Ch01. Analysis of Algorithms

What’s an Algorithm?

- Computer Science is about problem-solving using computers.
- Software is a solution to some problems.
- Algorithm is a recipe/design inside a software.
- Informally, an algorithm is a method for solving a well-specified computational problem.

- Algorithms become more and more important in digital age.
Organisms are algorithms, and as such homo sapiens (today’s human) may not be dominant in a universe where dataism becomes the paradigm.

Computers will do much better than organisms. Many professions will be out-of-date and labors become less worth.

Harari believes that humanism may push humans to search for immortality, happiness, and power.

Harari suggests the possibility of the replacement of humankind with a super-man, i.e. "homo deus“, endowed with abilities such as eternal life.
Algorithms and Data Structures

- **Algorithm** is a step-by-step procedure for performing some task in a finite amount of time.
  - Typically, an algorithm takes input data and produces an output based upon it.

- **Data structure** is a systematic way of organizing and accessing data.
Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition, noting the time needed:
- Plot the results
Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment
Pseudocode

- High-level description of an algorithm
  - More structured than English prose
  - Less detailed than a program
- Preferred notation for describing algorithms
- Easy map to primitive operations of CPU

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

$currentMax \leftarrow A[0]$
for $i \leftarrow 1$ to $n - 1$ do
  if $currentMax < A[i]$ then
    $currentMax \leftarrow A[i]$
return $currentMax$
Pseudocode Details

- **Control flow**
  - if ... then ... [else ...]
  - while ... do ...
  - for ... do ...
  - Indentation replaces braces

- **Method declaration**
  
  Algorithm *method* (*arg* [, *arg*...])
  
  Input ...
  
  Output ...

- **Method call**
  
  *method* (*arg* [, *arg*...])

- **Return value**
  
  return *expression*

- **Expressions:**
  
  ← Assignment
  
  = Equality testing
  
  $n^2$ Superscripts and other mathematical formatting allowed
The Random Access Machine (RAM) Model

A **RAM** consists of

- A **CPU**
- An potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character
- Memory cells are numbered and accessing any cell in memory takes unit time
Primitve Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

Examples:
- Arithmetic operations
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method
Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant $\approx 1$
  - Logarithmic $\approx \log n$
  - Linear $\approx n$
  - N-Log-N $\approx n \log n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$
  - Exponential $\approx 2^n$

- In a log-log chart, the slope of the line corresponds to the growth rate.
Functions Graphed Using “Normal” Scale

\[ g(n) = 1 \]

\[ g(n) = n \log n \]

\[ g(n) = 2^n \]

\[ g(n) = \log n \]

\[ g(n) = n^2 \]

\[ g(n) = n^3 \]

\[ g(n) = n \]
Example: By inspecting the pseudocode, we can determine the minimum and maximum number of primitive operations executed by an algorithm, as a function of the input size.

**Algorithm arrayMax(A, n):**
- **Input:** An array A storing \( n \geq 1 \) integers.
- **Output:** The maximum element in A.

```
currentMax \leftarrow A[0]
for i \leftarrow 1 \text{ to } n - 1 \text{ do}
    \text{if } currentMax < A[i] \text{ then}
        currentMax \leftarrow A[i]
return currentMax
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3n-1</td>
</tr>
<tr>
<td>3</td>
<td>2(n-1)</td>
</tr>
<tr>
<td>4</td>
<td>0 to 2(n-1)</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Minimum: \[ 2 + 3n-1 + 2(n-1) + 1 = 5n \]
Maximum: \[ 2 + 3n-1 + 4(n-1) + 1 = 7n - 2 \]
Estimating Running Time

- Algorithm \texttt{arrayMax} executes $7n - 2$ primitive operations in the worst case, $5n$ in the best case. Define:
  
  $a = \text{Time taken by the fastest primitive operation}$
  
  $b = \text{Time taken by the slowest primitive operation}$

- Let $T(n)$ be worst-case time of \texttt{arrayMax}. Then
  \[ a(5n) \leq T(n) \leq b(7n - 2) \]

- Hence, the running time $T(n)$ is bounded by two linear functions
Running Time

- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus primarily on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics
Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$

- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm arrayMax
## Why Growth Rate Matters

<table>
<thead>
<tr>
<th>if runtime is...</th>
<th>time for (n + 1)</th>
<th>time for (2n)</th>
<th>time for (4n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c \lg n)</td>
<td>(c \lg (n + 1))</td>
<td>(c (\lg n + 1))</td>
<td>(c(\lg n + 2))</td>
</tr>
<tr>
<td>(cn)</td>
<td>(c (n + 1))</td>
<td>(2cn)</td>
<td>(4cn)</td>
</tr>
<tr>
<td>(cn \lg n)</td>
<td>(~cn \lg n + cn)</td>
<td>(2cn \lg n + 2cn)</td>
<td>(4cn \lg n + 4cn)</td>
</tr>
<tr>
<td>(cn^2)</td>
<td>(~cn^2 + 2cn)</td>
<td>(~)</td>
<td>(4cn^2)</td>
</tr>
<tr>
<td>(cn^3)</td>
<td>(~cn^3 + 3cn^2)</td>
<td>(8cn^3)</td>
<td>(64cn^3)</td>
</tr>
<tr>
<td>(c2^n)</td>
<td>(c2^{n+1})</td>
<td>(c2^{2n})</td>
<td>(c2^{4n})</td>
</tr>
</tbody>
</table>

- **runtime quadruples when problem size doubles**
Analyzing Recursive Algorithms

- Use a function, $T(n)$, to derive a recurrence relation that characterizes the running time of the algorithm in terms of smaller values of $n$.

