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

Etymology
- Alteration of Middle English algorisme.
- From Old French & 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?

**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
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.
## 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 among all inputs of size $n$ the worst case 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.
- 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 machines, classroom models, today computers, and 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>$400n$</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**

$$f(n) = O(g(n)) \text{ if } f(n) \text{ asymptotically less than or equal to } g(n)$$

**Big-Omega**

$$f(n) = \Omega(g(n)) \text{ if } f(n) \text{ asymptotically greater than or equal to } g(n)$$

**Big-Theta**

$$f(n) = \Theta(g(n)) \text{ if } f(n) \text{ asymptotically equal to } g(n)$$

**Little-o**

$$f(n) = o(g(n)) \text{ if } f(n) \text{ asymptotically strictly less than } g(n)$$

**Little-omega**

$$f(n) = \omega(g(n)) \text{ if } f(n) \text{ asymptotically strictly greater than } g(n)$$
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 \( c', c'' > 0 \) 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 1: Comparing Growth Rates**

<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 2: Comparing Growth Rates**

<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** \( f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \)

- By the definition of \( O \), there exist \( c_1 > 0 \) and \( n_1 > 0 \) such that \( f(n) \leq c_1g(n) \) for every \( n \geq n_1 \)
- By the definition of \( \Omega \), there exist \( c_2 > 0 \) and \( n_2 > 0 \) such that \( f(n) \geq c_2g(n) \) for every integer \( n \geq n_2 \)
- Therefore, for \( n_0 \geq \max\{n_1, n_2\} \), it follows that \( c_2g(n) \leq f(n) \leq c_1g(n) \) for every \( n \geq n_0 \)
- By the definition of \( \Theta \), \( f(n) = \Theta(g(n)) \) for \( c' = c_1, c'' = c_2 \), and \( n_0 \)
**Θ** as 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, Ω, are reflexive relations**

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

**O, Ω, 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, Ω, are transitive relations**

- \( f(n) = O(g(n)) \land g(n) = O(h(n)) \Rightarrow f(n) = O(h(n)) \)
- \( f(n) = \Omega(g(n)) \land g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n)) \)
Growth of Functions

\[ n^2 \text{ vs. } n \]

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

**“Ignore” constants**

- $3n = \Theta(n/2)$
- $1000000n = \Theta(n/100000)$
- $n \log_2 n/100000 = \Omega(100000000n)$
- $\log_2(n) = \Theta(\log_{10}(n))$

**Polynomials**

- $a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0 = \Theta(n^d)$
  - for constants $a_0, a_1, \ldots, a_d$ and $a_d > 0$
- **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, 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 \( \omega \)
- \( f(n) = o(g(n)) \iff g(n) = \omega(f(n)) \)
- \( f(n) = o(g(n)) \land g(n) = o(h(n)) \Rightarrow f(n) = o(h(n)) \)
- \( f(n) = \omega(g(n)) \land g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n)) \)

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

\( \omega \) vs \( \Omega \)
- \( f(n) = \omega(g(n)) \Rightarrow f(n) = \Omega(g(n)) \)
- \( f(n) = \Omega(g(n)) \nLeftarrow f(n) = \omega(g(n)) \)
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
- \( \log_2 n = o(n) \)
- \( n \log_2 n = \omega(n) \)

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>Formula</th>
<th>Explanation</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>$2^2 \cdots ^2$</td>
<td>$n$ powers</td>
</tr>
</tbody>
</table>
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

A By Definition Algorithm

**Algorithm**

prefix-sum\( (A) \)

```pseudocode
for i = 1 to n do
    S[i] := 0
for i = 1 to n do
    for j = 1 to i do
```

**Correctness**

- By definition
A By Definition Algorithm

Complexity

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

- $\Theta(n) + \Theta(n^2) = \Theta(n^2)$ overall complexity
A By Induction Algorithm

Algorithm

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 \( i \):
  \[
  S[i] = S[i - 1] + A[i] = \sum_{j=1}^{i-1} A[j] + A[i] = \sum_{j=1}^{i} A[j]
  \]
The Prefix-Sum Problem

