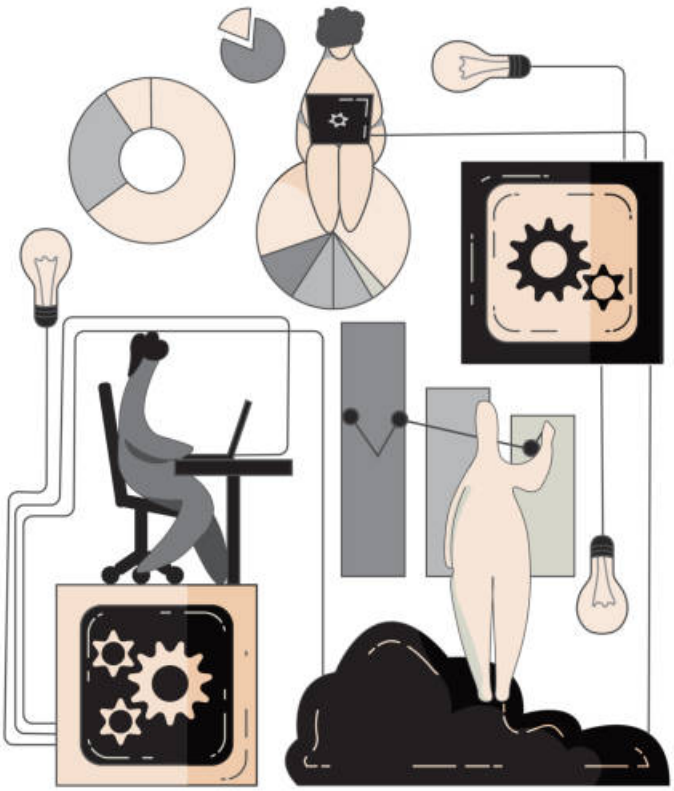
Algorithm Analysis

Ji Yong-Hyeon



Department of Information Security, Cryptology, and Mathematics
College of Science and Technology
Kookmin University

November 27, 2023

Contents

Part I

Basic Algorithms

Chapter 1

Basic Algorithms

1.1 Rotation Algorithm

Rotation

Problem. Rotate vector $x[n]$ by d positions.
Constraints. $O(n)$ time, $O(1)$ extra space.

Example 1.1. For $n = 8$ and $d = 3$, change

$$abcdefgh \rightarrow defghabc.$$

1.1.1 A Juggling Algorithm

Code 1.1: Juggling Rotation Algorithm

```
// Function to find the greatest common divisor (GCD)
int gcd(int a, int b) {
    return b ? gcd(b, a%b) : a;
}

// Function to rotate an array using Juggling Alg.
void juggling(int arr[], int n, int d) {
    int i, j, k, temp;

    // Find GCD of n and d
    int g = gcd(d, n);

    for (i = 0; i < g; i++) {
        // Store the first element of the current set
        temp = arr[i];

        j = i;

        // Shift each element of the set
        while (1) {
            k = j + d;

            // If k exceeds the array size, bring it within bounds
            if (k >= n) {
                k = k - n;
            }

            // If we are back to the initial position, break the loop
            if (k == i) {
                break;
            }

            // Move the element
            arr[j] = arr[k];

            // Update j for the next iteration
            j = k;
        }

        // Put the first element in its correct position within the set
        arr[j] = temp;
    }
}
```

1.1.2 The Block-Swap Algorithm

1.1.3 The Reversal Algorithm

Chapter 2

The Efficiency of Algorithms

2.1 Real-valued Functions of a Real Variable and Their Graphs

Graph

Definition 2.1. Let f be a real-valued function of a real variable. The graph of f is the set

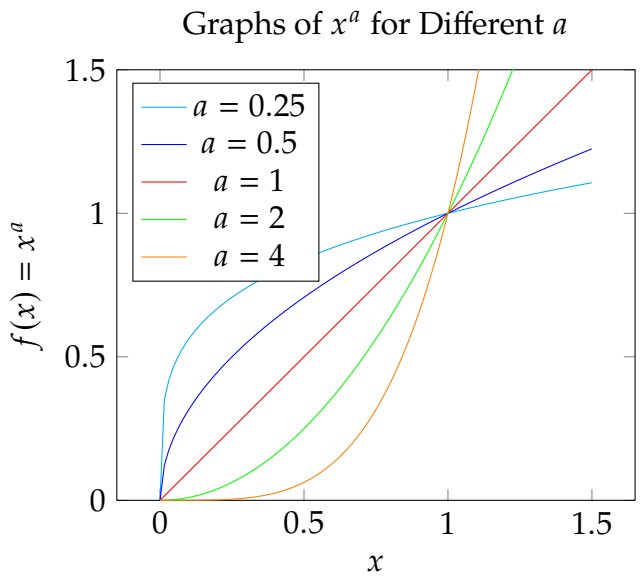
$$\Gamma_f := \left\{(x, y) \in \mathbb{R}^2 : y = f(x)\right\}.$$

Power Function

Definition 2.2. Let $a \in \mathbb{R}_{\geq 0}$. Define p_a, the power function with exponent a, as follows:

$$p_a(x) = x^a \quad \text{for} \quad x \in \mathbb{R}_{\geq 0}.$$

Example 2.1.