```
Algorithm recursiveMax(A, n):
    Input: An array A storing $n \geq 1$ integers.
    Output: The maximum element in A.
    if $n = 1$ then
        return $A[0]$
    return max{recursiveMax(A, $n - 1$), $A[n - 1]$}
```

$$T(n) = \begin{cases} 
3 & \text{if } n = 1 \\
T(n - 1) + 7 & \text{otherwise,}
\end{cases}$$
Constant Factors

- The growth rate is minimally affected by:
  - constant factors or
  - lower-order terms
- Examples
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function
Big-Oh Notation

- Given functions \( f(n) \) and \( g(n) \), we say that \( f(n) \) is \( O(g(n)) \) if there are positive constants \( c \) and \( n_0 \) such that
  \[ f(n) \leq cg(n) \text{ for } n \geq n_0 \]
- We also say \( g(n) \) is an asymptotic upper bound for \( f(n) \).

Example: \( 2n + 10 \) is \( O(n) \)

\[
\begin{align*}
2n + 10 & \leq cn \\
(c - 2) \cdot n & \geq 10 \\
n & \geq 10/(c - 2)
\end{align*}
\]
Pick \( c = 3 \) and \( n_0 = 10 \)
Relatives of Big-Oh

**big-Omega**
- $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

**big-Theta**
- $f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c'g(n) \leq f(n) \leq c''g(n)$ for $n \geq n_0$

**Theorem:** $\Theta$ is an equivalence relation.
(reflexive, symmetric, and transitive)
Intuition for Asymptotic Notation

big-Oh
- \( f(n) \) is \( O(g(n)) \) if \( f(n) \) is asymptotically less than or equal to \( g(n) \)

big-Omega
- \( f(n) \) is \( \Omega(g(n)) \) if \( f(n) \) is asymptotically greater than or equal to \( g(n) \)

big-Theta
- \( f(n) \) is \( \Theta(g(n)) \) if \( f(n) \) is asymptotically equal to \( g(n) \)
Example Uses of the Relatives of Big-Oh

- **5n^2** is \( \Omega(n^2) \)

  \( f(n) \) is \( \Omega(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \geq c g(n) \) for \( n \geq n_0 \)
  
  let \( c = 5 \) and \( n_0 = 1 \)

- **5n^2** is \( \Omega(n) \)

  \( f(n) \) is \( \Omega(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \geq c g(n) \) for \( n \geq n_0 \)
  
  let \( c = 1 \) and \( n_0 = 1 \)

- **5n^2** is \( \Theta(n^2) \)

  \( f(n) \) is \( \Theta(g(n)) \) if it is \( \Omega(n^2) \) and \( O(n^2) \). We have already seen the former, for the latter recall that \( f(n) \) is \( O(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \leq c g(n) \) for \( n \geq n_0 \)
  
  Let \( c = 5 \) and \( n_0 = 1 \)
Big-Oh, Big-Theta, Big Omega Rules

- If is \( f(n) \) a polynomial of degree \( d \), then \( f(n) \) is \( O(n^d) \), i.e.,
  1. Drop lower-order terms
  2. Drop constant factors

- Use the smallest possible class of functions
  - Say “2\(n\) is \( O(n) \)” instead of “2\(n\) is \( O(n^2) \)”

- Use the simplest expression of the class
  - Say “3\(n + 5\) is \( O(n) \)” instead of “3\(n + 5\) is \( O(n) \)”
Examples

**Θ(n^3):**
- \( n^3 \)
- \( 5n^3 + 4n \)
- \( 105n^3 + 4n^2 + 6n \)

**Θ(n^2):**
- \( n^2 \)
- \( 5n^2 + 4n + 6 \)
- \( n^2 + 5 \)

**Θ(log n):**
- \( \log n \)
- \( \log n^2 \)
- \( \log (n + n^3) \)
Little oh

\( f(n) \) grows slower than \( g(n) \) (or \( g(n) \) grows faster than \( f(n) \)) if

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0,
\]

Notation: \( f(n) = o(g(n)) \)

pronounced "little oh"
f(n) grows faster than g(n) (or g(n) grows slower than f(n)) if

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty, \]

Notation: \( f(n) = \omega(g(n)) \)
pronounced "little omega"
Relation Summary:

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = \begin{cases} \infty & \Rightarrow f(n) = \omega(g(n)) \\ C & \Rightarrow f(n) = \Theta(g(n)) \\ 0 & \Rightarrow f(n) = o(g(n)) \end{cases} \Rightarrow f(n) = \Omega(g(n)) \Rightarrow f(n) = O(g(n)) \]

Example: Decide the growth rate of \((\log n)^3n\) and \(n^{n/3}\)
A Case Study in Algorithm Analysis

- Given an array of \( n \) integers, find the subarray, \( A[j..k] \) that maximizes the sum

\[
s_{j,k} = a_j + a_{j+1} + \cdots + a_k = \sum_{i=j}^{k} a_i.
\]

