Discrete Structures: Algorithms

Amotz Bar-Noy

Department of Computer and Information Science
Brooklyn College
Outline

1. Introduction
3. Growth of Functions
4. The Prefix-Sum Problem
5. Dictionary Search
6. Sorting
7. Array Problems
Algorithm: Definitions

- 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, Process, Recipe, Routine, Solution, Technique, Mechanism, Scheme, Way, Design, Plan, Strategy, Construction, ...

Etymology

- Alteration of Middle English *algorisme*, from Old French & Medieval Latin; 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 do we need to solve problems?

Answer

Apply some combination of these five attributes!!!
The Ultimate Algorithmic Problem!?

**Question**
- What do we need to solve problems?

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

Apply some combination of these five attributes!!!
The Ultimate Algorithmic Problem!?

Question

What do we need to solve problems?

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.
Algorithms: Online Videos

- https://www.youtube.com/watch?v=Da5TOXCwLSg
- https://www.youtube.com/watch?v=6hfOvs8pY1k
- https://www.youtube.com/watch?v=CvSOaYi89B4&feature=youtu.be
- https://study.com/academy/lesson/
  what-is-an-algorithm-in-programming-definition-examples-analysis.html
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.
Introduction

The Babylonian Multiplication Algorithm

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 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.
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

- $T(n)$ is a worst case complexity:
  - If for all inputs of size $n$ the complexity is $T(n)$.

Average case

- $T(n)$ 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).
## Bounds

### Upper Bound
- A function $f(n)$ such that $T(n) \leq f(n)$ for all $n$.

### Lower bound
- A function $g(n)$ such that $T(n) \geq g(n)$ for all $n$.

### Tight bound
- A function $h(n)$ such that $T(n) \approx h(n)$ 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.
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

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>$400n$</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-o**

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

**Little-omega**

$f(n) = \omega(g(n))$ if $f(n)$ asymptotically strictly greater than $g(n)$.
**Growth of Functions**

### 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 \( c', c'' > 0 \) and an integer constant \( n_0 > 0 \) such that \( c''g(n) \leq f(n) \leq c'g(n) \) for every integer \( n \geq n_0 \).
### 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>
$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$

**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 $O$, $f(n) = O(g(n))$. 
**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:** \( (f(n) = \Theta(g(n))) \iff (g(n) = \Theta(f(n))) \).
- **Transitive:** \( f(n) = \Theta(g(n)) \land g(n) = \Theta(h(n)) \Rightarrow (f(n) = \Theta(h(n))) \).

\( O, \Omega \), are reflexive relations

- \( f(n) = O(f(n)) \).
- \( f(n) = \Omega(f(n)) \).

\( O, \Omega \), 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 \( g(n) = \Omega(f(n)) \).

\( O, \Omega \), 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 $n \geq n_1 > (1/c)$.
- 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

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

For $\Psi \in \{O, \Omega, \Theta\}$:

- $\Psi(f(n)) + \Psi(g(n)) = \Psi(f(n) + g(n))$.
- $\Psi(f(n)) \cdot \Psi(g(n)) = \Psi(f(n) \cdot g(n))$.
- $\Psi(f(n)) + \Psi(g(n)) = \Psi(\max \{f(n), g(n)\})$. 
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 $n \geq n_0$.
Propositions

**o and ω**
- \( f(n) = o(g(n)) \iff g(n) = \omega(f(n)) \).
- \( f(n) = o(g(n)) \land g(n) = o(h(n)) \rightarrow f(n) = o(h(n)) \).
- \( f(n) = \omega(g(n)) \land g(n) = \omega(h(n)) \rightarrow f(n) = \omega(h(n)) \).

**o vs. O**
- \( f(n) = o(g(n)) \rightarrow f(n) = O(g(n)) \).
- \( f(n) = O(g(n)) \nRightarrow f(n) = o(g(n)) \).

**ω vs Ω**
- \( f(n) = \omega(g(n)) \rightarrow f(n) = \Omega(g(n)) \).
- \( f(n) = \Omega(g(n)) \nRightarrow f(n) = \omega(g(n)) \).
Examples

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

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Example</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>$n^\varepsilon$</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>

Where $\varepsilon$ is a constant $0 < \varepsilon < 1$ and $k$ is a constant integer $k > 1$. The table lists various growth rates of functions, from the slowest (constant) to the fastest (exponential tower).
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
- \( A = [13, 34, -8, -55, -5, 21, \ldots] \)
- \( S = [13, 47, 39, -16, -21, 0 \ldots] \)
The Prefix-Sum Problem

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.
The Prefix-Sum Problem

By Definition Algorithm

Complexity

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

$\Theta(n) + \Theta(n^2) = \Theta(n^2)$ overall complexity.
The Prefix-Sum Problem

**By Induction Algorithm**

**Algorithm**

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

**Correctness**

- By Induction.
By Induction Algorithm

Complexity

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

Input
- Real numbers $a_0, a_1, \ldots, a_n$ such that $a_n \neq 0$ 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
- $a_4 = 5$, $a_3 = 0$, $a_2 = -7$, $a_1 = 3$, $a_0 = -11$, 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.
**The Prefix-Sum Problem**