The Floor and Ceiling Function

Definition 2.3.

$$f(x) := \lfloor x \rfloor := \sup\{n \in \mathbb{Z} : n \leq x\},$$
$$g(x) := \lceil x \rceil := \inf\{n \in \mathbb{Z} : x < n\}.$$

Example 2.2.

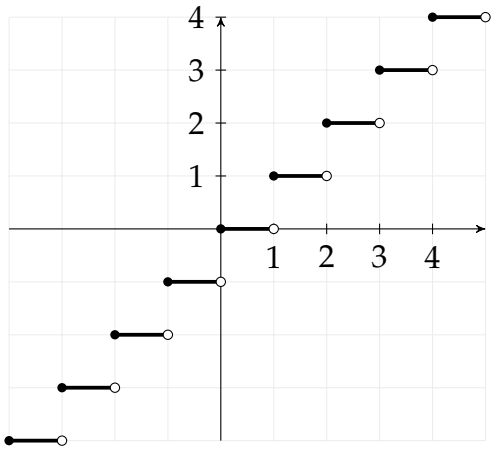


Figure 2.1: Graph of $f(x) = \lfloor x \rfloor$.

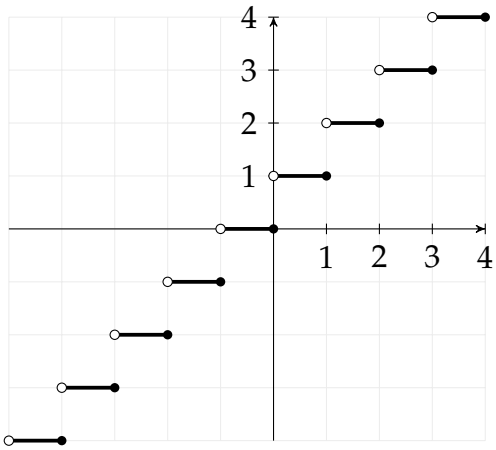


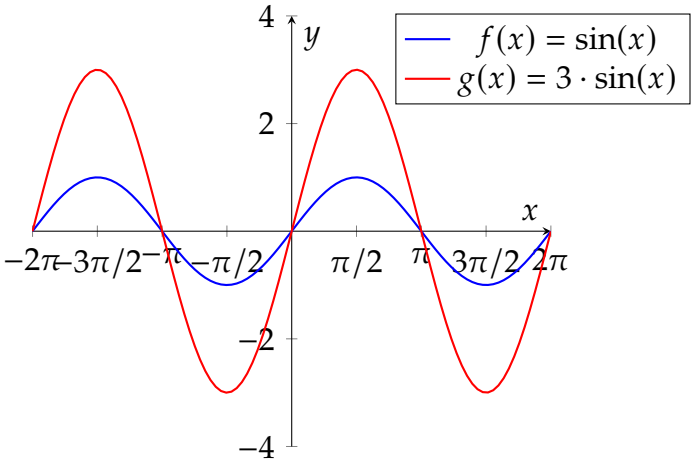
Figure 2.2: Graph of $f(x) = \lceil x \rceil$.

Graph of a Multiple of a Function

Definition 2.4. The function Mf, called the **multiple of** f **by** M or M **times** f, is the real-valued function with the as domain as f that is defined by the rule

$$\forall x \in \mathrm{Dom}(f) : (Mf)(x) = M \cdot f(x).$$

Example 2.3.



Absolute Function

Definition 2.5.

$$A(x) := |x| := \begin{cases} x & : x \geq 0 \\ -x & : x < 0 \end{cases}.$$

Increasing and Decreasing Function

Definition 2.6.

(1) A real-valued function f is **increasing** on S iff

$$\forall x_1, x_2 \in S : x_1 < x_2 \implies f(x_1) < f(x_2).$$

(2) A real-valued function g is **decreasing** on S iff

$$\forall x_1, x_2 \in S : x_1 < x_2 \implies g(x_1) > g(x_2).$$

2.2 O, Σ, Θ Notation

Note.