- In addition to being an interview question for testing the thinking skills of job candidates, this maximum subarray problem also has applications in pattern analysis in digitized images.
A First (Slow) Solution

Compute the maximum of every possible subarray summation $A[j, k]$ of the array $A$ separately.

- The outer loop, for index $j$, will iterate $n$ times, its middle-inner loop, for index $k$, will iterate $j \sim n$ times, and the inner-most loop, for index $i$, will iterate $j \sim k$ times.
- Thus, the running time of the MaxsubSlow algorithm is $O(n^3)$.

**Algorithm** MaxsubSlow($A$):

*Input:* An $n$-element array $A$ of numbers, indexed from 1 to $n$.

*Output:* The maximum subarray sum of array $A$.

$m \leftarrow 0$ // the maximum found so far

for $j \leftarrow 1$ to $n$ do
    for $k \leftarrow j$ to $n$ do
        $s \leftarrow 0$ // the next partial sum we are computing
        for $i \leftarrow j$ to $k$ do
            $s \leftarrow s + A[i]$  
        if $s > m$ then
            $m \leftarrow s$
    
return $m$
An Improved Algorithm

- A more efficient way to calculate these summations is to consider **prefix sums**

\[ S_t = a_1 + a_2 + \cdots + a_t = \sum_{i=1}^{t} a_i \]

- If we are given all such prefix sums (and assuming \( S_0 = 0 \)), we can compute any summation \( s_{j,k} \) in constant time as

\[ s_{j,k} = S_k - S_{j-1} \]

Example:

<table>
<thead>
<tr>
<th>i =</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-2</td>
<td>-4</td>
<td>3</td>
<td>-1</td>
<td>5</td>
<td>6</td>
<td>-7</td>
<td>-2</td>
<td>4</td>
<td>-3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>0</td>
<td>-2</td>
<td>-6</td>
<td>-3</td>
<td>-4</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>-2</td>
<td>2</td>
<td>-1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\text{Max} = S_6 - S_2 = 7 - (-6) = 13.
\]
An Improved Algorithm, cont.

- Compute all the prefix sums -- $O(n)$
- Then compute all the subarray sums -- $O(n^2)$

**Algorithm** MaxsubFaster($A$):

**Input:** An $n$-element array $A$ of numbers, indexed from 1 to $n$.

**Output:** The maximum subarray sum of array $A$.

$S_0 \leftarrow 0$  // the initial prefix sum

for $i \leftarrow 1$ to $n$ do
  $S_i \leftarrow S_{i-1} + A[i]$

$m \leftarrow 0$  // the maximum found so far

for $j \leftarrow 1$ to $n$ do
  for $k \leftarrow j$ to $n$ do
    $s = S_k - S_{j-1}$
    if $s > m$ then
      $m \leftarrow s$

return $m$
A Linear-Time Algorithm

- Instead of computing prefix sum $S_t = s_{1,t}$, let us compute a maximum suffix sum, $M_t$, which is the maximum of any subarray (including the empty one) ending at $t$:

\[
M_t = \max\{0, \max_{j=1,\ldots,t} \{s_{j,t}\}\}
\]

- If $M_t > 0$, then it is the summation value for a maximum subarray that ends at $t$, and if $M_t = 0$, then we can safely ignore any subarray that ends at $t$.

- If we know all the $M_t$ values, for $t = 1, 2, \ldots, n$, then the solution to the maximum subarray problem would simply be the maximum of all these values.
A Linear-Time Algorithm, cont.

- If $t = 0$, then $M_t = 0$.
- For $t \geq 1$, to compute $M_t$, the maximum subarray that ends at $t$, we can add $A[t]$ to $M_{t-1}$. If the result is a positive sum, then we are done; if it is negative, we let $M_t$ be 0, i.e., take the empty subarray, for there is no non-empty subarray that ends at $t$ with a positive summation.
- So we can define $M_0 = 0$ and recursively

$$M_t = \max\{0, M_{t-1} + A[t]\}$$

Example:

<table>
<thead>
<tr>
<th>$t$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>-2</td>
<td>-4</td>
<td>3</td>
<td>-1</td>
<td>5</td>
<td>6</td>
<td>-7</td>
<td>-2</td>
<td>4</td>
<td>-3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$M$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>13</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

Max = $M_6 = 13$. 
The MaxsubFastest algorithm consists of two loops, which each iterate exactly \( n \) times and take \( O(1) \) time in each iteration. Thus, the total running time of the MaxsubFastest algorithm is \( O(n) \).