**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

\[ a = a \cdot c \quad (\ast \ a = a_i c^i \ast) \]

\[ P(c) = P(c) + a \quad (\ast \ P(c) = a_i c^i + a_{i-1} c^{i-1} + \cdots + a_1 c + a_0 \ast) \]

return \( P(c) \)  

(\ast \ P(c) = a_n c^n + a_{n-1} c^{n-1} + \cdots + a_1 c + a_0 \ast)

**Correctness**

- By definition.
The Prefix-Sum Problem

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 of $\frac{1}{2}n^2 + \frac{3}{2}n$ operations.
- $\Theta(n^2)$ 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

\textbf{Polynomial-Evaluation}($P(x), c$)

\begin{align*}
P(c) &= a_0 \\ cc &= 1 \\
\text{for } i = 1 \text{ to } n \text{ do} \\
&\hspace{1cm} cc = cc \cdot c \quad (\ast \text{ cc } = c^i \ast) \\
&\hspace{1cm} P(c) = P(c) + a_i \cdot cc \quad (\ast \text{ P(c) } = a_i c^i + a_{i-1} c^{i-1} + \cdots + a_1 c + a_0 \ast) \\
\text{return} (P(c)) \quad (\ast \text{ P(c) } = a_n c^n + a_{n-1} c^{n-1} + \cdots + a_1 c + a_0 \ast)
\end{align*}

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 Prefix-Sum Problem

The Horner’s Algorithm

**Idea**

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

**Example**

\[ 4x^3 + 3x^2 + 2x + 1 = ((4x + 3)x + 2)x + 1. \]
The Prefix-Sum Problem

The Horner’s Algorithm

Algorithm

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

\[
P(c) = a_n
\]

for \(i = n - 1\) downto 0 do

\[
P(c) = P(c) \cdot c + a_i
\]

\((* \ P(c) = a_n c^{n−i} + a_{n−1} c^{n−i−1} + \cdots + a_{i+1} c + a_i *)\)

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

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

Running the algorithm

- $P_4(x) = a_4 = 5$
- $P_3(x) = P_4(x) \cdot c + a_3 = 5 \cdot 2 + 0 = 10$
- $P_2(x) = P_3(x) \cdot c + a_2 = 10 \cdot 2 - 7 = 13$
- $P_1(x) = P_2(x) \cdot c + a_1 = 13 \cdot 2 + 3 = 29$
- $P(x) = P_1(x) \cdot c + a_0 = 29 \cdot 2 - 11 = 47$
The Prefix-Sum Problem

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.
Dictioary Search

A Dictionary Search Problem

**Input**

- A key $K$.

**Output**

- Does $K$ appear in $A$? **YES** or **NO**.
- If **YES**: The first index $i$ such that $A[i] = K$.
- If **NO**: The largest index $i$ such that $A[i] < K$ or $i = 0$ if $K < A[1]$.

**Method**

- **Comparisons** between $K$ and the keys in the array.


**Complexity**

- Number of **comparisons**.
A Search Game

Game

- **Player 1:** Selects an integer \( x \) in the range \([1..n]\).
- **Player 2:** Searches for \( x \) with **comparisons** \( x \leq i \) for some \( 1 \leq i \leq n \).

Player 2 Goal

- Minimize the number of **comparisons** until finding \( x \).
  - In the worst case or in the average case.
  - As a function of \( n \).
The Two Models are “Equivalent”

- $x \leq i$ is “equivalent” to $K \leq A[i]$.
- Algorithms can be “converted” from one model to another while preserving the complexity.
- It is easier to design algorithms in the search game model.
- It is easier to prove complexity bounds in the search game model.
Sequential Search

Sequential-Search \((n,x)\)

\[
i = 0
\]

repeat

\[
i = i + 1
\]

until \(x \leq i\)  (* comparison *)

return \(i\)
Sequential Search – Correctness

**Induction hypothesis**

- $i \leq x \leq n$ after $i - 1$ comparisons with a NO answer.

**Termination**

- If $x \leq i$ then necessarily $x = i$.
- Eventually $x \leq n$. 
Sequential Search – Complexity

- **n comparisons** in the worst case when \( x = n \).
  - Possible \( n - 1 \) comparisons since there is no need for the last question when \( x = n \).

- Could be only 1 comparison when \( x = 1 \).

- \( (n + 1)/2 \) comparisons on average for a random \( x \) selected with a uniform distribution from the range \([1..n]\):

\[
\frac{1}{n} \left(1 + 2 + \cdots + n\right) = \frac{1}{n} \cdot \frac{n(n + 1)}{2} = \frac{n + 1}{2}
\]
Binary Search

**Binary-Search** \((n,x)\)

\(\ell = 1\)
\(u = n\)

while \(\ell < u\)

\[ m = \left\lfloor \frac{u + \ell}{2} \right\rfloor \]

if \(x \leq m\) (* comparison *)

then \(u = m\)

else \(\ell = m + 1\)

return \(\ell\)
Binary Search

**Notations**
- Let $u_j$ and $\ell_j$ be the values of $u$ and $\ell$ after iteration $j$ of the algorithm and let $\Delta_j = u_j - \ell_j + 1$.
- Initially $u_0 = n$, $\ell_0 = 1$, and $\Delta_0 = n$.