- **Algorithm** is methods for solving problems which are suited for computer implementation.
- **Algorithm Efficiency**
  - **Time efficiency** is a measure of amount of time for an algorithm to execute.
  - **Space efficiency** is a measure of amount of memory needed for algorithm to execute.
- O, Σ, Θ **notation** provide approximations that make it easy to evaluate large-scale differences in algorithm efficiency, while ignoring differences of a constant factor and differences that occur only for small sets of input data.

Ω, O and Θ

Definition 2.7. Let f and g be real-valued functions defined on the $\mathbb{R}_{\geq 0}$.

(1) f **is of order at least** g, written $\Omega(g(x))$, if and only if,

$$(\exists A \in \mathbb{R}_{>0})(\exists a \in \mathbb{R}_{\geq 0}) \quad x > a \implies A\left|g(x)\right| \leq \left|f(x)\right|.$$

(2) f **is of order at most** g, written $O(g(x))$, if and only if,

$$(\exists B \in \mathbb{R}_{>0})(\exists b \in \mathbb{R}_{\geq 0}) \quad x > b \implies \left|f(x)\right| \leq B\left|g(x)\right|.$$

(3) f **is of order** g, written $\Theta(g(x))$, if and only if,

$$(\exists A, B \in \mathbb{R}_{>0})(\exists k \in \mathbb{R}_{\geq 0}) \quad x > k \implies A\left|g(x)\right| \leq \left|f(x)\right| \leq B\left|g(x)\right|.$$

Example 2.4 (Translating to Θ-Notation)**.** Use Θ-notation to express the statement

$$10|x^6| \leq |17x^6 - 45x^3 + 2x + 8| \leq 30|x^6| \quad \text{for all real numbers } x > 2.$$

Solution. Let $A = 10 > 0$, $B = 30 > 0$ and $k = 2 \geq 0$. Then

$$A|x^6| \leq |17x^6 - 45x^3 + 2x + 8| \leq B|x^6| \quad \text{for all real numbers } x > k.$$

By definition of Θ-notation,

$$17x^6 - 45x^3 + 2x + 8 \quad \text{is} \quad \Theta(x^6).$$

□

Example 2.5 (Translating to O- and Θ-Notation)**.**

(i) Use Ω and O notations to express the statements

a. $15|\sqrt{x}| \leq \left|\dfrac{15\sqrt{x}(2x+9)}{x+1}\right|$ for all real numbers $x > 0$.

b. $\left|\dfrac{15\sqrt{x}(2x+9)}{x+1}\right| \leq 45|\sqrt{x}|$ for all real numbers $x > 7$.

(ii) Justify the statement: $\dfrac{15\sqrt{x}(2x+9)}{x+1}$ is $\Theta(\sqrt{x})$.

Solution.

(i) a. Let $A = 15$ and $a = 0$. Then

$$A|\sqrt{x}| \leq \left|\frac{15\sqrt{x}(2x+9)}{x+1}\right| \quad \text{for all real numbers } x > a.$$

By definition of Ω-notation,

$$\frac{15\sqrt{x}(2x+9)}{x+1} \quad \text{is} \quad \Omega(\sqrt{x}).$$

b. Let $B = 45$ and $b = 7$. Then

$$\left|\frac{15\sqrt{x}(2x+9)}{x+1}\right| \leq B|\sqrt{x}| \quad \text{for all real numbers } x > b.$$

By definition of O-notation,

$$\frac{15\sqrt{x}(2x+9)}{x+1} \quad \text{is} \quad O(\sqrt{x}).$$

(ii) Let $A = 15$, $B = 45$, and let $k = \max(0, 7) = 7$. Then

$$A|\sqrt{x}| \leq \left|\frac{15\sqrt{x}(2x+9)}{x+1}\right| \leq B|\sqrt{x}| \quad \text{for all real numbers } x > k.$$

Hence, by definition of O-notation,

$$\frac{15\sqrt{x}(2x+9)}{x+1} \quad \text{is} \quad \Theta(\sqrt{x}).$$

□

Theorem 2.1. *Let f and g be real-valued functions defined on $\mathbb{R}_{\geq 0}$.*

(1) $f(x)$ is $\Omega(g(x))$ and $f(x)$ is $O(g(x))$ $\iff$ $f(x)$ is $\Theta(g(x))$.

(2) $f(x)$ is $\Omega(g(x))$ $\iff$ $g(x)$ is $O(f(x))$.

(3) $f(x)$ is $O(g(x))$ and $g(x)$ is $O(h(x))$ $\implies$ $f(x)$ is $O(h(x))$.

Proof. (1) Clearly, it holds.

