Algorithms Slides

Emanuele Viola

2009 – present

Released under Creative Commons License
“Attribution-Noncommercial-No Derivative Works 3.0 United States”
http://creativecommons.org/licenses/by-nc-nd/3.0/us/

Also, let me know if you use them.
The slides are under construction.
The latest version is at http://www.ccs.neu.edu/home/viola/
Success stories of algorithms:

Shortest path (Google maps)

Pattern matching (Text editors, genome)

Fast-fourier transform (Audio/video processing)

http://cstheory.stackexchange.com/questions/19759/core-algorithms-deployed
This class:

- General techniques:
  - Divide-and-conquer,
  - dynamic programming,
  - data structures
  - amortized analysis

- Various topics:
  - Sorting
  - Matrixes
  - Graphs
  - Polynomials

. HARDNESS:
What is an algorithm?

- Informally, an algorithm for a function $f : A \rightarrow B$ (the problem) is a simple, step-by-step, procedure that computes $f(x)$ on every input $x$. 
What operations are simple?

- If, for, while, etc.  \textit{Control flow}

- Direct addressing: A[n], the n-entry of array A

- \underline{Basic arithmetic and logic on variables}
  - $x \times y$, $x + y$, $x \text{AND} y$, etc.
  - Simple in practice only if the variables are “small”. For example, 64 bits on current PC
  - Sometimes we get cleaner analysis if we consider them simple regardless of size of variables.
Measuring performance

- We bound the **running time**, or the **memory (space)** used.

- These are measured **as a function of the input length**.

- Makes sense: need to at least read the input!

- The input length is usually denoted $n$

- We are interested in which **functions of n** grow faster
Asymptotic analysis

• The exact time depends on the actual machine

• We ignore constant factors, to have more robust theory that applies to most computer

• Example:
  on my computer it takes $67n + 15$ operations,
  on yours $58n - 15$, but that's about the same

• We now give definitions that make this precise
Big-Oh

Definition:

\[ f(n) = O(g(n)) \text{ if there are } (\exists) \text{ constants } c, n_0 \text{ such that } f(n) \leq c \cdot g(n), \text{ for every } (\forall) n \geq n_0. \]

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

Definition:
f(n) = O(g(n)) if there are (\exists) constants c, n_0 such that
f(n) ≤ c \cdot g(n), for every (\forall) n ≥ n_0.

Example 1:
5n + 2n^2 + \log(n) = O(n^2) ?
Big-Oh

Definition:

\( f(n) = O(g(n)) \) if there are (\( \exists \)) constants \( c, n_0 \) such that

\( f(n) \leq c \cdot g(n) \), for every (\( \forall \)) \( n \geq n_0 \).

Example 1:

\( 5n + 2n^2 + \log(n) = O(n^2) \) True

Pick \( c = ? \)
Big-Oh

Definition:
\( f(n) = O(g(n)) \) if there are (\( \exists \)) constants \( c, n_0 \) such that
\[ f(n) \leq c \cdot g(n), \text{ for every (\( \forall \)) } n \geq n_0. \]

Example 1:
\[ 5n + 2n^2 + \log(n) = O(n^2) \] True

Pick \( c = 3 \). For large enough \( n \), \[ 5n + \log(n) \leq n^2. \]
Any \( c > 2 \) would work.
Example 2:

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

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

Pick c = ?
Example 2:

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

Pick $c = 1$.

Any $c > 0$ would work, for large enough $n$. 
Example 3:

\[ n^2 \log n = O(n^2) \]?
Example 3:

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

\[ \forall c, \exists n_0 \geq n_0 \text{ such that } n^2 \log n > c n^2. \]

\[ n > 2^c \Rightarrow n^2 \log n > n^2 c \]
Example 4:

\[ 2^n = O(2^{n/2}) \ ? \]
Example 4:

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

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

Pick any $n > 2 \log c$

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

$2.5 \sqrt{n} \ (n/e)^n \leq n! \leq 2.8 \sqrt{n} \ (n/e)^n$

• $n! = O(n^n)$ ?

