

# Analysis of Algorithms

## EXAMPLES

## The Prefix-Sum Problem

**Input:** An array  $A$  of  $n$  real numbers:  $A[1], A[2], \dots, A[n]$ .

**Output:** An array  $S$  of size  $n$  such that  $S[i] = \sum_{j=1}^i A[j]$   
for  $1 \leq i \leq n$ .

**Example:**

★  $A = [3, 1, 2, 3, 18, 100, \dots]$ .

★  $S = [3, 4, 6, 9, 27, 127, \dots]$ .

## Algorithm 1

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

**Correctness:** By definition.

## Algorithm I – Complexity

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

## Algorithm II

**prefix-sum**( $A$ )

$S[1] := A[1]$

**for**  $i = 2$  **to**  $n$  **do**

$S[i] := S[i - 1] + A[i]$

**Correctness:** By Induction.

## Algorithm II – Complexity

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

## Evaluating a Polynomial

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

**Output:** The value of the polynomial  $P(x)$  for  $x = c$ :

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0.$$

**Example:**

★  $a_3 = 5, a_2 = 7, a_1 = 3, a_0 = 11,$  and  $c = 2.$

★  $P(x) = 5x^3 + 7x^2 + 3x + 11.$

★  $P(2) = 5 \cdot 2^3 + 7 \cdot 2^2 + 3 \cdot 2 + 11 = 85.$

**Optimization goal:** Minimize the number of **operations** (**multiplications** and **additions**) between real numbers.

## Algorithm I: A Direct Approach

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

$$P(c) = a_0$$

for  $i = 1$  to  $n$  do

$$a = a_i$$

for  $j = 1$  to  $i$  do

$$a = a \cdot c$$

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

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

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

**Correctness:** By definition.

## Algorithm I – Complexity

- ★  $i$  **multiplications** in the  $i$ th iteration of the inner loop.
- ★  $1 + 2 + \dots + n = \frac{n(n+1)}{2}$  **multiplications** overall.
- ★  $n$  **additions** in the outer loop.
- ★ Total of  $\frac{1}{2}n^2 + \frac{3}{2}n$  **operations**.
- ★  $\Theta(n^2)$  time complexity.

## Algorithm II: An Improvement

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

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

$$P(c) = a_0$$

$$cc = 1$$

**for**  $i = 1$  **to**  $n$  **do**

$$cc = cc \cdot c \quad (* \text{ } cc = c^i \text{ } *)$$

$$P(c) = P(c) + a_i \cdot cc$$

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

**Correctness:** By induction.

## Algorithm II – Complexity

- ★ 2 **multiplications** in the  $i$ th iteration of the loop.
- ★ 1 **addition** in the  $i$ th iteration of the loop.
- ★ Total of  $3n$  **operations**:  $2n$  **multiplications** and  $n$  **additions**.
- ★  $\Theta(n)$  time complexity.

