Running Time Evaluation

Quadratic Vs. Linear Time

Lecturer: Georgy Gimel'farb

COMPSCI 220 Algorithms and Data Structures
1 Running time

2 Examples

3 “Big-Oh”, “Big-Omega”, and “Big-Theta” Tools

4 Time complexity
Running Time **$T(n)$**: Estimation Rules

It is proportional to the **most significant term** in $T(n)$:

- $n$ for a linear time, $T(n) = c_0 + c_1 n$; or
- $n^k$ if $T(n) = c_0 + c_1 n + \ldots + c_k n^k$ for a polynomial time.

Once a problem size $n$ becomes large, the most significant term is that which has the largest power of $n$.

- The most significant term increases faster than other terms which reduce in significance.

Constants of proportionality depend on a compiler, language, computer, programming, etc.

- It is useful to ignore the constants when analysing algorithms.
- Reducing constants of proportionality by using faster hardware or minimising time spent on the “inner loop” does not effect an algorithm’s behaviour for a large problem!
Elementary Operations and Data Inputs

Basic elementary computing operations

- Arithmetic operations (+; −; *; /; %)
- Relational operators (==; !=; >; <; ≥; ≤)
- Boolean operations (AND; OR; XOR; NOT)
- Branch operations
- Return

Input size for problem domains (meaning of $n$)

- **Sorting**: $n$ items
- **Graph / path**: $n$ vertices / edges
- **Image processing**: $n$ pixels (2D images) or voxels (3D images)
- **Text processing**: $n$ characters, i.e. the string length $n$
Estimating Running Time

**Simplifying assumptions**: all elementary statements / expressions take the same amount of time to execute, e.g. simple arithmetic assignments, return, etc.

- A single loop increases in time **linearly** as \( \lambda \cdot T_{\text{body of a loop}} \) where \( \lambda \) is number of times the loop is executed.
- Nested loops result in **polynomial** running time \( T(n) = cn^k \) if the number of elementary operations in the innermost loop is constant (\( k \) is the highest level of nesting and \( c \) is some constant).
- The first three values of \( k \) have special names:
  - **linear time** for \( k = 1 \) (a single loop);
  - **quadratic time** for \( k = 2 \) (two nested loops), and
  - **cubic time** for \( k = 3 \) (three nested loops).
Estimating Running Time

Conditional / switch statements like

\[
\text{if \{condition\} then \{const time } T_1 \text{\} else \{const time } T_2 \text{\} are more complicated.}
\]

- One has to account for branching frequencies \( f_{\text{condition=true}} \) and \( f_{\text{condition=false}} = 1 - f_{\text{condition=true}} \):

\[
T = f_{\text{true}}T_1 + (1 - f_{\text{true}})T_2 \leq \max\{T_1, T_2\}
\]

Function calls:

\[
T_{\text{function}} = \sum T_{\text{statements in function}}
\]

Function composition:

\[
T(f(g(n))) = T(g(n)) + T(f(n))
\]
Estimating Running Time

**Function calls** in more detail: \( T = \sum_i T_{\text{statement } i} \)

\[
\ldots \; \text{x.myMethod( 5, \ldots \ );}
\]

\[
\ldots \\
\text{public void myMethod( int a, \ldots \ ) } \{ \\
\text{statements 1, 2, \ldots, } M \\
\}
\]

**Function composition** in more detail: \( T(f(g(n))) \):

- Computation of \( x = g(n) \rightarrow T(g(n)) \)
- Computation of \( y = f(x) \rightarrow T(f(n)) \)
- \( T(f(g(n))) = T(g(n)) + T(f(n)) \)
Example 1.5: Textbook, p.19

Logarithmic time for a simple loop due to an exponential change

\[ i = 1, k, k^2, k^3, \ldots, k^m \]

of the control variable in the range \( 1 \leq i \leq n \):

```plaintext
for \( i \leftarrow 1 \) step \( i \leftarrow i \ast k \) until \( n \) do
    ...constant number of elementary operations
end for
```

\( m \) iterations such that \( k^{m-1} < n \leq k^m \) \( \rightarrow \) \( T(n) = c \lceil \log_k n \rceil \)

- The ceil \( \lceil z \rceil \) of the real number \( z \) is the least integer not less than \( z \).
- Additional conditions for executing inner loops only for special values of the outer variables also decrease running time.
Example 1.6: Textbook, p.19