• $n2^n = O(2^n \log n)$ ?
• \( n \log n = O(n^2) \).
• \( n^2 \neq O(n^{1.5} \log 10n) \).
• \( 2^n \neq O(n^{1000000}) \).
• \( (\sqrt{2})^{\log n} \neq O(n^{1/3}) \).
• \( n^{\log \log n} = O((\log n)^{\log n}) \).
• \( 2^n \neq O(4^{\log n}) \).
• \( n! \neq O(2^n) \).
• \( n! = O(n^n) \).
• \( n2^n = O(2^n \log n) \) ?
• $n \log n = O(n^2)$.
• $n^2 \neq O(n^{1.5} \log 10 n)$.
• $2^n \neq O(n^{1000000})$.
• $(\sqrt{2})^{\log n} \neq O(n^{1/3})$.
• $n^{\log \log n} = O((\log n)^{\log n})$.
• $2^n \neq O(4^{\log n})$.
• $n! \neq O(2^n)$.
• $n! = O(n^n)$.
• $n^2 \neq O(2^n) log n$ ? $n^{2n} = 2^{\log n + n}$. 
\[ n \log n = O(n^2). \]
\[ n^2 \neq O(n^{1.5} \log 10n). \]
\[ 2^n \neq O(n^{1000000}). \]
\[ (\sqrt{2})^{\log n} \neq O(n^{1/3}). \]
\[ n^{\log \log n} = O((\log n)^{\log n}). \]
\[ 2^n \neq O(4^{\log n}). \]
\[ n! \neq O(2^n). \]
\[ n! = O(n^n). \]
\[ n2^n = O(2^n \log n). \]
Big-omega

Definition:
\[ f(n) = \Omega (g(n)) \] means
\[ \exists \ c, \ n_0 > 0 \ \forall \ n \geq n_0, \ f(n) \geq c \cdot g(n). \]

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

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

Example 1:
\[ 0.01n = \Omega(\log n)? \]
**Big-omega**

**Definition:**

\[ f(n) = \Omega(g(n)) \] means  
\[ \exists c, \ n_0 > 0 \ \forall n \geq n_0, \ f(n) \geq c \cdot g(n). \]

**Example 1:**

0.01 n = \Omega(\log n) True

Pick c = 1. Any c > 0 would work
Example 2: 

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

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

\[ c = \frac{1}{100} \] Again, any \( c \) would work.
Example 2:

\( n^2/100 = \Omega (n \log n) \).

c = 1/100 Again, any c would work.

Example 3:

\( \sqrt{n} = \Omega(n/100) \) ?
Example 2:

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

\[ c = \frac{1}{100} \text{ Again, any c would work.} \]

Example 3:

\[ \sqrt{n} \neq \Omega(n/100) \]

\[ \forall c, \exists n_0 \geq n_0 \text{ such that, } \sqrt{n} < c \cdot n/100. \]
Example 4:

$2^{n/2} = \Omega(2^n)$ ?
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: \( f(n) = \Omega(g(n)) \iff g(n) = O(f(n)) \)
\( f(n) = O(g(n)) \iff g(n) = \Omega(f(n)) \).

Example:
\( 10 \log n = O(n) \), and \( n = \Omega(10 \log n) \).

\( 5n = O(n) \), and \( n = \Omega(5n) \)
**Theta**

**Definition:**

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

\[ \exists n_0, c_1, c_2 > 0 \quad \forall n \geq n_0, \]

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

**Meaning:** \( f \) grows like \( g \), up to constant factors
Theta

Definition:
\( f(n) = \Theta (g(n)) \) means
\[ \exists n_0, c_1, c_2 > 0 \quad \forall n \geq n_0, \]
\[ f(n) \leq c_1 \cdot g(n) \text{ and } g(n) \leq c_2 \cdot f(n). \]

Example:
\( n = \Theta (n + \log n) \) ?
**Theta**

**Definition:**

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

\[ \exists \ n_0, \ c_1, \ c_2 > 0 \ \ \forall \ n \geq n_0,\]

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

**Example:**

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

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

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

Definition:
\[ f(n) = \Theta(g(n)) \] means
\[ \exists \ n_0, c_1, c_2 > 0 \quad \forall \ n \geq n_0, \]
f(n) ≤ \(c_1 \cdot g(n)\) and \(g(n) \leq c_2 \cdot f(n)\).

Example:
n = \Theta(n + \log n) True
\[ c_1 = 1, \ c_2 = 2 \ n_0 = 2 \] such that \(\forall n \geq 2,\)
\[ n \leq 1 \ (n + \log n) \text{ and } n + \log n \leq 2 \ n. \]
**Definition:**

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

\[ \exists \ n_0, \ c_1, \ c_2 > 0 \ \ \forall \ n \geq n_0, \]

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

**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

- $n + \mathcal{O}(\log n) = \mathcal{O}(n)$
  
  Means $\forall c \exists c', n_0 : \forall n > n_0 \quad n + c \log n < c' n$

- $n^3 \log (n) = n^{\mathcal{O}(1)}$
  
  Means $\exists c, n_0 : \forall n > n_0 \quad n^3 \log (n) \leq n^c$
  
  $c = 4$

- $2^n + n^{\mathcal{O}(1)} = \Theta(2^n)$
  
  Means $\forall c \exists c_1, c_2, n_0 : \forall n > n_0$
  
  $c_2 2^n \leq 2^n + n^c \leq c_1 2^n$
Sorting
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:

Who cares about sorting?