A By Induction Algorithm

Algorithm

prefix-sum(A)


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


Complexity

- \( \Theta(1) \) complexity outside the loop
- \( n - 1 = \Theta(n) \) iterations of the only loop
- \( \Theta(1) \) complexity for each iteration
- \( \Theta(1) + \Theta(n) \cdot \Theta(1) = \Theta(n) \) overall 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** between real numbers
  - * multiplications and additions
The Prefix-Sum Problem

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

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

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

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

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 prefix-sum method

Algorithm

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

1. $P(c) = a_0$
2. $cc = 1$
3. for $i = 1$ to $n$ do
   1. $cc = cc \cdot c$ (* $cc = c^i$ *)
   2. $P(c) = P(c) + a_i \cdot cc$ (* $P(c) = a_i c^i + a_{i-1} c^{i-1} + \cdots + a_1 c + a_0$ *)
4. return($P(c)$) (* $P(c) = a_n c^n + a_{n-1} c^{n-1} + \cdots + a_1 c + a_0$ *)

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 Prefix-Sum Problem

The Horner’s Algorithm

Algorithm

- **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 (\ast P(c) = a_n c^{n-i} + a_{n-1} c^{n-i-1} + \cdots + a_{i+1} c + a_i \ast) \\
  \text{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 Induction
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
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 $x$ only with comparisons of the type $x \leq i$ for some $1 \leq i \leq n$

Players Goal

- **Player 1** tries to maximize the number of comparisons until Player 2 finds the value of $x$
- **Player 2** tries to minimize the number of comparisons until finding the value of $x$

Complexity: number of comparisons

- In the worst case or in the average case
- As a function of $n$
The Two Models are “Equivalent”

**Equivalence**
- $x \leq i$ is “equivalent” to $K \leq A[i]$
- Algorithms can be “converted” from one model to another while preserving the complexity

**Convinience**
- It is “easier” to design algorithms in the search game model
- It is “easier” to prove bounds and limitations on algorithms in the search game model
Algorithm outline

- Assume a search for $x$ in the range $[1..n]$
- Throughout the algorithm, maintain a lower bound $\ell$ on $x$ such that $\ell \leq x \leq n$
- Initially, $\ell = 1$
- In each round, compare $x$ with the lower bound $\ell$
  - If $x > \ell$ then increment $\ell$ by 1
  - If $x \leq \ell$ then return $\ell$
Sequential Search

Example

- **Input:** \( n = 10 \) and \( x = 7 \) \( \Rightarrow \ (x \in [1..10]) \)

- **Search procedure:**
  - Q1: \( x \leq 1 \) \( \Rightarrow \ A1: \text{NO} \) \( (x \in [2..10]) \)
  - Q2: \( x \leq 2 \) \( \Rightarrow \ A2: \text{NO} \) \( (x \in [3..10]) \)
  - Q3: \( x \leq 3 \) \( \Rightarrow \ A3: \text{NO} \) \( (x \in [4..10]) \)
  - Q4: \( x \leq 4 \) \( \Rightarrow \ A4: \text{NO} \) \( (x \in [5..10]) \)
  - Q5: \( x \leq 5 \) \( \Rightarrow \ A5: \text{NO} \) \( (x \in [6..10]) \)
  - Q6: \( x \leq 6 \) \( \Rightarrow \ A6: \text{NO} \) \( (x \in [7..10]) \)
  - Q7: \( x \leq 7 \) \( \Rightarrow \ A7: \text{YES} \) \( (x \in [7..7]) \)

- **Output:** \( x = 7 \)

- **Complexity:** 7 comparisons
Sequential Search

Algorithm Pseudocode

Sequential-Search (n,x)

\[ \ell = 0 \]

repeat

\[ \ell = \ell + 1 \]

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

return \( \ell \)
Sequential Search

Correctness
- By induction, \( \ell \leq x \leq n \) after \( \ell - 1 \) comparisons with a NO answer