**Observation**
- $\Delta_{j+1} \leq \left\lceil \frac{\Delta_j}{2} \right\rceil$ for $j \geq 0$.

**Corollary**
- $\Delta_k = 1$ for $k = \lceil \log_2 n \rceil$. 
Binary Search – Correctness and Complexity

**Correctness**
- By induction, always \( \ell \leq x \leq u \).
- At the end, \( \ell = u \) and therefore \( x = \ell = u \).

**Complexity**
- There are at most \( \lceil \log_2 n \rceil \) iterations and one comparison per iteration.
- Therefore, the **worst-case** complexity is \( \lceil \log_2 n \rceil \).
- If \( n \) is not a power of 2, then for some \( x \) there are only \( \lfloor \log_2 n \rfloor \) iterations.
- Therefore, the **average-case** complexity is approximately \( \log_2 n \).
## Binary-Search vs. Sequential-Search

<table>
<thead>
<tr>
<th></th>
<th>Binary-Search</th>
<th>Sequential-Search</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Best-Case</strong></td>
<td>$\lceil \log_2 n \rceil$</td>
<td>1</td>
</tr>
<tr>
<td><strong>Worst-Case</strong></td>
<td>$\lceil \log_2 n \rceil$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td><strong>Average-Case</strong></td>
<td>$\approx \log_2 n$</td>
<td>$\approx \frac{n+1}{2}$</td>
</tr>
</tbody>
</table>
Adversary Player I

**Goal**
- Maximize the number of *comparisons* until Player 1 finds $x$.

**Strategy**
- **Player 1 Does not** select $x$ at the beginning of the game. Instead, it maintains a set of candidates $S$ for $x$.
- Given a search question:
  - $S(Y)$ – the set of candidates if the answer is *YES*.
  - $S(N)$ – the set of candidates if the answer is *NO*.
- The adversary answer rule:
  - *YES* if $|S(Y)| \geq |S(N)|$.
  - *NO* if $|S(Y)| < |S(N)|$. 
Example

**Arbitrary algorithm**

- **Input:** \( n = 34 \) \((* x \in [1..34] *)\).
- **Search:**
  - Q1: \( x \leq 13 \) \(\Rightarrow\) A1: NO \((* x \in [14..34] *)\).
  - Q2: \( x \leq 26 \) \(\Rightarrow\) A2: YES. \((* x \in [14..26] *)\).
  - Q3: \( x \leq 18 \) \(\Rightarrow\) A3: NO. \((* x \in [19..26] *)\).
  - Q4: \( x \leq 23 \) \(\Rightarrow\) A4: YES. \((* x \in [19..23] *)\).
  - Q5: \( x \leq 20 \) \(\Rightarrow\) A5: NO. \((* x \in [21..23] *)\).
  - Q6: \( x \leq 22 \) \(\Rightarrow\) A6: YES. \((* x \in [21..22] *)\).
  - Q7: \( x \leq 21 \) \(\Rightarrow\) A7: YES. \((* x \in [21..21] *)\).
- **Output:** \( x = 21 \).
Example

Binary-Search

- **Input:** \( n = 34 \) \((* x \in [1..34] *)\).
- **Search:**
  - Q1: \( x \leq 17 \) ⇒ A1: YES \((* x \in [1..17] *)\).
  - Q2: \( x \leq 9 \) ⇒ A2: YES. \((* x \in [1..9] *)\).
  - Q3: \( x \leq 5 \) ⇒ A3: YES. \((* x \in [1..5] *)\).
  - Q4: \( x \leq 3 \) ⇒ A4: YES. \((* x \in [1..3] *)\).
  - Q5: \( x \leq 2 \) ⇒ A5: YES. \((* x \in [1..2] *)\).
  - Q6: \( x \leq 11 \) ⇒ A5: YES. \((* x \in [1..1] *)\).
- **Output:** \( x = 1 \).

Observation

- With Binary-Search the adversary always ends up with \( x = 1 \).
Theorem

There exists $1 \leq x \leq n$ for which the adversary forces the second player to ask at least $\lceil \log_2 n \rceil$ comparisons.

Proof

Assume that Player 2 asks $k$ comparisons to find $x$.
Let $S_i$ be the set of candidates after $i$ comparisons.
In particular, $|S_0| = n$ and $|S_k| = 1$.
$S = S(Y) \cup S(N)$ implies that $|S_{i+1}|/|S_i| \geq (1/2)$ for $1 \leq i \leq k - 1$.
$\lceil \log_2 n \rceil$ rounds are required to decrease $n$ to 1 by halving.
Therefore, $k \geq \lceil \log_2 n \rceil$. 
Remarks

- This is a **worst case** bound implying that no algorithm can guarantee less comparisons for all values of $x$.