- Sorting is a basic operation that shows up in countless other algorithms
- Often when you look at data you want it sorted
- It is also used in the theory of NP-hardness!
Bubblesort:
Input \((a[1], a[2], \ldots, a[n])\).
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];
Bubblesort:
Input \((a[1], a[2], \ldots, a[n])\).
for \((i=n; i > 1; i - -)\)
    for \((j=1; j < i; j++)\)
        if \((a[j] > a[j+1])\)
            swap \(a[j]\) and \(a[j+1]\);

Claim: Bubblesort sorts correctly
Bubblesort:
Input \((a[1], a[2], \ldots, a[n])\).

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

Claim: Bubblesort 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: \(a'[i] = ?\)
Bubblesort:
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 \(a[j+1]\);

Claim: Bubblesort sorts correctly
Proof: Fix \(i\). Let \(a'[1], ..., a'[n]\) 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
Bubblesort:
Input \((a[1], a[2], \ldots, a[n])\).
for \((i=n; i > 1; i - -)\)
    for \((j=1; j < i; j++)\)
        if \((a[j] > a[j+1])\)
            swap \(a[j]\) and \(a[j+1]\);

Claim: Bubblesort 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. \(\square\)
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 \( T(n) = \Omega(n^2) \)?
Analysis of running time

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

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

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

$\ldots$

$i = 1 \Rightarrow 1 \text{ comparison.}$

$T(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} = \Theta(n^2)$
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, under some assumptions
Countingsort:

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

- Idea: determine, for each $A[i]$, the number of elements in the input array that are smaller than $A[i]$.

- This way we can put element $A[i]$ directly into its position.
// Sorts A[1..n] into array B
Countingsort (A[1..n]) {

// Initializes C to 0
  for (i=0; k ; i++)  C[i] = 0;

// Set C[i] = number of elements = i.

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

  for (i=n; 1 ; i - -) {
  }
}
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])
for (i =0; i<k ; i++)
    C[i] = 0;
for (i =1; i<n ; i++)
    C[A[i]] =C[A[i]] +1;
for (i =1; i<k ; i++)
    C[i] = C[i] +C[i-1] ;
for (i =n; i>1 ; i--) {
}
```
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 $\Theta(n)$

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

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

- Idea: first sort by least significant digit, 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: elements with the same digit appear in the output array in the same order as in the input array.

- Fact: Counting sort is stable.
Radixsort(A[1..n]) {
    for i that goes from least significant digit to most {
        use counting sort algorithm to sort array A on digit i
    }
}

Example:
Sort in ascending order (3,2,1,0) (two binary digits).

```
\text{INPUT} \quad 11 \quad 10 \quad 00 \quad 00 = 0
\quad 10 \quad 00 \quad 11 \quad 01 = 1
\quad 01 \quad 01 \quad 11 \quad 10 = 2
\quad 00 \quad 01 \quad \uparrow \quad \uparrow \quad 3
```
Radixsort(A[1..n]) {
    for i that goes from least significant digit to most {
        use counting sort algorithm to sort array A on digit i
    }
}
Analysis of running time

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

- \( T(n) = d \cdot (\text{running time of Counting sort on } n \text{ elements}) = \Theta(d \cdot (n+k)) \)

Example: To sort numbers in range 0.. \( n^{10} \)
\[ T(n) = ? \]
(hint: think numbers in base \( n \))
Analysis of running time

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

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

Example: To sort numbers in range 0.. n^{10}
\[ T(n) = \Theta(10 \cdot n) = \Theta(n) \]

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

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

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

Example: To sort numbers in range 0..n^{10}
\[ T(n) = \Theta(10 \ n) = \Theta(n) \]

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

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

Space = \( O(d \cdot (n+k)) \)

Can you improve this?

```c
Radixsort(A[1..n]) {
    for i from least significant digit to most {
        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 with $O(n \log n)$ comparisons.

We introduce a new general paradigm.
Deleted scenes
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), $x_{i,j}$

$\begin{array}{cccc}
 x_{1,1} & x_{1,2} & \ldots & x_{1,T} \\
 & \ldots & \\
 x_{i,1} & x_{i,2} & \ldots & x_{i,T}
\end{array}$

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

$\varphi$ 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 $\varphi$:

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

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

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

Note only 1 memory location is represented per time step.

How do you check $C_i$ correct? What does $\varphi$ 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 3CNF $\varphi :$

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

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

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

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

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 $\phi$:

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

Better proof: $\varphi$ has $O(T \log^2 T)$ variables (and size),

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

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

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

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