## Algorithm III: A Sophisticated Method

**Idea and proof of correctness:**

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

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

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

$$P(c) = a_n$$

**for**  $i = n - 1$  **downto** 0 **do**

$$P(c) = P(c) \cdot c + a_i$$

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

## Algorithm III: Example

**Input:**  $P(x) = 5x^4 + 4x^3 + 3x^2 + 2x + 1$

**Algorithm:**

$$\star P_4(x) = a_4 = 5$$

$$\star P_3(x) = P_4(x)x + a_3 = 5x + 4$$

$$\star P_2(x) = P_3(x)x + a_2 = 5x^2 + 4x + 3$$

$$\star P_1(x) = P_2(x)x + a_1 = 5x^3 + 4x^2 + 3x + 2$$

$$\star P_1(x) = P_2(x)x + a_1 = 5x^4 + 4x^3 + 3x^2 + 2x + 1 = P(x)$$

## Algorithm III – Complexity

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

## Computing the Greatest Common Divisor

**Input:** Two integers  $x > 0$  and  $y > 0$ .

**Output:** The maximum integer  $z \geq 1$  such that  $z$  divides both  $x$  and  $y$ .

**Notation:**  $z = \gcd(x, y)$ .

**Examples:**

$$5 = \gcd(5, 15)$$

$$6 = \gcd(12, 18)$$

$$1 = \gcd(13, 21)$$

## Algorithm 1

**Fact:**  $1 \leq z \leq \min \{x, y\}$ .

**Idea and proof of correctness:** If  $z = \gcd(x, y)$  then there is no  $w > z$  that divides both  $x$  and  $y$ .

$\gcd(x, y)$

$z := \min \{x, y\} + 1$

**repeat**

$z := z - 1$

**until**  $(x \bmod z = 0)$  **and**  $(y \bmod z = 0)$

## Algorithm I – Complexity

- ★  $O(\min \{x, y\})$  iterations of the **repeat** loop.
- ★  $\Omega(\min \{x, y\})$  iterations when  $\gcd(x, y) = 1$ .
- ★  $\Theta(1)$  for each iteration.
- ★  $\Theta(\min \{x, y\})$  **overall** complexity.

## Algorithm II

**Idea and proof of correctness:**

$$\gcd(x, y) = \gcd(x - y, y) \text{ for } x > y.$$

**Proof:** If  $w$  divides  $x$  and  $y$  it divides  $x - y$ .

$\gcd(x, y)$

**if**  $x = y$  **then return**  $x$

**if**  $x < y$  **then**  $x \leftrightarrow y$

**return**  $\gcd(x - y, y)$

## Algorithm II – Complexity

- ★  $\Theta(1)$  for each recursive call.
- ★  $O(\max\{x, y\})$  recursive calls.
- ★  $\Omega(\max\{x, y\})$  recursive calls for  $\min\{x, y\} = 1$ .
- ★  $\Theta(\max\{x, y\})$  **overall** complexity.
- ★ **Usually** algorithm II is much **faster** than algorithm I.
- ★ Although the worst-case complexity of algorithm II is worse than the worst-case complexity of algorithm I!

# Euclid Algorithm

**Idea and proof of correctness:**

$$\gcd(x, y) = \gcd(y, x \bmod y) \text{ for } x > y.$$

**Proof:** If  $w$  divides  $x$  and  $y$  it divides  $x \bmod y$ .

$\gcd(x, y)$

**if**  $x \bmod y = 0$  **then return**  $y$

**return**  $\gcd(y, x \bmod y)$

**Complexity:**  $\Theta(\log(x + y))$ .

– **Usually** the algorithm terminates **faster**.

## Example 1

**Input:**  $x = 372$  and  $y = 138$ .

$x$	$y$
372	138
138	96
96	42
42	12
12	6

**Output:**  $\gcd(372, 138) = 6$ .

## Example II

**Input:**  $x = 21$  and  $y = 13$ .

$x$	$y$
21	13
13	8
8	5
5	3
3	2
2	1

**Output:**  $\gcd(21, 13) = 1$ .

## Time Complexity of Euclid Algorithm

**Lemma:** The Euclid algorithm for  $x = F_{k+1}$  and  $y = F_k$  has a time complexity  $\Omega(k) = \Omega(\log x)$ .

**Corollary:** The time complexity of the Euclid algorithm is  $\Omega(\log(\max\{x, y\}))$ .

**Lemma:** Pairs of Fibonacci numbers are the **worst case** for the Euclid algorithm.

**Corollary:** The time complexity of the Euclid algorithm is  $O(\log(\max\{x, y\}))$ .

**Theorem:** The time complexity of the Euclid algorithm is  $\Theta(\log(\max\{x, y\}))$ .

## Another proof for Time Complexity

- ★ There are  $O(1)$  operations in each recursive call.
- ★ If  $x < y$ , then after one iteration  $x > y$ .
- ★ Let  $x_i \geq y_i$  be the 2 inputs in the  $i$ th recursive call:
  - $x_0 = x$  and  $y_0 = y$ .
  - $x_{i+1} = y_i$  and  $y_{i+1} = x_i \bmod y_i$ .
  - $x_k \bmod y_k = 0 \Rightarrow y_k = \gcd(x, y)$ .
  - $x_0 > x_1 > \dots > x_k$ .

