Algorithms: Analysis of Algorithms

Amotz Bar-Noy

CUNY
A finite set of precise instructions for performing a computation or for solving a problem.

A specific set of instructions for carrying out a procedure or solving a problem, usually with the requirement that the procedure terminates at some point.

A procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation.

A step-by-step procedure for solving a problem or accomplishing some end especially by a computer.

A logical arithmetical or computational procedure that if correctly applied ensures the solution of a problem.

A finite set of unambiguous instructions performed in a prescribed sequence to achieve a goal, especially a mathematical rule or procedure used to compute a desired result.
Algorithm: Definitions

- A word used by programmers when they do not want to explain what they did.
- A word used by those whose program failed to justify what they did.
Algorithm

**Synonym?**

- Method, Procedure, Program, Code, Process, Recipe, Prescription, Routine, Solution, Technique, Mechanism, Scheme, Way, Design, Plan, Strategy, Construction, ...

**Etymology**

- Alteration of Middle English *algorisme*,
- from Medieval Latin *algorismus*,
- from Arabic *al-khuwarizmi*,
- from the name of the 9th-century Persian Mathematician Al-Khowârizmi who was the "*first*" to formalize the rules for the four basic arithmetic operations.
The Ultimate Algorithmic Problem!?  

**Question**

- What is required to **solve** problems and/or design **efficient** algorithms?

**Attributes**

1. Talent?
2. Intuition?
3. Luck?
4. Experience?
5. Hard work?

**Answer**

- Apply some combination of these five attributes!!!
Some Heuristics to Solve Problems

1. Search for a pattern.
2. Draw a figure.
3. Formulate an equivalent problem.
4. Modify the problem.
5. Choose effective notation.
7. Divide into cases.
8. Work backward.
10. Pursue parity.
11. Consider extreme cases.
Three Ancient Algorithms

The Babylonian Multiplication Algorithm
- Introduced around 3700 years ago.

The Euclid’s Greatest Common Divisor Algorithm
- Introduced around 2300 years ago.

The Sieve of Eratosthenes to Find Prime Numbers Algorithm
- Introduced around 2200 years ago.
Although there are some evidences of early multiplication algorithms in Egypt (around 1700-2000 BC) the oldest algorithm is widely accepted to have been found on a set of Babylonian clay tablets that date to around 1600-1800 BC.

Their true significance only came to light in 1972 when computer scientist & mathematician Donald E. Knuth published the first English translations of various Cuneiform mathematical tablets.

The Babylonians had developed a nice way to explain an algorithm by examples as the algorithm itself was being defined.

The tablets also appear to have been an early form of instruction manual.
The Euclid’s Greatest Common Divisor Algorithm

- The Euclidian algorithm is a procedure used to find the greatest common divisors (GCD) of two positive integers.

- It was first described by Euclid in his manuscript the Elements written around 300 BC.

- It is a very efficient computation that is still used today by computers in some form or other.
The Sieve of Eratosthenes Algorithm

- The Sieve of Eratosthenes is an ancient algorithm for finding all prime numbers up to any given limit.

- It is attributed to the Greek mathematician Eratosthenes of Cyrene and was “invented” around 200 BC.

- The algorithm iteratively marks as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2.

- The “less efficient” method sequentially tests each candidate number for divisibility by previously found prime.

A Demo

https://www.youtube.com/watch?v=dhfhu9Q5g8U
### Algorithms — Properties

#### Correctness
- For all valid inputs.

#### Termination
- Does not run forever on some inputs.

#### Complexity – Efficiency
- As a function of the input size.
- Worst-Case and/or Average-Case.

#### Scalability
- “Similar” structure and efficiency for any input size.

#### Limitations
- For the algorithm and for the problem.

#### Optimality
- Optimal or near-optimal or approximately optimal solutions.
Cost and Complexity

**Cost**
- How much resources does the algorithm require?
  - Usually time and space (memory).

**Complexity**
- As a function of the input size.
  - Usually an integer $n > 0$.
  - Usually a monotonic non-decreasing function.

**Terminology**
- Complexity is often called *running-time* because time is the dominating cost.
Worst Case and Average Case Complexity

**Worst case (informal definition)**
- \( T(n) \) is the **worst case complexity** if among all inputs of size \( n \) the worst case complexity is \( T(n) \).

**Average case (informal definition)**
- \( T(n) \) is the **average case complexity** if the average complexity over all length \( n \) inputs is \( T(n) \) where averaging is based on some distribution of the inputs, usually the uniform distribution.

**Bounds (informal definition)**
- A function \( f(n) \) is an **upper bound** if \( T(n) \leq f(n) \) for all \( n \).
- A function \( g(n) \) is a **lower bound** if \( T(n) \geq g(n) \) for all \( n \).
- A function \( h(n) \) is a **tight bound** if \( T(n) \approx h(n) \) for all \( n \).
Performance Evaluation of Algorithms

**Theoretical analysis**
- All possible inputs.
- Independent of hardware/software implementation.
- Based on a high level language.

**Experimental Study**
- Some typical inputs.
- Depends on hardware/software implementation.
- Based on a real program.
Growth of Functions

Objective
- Develop a language to express that Algorithm A is better than or worse than or equivalent to Algorithm B.

Technique
- Define a “≤” relation between functions measuring the growth of functions.

Robustness
- Being independent of the hardware/software environment: Turing machine, classroom model, today computers, and future super-computers.

An important property
- Constants that can be affected by changing the environment should be ignored.
Examples of Function Growth

<table>
<thead>
<tr>
<th>Running Time</th>
<th>1 second</th>
<th>1 minute</th>
<th>1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>1,000,000</td>
<td>60,000,000</td>
<td>3,600,000,000,000</td>
</tr>
<tr>
<td>$n^2$</td>
<td>1,000</td>
<td>7,745</td>
<td>60,000</td>
</tr>
<tr>
<td>$n^4$</td>
<td>31</td>
<td>88</td>
<td>244</td>
</tr>
<tr>
<td>$2^n$</td>
<td>19</td>
<td>25</td>
<td>31</td>
</tr>
</tbody>
</table>

Maximum size of a problem that can be solved in one second, one minute, and one hour, for various running times measured in microseconds.
Examples of Function Growth

<table>
<thead>
<tr>
<th>Running Time</th>
<th>New Maximum Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$256m$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$16m$</td>
</tr>
<tr>
<td>$n^4$</td>
<td>$4m$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$m + 8$</td>
</tr>
</tbody>
</table>

Increase in the maximum size of a problem that can be solved with a certain complexity, by using a computer that is **256 times faster** than the previous one.