Math you need to Review

- Summations
- Powers
- Logarithms
- Proof techniques
- Basic probability

- Properties of powers:
  \[ a^{(b+c)} = a^b a^c \]
  \[ a^{bc} = (a^b)^c \]
  \[ a^b / a^c = a^{(b-c)} \]
  \[ b = a^{\log_a b} \]
  \[ b^c = a^{c \log_a b} \]

- Properties of logarithms:
  \[ \log_b(xy) = \log_b x + \log_b y \]
  \[ \log_b (x/y) = \log_b x - \log_b y \]
  \[ \log_b x^a = a \log_b x \]
  \[ \log_b a = \log_x a / \log_x b \]
Logarithms

\[ \log_b y = x \iff b^x = y \iff b^{\log_b y} = y \]

\[ \log nm = \log n + \log m \]

\[ \log \frac{n}{m} = \log n - \log m \]

\[ \log n^r = r \log n \]

\[ \log_a n = \frac{\log_b n}{\log_b a} \]
Summations

\[ \sum_{i=1}^{n} f(i) = f(1) + f(2) + \ldots + f(n-1) + f(n) \]

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

\[ \sum_{i=1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6} \]

\[ \sum_{i=1}^{n} i^k = \Theta(n^{k+1}) \]

\[ \sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1} \text{ for } a > 1 \]
Summations

\[ \sum_{i=1}^{n} \frac{1}{i} = O(\ln n) \]

using Integral of 1/x.

\[ \sum_{i=1}^{n} \log i = O(n \log n) \]

using Stirling’s approximation

\[ n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \]
The Factorial Function

**Definition:**

\[ n! = 1 \cdot 2 \cdot 3 \cdot L \cdot (n - 1) \cdot n \]

**Stirling’s approximation:**

\[ n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \]

or

\[ \log(n!) = O(n \log n) \]

**How to approve it?**
Bounds of Factorial Function

Let 

\[
\log n! = \sum_{x=1}^{n} \log x.
\]

then 

\[
\int_{1}^{n} \log x \, dx \leq \sum_{x=1}^{n} \log x \leq \int_{0}^{n} \log(x + 1) \, dx
\]

which gives 

\[
n \log \left( \frac{n}{e} \right) + 1 \leq \log n! \leq (n + 1) \log \left( \frac{n + 1}{e} \right) + 1.
\]

so 

\[
e \left( \frac{n}{e} \right)^n \leq n! \leq e \left( \frac{n + 1}{e} \right)^{n+1}.
\]

\[
n! \approx \sqrt{2\pi n} \left( \frac{n}{e} \right)^n.
\]
Average Case Analysis

- In worst case analysis of time complexity we select the maximum cost among all possible inputs of size n.
- In average case analysis, the running time is taken to be the average time over all inputs of size n.
  - Unfortunately, there are infinite inputs.
  - It is necessary to know the probabilities of all input occurrences.
  - The analysis is in many cases complex and lengthy.
What is the average case of executing \( \text{currentMax} \leftarrow A[i] \)?

**Algorithm** `arrayMax(A, n)`:

- **Input:** An array \( A \) storing \( n \geq 1 \) integers.
- **Output:** The maximum element in \( A \).

\[
\text{currentMax} \leftarrow A[0]
\]

for \( i \leftarrow 1 \) to \( n - 1 \) do

if \( \text{currentMax} < A[i] \) then

\[
\text{currentMax} \leftarrow A[i]
\]

return \( \text{currentMax} \)

**Number of Assignments:** the worst case is \( n \). If numbers are randomly distributed, then the average case is \( 1 + 1/2 + 1/3 + 1/4 + \ldots + 1/n = O(\log n) \).

This is because \( A[i] \) has only \( 1/i \) probability to be the max of \( A[1], A[2], \ldots, A[i] \), under the assumption that all numbers are randomly distributed.
Amortized Analysis

- The **amortized running time** of an operation within a series of operations is the worst-case running time of the series of operations divided by the number of operations.

- Example: A growable array, $S$. When needing to grow:
  a. Allocate a new array $B$ of larger capacity.
  b. Copy $A[i]$ to $B[i]$, for $i = 0, \ldots, n - 1$, where $n$ is size of $A$.
  c. Let $A = B$, that is, we use $B$ as the array now supporting $A$. 

![Diagram](image)
Dynamic Array Description

- Let $\text{add}(e)$ be the operation that adds element $e$ at the end of the array.
- When the array is full, we replace the array with a larger one.
- But how large should the new array be?
  - **Incremental strategy**: increase the size by a constant $c$.
  - **Doubling strategy**: double the size.

Algorithm $\text{add}(e)$

