Analysis of Algorithms

Amotz Bar-Noy

CUNY
Algorithm: Definitions

1. A finite set of precise instructions for performing a computation or for solving a problem.
2. 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.
3. A procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation.
4. A step-by-step procedure for solving a problem or accomplishing some end especially by a computer.
5. A logical arithmetical or computational procedure that if correctly applied ensures the solution of a problem.
6. 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, Recipe, Routine, Solution, Technique . . .

- **Etymology:** Alteration of Middle English *algorisme*, from Old French & Medieval Latin; from Medieval Latin *algorismus*, from Arabic *al-khuwarizmi*, from the name of the Persian Mathematician *Al-Khowârizmi* who was the first to formalize the rules for the 4 basic arithmetical operations.
The Ultimate Algorithmic Problem!?

**Question:** how do we solve problems?

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

Answer: use a combination of these 5 factors!!!
**Question:** how do we solve problems?

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

Answer: use a combination of these 5 factors!!!
Question: how do we solve problems?

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

Answer: use a combination of these 5 factors!!!
How to solve a problem? Some Heuristics

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.
Correctness: for all valid inputs.

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

Scalability: “similar” efficiency for any input size.

Limitations: for the algorithm and for the problem.

Optimality: optimal or near-optimal or approximately optimal solutions.
Algorithm Complexity

- How much of resources does an algorithm require?
  - Usually: time and space (memory).

- Complexity: as a function of the input length.
  - Usually an integer $n > 0$.
  - Usually a monotonic non-decreasing function.
Worst Case and Average Case Complexity

- \( T(n) \) is a **worst case complexity**: If for all inputs of length \( n \) the complexity is \( T(n) \).

- \( T(n) \) is an **average case complexity**: If the **average** complexity over all length \( n \) inputs is \( T(n) \).
  Averaging based on some distribution of the inputs (usually the uniform distribution).
\textbf{Bounds}

- **An Upper bound:** A function $f(n)$ such that
  \[ T(n) \leq f(n) \text{ for all } n. \]

- **A Lower bound:** A function $g(n)$ such that
  \[ T(n) \geq g(n) \text{ for all } n. \]

- **A Tight bound:** A function $h(n)$ such that
  \[ T(n) \approx h(n) \text{ for all } n. \]
Performance Evaluation of Algorithms

- **Theoretical analysis:**
  - All possible inputs.
  - Independent of hardware/software implementation.
  - High level language.

- **Experimental Study:**
  - Some typical inputs.
  - Depends on hardware/software implementation.
  - A **real** program.
Objective: A language to express that Algorithm A is better than or worse than or equivalent to Algorithm B.

Need to define a “≤” relation between functions measuring the growth of functions.


Ignore constants that can be affected by changing the environment.
### 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>$400n$</td>
<td>2,500</td>
<td>150,000</td>
<td>9,000,000</td>
</tr>
<tr>
<td>$2n^2$</td>
<td>707</td>
<td>5,477</td>
<td>42,426</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>$400n$</td>
<td>$256m$</td>
</tr>
<tr>
<td>$2n^2$</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.
The “$O$, $\Omega$, $\Theta$, $o$, $\omega$” Notation

- **Big-Oh**: $f(n) = O(g(n))$ if $f(n)$ asymptotically less than or equal to $g(n)$.

- **Big-Omega**: $f(n) = \Omega(g(n))$ if $f(n)$ asymptotically greater than or equal to $g(n)$.

- **Big-Theta**: $f(n) = \Theta(g(n))$ if $f(n)$ asymptotically equal to $g(n)$.

- **Little-oh**: $f(n) = o(g(n))$ if $f(n)$ asymptotically strictly less than $g(n)$.