Each entry is given as a function of $m$, the previous maximum problem size.
### Examples of Function Growth

<table>
<thead>
<tr>
<th>Running Time</th>
<th>1 second</th>
<th>1 minute</th>
<th>1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n/256$</td>
<td>256,000,000</td>
<td>15,360,000,000</td>
<td>9,216,600,000,000</td>
</tr>
<tr>
<td>$n^2/256 = (n/16)^2$</td>
<td>16,000</td>
<td>123,925</td>
<td>960,000</td>
</tr>
<tr>
<td>$n^4/256 = (n/4)^4$</td>
<td>126</td>
<td>352</td>
<td>979</td>
</tr>
<tr>
<td>$2^n/256 = 2^{n-8}$</td>
<td>27</td>
<td>33</td>
<td>39</td>
</tr>
</tbody>
</table>

- Maximum size of a problem that can be solved by the faster computer in one second, one minute, and one hour, for various running times measured in **microseconds**.
The “$O$, $Ω$, $Θ$, $o$, $ω$” Notation

**Settings**

Let $f$ and $g$ be positive functions on the non-negative integers.

**Big-Oh**

$f(n) = O(g(n))$ if $f(n)$ is asymptotically less than or equal to $g(n)$.

**Big-Omega**

$f(n) = Ω(g(n))$ if $f(n)$ is asymptotically greater than or equal to $g(n)$.

**Big-Theta**

$f(n) = Θ(g(n))$ if $f(n)$ is asymptotically equal to $g(n)$.

**Little-o**

$f(n) = o(g(n))$ if $f(n)$ is asymptotically strictly less than $g(n)$.

**Little-omega**

$f(n) = ω(g(n))$ if $f(n)$ is asymptotically strictly greater than $g(n)$.
Big-Oh, Big-Omega, and Big-Theta

\[ f(n) = O(g(n)) \]
- **There exist** a real constant \( c > 0 \) and an integer constant \( n_0 > 0 \) such that \( f(n) \leq cg(n) \) for every integer \( n \geq n_0 \).

\[ f(n) = \Omega(g(n)) \]
- **There exist** a real constant \( c > 0 \) and an integer constant \( n_0 > 0 \) such that \( f(n) \geq cg(n) \) for every integer \( n \geq n_0 \).

\[ f(n) = \Theta(g(n)) \]
- **There exist** two real constants \( c' \geq c'' > 0 \) and an integer constant \( n_0 > 0 \) such that \( c''g(n) \leq f(n) \leq c'g(n) \) for every integer \( n \geq n_0 \).
Assume $\Psi \in \{O, \Omega, \Theta\}$.

$\Psi(g(n))$ is a set of functions.

$f_1(n) = \Psi(g(n))$ and $f_2(n) = \Psi(g(n))$ do not imply $f_1(n) = f_2(n)$.

$f(n) \in \Psi(g(n))$ is the more accurate notation.

$f(n) = \Psi(g(n))$ is often written as $f = \Psi(g)$. 
### Big-Oh and Big-Omega

<table>
<thead>
<tr>
<th></th>
<th>$f(n) = O(g(n))$</th>
<th>$g(n) = O(f(n))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g(n)$ grows faster</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>$f(n)$ grows faster</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>same growth</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$f(n) = \Omega(g(n))$</th>
<th>$g(n) = \Omega(f(n))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g(n)$ grows faster</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>$f(n)$ grows faster</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>same growth</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
Assume $f(n) = O(g(n))$

- By the definition of $O$, there exist $c > 0$ and $n_0 > 0$ such that $f(n) \leq cg(n)$ for every $n \geq n_0$.
- It follows that $g(n) \geq (1/c)f(n)$ for every $n \geq n_0$.
- Since $1/c > 0$, by the definition of $\Omega$, $g(n) = \Omega(f(n))$.

Assume $g(n) = \Omega(f(n))$

- By the definition of $\Omega$, there exist $c > 0$ and $n_0 > 0$ such that $g(n) \geq cf(n)$ for every $n \geq n_0$.
- It follows that $f(n) \leq (1/c)g(n)$ for every $n \geq n_0$.
- Since $1/c > 0$, by the definition of $O$, $f(n) = O(g(n))$. 
\( f(n) = \Theta(g(n)) \iff (f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))) \)

**Assume** \( f(n) = \Theta(g(n)) \)

- By the definition of \( \Theta \), there exist \( c', c'' > 0 \) and \( n_0 > 0 \) such that \( c''g(n) \leq f(n) \leq c'g(n) \) for every \( n \geq n_0 \).
- By the definition of \( O \), \( f(n) = O(g(n)) \) for \( c = c' \) and \( n_0 \).
- By the definition of \( \Omega \), \( f(n) = \Omega(g(n)) \) for \( c = c'' \) and \( n_0 \).

**Assume** \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \)

- By the definition of \( O \), there exist \( c_1 > 0 \) and \( n_1 > 0 \) such that \( f(n) \leq c_1 g(n) \) for every \( n \geq n_1 \).
- By the definition of \( \Omega \), there exist \( c_2 > 0 \) and \( n_2 > 0 \) such that \( f(n) \geq c_2 g(n) \) for every integer \( n \geq n_2 \).
- Therefore, for \( n_0 \geq \max \{n_1, n_2\} \), it follows that \( c_2 g(n) \leq f(n) \leq c_1 g(n) \) for every \( n \geq n_0 \).
- By the definition of \( \Theta \), \( f(n) = \Theta(g(n)) \) for \( c' = c_1, c'' = c_2 \), and \( n_0 \).
**$O$, $\Omega$, $\Theta$ as Relations**

**$\Theta$ is an equivalence relation**
- **Reflexive:** $f(n) = \Theta(f(n))$.
- **Symmetric:** $(f(n) = \Theta(g(n))) \Leftrightarrow (g(n) = \Theta(f(n)))$.
- **Transitive:** $f(n) = \Theta(g(n)) \land g(n) = \Theta(h(n)) \Rightarrow (f(n) = \Theta(h(n)))$.