(2) ($\Rightarrow$) Suppose $f(x)$ is $\Omega(g(x))$ then

$$(\exists A > 0)(\exists a \geq 0)(\forall x > a) \quad A|g(x)| \leq |f(x)|.$$

Divide both sides by A to obtain

$$(\forall x > a) \quad |g(x)| \leq \frac{1}{A}|f(x)|.$$

Let $B := 1/A$ and $b := a$. Then

$$(\forall x > b) \quad |g(x)| \leq B|f(x)|,$$

and so $g(x)$ is $O(f(x))$ by definition of O-notation.

($\Leftarrow$) Suppose $f(x)$ is $O(g(x))$ then

$$(\exists B > 0)(\exists b \geq 0)(\forall x > b) \quad |f(x)| \leq B|g(x)|.$$

Divide both sides by B to obtain

$$(\forall x > b) \quad \frac{1}{B}|f(x)| \leq |g(x)|.$$

Let $A := 1/B$ and $a := b$. Then

$$(\forall x > a) \quad A|f(x)| \leq |g(x)|,$$

and so $g(x)$ is $\Omega(f(x))$ by definition of Ω-notation.

(3) Suppose that $f(x)$ is $O(g(x))$ and $g(x)$ is $O(h(x))$. Then

$$(\exists B_1, B_2 \in \mathbb{R}_{>0})(\exists b_1, b_2 \in \mathbb{R}_{\geq 0}) \quad \begin{array}{l} x > b_1 \Rightarrow |f(x)| \leq B_1|g(x)| \quad \text{and} \\ x > b_2 \Rightarrow |g(x)| \leq B_2|h(x)|. \end{array}$$

Let $B = B_1B_2$ and $b = \max(b_1, b_2)$. Then

$$x > b \implies |f(x)| \leq B_1|g(x)| \leq B_1(B_2|h(x)|) \leq B|h(x)|.$$

Then, by definition of O-notation, $f(x)$ is $O(h(x))$.

□

Example 2.6. Show that $2x^4 + 3x^3 + 5$ is $\Theta(x^4)$.

Solution. Define functions f and g as follows:

$$f(x) := 2x^4 + 3x^3 + 5, \text{ and}$$
$$g(x) := x^4$$

for all $x \in \mathbb{R}_{\geq 0}$.

(i) For $x > 0$,

$$2x^4 \leq 2x^4 + 3x^3 + 5$$
$$2|x^4| \leq |2x^4 + 3x^3 + 5|.$$

Let $A = 2$ and $a = 0$. Then

$$A|x^4| \leq |2x^4 + 3x^3 + 5| \quad \text{for all } x > a,$$

and so $2x^4 + 3x^3 + 5$ is $\Omega(x^4)$.

(ii) For $x > 1$,

$$2x^4 + 3x^3 + 5 \leq 2x^4 + 3x^4 + 5x^4,$$
$$2x^4 + 3x^3 + 5 \leq 10x^4,$$
$$|2x^4 + 3x^3 + 5| \leq 10|x^4|.$$

Let $B = 10$ and $b = 1$. Then

$$|2x^4 + 3x^3 + 5| \leq B|x^4| \quad \text{for all } x > b,$$

and so $2x^4 + 3x^3 + 5$ is $O(x^4)$.

By (i) and (ii), we know that $2x^4 + 3x^3 + 5$ is $\Theta(x^4)$. □

Example 2.7.

a. Show that $3x^3 - 1000x - 200$ is $O(x^3)$.

b. Show that $3x^3 - 1000x - 200$ is $O(x^s)$ for all integer $s > 3$.

Solution. a. For $x > 1$,

$$\begin{aligned} |3x^3 - 1000x - 200| &\leq |3x^3| + |1000x| + |200| \\ &\leq 3x^3 + 1000x^3 + 200x^3 \\ &\leq 1203x^3 \end{aligned}$$

$\therefore 3x^3 - 1000x - 200$ is $O(x^3)$.

b. Suppose s is an integer with $s > 3$. For $x > 1$, we know $x^3 < x^s$. Then

$$B|x^3| < B|x^s|$$

for $x > b$ ($\because b = 1$). Thus,

$$|3x^3 - 1000x - 200| \leq B|x^s| \quad \text{for all } x > 1.$$

Hence, $3x^3 - 1000x - 200$ is $O(x^s)$ for all integer $s > 3$.

□

Example 2.8.

a. Show that $3x^3 - 1000x - 200$ is $\Omega(x^3)$.

b. Show that $3x^3 - 1000x - 200$ is $\Omega(x^r)$ for all integer $r < 3$.

