array are $d$-digit integers.

Idea: first sort the input array according to least significant digit of elements, then according to the next digit, ... and finally according to the most significant digit.

It is essential to use a digit sorting algorithm that is stable: numbers with the same value appear in the output array in the same order as in the input array.

Fact: Counting sort is stable.
Radix sort

Radixsort(A[1..n]) {
    for (i = 0; i < d; i++)
        use a stable sorting algorithm to sort array A on digit i
}

We use Counting sort.
Radix sort

Radix sort DEMO.

Note: first sort by least significant bit, i.e., puts even number first, and then odd (odd numbers are placed in temporary array)
Analysis of running time

\[ T(n) = \text{number of operations} \]

\[ T(n) = d \cdot (\text{running time of Counting sort on } n \text{ elements}) \]

\[ = \Theta(d.n) \]

Example: To sort numbers in range 0..n^{10}

\[ T(n) = ? \]

(hint: think numbers in base n)
Analysis of running time

T(n) = number of operations

T(n) = d.(running time of Counting sort on n elements)
   = $\Theta(d.n)$

Example: To sort numbers in range 0.. $n^{10}$

T(n) = $\Theta(10 \ n) = \Theta(n)$

While counting sort would take $T(n) = ?$
Analysis of running time

T(n) = number of operations

T(n) = d.(running time of Counting sort on n elements)

= \Theta(d.n)

Example: To sort numbers in range 0..n^{10}

T(n) = \Theta(10 \, n) = \Theta(n)

While counting sort would take T(n) = \Theta(n^{10})

Radixsort(A[1..n])

for (i = 0; i < d; i++)
    use Counting sort to sort array A on digit i
Space

We need as much space as we did for Counting sort on each digit.

Radixsort(A[1..n])
{
    for (i = 0; i < d; i++)
        use Counting sort to sort array A on digit i
}
Can we sort faster than $n^2$ without extra assumptions?

Next we show how to sort in time $O(n \log (n))$ arbitrary numbers.

We introduce a new general paradigm.