Termination
- If \( x \leq \ell \) then necessarily \( x = \ell \) because by the induction hypothesis \( x \geq \ell \)
- Eventually \( x \leq n \)
Sequential Search

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

**Best case complexity**
- Only 1 comparison when \( x = 1 \)

**Average case complexity**
- \((n + 1)/2\) 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

**Algorithm outline**

- Assume a search for $x$ in the range $[1..n]$
- Throughout the algorithm, maintain a range $[l..u]$ such that $l \leq x \leq u$
- Initially, $l = 1$ and $u = n$
- In each round, compare $x$ with the middle of the range $m = \left\lfloor \frac{u+l}{2} \right\rfloor$
  - If $x \leq m$ then update $u = m$
  - If $x > m$ then update $l = m + 1$
- Terminate when $l = u$
- Return $x = l = u$
**Binary Search – Example**

- **Input:** \( n = 128 \) and \( x = 50 \) \( \Rightarrow \) (* \( x \in [1..128] \) *)

- **Search procedure:**
  - Q1: \( x \leq 64 \) \( \Rightarrow \) A1: YES (* \( x \in [1..64] \) *)
  - Q2: \( x \leq 32 \) \( \Rightarrow \) A2: NO (* \( x \in [33..64] \) *)
  - Q3: \( x \leq 48 \) \( \Rightarrow \) A3: NO (* \( x \in [49..64] \) *)
  - Q4: \( x \leq 56 \) \( \Rightarrow \) A4: YES (* \( x \in [49..56] \) *)
  - Q5: \( x \leq 52 \) \( \Rightarrow \) A5: YES (* \( x \in [49..52] \) *)
  - Q6: \( x \leq 50 \) \( \Rightarrow \) A6: YES (* \( x \in [49..50] \) *)
  - Q7: \( x \leq 49 \) \( \Rightarrow \) A7: NO (* \( x \in [50..50] \) *)

- **Output:** \( x = 50 \)

- **Complexity:** \( 7 = \log_2(128) \) comparisons
Binary Search

Algorithm Pseudocode

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

\[
\begin{align*}
\ell &= 1 \\
u &= n \\
\text{while } \ell < u & \text{ do } \\
& \hspace{1em} m = \left\lfloor \frac{u + \ell}{2} \right\rfloor \\
& \hspace{1em} \text{if } x \leq m \quad (* \text{ comparison } *) \\
& \hspace{2em} \text{then } u = m \\
& \hspace{2em} \text{else } \ell = m + 1 \\
\text{return } \ell
\end{align*}
\]
**Binary Search**

**Notations**
- Let \( u_j \) and \( \ell_j \) be the values of \( u \) and \( \ell \) after iteration \( j \) of the algorithm.
- Let \( \Delta_j = u_j - \ell_j + 1 \) be the size of the range \([\ell_j..u_j]\).
- Initially \( \ell_0 = 1 \), \( u_0 = n \), and \( \Delta_0 = n \).

**Observation**
- \( \Delta_{j+1} \leq \left\lfloor \frac{\Delta_j}{2} \right\rfloor \) 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, $\Delta = 1$ and therefore $\ell = u$ which implies that $x = \ell = u$

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>

*Amotz Bar-Noy (Brooklyn College)*
Dictionary Search

Adversary Player I

Goal
- Maximize the number of comparisons until Player 2 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)| ≥ |S(N)|
  - NO if |S(Y)| < |S(N)|
Example: A possible 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:** \( x = 21 \)
Adversary Player I

Example: Binary-Search

- **Input:** $n = 34$ \hspace{1em} (* $x \in [1..34]$ *)

- **Search:**
  - Q1: $x \leq 17 \Rightarrow A1$: YES \hspace{1em} (* $x \in [1..17]$ *)
  - Q2: $x \leq 9 \Rightarrow A2$: YES \hspace{1em} (* $x \in [1..9]$ *)
  - Q3: $x \leq 5 \Rightarrow A3$: YES \hspace{1em} (* $x \in [1..5]$ *)
  - Q4: $x \leq 3 \Rightarrow A4$: YES \hspace{1em} (* $x \in [1..3]$ *)
  - Q5: $x \leq 2 \Rightarrow A5$: YES \hspace{1em} (* $x \in [1..2]$ *)
  - Q6: $x \leq 1 \Rightarrow A6$: YES \hspace{1em} (* $x \in [1..1]$ *)