**Linearithmic** \( n \log n \) running time of the conditional nested loops:

\[
m \leftarrow 2 \\
\text{for } j \leftarrow 1 \text{ to } n \text{ do} \\
\quad \text{if } j == m \text{ then} \\
\quad\quad m \leftarrow 2 \times m \\
\quad\quad \text{for } i \leftarrow 1 \text{ to } n \text{ do} \\
\quad\quad\quad \ldots \text{constant number of elementary operations} \\
\quad\quad \text{end for} \\
\quad \text{end if} \\
\text{end for}
\]

The inner loop is executed \( k \) times for \( j = m = 2, 4, \ldots, 2^k \)

- \( 2^k \leq n < 2^{k+1} \) implies that \( k \leq \log_2 n < k + 1 \)
- In total, \( T(n) \) is proportional to \( kn \), that is, \( T(n) = n[\log_2 n] \).
- The floor \( \lfloor z \rfloor \) is the greatest integer not greater than \( z \).
**Example 1.6: Textbook, p.19**

**Linearithmic $n \log n$ running time of the conditional nested loops:**

\[
\begin{align*}
m & \leftarrow 2 \\
\text{for } j & \leftarrow 1 \text{ to } n \text{ do} \\
\quad & \text{if } j == m \text{ then} \\
\quad & \quad m \leftarrow 2 \times m \\
\quad & \quad \text{for } i \leftarrow 1 \text{ to } n \text{ do} \\
\quad & \quad \quad \ldots \text{constant number of elementary operations} \\
\quad & \quad \text{end for} \\
\quad & \text{end if} \\
\text{end for}
\end{align*}
\]

The inner loop is executed $k$ times for $j = m = 2, 4, \ldots, 2^k$

- $2^k \leq n < 2^{k+1}$ implies that $k \leq \log_2 n < k + 1$
- In total, $T(n)$ is proportional to $kn$, that is, $T(n) = n \lfloor \log_2 n \rfloor$.
- The floor $\lfloor z \rfloor$ is the greatest integer not greater than $z$. 

Exercise 1.2.1: Textbook

Is the running time quadratic or linear for the nested loops below?

\[
m \leftarrow 1
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \text{ do}
\]

\[
\text{if } j == m \text{ then}
\]

\[
m \leftarrow (n - 1) \times m
\]

\[
\text{for } i \leftarrow 1 \text{ to } n \text{ do}
\]

\[
\ldots \text{constant number of operations}
\]

\[
\text{end for}
\]

\[
\text{end if}
\]

\[
} \text{ end for}
\]

The inner loop is executed only twice, for \( j = 1 \) and \( j = n - 1 \); in total: \( T(n) = 2n \rightarrow \textit{linear running time} \).
Exercise 1.2.1: Textbook

Is the running time quadratic or linear for the nested loops below?

\[
m \leftarrow 1
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \text{ do}
\]
\[
\text{if } j == m \text{ then}
\]
\[
m \leftarrow (n - 1) \times m
\]

\[
\text{for } i \leftarrow 1 \text{ to } n \text{ do}
\]
\[
\ldots \text{constant number of operations}
\]

\[
\text{end for}
\]

\[
\text{end if}
\]

\[
\text{end for}
\]

The inner loop is executed only twice, for \( j = 1 \) and \( j = n - 1 \); in total: \( T(n) = 2n \rightarrow \text{linear running time} \).
“Big-Oh”, “Big-Omega”, and “Big-Theta” Tools

How does the relative running time change if the input size, \( n \), increases from \( n_1 \) to \( n_2 \), all other things equal?

By a factor of

\[
\frac{T(n_2)}{T(n_1)} = \frac{cf(n_1)}{cf(n_1)} = \frac{f(n_2)}{f(n_1)}
\]

- “Big-Oh”, “Big-Omega”, and “Big-Theta” help to avoid imprecise statements like “roughly proportional to…”
- Can be applied to all non-negative-valued functions, \( f(n) \) and \( g(n) \), defined on non-negative integers, \( n \).
- Running time is such a function, \( T(n) \), of data size, \( n; n > 0 \).

Basic assumption:

Two algorithms have essentially the same complexity if their running times as functions of \( n \) differ only by a constant factor.
Definition of “Big-Oh”, $g(n)$ is $O(f(n))$

Let $f(n)$ and $g(n)$ be non-negative-valued functions, defined on non-negative integers, $n$.