- The theorem holds for a “**stronger**” Player 2. One that may ask any **YES/NO** questions. For example,
  - Is $x$ even?
  - Is $x \in \{1, 2, 3, 5, 8, 13, 21, 34, 55\}$?
Searching with “Clues”

Clue

- **Player 1** selects only odd numbers $1, 3, \ldots$ between 1 and $n$.

A modified Binary Search

- The search domain is $1, 2, \ldots, \lceil n/2 \rceil$.
- **Player 2**: Instead of asking “if $x \leq i$” asks “if $x \leq 2i - 1$”.
- When the search outputs $x = i$ the modified search outputs $2i - 1$.

Complexity

- $\lceil \log_2(\lceil n/2 \rceil) \rceil \approx \log_2(n/2) = \log_2(n) - 1$ comparisons.
- The saving is only 1 comparison although the clue “eliminated” about half of the candidates!
Search with “Clues”

**Clue**
- **Player 1** selects only odd numbers 1, 3, … between 1 and \( n \).

**Example**
- \( n = 31 = 2 \cdot 16 - 1 \) and \( x = 19 = 2 \cdot 10 - 1 \)
- The possible 16 values for \( x \) are 1, 3, 5, …, 31 and the search domain is 1, 2, …, 16.

**Running the algorithm**
- **Question 1:** \( x \leq (2 \cdot 8 - 1 = 15) \) because 8 = \( \left\lfloor (1 + 16)/2 \right\rfloor \).
- **Question 2:** \( x \leq (2 \cdot 12 - 1 = 23) \) because 12 = \( \left\lfloor (9 + 16)/2 \right\rfloor \).
- **Question 3:** \( x \leq (2 \cdot 10 - 1 = 19) \) because 10 = \( \left\lfloor (9 + 12)/2 \right\rfloor \).
- **Question 4:** \( x \leq (2 \cdot 9 - 1 = 17) \) because 9 = \( \left\lfloor (9 + 10)/2 \right\rfloor \).
- \( x = 21 \) found with \( 4 = \log_2 16 \approx \log_2 31 - 1 \) comparisons.
Searching with “Clues”

Clue
- **Player 1** selects only square numbers 1, 4, 9, … between 1 and $n$.

A modified Binary Search
- The search domain is $1, 2, \ldots, \lfloor \sqrt{n} \rfloor$.
- **Player 2**: Instead of asking “if $x \leq i$” asks “if $x \leq i^2$”.
- When the search outputs $x = i$ the modified search outputs $i^2$.

Complexity
- $\lceil \log_2(\lfloor \sqrt{n} \rfloor) \rceil \approx \log_2(\sqrt{n}) = \frac{1}{2} \log_2(n)$ comparisons.
- The saving is only half of the comparisons although the clue “eliminated” almost all the candidates!
Searching with “Clues”

**Clue**
- **Player 1** selects only square numbers 1, 4, 9, \ldots between 1 and \( n \).

**Example**
- \( n = 256 = 16^2 \) and \( x = 100 = 10^2 \)
- The possible 16 values for \( x \) are 1, 4, 9, \ldots, 256 and the search domain is 1, 2, \ldots, 16.

**Running the algorithm**
- Question 1: \( x \leq (8^2 = 64) \)? because \( 8 = \lceil (1 + 16)/2 \rceil \).
- Question 2: \( x \leq (12^2 = 144) \)? because \( 12 = \lceil (9 + 16)/2 \rceil \).
- Question 3: \( x \leq (10^2 = 100?) \) because \( 10 = \lceil (9 + 12)/10 \rceil \).
- Question 4: \( x \leq (9^2 = 81?) \) because \( 9 = \lceil (9 + 10)/10 \rceil \).
- \( x = 100 \) found with \( 4 = \log_2 16 = (1/2) \log_2 256 \) comparisons.
Searching with “Clues”

Clue

- **Player 1** selects only power of 2 numbers 1, 2, 4, 8, … between 1 and $n$.

A modified Binary Search

- The search domain is $1, 2, \ldots, \lfloor \log_2 n + 1 \rfloor$.
- **Player 2**: Instead of asking “if $x \leq i$” asks “if $x \leq 2^{i-1}$”.
- When the search outputs $x = i$ the modified search outputs $2^{i-1}$.

Complexity

- $\lceil \log_2(\lfloor \log_2(n) \rfloor + 1) \rceil \approx \log_2(\log_2(n))$ comparisons.
- For $n = 1000000$ the saving is only from 20 to 5 comparisons although there are only 20 candidates!
Searching with “Clues”

**Clue**

- **Player 1** selects only power of 2 numbers 1, 2, 4, 8, \ldots between 1 and \( n \).

**Example**

- \( n = 32768 = 2^{15} \) and \( x = 512 = 2^9 \)
- The possible 16 values for \( x \) are 1, 2, 4, \ldots, 32768 and the search domain is 1, 2, \ldots, 16.