- **Output:** $x = 1$

Observation

- With Binary-Search the search always ends up with $x = 1$
Impossible to Search Faster than Binary Search

**Theorem**

There exists \(1 \leq x \leq n\) for which the Adversary Player 1 forces Player 2 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

Worst case
- The $\lceil \log_2 n \rceil$ lower bound is a worst case bound.
- No algorithm can guarantee less comparisons for all values of $x$.

Average case
- It is possible to prove an $\Omega(\log n)$ average case lower bound.

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

Clue
- **Player 1** selects only even numbers 2, 4, 6, 8, ... between 1 and an even $n$

A modified Binary Search
- The search domain is 1, 2, ..., $n/2$
- Instead of asking “if $x \leq i$”, **Player 2** asks “if $x \leq 2i$” and then considers the answer as if it was the answer to “if $x \leq i$”
- When the search outputs $x = i$ the modified search outputs $2i$

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

**Clue**
- Player 1 selects only even numbers 2, 4, 6, 8, ... between 1 and an even \( n \)

**Example**
- \( n = 32 = 2 \cdot 16 \) and \( x = 20 = 2 \cdot 10 \)
- The possible 16 values for \( x \) are 2, 4, 6, ..., 32 and the search domain is 1, 2, ..., 16

**Running the algorithm**
- Question 1: \( x \leq (2 \cdot 8 = 16) \)? because \( 8 = \lceil (1 + 16)/2 \rceil \)
- Question 2: \( x \leq (2 \cdot 12 = 24) \)? because \( 12 = \lceil (9 + 16)/2 \rceil \)
- Question 3: \( x \leq (2 \cdot 10 = 20) \)? because \( 10 = \lceil (9 + 12)/2 \rceil \)
- Question 4: \( x \leq (2 \cdot 9 = 18) \)? because \( 9 = \lceil (9 + 10)/2 \rceil \)
- \( x = 20 \) found with \( 4 = \log_2 16 = \log_2 31 - 1 \) comparisons
Searching with “Clues”

Clue
- **Player 1** selects only square numbers 1, 4, 9, 16, ... between 1 and a square number $n$

A modified Binary Search
- The search domain is $1, 2, \ldots, \sqrt{n}$
- Instead of asking “if $x \leq i$”, **Player 2** asks “if $x \leq i^2$” and then considers the answer as if it was the answer to “if $x \leq i$”
- When the search outputs $x = i$ the modified search outputs $i^2$

Complexity
- $\lceil \log_2(\sqrt{n}) \rceil \approx \log_2(\sqrt{n}) = \frac{1}{2} \log_2(n)$ 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, 16, ... between 1 and a square number \( n \)

Example

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

Running the algorithm

- Question 1: \( x \leq (8^2 = 64) \)? because \( 8 = \lfloor (1 + 16)/2 \rfloor \)
- Question 2: \( x \leq (12^2 = 144) \)? because \( 12 = \lfloor (9 + 16)/2 \rfloor \)
- Question 3: \( x \leq (10^2 = 100) \)? because \( 10 = \lfloor (9 + 12)/10 \rfloor \)
- Question 4: \( x \leq (9^2 = 81) \)? because \( 9 = \lfloor (9 + 10)/10 \rfloor \)
- \( x = 100 \) found with \( 4 = \log_2 16 = (1/2) \log_2 256 \) comparisons
**Dictionary Search**

**Searching with “Clues”**

**Clue**

- **Player 1** selects only powers of 2 numbers $2, 4, 8, 16, \ldots$ between 2 and a power of 2 number $n$

**A modified Binary Search**

- The search domain is $1, 2, \ldots, \log_2 n$
- Instead of asking “if $x \leq i$”, **Player 2** asks “if $x \leq 2^i$” and then considers the answer as if it was the answer to “if $x \leq i$”
- When the search outputs $x = i$ the modified search outputs $2^i$