Then $g(n)$ is $O(f(n))$ (read “$g(n)$ is Big Oh of $f(n)$) iff there exists a positive real constant, $c$, and a positive integer, $n_0$, such that $g(n) \leq cf(n)$ for all $n > n_0$.

- The notation “iff” is an abbreviation of “if and only if”.
- Meaning: $g(n)$ is a member of the set $O(f(n))$ of functions that increase at most as fast as $f(n)$, when $n \rightarrow \infty$.
- In other words, $g(n) \in O(f(n))$ if $g(n)$ increases eventually at the same or lesser rate than $f(n)$, to within a constant factor.
- $g(n) \in O(f(n))$ specifies a generalised “asymptotic upper bound”, such that $g(n)$ for large $n$ may approach closer and closer to $cf(n)$.
Definition of “Big-Omega”, $g(n)$ is $\Omega(f(n))$

$g(n)$ is $\Omega(f(n))$ (read “$g(n)$ is Big Omega of $f(n)$) iff there exists a positive real constant, $c$, and a positive integer, $n_0$, such that $g(n) \geq cf(n)$ for all $n > n_0$.

- Meaning: $g(n)$ is a member of the set $\Omega(f(n))$ of functions that increase at least as fast as $f(n)$, when $n \to \infty$.
- In other words, $g(n) \in \Omega(f(n))$ if $g(n)$ increases eventually at the same or larger rate than $f(n)$, to within a constant factor.
- “Big Omega” is complementary to “Big Oh” and generalises the concept of “asymptotic lower bound” ($\geq_{n \to \infty}$) just as “Big Oh” generalises the asymptotic upper bound ($\leq_{n \to \infty}$).
- If $g(n)$ is $O(f(n))$, then $f(n)$ is $\Omega(g(n))$. 
Definition of “Big Theta”, $g(n)$ is $\Theta(f(n))$

$g(n)$ is $\Theta(f(n))$ (read “$g(n)$ is Big Theta of $f(n)$”) iff there exist two positive real constants, $c_1$ and $c_2$, and a positive integer, $n_0$, such that $c_1 f(n) \leq g(n) \leq c_2 f(n)$.

- **Meaning**: $g(n)$ is a member of the set $\Theta(f(n))$ of functions that increase as fast as $f(n)$, when $n \to \infty$.

- In other words, $g(n) \in \Theta(f(n))$ if $g(n)$ increases eventually at the same rate as $f(n)$, to within a constant factor.

- “Big Theta” generalises the concept of “asymptotic tight bound”.

- If $g(n) \in O(f(n))$ and $f(n) \in O(g(n))$, then $f(n) \in \Theta(g(n))$ and $g(n) \in \Theta(f(n))$, i.e. both algorithms are of the same time complexity.
Proving $g(n)$ is $O(f(n))$, or $\Omega(f(n))$, or $\Theta(f(n))$

Proving the ‘Big-X” property means finding constants, $(c, n_0)$ or $(c_1, c_2, n_0)$ specified in Definitions.

- It might be done by a chain of inequalities, starting from $f(n)$.
- Mathematical induction can be used in more intricate cases.

Proving $g(n)$ is not “Big-X” of $f(n)$ finds the required constants do not exist, i.e. lead to a contradiction.

**Example 1: Prove that $g(n) = 5n^2 + 3n$ is not $O(n)$.**

If $g(n) = 5n^2 + 3n \leq c \cdot n$ for $n > n_0$, then for any $n_0$ the factor $c > 5n_0 + 3$, i.e. it cannot be constant. Therefore, $g(n) \notin O(n)$.

**Example 2: Prove that $g(n) = 5n^2 + 3n$ is $\Omega(n)$.**

If $g(n) = 5n^2 + 3n \geq c \cdot n$ for $n > n_0$, then for any $n_0$ there exist the required factor $c < 5n_0 + 3$. Therefore, $g(n) \in \Omega(n)$. 
Time Complexity of Algorithms

In analysing running time, \( T(n) \in O(f(n)) \), functions \( f(n) \) measure approximate time complexity like \( \log n \), \( n \), \( n^2 \) etc.

- Polynomial algorithms: 
  \( T(n) \) is \( O(n^k) \); \( k = \text{const} \).
- Exponential algorithms otherwise.