- **Little-omega**: $f(n) = \omega(g(n))$ if $f(n)$ asymptotically strictly greater than $g(n)$. 
Big-Oh, Big-Omega, and Big-Theta

- **$f(n) = O(g(n))$:**
  - There exists 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 exists 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 $c', c'' > 0$ and an integer constant $n_0 > 0$ such that $c''g(n) \leq f(n) \leq c'g(n)$ for every integer $n \geq n_0$. 
Propositions

- \( f(n) = O(g(n)) \) iff \( g(n) = \Omega(f(n)) \).
- \( f(n) = \Theta(g(n)) \) iff \( g(n) = \Theta(f(n)) \).
- \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \) iff \( f(n) = \Theta(g(n)) \).
- \( f(n) = O(g(n)) \) and \( g(n) = O(h(n)) \) \( \Rightarrow \) \( f(n) = O(h(n)) \).
- \( f(n) = \Omega(g(n)) \) and \( g(n) = \Omega(h(n)) \) \( \Rightarrow \) \( f(n) = \Omega(h(n)) \).
- \( f(n) = \Theta(g(n)) \) and \( g(n) = \Theta(h(n)) \) \( \Rightarrow \) \( f(n) = \Theta(h(n)) \).
Examples

- $3n = \Theta(n/2)$.
- $1000000n = \Theta(n/100000)$.
- $n \log_2 n / 100000 = \Omega(10000000n)$.
- $\log_2(n) = \Theta(\log_{10}(n))$.
- $a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0 = \Theta(n^d)$
  - for constants $a_0, a_1, \ldots, a_d$ and $a_d > 0$. 
More Propositions

For any real constant $c$:
- $O(cf(n)) = O(f(n))$.
- $O(f(n)/c) = O(f(n))$.
- $O(c) = O(1)$.

$O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max \{f(n), g(n)\})$.

$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$. 
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 \( f(n) \leq cg(n) \) 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 \( f(n) \geq cg(n) \) for every integer \( n \geq n_0 \).
Examples and Propositions

Examples:

- \( \log_2 n = o(\sqrt{n}) \).
- \( n^3 = \omega(n^2) \).
- \( 10^{100} n = o(n^2/10^{100}) \).

Propositions:

- \( f(n) = o(g(n)) \) iff \( g(n) = \omega(f(n)) \).
- \( f(n) = o(g(n)) \) and \( g(n) = o(h(n)) \) \( \Rightarrow \) \( f(n) = o(h(n)) \).
- \( f(n) = \omega(g(n)) \) and \( g(n) = \omega(h(n)) \) \( \Rightarrow \) \( f(n) = \omega(h(n)) \).
## Hierarchy of Functions

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Expression</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Log star</td>
<td>$\log^* n$</td>
<td></td>
</tr>
<tr>
<td>Loglog</td>
<td>$\log \log n$</td>
<td></td>
</tr>
<tr>
<td>Logarithmic</td>
<td>$\log n$</td>
<td></td>
</tr>
<tr>
<td>Poly-logarithmic</td>
<td>$\log^k n$</td>
<td>Constant integer $k &gt; 1$</td>
</tr>
<tr>
<td>Sub-linear</td>
<td>$n^\varepsilon$</td>
<td>Constant $0 &lt; \varepsilon &lt; 1$</td>
</tr>
<tr>
<td>Linear</td>
<td>$n$</td>
<td></td>
</tr>
<tr>
<td>Above-linear</td>
<td>$n \log n$</td>
<td></td>
</tr>
<tr>
<td>Quadratic</td>
<td>$n^2$</td>
<td></td>
</tr>
<tr>
<td>Cubic</td>
<td>$n^3$</td>
<td></td>
</tr>
<tr>
<td>Polynomial</td>
<td>$n^k$</td>
<td>Constant integer $k &gt; 1$</td>
</tr>
<tr>
<td>Super-polynomial</td>
<td>$n^{\log n}$</td>
<td></td>
</tr>
<tr>
<td>Exponential</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>Factorial</td>
<td>$n!$</td>
<td></td>
</tr>
<tr>
<td>Super-exponential</td>
<td>$n^n$</td>
<td></td>
</tr>
<tr>
<td>Exponential tower</td>
<td>$2^2 \cdots n$</td>
<td>$n$ powers</td>
</tr>
</tbody>
</table>
Consider the following recursive formula:

- $T(1) = 0$.
- $T(n) = 2T(n/2) + n$.