**$O$ and $\Omega$, are reflexive relations**
- $f(n) = O(f(n))$.
- $f(n) = \Omega(f(n))$.

**$O$ and $\Omega$, are not symmetric relations**
- $f(n) = O(g(n))$ does not imply that $g(n) = O(f(n))$.
- $f(n) = \Omega(g(n))$ does not imply that $g(n) = \Omega(f(n))$.

**$O$ and $\Omega$, are transitive relations**
- $f(n) = O(g(n)) \land g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$.
- $f(n) = \Omega(g(n)) \land g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$. 
\( n^2 \) vs. \( n \)

**\( n = O(n^2) \) and \( n^2 = \Omega(n) \)**

- Observe that \( n \leq n^2 \) for integer \( n \geq 1 \) (\( n < n^2 \) for integer \( n > 1 \)).
- Therefore, for \( c = 1 \) and \( n_0 = 1 \), the definition of \( O \) implies that \( n = O(n^2) \) and the definition of \( \Omega \) implies that \( n^2 = \Omega(n) \).

**\( n^2 \neq O(n) \) and \( n \neq \Omega(n^2) \)**

- Observe that if \((1/c) < n\) for a constant \( c > 0 \), then by multiplying both sides of the inequality by \( cn \), it follows that \( n < cn^2 \).
- Therefore, \( n < cn^2 \) for every real constant \( c > 0 \) and integer \( n > (1/c) \).
- As a result, there are no real constant \( c > 0 \) and integer \( n_0 \) such that \( n \geq cn^2 \) for every integer \( n \geq n_0 \).
- Consequently, the definitions of \( O \) and \( \Omega \) cannot be applied to get \( n^2 = O(n) \) or \( n = \Omega(n^2) \).
Examples

“Ignore” constants

- $3n = \Theta(n/2)$.
- $1000000n = \Theta(n/100000)$.
- $\log_2(n) = \Theta(\log_{10}(n))$.
- $n \log_2 n / 100000 = \Omega(10000000n)$.
- $10^{100}n = O(n^2)$.

Polynomials

- $a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0 = \Theta(n^d)$
  - for constants $a_0, a_1, \ldots, a_{d-1}$ and a positive constant $a_d$.
- **Example:** $5n^3 + 1000n^2 - 345n + 7 = \Theta(n^3)$
Observations

Eliminating constants

- For any real constant $c$ and $\Psi \in \{O, \Omega, \Theta\}$:
  * $\Psi(cf(n)) = \Psi(f(n))$.
  * $\Psi(f(n)/c) = \Psi(f(n))$.
  * $\Psi(c) = \Psi(1)$.

Addition and multiplication rules

- For $\Psi \in \{O, \Omega, \Theta\}$, if $f_1(n) = \Psi(g_1(n))$ and $f_2(n) = \Psi(g_2(n))$ then
  * $f_1(n) \cdot f_2(n) = \Psi(g_1(n) \cdot g_2(n))$
  * $f_1(n) + f_2(n) = \Psi(\max \{g_1(n) + g_2(n)\})$
Little-oh and Little-omega

\[ f(n) = o(g(n)) \]

- If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \):
  - For any constant \( c > 0 \) there exists an integer constant \( n_0 > 0 \) such that \( f(n) < cg(n) \) for every integer \( n \geq n_0 \).

\[ f(n) = \omega(g(n)) \]

- If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \):
  - For any constant \( c > 0 \) there exists an integer constant \( n_0 > 0 \) such that \( f(n) > cg(n) \) for every integer \( n \geq n_0 \).
Propositions

\( o \) and \( \omega \)

- \( f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n)) \).
- \( f(n) = o(g(n)) \land g(n) = o(h(n)) \Rightarrow f(n) = o(h(n)) \).
- \( f(n) = \omega(g(n)) \land g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n)) \).

\( o \) vs. \( O \)

- \( f(n) = o(g(n)) \Rightarrow f(n) = O(g(n)) \).
- \( f(n) = O(g(n)) \nRightarrow f(n) = o(g(n)) \).

\( \omega \) vs \( \Omega \)

- \( f(n) = \omega(g(n)) \Rightarrow f(n) = \Omega(g(n)) \).
- \( f(n) = \Omega(g(n)) \nRightarrow f(n) = \omega(g(n)) \).
Examples

**Polynomials**
- $n^3 = \omega(n^2)$
- $10^{100} n = o(n^2/10^{100})$
- $1 + n + n^2 + n^3 + \cdots + n^{k-1} = o(n^k)$

**The logarithmic function**
- $\log_2 n = o(n)$
- $n \log_2 n = \omega(n)$
More Examples

The sqrt function
- \( \log_2 n = o(\sqrt{n}) \)
- \( n = \omega(\sqrt{n}) \)

Beyond polynomial function
- \( n^k = o(2^n) \) for any integer \( k \geq 0 \)
- \( 2^n = o(3^n) = o(4^n) = \cdots = o(k^n) \)
- \( n^n = \omega(n!) = \omega(2^n) \)
## Hierarchy of Functions

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>$1$</td>
</tr>
<tr>
<td>Log star</td>
<td>$\log^* n$</td>
</tr>
<tr>
<td>Loglog</td>
<td>$\log \log n$</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>$\log n$</td>
</tr>
<tr>
<td>Poly-logarithmic</td>
<td>$\log^k n$</td>
</tr>
<tr>
<td>Sub-linear</td>
<td>$n^\epsilon$</td>
</tr>
<tr>
<td>Linear</td>
<td>$n$</td>
</tr>
<tr>
<td>Above-linear</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>Quadratic</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Cubic</td>
<td>$n^3$</td>
</tr>
<tr>
<td>Polynomial</td>
<td>$n^k$</td>
</tr>
<tr>
<td>Super-polynomial</td>
<td>$n^{\log n}$</td>
</tr>
<tr>
<td>Exponential</td>
<td>$2^n$</td>
</tr>
<tr>
<td>Factorial</td>
<td>$n!$</td>
</tr>
<tr>
<td>Super-exponential</td>
<td>$n^n$</td>
</tr>
<tr>
<td>Exponential tower</td>
<td>$2^{2 \cdots 2}$</td>
</tr>
</tbody>
</table>

- $\log^* n$ denotes the iterated logarithm, also known as the Ackermann function.