Intractable problems: if no polynomial algorithm is known for solution.

\[
T(n) = 100 \log_{10} n \\
T(n) \leq n \text{ for all } n > 238 \\
T(n) \leq 0.3n \text{ for all } n > 1000 \\
T(n) \in O(n)
\]
Time Complexity Growth

<table>
<thead>
<tr>
<th>$f(n)$</th>
<th>Approximate number of data items processed per:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 minute</td>
</tr>
<tr>
<td>$n$</td>
<td>10</td>
</tr>
<tr>
<td>$n \log_{10} n$</td>
<td>10</td>
</tr>
<tr>
<td>$n^{1.5}$</td>
<td>10</td>
</tr>
<tr>
<td>$n^2$</td>
<td>10</td>
</tr>
<tr>
<td>$n^3$</td>
<td>10</td>
</tr>
<tr>
<td>$2^n$</td>
<td>10</td>
</tr>
</tbody>
</table>

Beware Exponential Complexity!

- A linear, $O(n)$, algorithm processing 10 items per minute, can process $1.4 \times 10^4$ items per day, $5.3 \times 10^6$ items per year, and $5.3 \times 10^8$ items per century.

- An exponential, $O(2^n)$, algorithm processing 10 items per minute, can process only 20 items per day and only 35 items per century...
Time Complexity Growth

<table>
<thead>
<tr>
<th>$f(n)$</th>
<th>Approximate number of data items processed per:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 minute</td>
</tr>
<tr>
<td>$n$</td>
<td>10</td>
</tr>
<tr>
<td>$n \log_{10} n$</td>
<td>10</td>
</tr>
<tr>
<td>$n^{1.5}$</td>
<td>10</td>
</tr>
<tr>
<td>$n^2$</td>
<td>10</td>
</tr>
<tr>
<td>$n^3$</td>
<td>10</td>
</tr>
<tr>
<td>$2^n$</td>
<td>10</td>
</tr>
</tbody>
</table>

Beware Exponential Complexity!

- A linear, $O(n)$, algorithm processing 10 items per minute, can process $1.4 \times 10^4$ items per day, $5.3 \times 10^6$ items per year, and $5.3 \times 10^8$ items per century.

- An exponential, $O(2^n)$, algorithm processing 10 items per minute, can process only 20 items per day and only 35 items per century...
Big-Oh vs. Actual Running Time

Example 1:

Algorithms $A$ and $B$ with running times $T_A(n) = 20n$ time units and $T_B(n) = 0.1n \log_2 n$ time units, respectively.

- In the “Big-Oh” sense, the linear algorithm $A$ is better than the linearithmic algorithm $B$...

- **But:** on which data volume can $A$ outperform $B$, i.e. for which value $n$ the running time for $A$ is less than for $B$?

  $T_A(n) < T_B(n)$ if $20n < 0.1n \log_2 n$, or $\log_2 n > 200$, that is, when $n > 2^{200} \approx 10^{60}$!

Thus, in all practical cases the algorithm $B$ is better than $A$...
Big-Oh vs. Actual Running Time

Example 1:

Algorithms A and B with running times $T_A(n) = 20n$ time units and $T_B(n) = 0.1n \log_2 n$ time units, respectively.

- In the “Big-Oh” sense, the linear algorithm A is better than the linearithmic algorithm B...

- But: on which data volume can A outperform B, i.e. for which value $n$ the running time for A is less than for B?

  $T_A(n) < T_B(n)$ if $20n < 0.1n \log_2 n$,  
  or $\log_2 n > 200$,  
  that is, when $n > 2^{200} \approx 10^{60}$!

Thus, in all practical cases the algorithm B is better than A...
Big-Oh vs. Actual Running Time

Example 1:

Algorithms $A$ and $B$ with running times $T_A(n) = 20n$ time units and $T_B(n) = 0.1n \log_2 n$ time units, respectively.

- In the “Big-Oh” sense, the linear algorithm $A$ is better than the linearithmic algorithm $B$...

- **But:** on which data volume can $A$ outperform $B$, i.e. for which value $n$ the running time for $A$ is less than for $B$?

  $$T_A(n) < T_B(n) \quad \text{if} \quad 20n < 0.1n \log_2 n,$$

  or  $$\log_2 n > 200, \quad \text{that is, when} \quad n > 2^{200} \approx 10^{60}!$$

Thus, in all practical cases the algorithm $B$ is better than $A$...
Example 1:

Algorithms $A$ and $B$ with running times $T_A(n) = 20n$ time units and $T_B(n) = 0.1n \log_2 n$ time units, respectively.