**Running the algorithm**

- **Question 1:** \( x \leq (2^8 - 1 = 128)? \) \( 8 = \lceil (1 + 16)/2 \rceil \).
- **Question 2:** \( x \leq (2^{12} - 1 = 2048)? \) \( 12 = \lceil (9 + 16)/2 \rceil \).
- **Question 3:** \( x \leq (2^{10} - 1 = 512)? \) \( 10 = \lceil (9 + 12)/10 \rceil \).
- **Question 4:** \( x \leq (2^9 - 1 = 256)? \) \( 9 = \lceil (9 + 10)/10 \rceil \).
- \( x = 512 \) found with \( 4 = \log_2 16 \approx \log_2 \log_2 32768 \) comparisons
Searching an Unbounded Domain

Game
- **Player 1**: Selects any positive integer $x$.
- **Player 2**: Searches for $x$ with comparisons $x \leq i$ for some integer $i$.

Adversary Player 1
- Always answers **NO**.
- **Player 2** will never find $x$!

Player 2 Goal
- Find $x$ with as minimum possible comparisons as a function of $x$.
- Ask "few" comparisons when $x$ is small and ask "more" comparisons when $x$ is large.
Searching an Unbounded Domain

Sequential search
- Sequential search finds \( x \) with exactly \( x \) comparisons.

The doubling technique
- A strategy that finds \( x \) with approximately \( 2 \log_2(x) \) comparisons.

A more sophisticated doubling technique
- A strategy that finds \( x \) with approximately \( \log_2(x) + 2 \log_2 \log_2(x) \) comparisons.

Optimal solution
- A strategy that finds \( x \) with approximately

\[
\log_2(x) + \log_2 \log_2(x) + \log_2 \log_2 \log_2(x) + \cdots
\]

comparisons.
The Doubling Technique

Strategy

Phase 1: Ask the following comparisons until the answer is YES:

\[ x \leq 1? \quad x \leq 2? \quad x \leq 4? \quad x \leq 8? \quad \cdots \quad x \leq 2^j? \quad \cdots \]

Assume \( 2^{k-1} < x \leq 2^k \)

Phase 2: Apply binary search on the domain

\[ 2^{k-1} + 1, 2^{k-1} + 2, \ldots, 2^k \]

Complexity

- \( k + 1 \) comparisons are asked in Phase 1.
- The number of comparisons asked in Phase 2 is
  \[ \log_2(2^k - (2^{k-1} + 1) + 1) = \log_2(2^{k-1}) = k - 1 \]
- Total number of comparisons:
  \[ (k + 1) + (k - 1) = 2k = 2 \left \lceil \log_2(x) \right \rceil \]
The Sorting Problem

**Keys**
- Entities from a **well ordered** domain.

**Comparisons**
- Between two keys $K_1$ and $K_2$
  
  $K_1 < K_2$? $K_1 \leq K_2$? $K_1 = K_2$?

**Input**

**Output**

**Goal**
- Minimize number of comparisons between keys.
Complexity of the Sorting Problem

**Lower bound**
- $\Omega(n \log n)$ comparisons are required by any algorithm.

**Upper bounds**
- $O(n^2)$ comparisons with “simple” algorithms.
- $O(n \log n)$ comparisons with more “sophisticated” algorithms.

**Tight bound**
- $\Theta(n \log n)$ overall complexity.

**Models**
- Bounds are for both worst case and average case complexity.
Some Sorting Algorithms

Simple algorithms
- **Bubble-Sort**: $\Theta(n^2)$ worst & average case.
- **Insertion-Sort**: $\Theta(n^2)$ worst & average case.

Efficient deterministic sorting algorithms
- **Merge-Sort**: $\Theta(n \log n)$ worst & average case.
- **Heap-Sort**: $\Theta(n \log n)$ worst & average case.
- **Balanced-Tree-Sort**: $\Theta(n \log n)$ worst & average case.

Efficient randomized sorting algorithms
- **Quick-Sort**: $\Theta(n \log n)$ average case; $\Theta(n^2)$ worst case.
- **Binary-Tree-Sort**: $\Theta(n \log n)$ average case; $\Theta(n^2)$ worst case.
Bubble Sort

Input

Ideas
- Find the minimum $n - 1$ times.
- Compare and exchange only adjacent keys.