Solution.

a. Let $a := 2 \times \dfrac{(1000 + 200)}{3} = 800$. Then

$$\begin{aligned} x &> a, \\ x &> 800, \\ x &> \frac{2 \cdot 1000}{3} + \frac{2 \cdot 200}{3}, \\ x &> \frac{2 \cdot 1000}{3} \cdot \frac{1}{x} + \frac{2 \cdot 200}{3} \frac{1}{x^2}, \\ \frac{3}{2}x^3 &> 1000x + 200, \\ 3x^3 - \frac{3}{2}x^3 &> 1000x + 200, \\ 3x^3 - 1000x - 200 &> \frac{3}{2}x^3, \\ |3x^3 - 1000x - 200| &> \frac{3}{2}|x^3|. \end{aligned}$$

Let $A = 3/2$ and let $a = 800$. Then

$$A|x^3| \leq |3x^3 - 1000x - 200| \quad \text{for all } x > a.$$

$\therefore 3x^3 - 1000x - 200$ is $\Omega(x^3)$.

b. Suppose that $r < 3$. Since $x^r < x^3$, we have $A|x^r| < A|x^3|$ for $x > 1$. Thus

$$A|x^r| \leq |3x^3 - 1000x - 200| \quad \text{for all } x > a = 800 > 1$$

Hence, $3x^3 - 1000x - 200$ is $\Omega(x^r)$ for all integer $r < 3$.

□

On Polynomial Orders

Theorem 2.2. *Let $a_i \in \mathbb{R}$ for $i = 0, \ldots, n$ with $a_n \neq 0$.*

(1) $\sum_{i=0}^{n} a_i x^i$ *is* $O(x^s)$ *for all integers* $s \geq n$.

(2) $\sum_{i=0}^{n} a_i x^i$ *is* $\Omega(x^r)$ *for all integers* $r \leq n$.

(3) $\sum_{i=0}^{n} a_i x^i$ *is* $\Theta(x^n)$.

Proof. Let $A = \sum_{i=0}^{n} |a_i|$ and $a = 1$. Then $\left|\sum_{i=0}^{n} a_i x^i\right| \leq A|x^n|$ for all $x > 1$. □

Example 2.9. Show that x^2 is not $O(x)$, and deduce that x^2 is not $\Theta(x)$.

Solution. Suppose that x^2 is $O(x)$. Then

$$(\exists B > 0)(\exists b \geq 0)(\forall x > b) \quad |x^2| \leq B|x|. \tag{*}$$

Since

$$x \cdot x > B \cdot x \implies |x^2| > B|x| \implies \exists x > b : |x^2| > B|x|.$$

This contradicts (*). Hence, x^2 is not $O(x)$. □

Limitation on Orders of Polynomial Functions

Theorem 2.3. *Let $n \in \mathbb{Z}^+$, and let $a_i \in \mathbb{R}$ for $i = 0, \ldots, n$ with $a_n \neq 0$. If $m < n$, then*

(1) $\sum_{i=0}^{n} a_i x^i$ *is not* $O(x^m)$ *and*

(2) $\sum_{i=0}^{n} a_i x^i$ *is not* $\Omega(x^m)$.

2.3 Application: Efficiency of Algorithm I

Note. Time efficiency of algorithm counting the number of elementary operations.

Definition 2.8. Let A be an algorithm.

- $b(n)$ = Minimum number of elementary operation.
  - If $b(n) = \Theta(g(n))$, we say in the bast case A is $\Theta(g(n))$.
  - A has a best case order of $g(n)$.
- $w(n)$ = Maximum number of elementary operation.
  - If $w(n) = \Theta(g(n))$, we say in the worst case A is $\Theta(g(n))$.
  - A has a worst case order of $g(n)$.

Example 2.10. Assume n is a positive integer and consider the following algorithm segment:

$p := 0, x := 2$
for $i := 2$ **to** n
  $p := (p + i) \cdot x$
next i

a. Compute the actual number of additions and multiplications that must be performed when this algorithm segment is executed.

b. Use the theorem on polynomial orders to find an order for this algorithm segment.

Solution.

a. There are two operations $(+, \times)$ for each iterations of the loop. The number of iterations of the **for-next** loop equals

$$\text{the top index} \; - \; \text{the bottom index} \; + 1 = n - 2 + 1 = n - 1.$$

Hence there are $2(n-1) = 2n - 2$ multiplications and additions.

b. By the theorem on polynomial orders,

$$2n - 2 \quad \text{is} \quad \Theta(n),$$

and so this algorithm segment is $\Theta(n)$.

□

Example 2.11. Assume n is a positive integer and consider the following algorithm segment:

```
s := 0
for i := 1 to n
  for j := 1 to i
    s := s + j · (i − j + 1)
  next j
next i
```

a. Compute the actual number of additions and multiplications that must be performed when this algorithm segment is executed.

b. Use the theorem on polynomial orders to find an order for this algorithm segment.