- $\log n$, $\log \log n$, ... denote the iterated logarithms.

- $n^\epsilon$ where $0 < \epsilon < 1$ for sub-linear functions.

- $n^k$ for polynomial functions, where $k$ is a constant integer.

- $2^{2 \cdots 2}$ for exponential towers, representing $n$ powers of 2.
Analysis of Algorithms

Algorithm $A$ has a worst case complexity $T(n)$

- To prove that $T(n) = O(f(n))$, show this for all size $n$ inputs for all $n \geq n_0$ for some integer $n_0$.
- To prove that $T(n) = \Omega(f(n))$, show this for one size $n$ input for infinitely many $n$.
- To prove that $T(n) = \Theta(f(n))$, show that $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

Algorithm $A$ has an average case complexity $T(n)$

- To prove that $T(n) = O(f(n))$ for a given distribution, show this by averaging over all size $n$ inputs for all $n \geq n_0$ for some integer $n_0$.
- To prove that $T(n) = \Omega(f(n))$ for a given distribution, show this by averaging over all size $n$ inputs for infinitely many $n$.
- To prove that $T(n) = \Theta(f(n))$ for a given distribution, show that $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$. 
Recursion

Poem

- Great fleas have little fleas upon their backs to bite ’em, And little fleas have lesser fleas, and so ad infinitum. And the great fleas themselves in turn have greater fleas to go on; While these again have greater still, and greater still, and so on.

Definition

- Recursion occurs when a “thing” is defined in terms of its type.

Focus

- Recursive formulas in mathematics.

Illustrations

- https://storage.googleapis.com/algodailyrandomassets/curriculum/recursion/cover.jpg
Recursive Formulas

Definition
- A recursive formula is defined on the set of integers greater than or equal to some number $m$ (usually 0 or 1).
- The formula computes the $n$th value based on some or all of the previous $n-1$ values.

Goal
- Given initial values and a recursive formula, find an equivalent closed-form expression as a function of $n$ that does not depend on previous values.

Recursion and induction
- Usually proving the correctness of a solution (a closed-form expression) to a recursive formula is done by induction.
The Non-Negative Integers

The recursive formula:

\[ N(n) = \begin{cases} 
0 & \text{for } n = 0 \\
N(n - 1) + 1 & \text{for } n > 0 
\end{cases} \]

The solution

\[ N(n) = n \]

Proof by induction

- Induction base: \( N(0) = 0 \)
- Induction hypothesis: \( N(k) = k \) for all \( 0 \leq k < n \)
- Inductive step for \( n \geq 1 \):

\[
N(n) = N(n - 1) + 1 \\
= (n - 1) + 1 \\
= n
\]
**A Generalization**

**The recursive formula**

\[
T(n) = \begin{cases} 
  b & \text{for } n = 0 \text{ and a real number } b \\
  T(n - 1) + a & \text{for } n > 0 \text{ and a real number } a 
\end{cases}
\]

**The solution**

\[ T(n) = b + an \]

**Proof by induction**

- Induction base: \( T(0) = b = b + a \cdot 0 \)
- Induction hypothesis: \( T(k) = b + ak \) for all \( 0 \leq k < n \)
- Inductive step for \( n \geq 1 \):
  \[
  T(n) = T(n - 1) + a \\
  = b + a(n - 1) + a \\
  = b + an
  \]
Special Cases

The recursive formula

\[ T(n) = \begin{cases} 
  b & \text{for } n = 0 \text{ and a real number } b \\
  T(n - 1) + a & \text{for } n > 0 \text{ and a real number } a 
\end{cases} \]

The solution

\[ T(n) = b + an \]

Examples

- All integers: \( b = 0 \) and \( a = 1 \) \( \implies \) \( T(n) = n \)
- Even integers: \( b = 0 \) and \( a = 2 \) \( \implies \) \( T(n) = 2n \)
- Odd integers: \( b = 1 \) and \( a = 2 \) \( \implies \) \( T(n) = 2n + 1 \)

An infinite arithmetic progression

- \((X(0), X(1), \ldots, X(n), \ldots) = (x, x + d, x + 2d, \ldots, x + (n - 1)d, \ldots)\)
- \( b = x \) and \( a = d \) \( \implies \) \( X(n) = x + dn \)
Powers of integers

The recursive formula

\[ P(n) = \begin{cases} 
1 & \text{for } n = 0 \\
 d \cdot P(n - 1) & \text{for } n \geq 1 
\end{cases} \]

The solution

\[ P(n) = d^n \]

Proof by induction

- Induction base: \( P(0) = 1 = d^0 \)
- Induction hypothesis: \( P(k) = d^k \) for all \( 0 \leq k < n \)
- Inductive step for \( n \geq 1 \): \( P(n) = d \cdot P(n - 1) = d \cdot d^{n-1} = d^n \)
Factorials

The recursive formula

\[ F(n) = \begin{cases} 
1 & \text{for } n = 0 \\
 nF(n - 1) & \text{for } n \geq 1 
\end{cases} \]

The solution

\[ F(n) = n! \]

Proof by induction

- Induction base: \( F(0) = 1 = 0! \)
- Induction hypothesis: \( F(k) = k! \) for all \( 0 \leq k < n \)
- Inductive step for \( n \geq 1 \): \( F(n) = nF(n - 1) = n(n - 1)! = n! \)
The $\log_2$ Function

The recursive formula

$$L(n) = \begin{cases} 
0 & \text{for } n = 1 \\
L(n/2) + 1 & \text{for } n = 2^h \text{ and } h > 0 
\end{cases}$$