**Complexity**

- $\lceil \log_2(\log_2(n)) \rceil \approx \log_2(\log_2(n))$ comparisons
- For $n = 2^{32} = 4294967296$ the saving is from 32 to 5 comparisons although there are only 32 candidates!
Searching with “Clues”

Clue
- **Player 1** selects only powers of 2 numbers 2, 4, 8, 16, ... between 2 and a power of 2 number $n$

Example
- $n = 65536 = 2^{16}$ and $x = 1024 = 2^{10}$
- The possible 16 values for $x$ are 2, 4, 8, ..., 65536 and the search domain is 1, 2, ..., 16

Running the algorithm
- Question 1: $x \leq (2^8 = 256)$? $8 = \lfloor (1 + 16)/2 \rfloor$
- Question 2: $x \leq (2^{12} = 4096)$? $12 = \lfloor (9 + 16)/2 \rfloor$
- Question 3: $x \leq (2^{10} = 1024)$? $10 = \lfloor (9 + 12)/10 \rfloor$
- Question 4: $x \leq (2^9 = 512)$? $9 = \lfloor (9 + 10)/10 \rfloor$
- $x = 1024$ found with $4 = \log_2 16 = \log_2 \log_2 65536$ comparisons
Searching with “Clues”

**Clue**
- **Player 1** selects only primes 2, 3, 5, 7, ... not larger than \( n \)

**A modified Binary Search**
- The search domain is \( 1, 2, \ldots, \pi(n) \) where \( \pi(n) \) is the number of primes between 2 and \( n \)
- Instead of asking “if \( x \leq i \)” , **Player 2** asks “if \( x \leq p_i \)” where \( p_i \) is the \( i \)th prime and then considers the answer as if it was the answer to “if \( x \leq i \)”
- When the search outputs \( x = i \) the modified search outputs \( p_i \)

**Complexity**
- \( \log_2(n/\ln(n)) \approx \log_2(n) - \log_2 \log_2(n) \) comparisons because there are approximately \( n/\ln(n) \) primes between 2 and \( n \)
- There are 78498 primes between 2 and 1000000. The clue saves only 3 comparisons, because \( \lceil \log_2(1000000) \rceil = 20 \) and \( \lceil \log_2(78498) \rceil = 17 \)
Searching with “Clues”

**Clue**
- **Player 1** selects only primes 2, 3, 5, 7, … not larger than \(n\)

**Example**
- \(n = 53\) and \(x = 29\)
- The possible 16 values for \(x\) are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53
- and the search domain is 1, 2, …, 16

**Running the algorithm**
- **Question 1:** \(x \leq (p_8 = 19)?\) \(8 = \left\lfloor (1 + 16)/2 \right\rfloor\)
- **Question 2:** \(x \leq (p_{12} = 37)?\) \(12 = \left\lfloor (9 + 16)/2 \right\rfloor\)
- **Question 3:** \(x \leq (p_{10} = 29)?\) \(10 = \left\lfloor (9 + 12)/10 \right\rfloor\)
- **Question 4:** \(x \leq (p_9 = 23)?\) \(9 = \left\lfloor (9 + 10)/10 \right\rfloor\)
- \(x = 29\) found with \(4 \approx \log_2(53) - \log_2 \log_2(53)\) 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:
  
  \[\begin{align*}
  x \leq 1? & \quad x \leq 2? & \quad x \leq 4? & \quad x \leq 8? & \cdots & \quad x \leq 2^j? & \cdots \\
  \end{align*}\]

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

  **Phase 2:** Apply binary search on the domain
  
  \[\left[2^{k-1} + 1..2^k\right]\]

**Complexity**
- \(k + 1\) comparisons are asked in **Phase 1**
- The number of comparisons asked in **Phase 2** is
  
  \[
  \left\lfloor \log_2(2^k - (2^{k-1} + 1) + 1) \right\rfloor = \left\lfloor \log_2(2^{k-1}) \right\rfloor = k - 1
  \]