Implementation
- **Bubble-Sort**($A[1], \ldots, A[n]$)
  
  for $i = 1$ to $n - 1$
  
  for $j = n$ downto $i + 1$
  
  
  then $A[j] \leftrightarrow A[j - 1]$
Example

Initial array

[34, 89, 13, 55, 21]

Round 1

21 < 55  \implies  [34, 89, 13, 21, 55]
13 < 21  \implies  [34, 89, 13, 21, 55]
13 < 89  \implies  [34, 13, 21, 89, 55]
13 < 34  \implies  [13, 34, 89, 21, 55]

Round 2

21 < 55  \implies  [13, 34, 89, 21, 55]
21 < 89  \implies  [13, 34, 21, 89, 55]
21 < 34  \implies  [13, 21, 34, 89, 55]

Round 3

55 < 89  \implies  [13, 21, 34, 55, 89]
34 < 55  \implies  [13, 21, 34, 55, 89]

Round 4

55 < 89  \implies  [13, 21, 34, 55, 89]

Sorted array

[13, 21, 34, 55, 89]
Example

- **Initial array:** [8, 21, 1, 3, 2, 13, 5].
- **After round 1:** [1, 8, 21, 2, 3, 5, 13]
- **After round 2:** [1, 2, 8, 21, 3, 5, 13]
- **After round 3:** [1, 2, 3, 8, 21, 5, 13]
- **After round 4:** [1, 2, 3, 5, 8, 21, 13]
- **After round 5:** [1, 2, 3, 5, 8, 13, 21]
- **Sorted array after round 6:** [1, 2, 3, 5, 8, 21, 13]
Correctness

- By induction, for $1 \leq i \leq n - 1$, after round $i$:
  - $A[i] \leq A[j]$ for all $i < j \leq n$


Complexity

- For $1 \leq i \leq n - 1$, in round $i$: exactly $n - i$ comparisons.
- The total number of comparisons is always

$$
(n - 1) + (n - 2) + \cdots + 1 = \frac{n(n - 1)}{2} = \Theta(n^2)
$$
Merge-Sort

Input

- An unsorted array of \( n \) keys \( A[1], A[2], \ldots, A[n] \).

Divide and Conquer

- For \( n \geq 2 \) and \( q = \left\lfloor \frac{n+1}{2} \right\rfloor \) recursively sort the sub-arrays \( A[1..q] \) and \( A[q+1..n] \).
- **Merge** the sub-arrays \( A[1..q] \) and \( A[q+1..n] \) into a **sorted** array \( A[1..n] \).
The Merge Procedure

Global array

- $A[1], A[2], \ldots, A[n]$

Procedure

- **Merge** $(p, q, r)$
  - $1 \leq p \leq q < r \leq n$.
  - Merge the two sorted sub-arrays $A[p] \leq \cdots \leq A[q]$ and $A[q + 1] \leq \cdots \leq A[r]$ into a sorted sub-array $A[p] \leq \cdots \leq A[r]$.

Complexity

- Number of comparisons is at most $(r - p)$.

Implementation

- [https://www.tutorialspoint.com/data_structures_algorithms/merge_sort_algorithm.htm](https://www.tutorialspoint.com/data_structures_algorithms/merge_sort_algorithm.htm)
The Recursive Merge-Sort Procedure

**Initial recursive call**
- Merge-Sort(1, n).

**Recursive procedure**
- Merge-Sort(p, r)
  - if \(r > p\) then
  - \(q = \left\lfloor \frac{p+r}{2} \right\rfloor\) \((^* p \leq q < q + 1 \leq r ^*)\)
  - Merge-Sort(p, q)
  - Merge-Sort(q + 1, r)
  - Merge(p, q, r)
Merge-Sort – Correctness

- Proof by induction on $r - p$.
- Case $r = p$ the array is sorted trivially.
- Case $p \leq q < r$, the induction hypothesis holds:
  - For sub-array $A[q + 1..r]$ since $r - (q + 1) < r - p$.
- The inductive step is correct due to the correctness of procedure Merge.
MergeSort – Complexity

Notation
- \( T(n) \) - upper bound on the number of comparisons for an array with \( n \) keys.

Recursive formula
- \( T(1) = 0 \)
- \( T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + (n - 1) \)

Solution
- \( T(n) = \Theta(n \log(n)) \)
Merge-Sort – Complexity for $n = 2^k$

**Bottom-Up evaluation**

<table>
<thead>
<tr>
<th>$T(1)$</th>
<th>$=$</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(2)$</td>
<td>$\leq$</td>
<td>$2 \cdot T(1) + (2 - 1)$</td>
</tr>
<tr>
<td>$T(4)$</td>
<td>$\leq$</td>
<td>$2 \cdot T(2) + (4 - 1)$</td>
</tr>
<tr>
<td>$T(8)$</td>
<td>$\leq$</td>
<td>$2 \cdot T(4) + (8 - 1)$</td>
</tr>
<tr>
<td>$T(16)$</td>
<td>$\leq$</td>
<td>$2 \cdot T(8) + (16 - 1)$</td>
</tr>
<tr>
<td>$T(32)$</td>
<td>$\leq$</td>
<td>$2 \cdot T(16) + (32 - 1)$</td>
</tr>
</tbody>
</table>