Solution.

a. There are four operations $(+, \cdot, -, +)$ for each iterations of the loop. Note that Hence

i	1	2		3			$\cdots$	n			
j	1	1	2	1	2	3	$\cdots$	1	2	$\cdots$	n

the total number of iterations of the inner loop is

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2},$$

and so the number of operations is $4 \cdot \frac{n(n+1)}{2} = 2n(n+1)$.

b. By the theorem on polynomial orders, $2n(n+1)$ is $\Theta(n^2)$, and so this algorithm segment is $\Theta(n^2)$.

□

Asymptotic Upper Bound

Definition 2.9.

$$O(g(n)) := \left\{ f(n) : (\exists c, n_0 > 0)(\forall n \geq n_0)\ 0 \leq f(n) \leq cg(n) \right\}.$$

Asymptotic Lower Bound

Definition 2.10.

$$\Omega(g(n)) := \left\{ f(n) : (\exists c, n_0 > 0)(\forall n \geq n_0)\ 0 \leq cg(n) \leq f(n) \right\}.$$

Asymptotic Tight Bound

Definition 2.11.

$$\Theta(g(n)) := \left\{ f(n) : (\exists c_1, c_2, n_0 > 0)(\forall n \geq n_0)\ 0 \leq c_g(n) \leq f(n) \leq c_2 g(n) \right\}.$$

Chapter 3

Sorting of Numbers

- **Input:** A sequence of n numbers $[a_1, a_2, \ldots, a_n]$.
- **Output:** A sorted permutation $[a'_1, a'_2, \ldots, a'_n]$ s.t. $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

3.1 The Insertion Sort Algorithm

Algorithm 1: Insertion-Sort (A)

Input: $A = [a_1, a_2, \ldots, a_n]$
Output: $A' = [a'_1, a'_2, \ldots, a'_n]$ s.t. $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

```
1 for j ← 2 to n do
2 |   key← A[j];
  |   /* Insert A[j] into the sorted sequence A[1,...,j − 1]   */
3 |   i ← j − 1;
4 |   while i > 0 and A[i] > key do
5 |   |   A[i + 1] ← A[i];
6 |   |   i ← i − 1;
7 |   end
8 |   A[i + 1] ← key;
9 end
```

```c
void insertion(int* arr, int num) {
    int key;

    for(int i=1; i<num; i++) {
        key = *(arr+i);
        int j = i-1;
        while (j>=0 && *(arr+j) > key) {
            *(arr+j+1) = *(arr+j);
            j -= 1;
        }
        *(arr+j+1) = key;
    }
}

```

```
/************************************************
* Input: 126 062 214 103 004 098 150 055 136 077
*
* [i=1] 062 126 214 103 004 098 150 055 136 077
* [i=2] 062 126 214 103 004 098 150 055 136 077
* [i=3] 062 103 126 214 004 098 150 055 136 077
* [i=4] 004 062 103 126 214 098 150 055 136 077
* [i=5] 004 062 098 103 126 214 150 055 136 077
* [i=6] 004 062 098 103 126 150 214 055 136 077
* [i=7] 004 055 062 098 103 126 150 214 136 077
* [i=8] 004 055 062 098 103 126 136 150 214 077
* [i=9] 004 055 062 077 098 103 126 136 150 214
*
* Output: 004 055 062 077 098 103 126 136 150 214
************************************************/
```

3.1.1 Time Analysis 1

- The running time of Insertion-Sort on an input of n values, $T(n)$, is the sum of the products of the *cost* and *times* columns:

$$\begin{aligned} T(n) =& c_1 n + c_2(n-1) + c_4(n-1) \\ &+ c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) \\ &+ c_8(n-1). \end{aligned}$$

- Best-case running time:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

- $T(n) = \Omega(n)$.

- Worst-case running time:

$$\begin{aligned} T(n) =& c_1 n + c_2(n-1) + c_4(n-1) \\ &+ c_5 \left(\frac{n(n+1)}{2} - 1 \right) + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) \\ &+ c_8(n-1) \\ =& \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &- (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

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

3.1.2 Complexity Analysis

- The best-case running time of insertion sort is $\Omega(n)$. The running time of insertion sort therefore falls between $\Omega(n)$ and $O(n^2)$. The worst-case running time of insertion sort is $\Omega(n^2)$, since there exists an input that causes the algorithm to take $\Omega(n^2)$ time.
- When we say that the running time of an algorithm is $\Omega(g(n))$, we mean that no matter what particular input size is chosen for each value of n, the running time on that input is at least a constant times $g(n)$, for sufficiently large n.