  Total number of comparisons:
  
  \[(k + 1) + (k - 1) = 2k = 2\left\lceil \log_2(x) \right\rceil\]
**Input:** $x = 50$

**Search procedure:**

- Q1: $x \leq 1 \Rightarrow A1$: NO (* $x \in [2..\infty]$ *

- Q2: $x \leq 2 \Rightarrow A2$: NO (* $x \in [3..\infty]$ *

- Q3: $x \leq 4 \Rightarrow A3$: NO (* $x \in [5..\infty]$ *

- Q4: $x \leq 8 \Rightarrow A4$: NO (* $x \in [9..\infty]$ *

- Q5: $x \leq 16 \Rightarrow A5$: NO (* $x \in [17..\infty]$ *

- Q6: $x \leq 32 \Rightarrow A6$: NO (* $x \in [33..\infty]$ *

- Q7: $x \leq 64 \Rightarrow A7$: YES (* $x \in [33..64]$ *

- Q8: $x \leq 48 \Rightarrow A8$: NO (* $x \in [49..64]$ *

- Q9: $x \leq 56 \Rightarrow A9$: YES (* $x \in [49..56]$ *

- Q10: $x \leq 52 \Rightarrow A10$: YES (* $x \in [49..52]$ *

- Q11: $x \leq 50 \Rightarrow A11$: YES (* $x \in [49..50]$ *

- Q12: $x \leq 49 \Rightarrow A12$: NO (* $x \in [50..50]$ *

**Output:** $x = 50$

**Complexity:** $12 = \lceil 2\log_2 50 \rceil$ comparisons
The Sorting Problem

Keys
- Entities from a well ordered domain

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

Input
- An unsorted array of $n$ keys $A[1], A[2], \ldots, A[n]$

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
- An unsorted array of $n$ keys $A[1], A[2], \ldots, A[n]$

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: [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]
- Final sorted array: [1, 2, 3, 5, 8, 13, 21]
## Example

<table>
<thead>
<tr>
<th>Initial array</th>
<th>[34, 89, 13, 55, 21]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Round 1</strong></td>
<td></td>
</tr>
<tr>
<td>21 &lt; 55</td>
<td>[34, 89, 13, 21, 55]</td>
</tr>
<tr>
<td>13 &lt; 21</td>
<td>[34, 89, 13, 21, 55]</td>
</tr>
<tr>
<td>13 &lt; 89</td>
<td>[34, 13, 89, 21, 55]</td>
</tr>
<tr>
<td>13 &lt; 34</td>
<td>[13, 34, 89, 21, 55]</td>
</tr>
<tr>
<td><strong>Round 2</strong></td>
<td></td>
</tr>
<tr>
<td>21 &lt; 55</td>
<td>[13, 34, 89, 21, 55]</td>
</tr>
<tr>
<td>21 &lt; 89</td>
<td>[13, 34, 21, 89, 55]</td>
</tr>
<tr>
<td>21 &lt; 34</td>
<td>[13, 21, 34, 89, 55]</td>
</tr>
<tr>
<td><strong>Round 3</strong></td>
<td></td>
</tr>
<tr>
<td>55 &lt; 89</td>
<td>[13, 21, 34, 55, 89]</td>
</tr>
<tr>
<td>34 &lt; 55</td>
<td>[13, 21, 34, 55, 89]</td>
</tr>
<tr>
<td><strong>Round 4</strong></td>
<td></td>
</tr>
<tr>
<td>55 &lt; 89</td>
<td>[13, 21, 34, 55, 89]</td>
</tr>
<tr>
<td><strong>Sorted array</strong></td>
<td>[13, 21, 34, 55, 89]</td>
</tr>
</tbody>
</table>
**Bubble Sort: Correctness and Complexity**