The solution

$$L(n) = \log_2(n)$$
$$L(2^h) = \log_2(2^h) = h$$
The $\log_2$ Function

Proof by induction

- Induction base: $L(1) = 0 = \log_2(1)$
- Induction hypothesis: $L(2^k) = k$ for all $0 \leq k < \log_2(n)$
- Inductive step: Assume $n = 2^h$ for $h > 0$ and therefore $n/2 = 2^{h-1}$

\[
\begin{align*}
L(n) & = L(2^h) \\
& = L(2^{h-1}) + 1 \\
& = \log_2(2^{h-1}) + 1 \\
& = (h - 1) + 1 \\
& = h \\
& = \log_2(n)
\end{align*}
\]
The Sum $1 + 2 + \cdots + n$

The recursive formula

$$S(n) = \begin{cases} 
0 & \text{for } n = 0 \\
S(n - 1) + n & \text{for } n \geq 1 
\end{cases}$$

Small values of $n$

- $S(0) = 0$.
- $S(1) = S(0) + 1 = 0 + 1 = 1$
- $S(2) = S(1) + 2 = 1 + 2 = 3$
- $S(3) = S(2) + 3 = 3 + 3 = 6$
- $S(4) = S(3) + 4 = 6 + 4 = 10$

The solution

$$S(n) = \frac{n(n + 1)}{2}$$
The Sum $1 + 2 + \cdots + n$

Proof by induction

- Induction base: $S(0) = 0 = \frac{0 \cdot 1}{2}$
- Induction hypothesis: $S(k) = \frac{k(k+1)}{2}$ for all $0 \leq k < n$
- Inductive step for $n \geq 1$:

$$S(n) = S(n-1) + n = \frac{(n-1)n}{2} + n$$
$$= \frac{n^2 - n}{2} + \frac{2n}{2}$$
$$= \frac{n^2 + n}{2}$$
$$= \frac{n(n+1)}{2}$$
The Sum $1 + 3 + \cdots + (2k - 1)$

The recursive formula

$$S(n) = \begin{cases} 
1 & \text{for } n = 1 \\
S(n - 2) + n & \text{for odd } n > 1
\end{cases}$$

Small values of $n$

- $S(1) = 1$
- $S(3) = S(1) + 3 = 1 + 3 = 4$
- $S(5) = S(3) + 5 = 4 + 5 = 9$
- $S(7) = S(5) + 7 = 9 + 7 = 16$

The solution

Assume $n = 2h - 1$,

$$S(n) = S(2h - 1) = h^2 = \left( \frac{n + 1}{2} \right)^2$$
The Sum $1 + 3 + \cdots + (2k - 1)$

**Proof by induction**

- **Induction base:** $S(1) = S(2 \cdot 1 - 1) = 1 = 1^2$
- **Induction hypothesis:** $S(2k - 1) = k^2$ for all $1 \leq k < h$
- **Inductive step for $n = 2h - 1$ and $h > 1$:**

\[
S(2h - 1) = S(2h - 3) + (2h - 1)
\]
\[
= S(2(h - 1) - 1) + (2h - 1)
\]
\[
= (h - 1)^2 + (2h - 1)
\]
\[
= (h^2 - 2h + 1) + (2h - 1)
\]
\[
= h^2
\]
A General Recursive Formula

**Theorem**

- Let $a$ and $b$ be real numbers.
- Let $r \neq 1$ be a positive real number.
- Assume

$$T(n) = \begin{cases} 
    b & \text{for } n = 0 \\
    rT(n - 1) + a & \text{for } n \geq 1 
\end{cases}$$

- Then

$$T(n) = br^n + a \frac{r^n - 1}{r - 1}$$
A General Recursive Formula

The recursive formula

\[ T(n) = \begin{cases} 
  b & \text{for } n = 0 \\
  rT(n - 1) + a & \text{for } n \geq 1
\end{cases} \]

Top-Down evaluation

\[
T(n) = rT(n - 1) + a \\
= r^2T(n - 2) + ar + a \\
= r^3T(n - 3) + ar^2 + ar + a \\
\vdots \\
= r^n T(0) + ar^{n-1} + ar^{n-2} + \cdots + ar + a \\
= br^n + a \sum_{i=0}^{n-1} r^i \\
= br^n + a \frac{r^n - 1}{r - 1}
\]
A General Recursive Formula

The recursive formula

\[ T(n) = \begin{cases} 
  b & \text{for } n = 0 \\
  rT(n-1) + a & \text{for } n \geq 1
\end{cases} \]

Bottom-Up evaluation

\[
T(0) = b \\
T(1) = rT(0) + a = br + a \\
T(2) = rT(1) + a = br^2 + ar + a \\
T(3) = rT(2) + a = br^3 + ar^2 + ar + a \\
\vdots \\
T(n) = br^n + a \sum_{i=0}^{n-1} r^i \\
T(n) = br^n + a \frac{r^n - 1}{r - 1}
\]
First Special Case of the General Formula

**Theorem**
- Recursive formula: \( T(0) = b \) and \( T(n) = rT(n - 1) + a \)
- Closed-form formula: \( T(n) = br^n + a\frac{r^n - 1}{r - 1} \)

\( b = 1, \ a = 0, \text{ and } r \neq 1 \)
- Recursive formula: \( T(0) = 1 \) and \( T(n) = rT(n - 1) \)
- Closed-form formula:

\[
T(n) = 1 \cdot r^n + 0 \cdot \frac{r^n - 1}{r - 1} = r^n
\]
Second Special Case of the General Formula

**Theorem**
- Recursive formula: \( T(0) = b \) and \( T(n) = rT(n - 1) + a \)
- Closed-form formula: \( T(n) = br^n + a \frac{r^n - 1}{r - 1} \)

**Example**

\( b = 0, \ a = 1, \text{ and } r = 2 \)
- Recursive formula: \( T(0) = 0 \) and \( T(n) = 2T(n - 1) + 1 \)
- Closed-form formula:

\[
T(n) = 0 \cdot r^n + 1 \cdot \frac{2^n - 1}{2 - 1} = 2^n - 1
\]
Third Special Case of the General Formula