3.2 The Bubble Sort Algorithm

Algorithm 2: Bubble-Sort (A)

Input: $A = [a_1, a_2, \ldots, a_n]$
Output: $A' = [a'_1, a'_2, \ldots, a'_n]$ s.t. $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

```
1 for i ← 1 to n do
2 |   for j ← n downto i + 1 do
3 |   |   if A[j] < A[j − 1] then
4 |   |   |   exchange A[j] ↔ A[j − 1];
5 |   |   end
6 |   end
7 end
```

```c
void bubble(int* arr, int num) {
    for(int i=0; i<num; i++) {
        for(int j=num; j>i+1; j--) {
            if(*(arr+j) < *(arr+j-1)) {
                int tmp = *(arr+j);
                *(arr+j) = *(arr+j-1);
                *(arr+j-1) = tmp;
            }
        }
    }
}
/*********************************************
* Input: 92 54 99 02 47 66 66 05 20 72
*
* [i j = 00 09] 92 54 99 02 47 66 66 05 20 72
* [i j = 00 08] 92 54 99 02 47 66 66 05 20 72
* [i j = 00 07] 92 54 99 02 47 66 05 66 20 72
* [i j = 00 06] 92 54 99 02 47 05 66 66 20 72
* [i j = 00 05] 92 54 99 02 05 47 66 66 20 72
* [i j = 00 04] 92 54 99 02 05 47 66 66 20 72
* [i j = 00 03] 92 54 02 99 05 47 66 66 20 72
* [i j = 00 02] 92 02 54 99 05 47 66 66 20 72
* [i j = 00 01] 02 92 54 99 05 47 66 66 20 72
* [i j = 01 09] 02 92 54 99 05 47 66 66 20 72
* [i j = 01 08] 02 92 54 99 05 47 66 20 66 72
* [i j = 01 07] 02 92 54 99 05 47 20 66 66 72
* [i j = 01 06] 02 92 54 99 05 20 47 66 66 72
* [i j = 01 05] 02 92 54 99 05 20 47 66 66 72
* [i j = 01 04] 02 92 54 05 99 20 47 66 66 72
* [i j = 01 03] 02 92 05 54 99 20 47 66 66 72
* [i j = 01 02] 02 05 92 54 99 20 47 66 66 72
* [i j = 02 09] 02 05 92 54 99 20 47 66 66 72
* [i j = 02 08] 02 05 92 54 99 20 47 66 66 72
* [i j = 02 07] 02 05 92 54 99 20 47 66 66 72
* [i j = 02 06] 02 05 92 54 99 20 47 66 66 72
* [i j = 02 05] 02 05 92 54 20 99 47 66 66 72
* [i j = 02 04] 02 05 92 20 54 99 47 66 66 72
* [i j = 02 03] 02 05 20 92 54 99 47 66 66 72
```

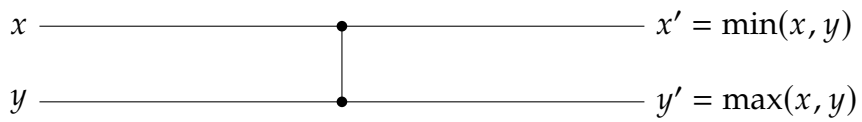
```
39 * [i j = 03 09] 02 05 20 92 54 99 47 66 66 72
40 * [i j = 03 08] 02 05 20 92 54 99 47 66 66 72
41 * [i j = 03 07] 02 05 20 92 54 99 47 66 66 72
42 * [i j = 03 06] 02 05 20 92 54 47 99 66 66 72
43 * [i j = 03 05] 02 05 20 92 47 54 99 66 66 72
44 * [i j = 03 04] 02 05 20 47 92 54 99 66 66 72
45 * [i j = 04 09] 02 05 20 47 92 54 99 66 66 72
46 * [i j = 04 08] 02 05 20 47 92 54 99 66 66 72
47 * [i j = 04 07] 02 05 20 47 92 54 66 99 66 72
48 * [i j = 04 06] 02 05 20 47 92 54 66 99 66 72
49 * [i j = 04 05] 02 05 20 47 54 92 66 99 66 72
50 * [i j = 05 09] 02 05 20 47 54 92 66 99 66 72
51 * [i j = 05 08] 02 05 20 47 54 92 66 66 99 72
52 * [i j = 05 07] 02 05 20 47 54 92 66 66 99 72
53 * [i j = 05 06] 02 05 20 47 54 66 92 66 99 72
54 * [i j = 06 09] 02 05 20 47 54 66 92 66 72 99
55 * [i j = 06 08] 02 05 20 47 54 66 92 66 72 99
56 * [i j = 06 07] 02 05 20 47 54 66 66 92 72 99
57 * [i j = 07 09] 02 05 20 47 54 66 66 92 72 99
58 * [i j = 07 08] 02 05 20 47 54 66 66 72 92 99
59 * [i j = 08 09] 02 05 20 47 54 66 66 72 92 99
60 *
61 * Output: 02 05 20 47 54 66 66 72 92 99
62 ********************************************/
```