- In the “Big-Oh” sense, the linear algorithm $A$ is better than the linearithmic algorithm $B$...
- **But:** on which data volume can $A$ outperform $B$, i.e. for which value $n$ the running time for $A$ is less than for $B$?

\[
T_A(n) < T_B(n) \quad \text{if} \quad 20n < 0.1n \log_2 n,
\]

or \quad $\log_2 n > 200$, \quad that is, when $n > 2^{200} \approx 10^{60}$!

Thus, in all practical cases the algorithm $B$ is better than $A$...
Example 1:

Algorithms $A$ and $B$ with running times $T_A(n) = 20n$ time units and $T_B(n) = 0.1n \log_2 n$ time units, respectively.

- In the “Big-Oh” sense, the linear algorithm $A$ is better than the linearithmic algorithm $B$.

- **But:** on which data volume can $A$ outperform $B$, i.e. for which value $n$ the running time for $A$ is less than for $B$?

\[
T_A(n) < T_B(n) \quad \text{if} \quad 20n < 0.1n \log_2 n, \\
\text{or} \quad \log_2 n > 200, \quad \text{that is, when} \quad n > 2^{200} \approx 10^{60}!
\]

Thus, in all practical cases the algorithm $B$ is better than $A$.\ldots
Example 2:

Algorithms $A$ and $B$ with running times $T_A(n) = 20n$ time units and $T_B(n) = 0.1n^2$ time units, respectively.

- In the “Big-Oh” sense, the linear algorithm $A$ is better than the quadratic algorithm $B$.
- **But:** on which data volume can $A$ outperform $B$, i.e. for which value $n$ the running time for $A$ is less than for $B$?

$$T_A(n) < T_B(n) \text{ if } 20n < 0.1n^2, \text{ or } n > 200$$

Thus the algorithm $A$ is better than $B$ in most of practical cases except for $n < 200$ when $B$ becomes faster.
Example 2:

Algorithms $A$ and $B$ with running times $T_A(n) = 20n$ time units and $T_B(n) = 0.1n^2$ time units, respectively.

- In the “Big-Oh” sense, the linear algorithm $A$ is better than the quadratic algorithm $B$.

- But: on which data volume can $A$ outperform $B$, i.e. for which value $n$ the running time for $A$ is less than for $B$?

$$T_A(n) < T_B(n) \text{ if } 20n < 0.1n^2, \text{ or } n > 200$$

Thus the algorithm $A$ is better than $B$ in most of practical cases except for $n < 200$ when $B$ becomes faster...
Example 2:

Algorithms $A$ and $B$ with running times $T_A(n) = 20n$ time units and $T_B(n) = 0.1n^2$ time units, respectively.

- In the “Big-Oh” sense, the linear algorithm $A$ is better than the quadratic algorithm $B$...

- **But:** on which data volume can $A$ outperform $B$, i.e. for which value $n$ the running time for $A$ is less than for $B$?

  $$T_A(n) < T_B(n) \text{ if } 20n < 0.1n^2, \text{ or } n > 200$$

Thus the algorithm $A$ is better than $B$ in most of practical cases except for $n < 200$ when $B$ becomes faster...
Example 2:

Algorithms $A$ and $B$ with running times $T_A(n) = 20n$ time units and $T_B(n) = 0.1n^2$ time units, respectively.

- In the “Big-Oh” sense, the linear algorithm $A$ is better than the quadratic algorithm $B$...
- **But:** on which data volume can $A$ outperform $B$, i.e. for which value $n$ the running time for $A$ is less than for $B$?

$$T_A(n) < T_B(n) \text{ if } 20n < 0.1n^2, \text{ or } n > 200$$

Thus the algorithm $A$ is better than $B$ in most of practical cases except for $n < 200$ when $B$ becomes faster...
Big-Oh vs. Actual Running Time

Example 2:

Algorithms $A$ and $B$ with running times $T_A(n) = 20n$ time units and $T_B(n) = 0.1n^2$ time units, respectively.

- In the “Big-Oh” sense, the linear algorithm $A$ is better than the quadratic algorithm $B$.
- **But:** on which data volume can $A$ outperform $B$, i.e. for which value $n$ the running time for $A$ is less than for $B$?

$$T_A(n) < T_B(n) \text{ if } 20n < 0.1n^2, \text{ or } n > 200$$

Thus the algorithm $A$ is better than $B$ in most of practical cases except for $n < 200$ when $B$ becomes faster.