```plaintext
if $n = A.length$ then
    $B \leftarrow$ new array of size ...
    for $i \leftarrow 0$ to $n-1$ do
        $B[i] \leftarrow A[i]$
    $A \leftarrow B$
    $n \leftarrow n + 1$
    $A[n-1] \leftarrow e$
```

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of $n$ add operations.
- We assume that we start with an empty list represented by a growable array of size 1.
- We call amortized time of an add operation the average time taken by an add operation over the series of operations, i.e., $T(n)/n$. 
Incremental Strategy Analysis

- Over $n$ add operations, we replace the array $k = n/c$ times, where $c$ is a constant.
- The total time $T(n)$ of a series of $n$ add operations is proportional to
  
  \[ n + c + 2c + 3c + 4c + \ldots + kc = \]
  
  \[ n + c(1 + 2 + 3 + \ldots + k) = \]
  
  \[ n + ck(k + 1)/2 \]

- Since $c$ is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$.
- Thus, the amortized time of an add operation is $O(n)$.
Doubling Strategy Analysis:
The Aggregate Method

- We replace the array $k = \log_2 n$ times.
- The total time $T(n)$ of a series of $n$ push operations is proportional to

\[
n + 1 + 2 + 4 + 8 + \ldots + 2^k =
\]

\[
n + 2^{k+1} - 1 = 3n - 1
\]

- $T(n)$ is $O(n)$.
- The amortized time of an add operation is $O(1)$.
Doubling Strategy Analysis: The Accounting Method

- We view the computer as a coin-operated appliance that requires one cyber-dollar for a constant amount of computing time.
- For this example, we shall pay each add operation 3 cyber-dollars.
  - Set a saving account with $s_0 = 0$ initially.
  - The $i^{th}$ operation has a budget cost of $a_i = 3$, which is the amortized cost of each operation.
  - The account value after the $i^{th}$ add operation is
    \[ s_i = s_{i-1} + a_i - c_i \]

| i  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Array size</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| $c_i$ | 1 | 2 | 3 | 1 | 5 | 1 | 1 | 1 | 9 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 17 | 1 | 1 | 1 | ...
| $s_i$ | 2 | 3 | 3 | 5 | 3 | 5 | 7 | 9 | 3 | 5 | 7 | 9 | 11 | 15 | 17 | 19 | 5 | 7 | 9 | 11 | ...


Doubling Strategy Analysis: The Accounting Method

- We shall pay each add operation 3 cyber-dollars, that is, it will have an amortized $O(1)$ amortized running time.
  - We over-pay each add operation not causing an overflow 2 cyber-dollars.
  - An overflow occurs when the array $A$ has $2^i$ elements.
  - Thus, doubling the size of the array will require $2^i$ cyber-dollars.
  - These cyber-dollars are at the elements stored in cells $2^{i-1}$ through $2^i-1$. 

![Diagram](image)
Summary

- **Worst-case complexity**: given an upper bound at the worst case.

- **Average complexity**: Assume a probability distribution of all inputs, give the complexity under this distribution.

- **Amortized complexity**: Compute the worst case of the sum of a sequence of operations, and then divide it by the number of operations.