**Guess**

$T(n) \leq n \log_2 n - (n - 1)$. 
Guessing by Unfolding the Recursion

Top-Down evaluation

\[ T(2^k) \leq 2T(2^{k-1}) + (2^k - 1) \]
\[ = 2T(2^{k-1}) + (1 \cdot 2^k - 1) \]
\[ \leq 2(2T(2^{k-2}) + (2^{k-1} - 1)) + (2^k - 1) \]
\[ = 4T(2^{k-2}) + (2 \cdot 2^k - 3) \]
\[ \leq 4(2T(2^{k-3}) + (2^{k-2} - 1)) + (2 \cdot 2^k - 3) \]
\[ = 8T(2^{k-3}) + (3 \cdot 2^k - 7) \]
\[ \vdots \]
\[ = 2^i T(2^{k-i}) + (i \cdot 2^k - (2^i - 1)) \]
\[ \vdots \]
\[ = 2^k T(2^0) + (k \cdot 2^k - (2^k - 1)) \]
\[ = n \log_2 n - (n - 1) \]
Proof By Induction for $n = 2^k$

**Theorem**

$T(n) \leq n\log_2 n - (n - 1)$

**Induction base**

$n = 1$: $T(1) \leq 0 = 1 \cdot 1 - 0 = 1\log_2 1 - (1 - 1)$

**Induction hypothesis**

\[
T(n/2) \leq (n/2)\log_2(n/2) - (n/2 - 1)
\]
\[
= (n/2)(\log_2 n - 1) - (n/2 - 1)
\]
\[
= (n/2)\log_2 n - (n - 1)
\]

**Inductive step**

\[
T(n) \leq 2T(n/2) + (n - 1)
\]
\[
\leq 2((n/2)\log_2 n - (n - 1)) + (n - 1)
\]
\[
= n\log_2 n - (n - 1).
\]
Merge-Sort – Complexity for $n \neq 2^k$

**Bottom-Up evaluation**

\[
\begin{align*}
T(1) &= 0 \\
T(2) &\leq T(1) + T(1) + (2 - 1) = 1 \\
T(3) &\leq T(2) + T(1) + (3 - 1) = 3 \\
T(4) &\leq T(2) + T(2) + (4 - 1) = 5 \\
T(5) &\leq T(3) + T(2) + (5 - 1) = 8 \\
T(6) &\leq T(3) + T(3) + (6 - 1) = 11 \\
T(7) &\leq T(4) + T(3) + (7 - 1) = 14 \\
T(8) &\leq T(4) + T(4) + (8 - 1) = 17 \\
T(9) &\leq T(5) + T(4) + (9 - 1) = 21
\end{align*}
\]
Merge-Sort – Complexity for $n \neq 2^k$

**Guess**

- $T(n) \leq n \lfloor \log_2 n \rfloor - (2^{\lfloor \log_2 n \rfloor} - 1)$

**Verification**

- $T(1) = 0 = 1 \lfloor \log_2 1 \rfloor - (2^{\lfloor \log_2 1 \rfloor} - 1)$
- $T(2) \leq 1 = 2 \lfloor \log_2 2 \rfloor - (2^{\lfloor \log_2 2 \rfloor} - 1)$
- $T(3) \leq 3 = 3 \lfloor \log_2 3 \rfloor - (2^{\lfloor \log_2 3 \rfloor} - 1)$
- $T(4) \leq 5 = 4 \lfloor \log_2 4 \rfloor - (2^{\lfloor \log_2 4 \rfloor} - 1)$
- $T(5) \leq 8 = 5 \lfloor \log_2 5 \rfloor - (2^{\lfloor \log_2 5 \rfloor} - 1)$
- $T(6) \leq 11 = 6 \lfloor \log_2 6 \rfloor - (2^{\lfloor \log_2 6 \rfloor} - 1)$
- $T(7) \leq 14 = 7 \lfloor \log_2 7 \rfloor - (2^{\lfloor \log_2 7 \rfloor} - 1)$
- $T(8) \leq 17 = 8 \lfloor \log_2 8 \rfloor - (2^{\lfloor \log_2 8 \rfloor} - 1)$
- $T(9) \leq 21 = 9 \lfloor \log_2 9 \rfloor - (2^{\lfloor \log_2 9 \rfloor} - 1)$
Ceilings of Logarithms

Observations

- $\lceil \log_2 (k + 1) \rceil = \lceil \log_2 k \rceil$ for $k \neq 2^h$.
- $\lceil \log_2 (k + 1) \rceil = \lceil \log_2 k \rceil + 1$ for $k = 2^h$.
- $\lceil \log_2 (2k) \rceil = \lceil \log_2 k \rceil + 1$.
- $\lceil \log_2 (2k + 1) \rceil = \lceil \log_2 k \rceil + 1$ for $k \neq 2^h$.
- $\lceil \log_2 (2k + 1) \rceil = \lceil \log_2 k \rceil + 2$ for $k = 2^h$. 