**Theorem**
- Recursive formula: \( T(0) = b \) and \( T(n) = rT(n - 1) + a \)
- Closed-form formula: \( T(n) = br^n + a \frac{r^n - 1}{r - 1} \)

**b = 1, a = -1, and r = 2**
- Recursive formula: \( T(0) = 1 \) and \( T(n) = 2T(n - 1) - 1 \)
- Closed-form formula:
  \[
  T(n) = 1 \cdot 2^n - 1 \cdot \frac{2^n - 1}{2 - 1} = 1
  \]

**Intuition**
- \( T(n) = 1 \) because \( 2 \cdot 1 - 1 = 1 \)
Fourth Special Case of the General Formula

Theorem

- Recursive formula: \( T(0) = b \) and \( T(n) = rT(n - 1) + a \)
- Closed-form formula: \( T(n) = br^n + a \frac{r^n-1}{r-1} \)

\( b = 0, \ a = 1/2, \ \text{and} \ r = 1/2 \)
- Recursive formula: \( T(0) = 0 \) and \( T(n) = (1/2)T(n - 1) + 1/2 \)
- Closed-form formula:

\[
T(n) = 0 \cdot r^n + \frac{1}{2} \cdot \frac{\left(\frac{1}{2}\right)^n - 1}{\frac{1}{2} - 1} = \frac{1}{2} \cdot \frac{1 - \left(\frac{1}{2}\right)^n}{\frac{1}{2}}
\]

\[
= 1 - \left(\frac{1}{2}\right)^n = 1 - 2^{-n}
\]
A Divide-and-Conquer Recursive Formula

The recursive formula

\[ T(n) = \begin{cases} 
0 & \text{for } n = 1 \\
2T(n/2) + n & \text{for } n > 1 \text{ a power of 2}
\end{cases} \]

Small values of $n = 2^h$

- $T(2^0) = T(1) = 0$
- $T(2^1) = T(2) = 2T(1) + 2 = 2$
- $T(2^2) = T(4) = 2T(2) + 4 = 8$
- $T(2^3) = T(8) = 2T(4) + 8 = 24$
- $T(2^4) = T(16) = 2T(8) + 16 = 64$
- $T(2^5) = T(32) = 2T(16) + 32 = 160$
A Divide-and-Conquer Recursive Formula

Guessing the solution

For $n = 2^h$ guess:

$$T(n) = n \log_2 n$$

Verify the guess for small numbers

- $1 \log_2 1 = 0 = T(1)$
- $2 \log_2 2 = 2 = T(2)$
- $4 \log_2 4 = 8 = T(4)$
- $8 \log_2 8 = 24 = T(8)$
- $16 \log_2 16 = 64 = T(16)$
- $32 \log_2 32 = 160 = T(32)$
A Divide-and-Conquer Recursive Formula

Proof by induction

\[ T(n) = 2T\left(\frac{n}{2}\right) + n \]
\[ = 2\left(\frac{n}{2}\right) \log_2 \left(\frac{n}{2}\right) + n \]
\[ = n \log_2 n - 1 + n \]
\[ = n \log_2 n \]
A Generalized Divide-and-Conquer Formula

The recursive formula
For real numbers $a$ and $b$ (independent of $n$)

$$T(n) = \begin{cases} 
a & \text{for } n = 1 \\
2T(n/2) + bn & \text{for } n > 1 \text{ a power of } 2
\end{cases}$$

Small values of $n = 2^h$

- $T(2^0) = T(1) = a$
- $T(2^1) = T(2) = 2T(1) + 2b = 2b + 2a$
- $T(2^2) = T(4) = 2T(2) + 4b = 8b + 4a$
- $T(2^3) = T(8) = 2T(4) + 8b = 24b + 8a$
- $T(2^4) = T(16) = 2T(8) + 16b = 64b + 16a$
- $T(2^5) = T(32) = 2T(16) + 32b = 160b + 32a$
A Generalized Divide-and-Conquer Formula

Guessing the solution

- For $n = 2^h$ guess:

$$T(n) = bn \log_2 n + an$$

Verify the guess for small numbers

- $b \cdot 1 \log_2 1 + a \cdot 1 = a = T(1)$
- $b \cdot 2 \log_2 2 + a \cdot 2 = 2b + 2a = T(2)$
- $b \cdot 4 \log_2 4 + a \cdot 4 = 8b + 4a = T(4)$
- $b \cdot 8 \log_2 8 + a \cdot 8 = 24b + 8a = T(8)$
- $b \cdot 16 \log_2 16 + a \cdot 16 = 64b + 16a = T(16)$
- $b \cdot 32 \log_2 32 + a \cdot 32 = 160b + 32a = T(32)$
A Generalized Divide-and-Conquer Formula

Proof by induction

\[
T(n) = 2T(n/2) + bn \\
= 2(b(n/2) \log_2(n/2) + a(n/2)) + bn \\
= (bn(\log_2 n - 1) + an) + bn \\
= bn \log_2 n - bn + an + bn \\
= bn \log_2 n + an
\]
Another Divide-and-Conquer Recursive Formula

The recursive formula
For real numbers $a$ and $b$ (independent of $n$)

$$T(n) = \begin{cases} \ a & \text{for } n = 1 \\ T(n/2) + b & \text{for } n > 1 \text{ a power of 2} \end{cases}$$

Small values of $n = 2^h$

- $T(2^0) = T(1) = a$
- $T(2^1) = T(2) = T(1) + b = b + a$
- $T(2^2) = T(4) = T(2) + b = 2b + a$
- $T(2^3) = T(8) = T(4) + b = 3b + a$
- $T(2^4) = T(16) = T(8) + b = 4b + a$
- $T(2^5) = T(32) = T(16) + b = 5b + a$
Another Divide-and-Conquer Recursive Formula