Compute the solution for small powers of 2:

- $T(2) = 2T(1) + 2 = 2$.
- $T(4) = 2T(2) + 4 = 8$.
- $T(8) = 2T(4) + 8 = 24$.
- $T(16) = 2T(8) + 16 = 64$. 
Guessing the Solution

For \( n = 2^k \) (power of 2), guess:

\[ T(n) = n \log_2 n. \]

Verify the guess for small numbers:

- \( 1 \log_2 1 = 0 \).
- \( 2 \log_2 2 = 2 \).
- \( 4 \log_2 4 = 8 \).
- \( 8 \log_2 8 = 24 \).
- \( 16 \log_2 16 = 64 \).
A Proof by Induction

\[ T(n) = 2T(n/2) + n \]
\[ = 2(n/2) \log_2(n/2) + n \]
\[ = n(\log_2 n - 1) + n \]
\[ = n \log_2 n \]
Consider the following recursive formula:

\[ T(1) = a. \]
\[ T(n) = 2T(n/2) + bn. \]

For some constants \( a, b \) (independent of \( n \)).

Compute the solution for small powers of 2:

\[ T(2) = 2T(1) + 2b = 2b + 2a. \]
\[ T(4) = 2T(2) + 4b = 8b + 4a. \]
\[ T(8) = 2T(4) + 8b = 24b + 8a. \]
\[ T(16) = 2T(8) + 16b = 64b + 16a. \]
Guessing the Solution

For $n = 2^k$ (power of 2), guess:

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

Verify the guess for small numbers:

- $b \cdot 1 \log_2 1 + a \cdot 1 = a$.
- $b \cdot 2 \log_2 2 + a \cdot 2 = 2b + 2a$.
- $b \cdot 4 \log_2 4 + a \cdot 4 = 8b + 4a$.
- $b \cdot 8 \log_2 8 + a \cdot 8 = 24b + 8a$.
- $b \cdot 16 \log_2 16 + a \cdot 16 = 64b + 16a$. 
A Proof by Induction

\[ T(n) = 2T(n/2) + bn \]
\[ = 2(b(n/2) \log_2(n/2) + a(n/2)) + bn \]
\[ = bn(\log_2 n - 1) + an + bn \]
\[ = bn \log_2 n + an \]
Consider the following recursive formula:

- \( T(1) = a. \)
- \( T(n) = T(n/2) + b. \)

For some constants \( a, b \) (independent of \( n \)).

Compute the solution for small powers of 2:

- \( T(2) = T(1) + b = b + a. \)
- \( T(4) = T(2) + b = 2b + a. \)
- \( T(8) = T(4) + b = 3b + a. \)
- \( T(16) = T(8) + b = 4b + a. \)
Guessing the Solution

For \( n = 2^k \) (power of 2), guess:

\[ T(n) = b \log_2 n + a. \]

Verify the guess for small numbers:

- \( b \cdot \log_2 1 + a = a \).
- \( b \cdot \log_2 2 + a = b + a \).
- \( b \cdot \log_2 4 + a = 2b + a \).
- \( b \cdot \log_2 8 + a = 3b + a \).
- \( b \cdot \log_2 16 + a = 4b + a \).
A Proof by Induction

\[
T(n) = T(n/2) + b = (b \log_2(n/2) + a) + b = b(\log_2 n - 1) + a + b = b \log_2 n + a
\]
Consider the following recursive formula:

- $T(1) = 0$.
- $T(n) = 4T(n/2) + n$.