**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
  $$\frac{n(n-1)}{2} = \Theta(n^2)$$
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
    - \( q = \lfloor \frac{p+r}{2} \rfloor \) (* \( p \leq q < q + 1 \leq r \) *)
    - **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 \( A[p..q] \) since \( q - p < r - p \)
  - 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**

\[
\begin{align*}
T(1) &= 0 \\
T(2) &= 2 \cdot T(1) + (2 - 1) = 1 \\
T(4) &= 2 \cdot T(2) + (4 - 1) = 5 \\
T(8) &= 2 \cdot T(4) + (8 - 1) = 17 \\
T(16) &= 2 \cdot T(8) + (16 - 1) = 49 \\
T(32) &= 2 \cdot T(16) + (32 - 1) = 129
\end{align*}
\]

**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

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

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

$T(n) \leq 2T(n/2) + (n - 1)$

$\leq 2 \left( (n/2) \log_2 n - (n - 1) \right) + (n - 1)$

$= n \log_2 n - (n - 1)$
Merge-Sort – Complexity for $n \neq 2^k$

Bottom-Up evaluation

$T(1) \leq 0$

$T(2) \leq T(1) + T(1) + (2 - 1) = 1$

$T(3) \leq T(2) + T(1) + (3 - 1) = 3$

$T(4) \leq T(2) + T(2) + (4 - 1) = 5$

$T(5) \leq T(3) + T(2) + (5 - 1) = 8$

$T(6) \leq T(3) + T(3) + (6 - 1) = 11$

$T(7) \leq T(4) + T(3) + (7 - 1) = 14$

$T(8) \leq T(4) + T(4) + (8 - 1) = 17$

$T(9) \leq T(5) + T(4) + (9 - 1) = 21$
Merge-Sort – Complexity for $n \neq 2^k$

**Guess**

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

**Verification**

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

Observations

\[ \lceil \log_2 (k + 1) \rceil = \lceil \log_2 k \rceil \text{ for } k \neq 2^h \]

\[ \lceil \log_2 (k + 1) \rceil = \lceil \log_2 k \rceil + 1 \text{ for } k = 2^h \]

\[ \lceil \log_2 (2k) \rceil = \lceil \log_2 k \rceil + 1 \]

\[ \lceil \log_2 (2k + 1) \rceil = \lceil \log_2 k \rceil + 1 \text{ for } k \neq 2^h \]

\[ \lceil \log_2 (2k + 1) \rceil = \lceil \log_2 k \rceil + 2 \text{ for } k = 2^h \]
Proof for $n = 2k$

**Theorem**

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

**Inductive step**

$$T(n) \leq 2T(k) + (n - 1)$$

$$\leq 2(k \lceil \log_2 k \rceil - (2^\lceil \log_2 k \rceil - 1)) + (n - 1)$$

$$= 2k \lceil \log_2 k \rceil + n - 2 \cdot 2^\lceil \log_2 k \rceil + 2 - 1$$

$$= n(\lceil \log_2 k \rceil + 1) - (2^\lceil \log_2 k \rceil + 1 - 1)$$

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

**Theorem**

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

**Inductive step**

\[
T(n) \leq T(k + 1) + T(k) + (n - 1) \\
\leq ((k + 1) \left\lceil \log_2 (k + 1) \right\rceil - (2^{\left\lceil \log_2 (k+1) \right\rceil} - 1)) \\
+ (k \left\lceil \log_2 k \right\rceil - (2^{\left\lceil \log_2 k \right\rceil} - 1)) + (n - 1) \\
= (2k + 1) \left\lceil \log_2 k \right\rceil + n - 2 \cdot 2^{\left\lceil \log_2 k \right\rceil} + 1 \\
= n(\left\lceil \log_2 k \right\rceil + 1) - (2^{\left\lceil \log_2 k \right\rceil+1} - 1) \\
= n \left\lceil \log_2 n \right\rceil - (2^{\left\lceil \log_2 n \right\rceil} - 1) \]
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 is assumed
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 *)

$j = 0$

for $i = 1$ to $k$ do  (* spill all the buckets *)
    while $B[i] > 0$ do
        $j = j + 1$
        $A[j] = i$
        $B[i] = B[i] - 1$

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

**Arbitrary arrays**
- Can the problem be solved by inspecting only $\Theta(1)$ numbers avoiding more involved procedures?

**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?

**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?