## Another proof for Time Complexity

**Claim:**  $x_{i+2} \leq \frac{x_i}{2}$ .

– If  $y_i \leq \frac{x_i}{2}$  then already  $x_{i+1} = y_i \leq \frac{x_i}{2}$ .

– Otherwise,  $y_{i+1} = x_i \bmod y_i \leq \frac{x_i}{2} \Rightarrow x_{i+2} = y_{i+1} \leq \frac{x_i}{2}$ .

★ In every 2 recursive calls  $x$  is divided by at least 2.

★ At most  $k = 2 \log_2(x)$  recursive calls.

★ Overall complexity:  $O(\log(\max\{x, y\}))$ .

## Binary Euclid Algorithm

*bingcd*( $x, y$ )

- (a) **if**  $x = y$   
    **then return**  $x$
- (b) **if**  $x$  and  $y$  are even  
    **then return**  $2 \cdot \text{bingcd}(\frac{x}{2}, \frac{y}{2})$
- (c) **if**  $x$  is even and  $y$  is odd  
    **then return**  $\text{bingcd}(\frac{x}{2}, y)$
- (d) **if**  $x$  is odd and  $y$  is even  
    **then return**  $\text{bingcd}(x, \frac{y}{2})$
- (e) **if**  $x$  and  $y$  are odd  
    **then return**  $\text{bingcd}(\frac{|x-y|}{2}, \min\{x, y\})$

## Correctness

All the **return** decisions are **correct**:

- (a) By definition.
- (b) A 2 factor was found.
- (c) There are no more 2 factors.
- (d) There are no more 2 factors.
- (e)  $\gcd(x, y) = \gcd\left(\frac{|x-y|}{2}, \min\{x, y\}\right)$   
since both  $x$  and  $y$  are odd.

## Example 1

**Input:**  $x = 126$  and  $y = 48$ .

$x$	$y$	found factor
126	48	2
63	24	1
63	12	1
63	6	1
63	3	1
30	3	1
15	3	1
6	3	1
3	3	3

**Output:**  $\gcd(126, 48) = 2 \cdot 3 = 6$ .

## Example II

**Input:**  $x = 21$  and  $y = 13$ .

$x$	$y$	found factor
21	13	1
4	13	1
2	13	1
1	13	1
1	6	1
1	3	1
1	1	1

**Output:**  $\gcd(21, 13) = 1$ .

## Complexity: Upper Bound

**Assumption:** Division by 2 can be done in  $O(1)$  (shifting).

- ★ The complexity of each recursive call is  $O(1)$ .
- ★ In each recursive call, either  $x$  or  $y$  or both are divided by at least 2. Note that  $\frac{|x-y|}{2} < \frac{\max\{x,y\}}{2}$ .
- ★ At most  $\log_2 x + \log_2 y \leq 2 \log_2(\max\{x, y\})$  recursive calls.
- ★ Overall  $O(\log(\max\{x, y\}))$  time complexity.

## Complexity: Lower Bound

**Input:**  $x = 2^k + 1$  and  $y = 2^k$ .

$x$	$y$	found factor
$2^k + 1$	$2^k$	1
$2^k + 1$	$2^{k-1}$	1
$\vdots$	$\vdots$	$\vdots$
$2^k + 1$	1	1
$2^{k-1}$	1	1
$2^{k-2}$	1	1
$\vdots$	$\vdots$	$\vdots$
1	1	1

**Output:**  $\gcd(2^k + 1, 2^k) = 1$ .

**Complexity:**  $2k \approx 2 \log_2(\max\{x, y\})$  recursive calls implying  $\Omega(\log(\max\{x, y\}))$  complexity.