Compute the solution for small powers of 2:

- $T(2) = 4T(1) + 2 = 2$.
- $T(4) = 4T(2) + 4 = 12$.
- $T(8) = 4T(4) + 8 = 56$.
- $T(16) = 4T(8) + 16 = 240$. 
Guessing the Solution

For $n = 2^k$ (power of 2), guess:

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

Verify the guess for small numbers:

- $1^2 - 1 = 0$.
- $2^2 - 2 = 2$.
- $4^2 - 4 = 12$.
- $8^2 - 8 = 56$.
- $16^2 - 16 = 240$. 
A Proof by Induction

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

**Assumptions:**
- Let \( a > 0 \) and \( b > 1 \) and \( d \geq 0 \) be constants (Independent of \( n \)).
- Let \( T(1) = \Theta(1) \).
- Let \( T(n) = aT(n/b) + \Theta(n^d) \) for \( n > 1 \).
  - \( n/b \) 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

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

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

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

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

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

\[ \implies \textbf{Case II: } T(n) = \Theta(\log n). \]
Example III

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

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

\(\implies\) **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:

\[
w(k) = a^k \cdot \Theta \left( \frac{n}{b^k} \right)^d = \Theta(n^d) \cdot \left( \frac{a}{b^d} \right)^k
\]
Proof Outline for the Master Theorem

- 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 $a/b^d$. 

Amotz Bar-Noy (CUNY)
Proof Outline for the Master Theorem

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

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

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

Comparing \( a/b^d \) to 1 is equivalent to comparing \( a \) to \( b^d \) which is equivalent to comparing \( \log_b(a) \) to \( d \).
Algorithm $A$ has a **worst case** complexity $T(n)$:

- To prove that $T(n) = O(f(n))$,
  - show this for **all** inputs of size $n$ for **all** $n$.

- To prove that $T(n) = \Omega(f(n))$,
  - show this for **one** input of size $n$ 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 $\mathcal{A}$ has an **average case** complexity $T(n)$ for a given distribution:

- To prove that $T(n) = O(f(n))$,
  - show this by **averaging over all** inputs of size $n$ for **all** $n$.

- To prove that $T(n) = \Omega(f(n))$,
  - show this by **averaging over all** inputs of size $n$ for **infinitely many** $n$.

- To prove that $T(n) = \Theta(f(n))$,
  - show that $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$. 
The Prefix-Sum Problem

- **Output:** An array $S$ of size $n$ such that for $1 \leq i \leq n$,

$$S[i] = \sum_{j=1}^{i} A[j].$$
The Prefix-Sum Problem

- **Output:** An array $S$ of size $n$ such that for $1 \leq i \leq n$,

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

- **Example:**
  - $A = [3, 1, 2, 3, 18, 100, \ldots]$  
  - $S = [3, 4, 6, 9, 27, 127, \ldots]$
Algorithm I

**prefix-sum**\((A)\)

\[
\begin{align*}
&\text{for } i = 1 \text{ to } n \text{ do} \\
&S[i] := 0 \\
&\text{for } i = 1 \text{ to } n \text{ do} \\
&\quad \text{for } j = 1 \text{ to } i \text{ do} \\
&\quad \quad S[i] := S[i] + A[j]
\end{align*}
\]
Algorithm I

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.
Algorithm I – Complexity

- $\Theta(n)$ time for the first loop.

- $\Theta(n^2)$ time for the second loop.
  - $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ iterations of the inner loop.
  - $\Theta(1)$ time for each iteration.

- $\Theta(n) + \Theta(n^2) = \Theta(n^2)$ time complexity.
Algorithm II

prefix-sum\(A\)

\[
\text{for } i = 2 \text{ to } n \text{ do} \\
S[i] := S[i - 1] + A[i]
\]
Algorithm II

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

Correctness: By Induction.
Algorithm II – Complexity

- $n - 1$ iterations of the only loop.
- $\Theta(1)$ time for each iteration.
- $\Theta(n)$ time complexity.
Evaluating a Polynomial

- **Input:** Real numbers $a_0, a_1, \ldots, a_n$ and $c$.
- **Output:** The value of the polynomial $P(x)$ for $x = c$:
  $$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0.$$
Evaluating a Polynomial

**Input:** Real numbers $a_0, a_1, \ldots, a_n$ and $c$.

**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:**

- $a_3 = 5$, $a_2 = 7$, $a_1 = 3$, $a_0 = 11$, and $c = 2$.
- $P(x) = 5x^3 + 7x^2 + 3x + 11$.
- $P(2) = 5 \cdot 2^3 + 7 \cdot 2^2 + 3 \cdot 2 + 11 = 85$. 