**Guessing the solution**

- For $n = 2^h$ guess:

  $$T(n) = b \log_2 n + a$$

**Verify the guess for small numbers**

- $b \log_2 1 + a = a = T(1)$
- $b \log_2 2 + a = b + a = T(2)$
- $b \log_2 4 + a = 2b + a = T(4)$
- $b \log_2 8 + a = 3b + a = T(8)$
- $b \log_2 16 + a = 4b + a = T(16)$
- $b \log_2 32 + a = 5b + a = T(32)$
Another Divide-and-Conquer Recursive Formula

Proof by induction

\[ T(n) = T(n/2) + b \]
\[ = (b \log_2(n/2) + a) + b \]
\[ = b(\log_2 n - 1) + a + b \]
\[ = b \log_2 n - b + a + b \]
\[ = b \log_2 n + a \]
A Third Divide-and-Conquer Recursive Formula

The recursive formula

\[ T(n) = \begin{cases} 
1 & \text{for } n = 1 \\
4T(n/2) & \text{for } n > 1 \text{ a power of 2}
\end{cases} \]

Small values of \( n = 2^h \)

- \( T(2^0) = T(1) = 1 \)
- \( T(2^1) = T(2) = 4T(1) = 4 \)
- \( T(2^2) = T(4) = 4T(2) = 16 \)
- \( T(2^3) = T(8) = 4T(4) = 64 \)
- \( T(2^4) = T(16) = 4T(8) = 256 \)
- \( T(2^5) = T(32) = 4T(16) = 1024 \)
A Third Divide-and-Conquer Recursive Formula

Guessing the solution
- For $n = 2^h$ guess:

$$T(n) = n^2$$

Verify the guess for small numbers
- $1^2 = 1 = T(1)$
- $2^2 = 4 = T(2)$
- $4^2 = 16 = T(4)$
- $8^2 = 64 = T(8)$
- $16^2 = 256 = T(16)$
- $32^2 = 1024 = T(32)$
A Third Divide-and-Conquer Recursive Formula

Proof by induction

\[
T(n) = 4T(n/2) \\
= 4(n/2)^2 \\
= 4(n^2/4) \\
= n^2
\]
The Master Theorem

Assumptions

- For real numbers $a > 0$, $b > 1$, and $d \geq 0$ (independent of $n$):

$$T(n) = \begin{cases} 
\Theta(1) & \text{for } n = 1 \\
aT(n/b) + \Theta(n^d) & \text{for } n > 1 
\end{cases}$$

- $n/b$ can be either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.

Theorem

**Case I:** If $d < \log_b a$
- then $T(n) = \Theta(n^{\log_b a})$.

**Case II:** If $d = \log_b a$
- then $T(n) = \Theta(n^{\log_b a \log n}) = \Theta(n^d \log n)$.

**Case III:** If $d > \log_b a$
- then $T(n) = \Theta(n^d)$. 
Example I

\[
T(1) = 1 \\
T(n) = 9T(n/3) + n
\]

- \( a = 9 \).
- \( b = 3 \).
- \( d = 1 \).
- \( \log_b a = \log_3 9 = 2 > 1 = d \).

\[\implies \textbf{Case I: } T(n) = \Theta(n^2).\]
Example II

\[
T(1) = 1 \\
T(n) = T\left(\frac{2n}{3}\right) + 1
\]

- \(a = 1\).
- \(b = \frac{3}{2}\).
- \(d = 0\).
- \(\log_b a = \log_{\frac{3}{2}} 1 = 0 = d\).

\[\Rightarrow \text{Case II: } T(n) = \Theta(n^0 \log n) = \Theta(\log n).\]
Example III

\[
T(1) = 1 \\
T(n) = 3T(n/4) + n
\]

- \( a = 3 \).
- \( b = 4 \).
- \( d = 1 \).
- \( \log_b a = \log_4 3 \approx 0.793 < 1 = d \).

\[\implies \text{Case III: } T(n) = \Theta(n)\]
Proof Outline for the Master Theorem

- Assume that \( n \) is a power of \( b \).
- There are \( \log_b(n) \) levels to the recursion.
- The \( k^{th} \) level is made up of \( a^k \) subproblems.
- Each subproblem at level \( k \) is of size \( n/b^k \).
- The total work done at level \( k \) is:

\[
w(k) = a^k \cdot \Theta \left( \frac{n}{b^k} \right)^d = \Theta(n^d) \cdot \left( \frac{a}{b^d} \right)^k
\]
The numbers \( w(0), w(1), \ldots, w(\log_b(n)) \) form a geometric series with ratio \( a/b^d \).

\( w(0) = \Theta(n^d) \).

\( w(\log_b(n)) = \Theta(a^{\log_b(n)}) = \Theta(n^{\log_b(a)}) \).

\( T(n) = \sum_{k=0}^{\log_b(n)} w(k) = \Theta(n^d) \sum_{k=0}^{\log_b(n)} \left( \frac{a}{b^d} \right)^k \).

The sum depends on the ratio \( a/b^d \).
Proof Outline for the Master Theorem

- If \( \frac{a}{b^d} < 1 \) then the sum is dominated by the first term.
  \[ T(n) = \Theta(w(0)) = \Theta(n^d). \]

- If \( \frac{a}{b^d} = 1 \) then all \( \Theta(\log(n)) \) terms are equal to \( \Theta(n^d) \).
  \[ T(n) = \Theta(n^d \log(n)). \]

- If \( \frac{a}{b^d} > 1 \) then the sum is dominated by the last term.
  \[ T(n) = \Theta(w(\log_b(n))) = \Theta(n^{\log_b(a)}). \]

- Comparing \( \frac{a}{b^d} \) to 1 is equivalent to comparing \( a \) to \( b^d \) which is equivalent to comparing \( \log_b(a) \) to \( d \).
The Prefix-Sum Problem

Input
- An array \( A = [A[1], A[2], \ldots, A[n]] \) with \( n \geq 1 \) real numbers.