Proof for $n = 2k$

Theorem

\[ T(n) \leq n \lceil \log_2 n \rceil - (2^{\lceil \log_2 n \rceil} - 1) \]

Inductive step

\[
\begin{align*}
T(n) & \leq 2T(k) + (n - 1) \\
& \leq 2(k \lceil \log_2 k \rceil - (2^{\lceil \log_2 k \rceil} - 1)) + (n - 1) \\
& = 2k \lceil \log_2 k \rceil + n - 2 \cdot 2^{\lceil \log_2 k \rceil} + 2 - 1 \\
& = n(\lceil \log_2 k \rceil + 1) - (2^{\lceil \log_2 k \rceil + 1} - 1) \\
& = n \lceil \log_2 n \rceil - (2^{\lceil \log_2 n \rceil} - 1)
\end{align*}
\]
Proof for $n = 2k + 1$ and $k \neq 2^h$

**Theorem**

$$T(n) \leq n \lceil \log_2 n \rceil - (2^{\lceil \log_2 n \rceil} - 1)$$

**Inductive step**

$$T(n) \leq T(k+1) + T(k) + (n-1)$$
$$\leq ((k + 1) \lceil \log_2 (k + 1) \rceil - (2^{\lceil \log_2 (k+1) \rceil} - 1))$$
$$+ (k \lceil \log_2 k \rceil - (2^{\lceil \log_2 k \rceil} - 1)) + (n - 1)$$
$$= (2k + 1) \lceil \log_2 k \rceil + n - 2 \cdot 2^{\lceil \log_2 k \rceil} + 1$$
$$= n(\lceil \log_2 k \rceil + 1) - (2^{\lceil \log_2 k \rceil+1} - 1)$$
$$= n \lceil \log_2 n \rceil - (2^{\lceil \log_2 n \rceil} - 1)$$
Proof for $n = 2k + 1$ and $k = 2^h$

**Theorem**

$T(n) \leq n \lceil \log_2 n \rceil - (2^{\lceil \log_2 n \rceil} - 1)$

**Inductive step**

\[
T(n) \leq T(k + 1) + T(k) + (n - 1) \\
\leq ((k + 1) \lceil \log_2 (k + 1) \rceil - (2^{\lceil \log_2 (k+1) \rceil} - 1)) \\
+ (k \lceil \log_2 k \rceil - (2^{\lceil \log_2 k \rceil} - 1)) + (n - 1) \\
= (k + 1)(h + 1) - (2k - 1) + kh - (k - 1) + 2k \\
= (2k + 1)h + 3 \\
= n(h + 2) - (2n - 3) \\
= n \lceil \log_2 n \rceil - (2^{\lceil \log_2 n \rceil} - 1)
\]

**Observation**

$2^{\lceil \log_2 n \rceil} = 2^{\lceil \log_2 (2k+1) \rceil} = 2^{h+2} = 4k = 2n - 2$
Sort in Linear Time

**Idea**
- Sort *without* comparisons by using memory locations.

**Complexity**
- Often $o(n \log n)$ and even $O(n)$ for sorting an array of $n$ keys.

**A contradiction?**
- A *different* model.
- A *bounded* range for the keys.
Bucket Sort

Input
- Keys belong to a bounded domain of size $k$:
  - Without loss of generality the keys are $1, 2, \ldots, k$.

Idea
- For each key between 1 and $k$, **count** the number of times it appears in $A$ and then **rearrange** $A$.

Complexity
- $\Theta(n + k)$.
- $O(n)$ for $k = O(n)$. 
Bucket Sort

Implementation

**Bucket-Sort**\( (A[1], \ldots, A[n]) \)

for \( i = 1 \) to \( k \) do (* prepare \( k \) empty buckets *)

\[ B[i] = 0 \]

for \( j = 1 \) to \( n \) do (* fill the buckets *)

\[ B[A[j]] = B[A[j]] + 1 \]

\( j = 0 \)

for \( i = 1 \) to \( k \) do (* spill all the buckets *)

while \( B[i] > 0 \) do (* spill bucket \( i \) *)

\[ j = j + 1 \]

\[ A[j] = i \]

\[ B[i] = B[i] - 1 \]

Complexity

\[ \Theta(k) + \Theta(n) + \Theta(n + k) = \Theta(n + k). \]
Solving Array Problems

Model

- The input is an array containing \( n \) numbers.
- Sometimes the input includes several arrays with the same or different sizes.

Goal

- **Efficiently** do something with the array and/or find something that is based on some or all the numbers in the array(s).
- Determine if the complexity of the most efficient solution is \( \Theta(1) \), \( \Theta(\log(n)) \), \( \Theta(n) \), \( \Theta(n \log(n)) \), or \( \Theta(n^2) \).
Sorted Arrays Vs. Unsorted Arrays

**Sorted arrays**

- Can a binary-search like procedure solve the problem with complexity $\Theta(\log n)$ instead of a possible "trivial" solution that scans the array and examines all the $n$ numbers?
- Can the problem be solved by inspecting only $\Theta(1)$ numbers avoiding more involved search procedures?

**Unsorted arrays**

- Is it possible to solve the problem with complexity $\Theta(n)$ avoiding sorting the array?
- Will sorting the array yield a solution with complexity $\Theta(n \log n)$ instead of a possible "trivial" $\Theta(n^2)$ solution that examines all pairs of numbers?