Optimization goal:

Minimize the number of operations (multiplications and additions) between real numbers.
Evaluating a Polynomial

Input: Real numbers $a_0, a_1, \ldots, a_n$ and $c$.

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:
- $a_3 = 5$, $a_2 = 7$, $a_1 = 3$, $a_0 = 11$, and $c = 2$.
- $P(x) = 5x^3 + 7x^2 + 3x + 11$.
- $P(2) = 5 \cdot 2^3 + 7 \cdot 2^2 + 3 \cdot 2 + 11 = 85$.

Optimization goal: Minimize the number of operations (multiplications and additions) between real numbers.
Algorithm I: A Direct Approach

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

$P(c) = a_0$

for $i = 1$ to $n$ do

$a = a_i$

for $j = 1$ to $i$ do

$a = a \cdot c$

(* $a = a_i c^i$ *)

$P(c) = P(c) + a$

(* $P(c) = a_i c^i + a_{i-1} c^{i-1} + \cdots + a_1 c + a_0$ *)

Correctness:
By definition.
Algorithm I: A Direct Approach

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

\[
P(c) = a_0 \\
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad a = a_i \\
\quad \text{for } j = 1 \text{ to } i \text{ do} \\
\quad \quad a = a \cdot c \\
\quad (*) a = a_i c^i (*) \\
P(c) = P(c) + a \\
(*) P(c) = a_i c^i + a_{i-1} c^{i-1} + \cdots + a_1 c + a_0 (*)
\]

Correctness: By definition.
Algorithm I – 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 of $\frac{1}{2}n^2 + \frac{3}{2}n$ operations.

- $\Theta(n^2)$ time complexity.
Algorithm II: A Prefix-Sum Approach

Idea: Compute $c, c^2, c^3, \ldots, c^n$ all the powers of $c$ using the efficient prefix-sum method.
Algorithm II: A Prefix-Sum Approach

- **Idea:** Compute $c, c^2, c^3, \ldots, c^n$ all the powers of $c$ using the efficient prefix-sum method.

- **Polynomial-Evaluation** ($P(x), c$)
  
  $P(c) = a_0$
  
  $cc = 1$
  
  for $i = 1$ to $n$ do
    
    $cc = cc \cdot c$
    
    (* $cc = c^i$ *)
  
  $P(c) = P(c) + a_i \cdot cc$
  
  (* $P(c) = a_i c^i + a_{i-1} c^{i-1} + \cdots + a_1 c + a_0$ *)

- **Correctness:** By induction.
Algorithm II: A Prefix-Sum Approach

- **Idea:** Compute \( c, c^2, c^3, \ldots, c^n \) all the powers of \( c \) using the efficient prefix-sum method.

- **Polynomial-Evaluation** \((P(x), c)\)
  
  \[
  P(c) = a_0 \\
  cc = 1 \\
  \text{for } i = 1 \text{ to } n \text{ do} \\
  \quad cc = cc \cdot c \\
  \quad (* cc = c^i *) \\
  \quad P(c) = P(c) + a_i \cdot cc \\
  \quad (* P(c) = a_i c^i + a_{i-1} c^{i-1} + \cdots + a_1 c + a_0 *)
  \]

- **Correctness:** By induction.
Algorithm II – 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.