Output
- An array \( S = [S[1], S[2], \ldots, S[n]] \) such that for all \( 1 \leq i \leq n \),

\[
S[i] = \sum_{j=1}^{i} A[j]
\]

Example
- \( A = [13, 34, -8, -55, -5, 21, \ldots] \)
- \( S = [13, 47, 39, -16, -21, 0 \ldots] \)

Optimization goal
- Minimize the number of additions between numbers
A By Definition Algorithm

Algorithm

prefix-sum(A)
  for $i = 1$ to $n$ do
    $S[i] := 0$
  for $i = 1$ to $n$ do
    for $j = 1$ to $i$ do

Correctness

• By definition

Complexity

• $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ additions in the inner loop
• $\Theta(n^2)$ complexity
A By Induction Algorithm

Algorithm

\[
\text{prefix-sum}(A) \\
\text{for } i = 2 \text{ to } n \text{ do} \\
S[i] := S[i-1] + A[i]
\]

Correctness

- Induction hypothesis, for \( 1 \leq i \leq n - 1 \), after iteration \( i - 1 \):
  \[
  S[i - 1] = \sum_{j=1}^{i-1} A[j]
  \]
- By Induction for \( 2 \leq i \leq n \), after iteration \( i \):
  \[
  S[i] = S[i - 1] + A[i] = \sum_{j=1}^{i-1} A[j] + A[i] = \sum_{j=1}^{i} A[j]
  \]
A By Induction Algorithm

Algorithm

prefix-sum(A)


for \( i = 2 \) to \( n \) do


Complexity

- \( n - 1 \) additions in the loop
- \( \Theta(n) \) complexity
Evaluating a Polynomial

**Input**
- Real numbers $c$ and $a_0, a_1, \ldots, a_n$ such that $a_n \neq 0$.

**Output**
- The value of the polynomial $P(x)$ for $x = c$:
  \[ P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \]

**Example**
- $a_4 = 5$, $a_3 = 0$, $a_2 = -7$, $a_1 = 3$, $a_0 = -11$, and $c = 2$.
- $P(x) = 5x^4 - 7x^2 + 3x - 11$
- $P(2) = 5 \cdot 2^4 - 7 \cdot 2^2 + 3 \cdot 2 - 11 = 47$

**Optimization goal**
- Minimize the number of operations (multiplications and additions) between real numbers.
A By Definition Algorithm

Algorithm

Polynomial-Evaluation\((P(x), c)\)

\[
P(c) = a_0
\]

for \(i = 1\) to \(n\) do

\[
a = a_i
\]

for \(j = 1\) to \(i\) do

\[
a = a \times c
\]

\[
P(c) = P(c) + a
\]

return \(P(c)\)

Correctness

- By definition.
A By Definition Algorithm

**Complexity**

- \( i \) multiplications in the \( i^{th} \) iteration of the inner loop.
- \( 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \) multiplications overall.
- \( n \) additions in the outer loop.

Total number of operations:

\[
\frac{n(n + 1)}{2} + n = \frac{n^2 + n}{2} + \frac{2n}{2} = \frac{n^2 + 3n}{2} = \frac{1}{2}n^2 + \frac{3}{2}n
\]

- \( \Theta(n^2) \) overall complexity.
A Prefix-Sum Algorithm

Idea

- Compute \( c, c^2, c^3, \ldots, c^n \) all the powers of \( c \) using the efficient \textit{prefix-sum} method.

Algorithm

**Polynomial-Evaluation** \((P(x), c)\)

\[
P(c) = a_0 \\
c_{c} = 1 \\
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad c_{c} = c_{c} \times c \quad (* \quad c_{c} = c^{i} *) \\
\quad P(c) = P(c) + a_i \times c_{c} \quad (* \quad P(c) = a_i c^i + a_{i-1} c^{i-1} + \cdots + a_1 c + a_0 *) \\
\text{return} (P(c)) \quad (* \quad P(c) = a_n c^n + a_{n-1} c^{n-1} + \cdots + a_1 c + a_0 *)
\]

Correctness

- By induction.
A Prefix-Sum Algorithm

Complexity

- 2 multiplications in the $i^{th}$ iteration of the loop
- 1 addition in the $i^{th}$ iteration of the loop
- Total of $3n$ operations: $2n$ multiplications and $n$ additions
- $\Theta(n)$ overall complexity
The Horner’s Algorithm

Idea

\[ P(x) = (\cdots ((a_n x + a_{n-1}) x + a_{n-2}) x + \cdots ) x + a_0 \]

Example I

\[ 4x^3 + 3x^2 + 2x + 1 = ((4x + 3)x + 2)x + 1 \]

Example II

\[ 5x^4 - 7x^2 + 3x - 11 = (((5x + 0)x - 7)x + 3)x - 11 \]
The Horner’s Algorithm

**Algorithm**

- **Polynomial-Evaluation** \((P(x), c)\)

  \[
  P(c) = a_n \\
  \text{for } i = n - 1 \text{ downto } 0 \text{ do} \\
  P(c) = P(c) \cdot c + a_i \\
  (* P(c) = a_n c^{n-i} + a_{n-1} c^{n-i-1} + \cdots + a_{i+1} c + a_i *) \\
  \text{return}(P(c)) \\
  (* P(c) = a_n c^n + a_{n-1} c^{n-1} + \cdots + a_1 c + a_0 *)
  \]

**Correctness**

- By Induction.
The Horner’s Algorithm

Example II

**Input:** Evaluate $P(x) = 5x^4 - 7x^2 + 3x - 11$ for $c = 2$.
* In the above polynomial $a_3 = 0$.

**Running the algorithm**

\[
\begin{align*}
P_4(x) &= a_4 = 5 \\
P_3(x) &= P_4(x) \cdot c + a_3 = 5 \cdot 2 + 0 = 10 \\
P_2(x) &= P_3(x) \cdot c + a_2 = 10 \cdot 2 - 7 = 13 \\
P_1(x) &= P_2(x) \cdot c + a_1 = 13 \cdot 2 + 3 = 29 \\
P(x) &= P_1(x) \cdot c + a_0 = 29 \cdot 2 - 11 = 47
\end{align*}
\]
The Horner’s Algorithm

**Complexity**

- 1 multiplication in the $i$th iteration of the loop.
- 1 addition in the $i$th iteration of the loop.
- Total of $2n$ operations: $n$ multiplications and $n$ additions.
- $\Theta(n)$ overall complexity.