$\Theta(n)$ time complexity.
**Algorithm III: A Sophisticated Method**

- **Idea and proof of correctness:**
  \[ P(x) = (\cdots ((a_n x + a_{n-1}) x + a_{n-2}) x + \cdots ) x + a_0. \]
Algorithm III: A Sophisticated Method

- **Idea and proof of correctness:**
  \[ P(x) = (\cdots ((a_n x + a_{n-1}) x + a_{n-2}) x + \cdots ) x + a_0. \]

- **Example:**
  \[ 4x^3 + 3x^2 + 2x + 1 = ((4x + 3)x + 2)x + 1. \]
Algorithm III: A Sophisticated Method

Idea and proof of correctness:

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

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

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

\[
P(c) = a_n \\
\text{for } i = n - 1 \text{ downto } 0 \text{ do} \\
\quad P(c) = P(c) \cdot c + a_i \\
\quad (* P(c) = a_n c^{n-i} + a_{n-1} c^{n-i-1} + \cdots + a_{i+1} c + a_i *)
\]
Algorithm III: A Sophisticated Method

**Idea and proof of correctness:**

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

**Example:** \( 4x^3 + 3x^2 + 2x + 1 = ((4x + 3)x + 2)x + 1. \)

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

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

**Correctness:** By Induction.
Algorithm III: Example

- **Input:** $P(x) = 16x^4 + 8x^3 + 4x^2 + 2x + 1$
Algorithm III: Example

- **Input:** \( P(x) = 16x^4 + 8x^3 + 4x^2 + 2x + 1 \)

- **Algorithm:**
Algorithm III: Example

**Input:** \( P(x) = 16x^4 + 8x^3 + 4x^2 + 2x + 1 \)

**Algorithm:**
- \( P_4(x) = a_4 = 16 \)
Algorithm III: Example

**Input:** \( P(x) = 16x^4 + 8x^3 + 4x^2 + 2x + 1 \)

**Algorithm:**
- \( P_4(x) = a_4 = 16 \)
- \( P_3(x) = P_4(x)x + a_3 = 16x + 8 \)
Algorithm III: Example

- **Input:** $P(x) = 16x^4 + 8x^3 + 4x^2 + 2x + 1$

- **Algorithm:**
  - $P_4(x) = a_4 = 16$
  - $P_3(x) = P_4(x)x + a_3 = 16x + 8$
  - $P_2(x) = P_3(x)x + a_2 = 16x^2 + 8x + 4$
Algorithm III: Example

**Input:** \( P(x) = 16x^4 + 8x^3 + 4x^2 + 2x + 1 \)

**Algorithm:**

- \( P_4(x) = a_4 = 16 \)
- \( P_3(x) = P_4(x)x + a_3 = 16x + 8 \)
- \( P_2(x) = P_3(x)x + a_2 = 16x^2 + 8x + 4 \)
- \( P_1(x) = P_2(x)x + a_1 = 16x^3 + 8x^2 + 4x + 2 \)
Algorithm III: Example

- **Input:** \( P(x) = 16x^4 + 8x^3 + 4x^2 + 2x + 1 \)

- **Algorithm:**
  - \( P_4(x) = a_4 = 16 \)
  - \( P_3(x) = P_4(x)x + a_3 = 16x + 8 \)
  - \( P_2(x) = P_3(x)x + a_2 = 16x^2 + 8x + 4 \)
  - \( P_1(x) = P_2(x)x + a_1 = 16x^3 + 8x^2 + 4x + 2 \)
  - \( P(x) = P_1(x)x + a_0 = 16x^4 + 8x^3 + 4x^2 + 2x + 1 \)
Algorithm III – 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.
- \( \Theta(n) \) time complexity.
The Josephus Problem

**Story:** \( n \) People are standing in a circle waiting to be executed. Counting begins at a specified point in the circle and proceeds around the circle clockwise. After \( k \) people are skipped, the next person is executed. The procedure is repeated with the remaining people starting with the next person until only one person remains, and is freed.

**Problem:** Given the starting point \( n \) and \( k \) find the position in the initial circle to avoid execution.

**A video lecture for** \( k = 0 \): [https://www.youtube.com/watch?v=uCsD3ZGzMgE](https://www.youtube.com/watch?v=uCsD3ZGzMgE)