3.3 Sorting Networks

Comparator

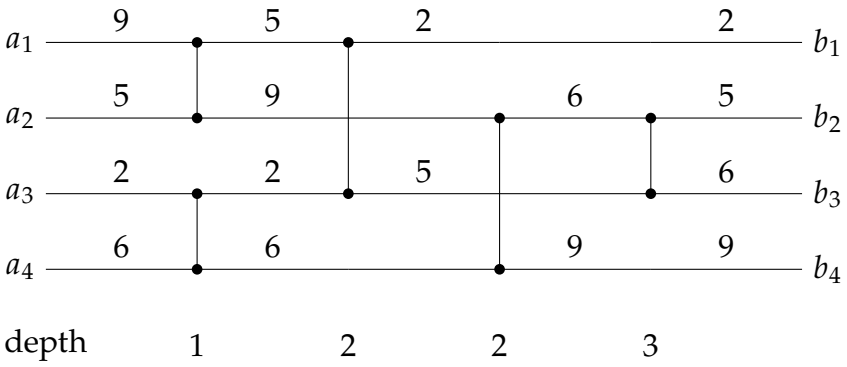
Definition 3.1. A **comparator** is a device with two inputs, x and y, and two outputs, x' and y', that performs the following function:

$$x' = \min(x, y),$$
$$y' = \max(x, y).$$



Remark 3.1. Works in $O(1)$ time.

Example 3.1.



- Wires go straight, left to right.
- Each comparator has inputs/outputs on some pair of wires.

Depth

Definition 3.2. We define the **depth** of a wire as follows:

(1) An input wire of a comparison network has depth 0.

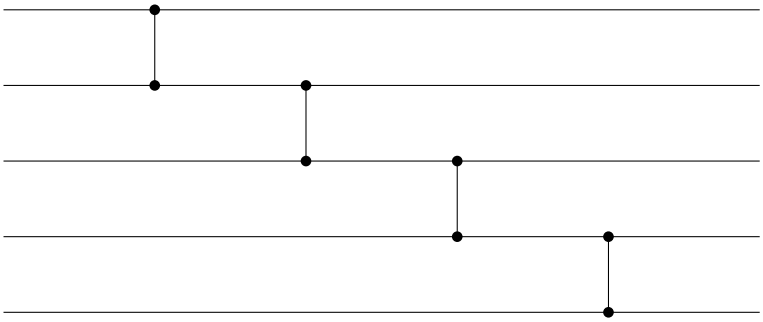
(2) If a comparator has two input wires with depths d_x and d_y, then its output wires have depth $\max(d_x, d_y) + 1$.

Remark 3.2.

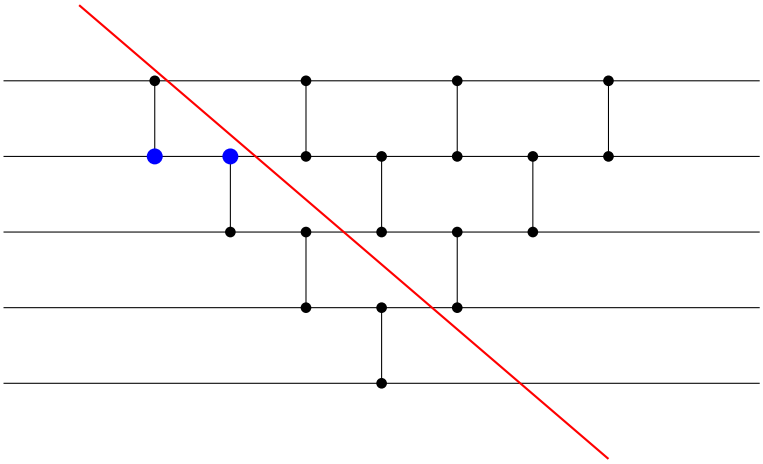
(1) **Depth of a Comparator** := depth of its output wire.

(2) **Depth of a Network** := maximum depth of an output of the network.

Example 3.2 (Bouble-Sort)**.** Find the maximum of 5 values:



We extend our idea:



Depth:

$$\begin{cases} D(n) = D(n-1) + 2 \\ D(2) = 1 \end{cases}.$$

Then

$$\begin{aligned} D(2) &= 1 \\ D(3) &= D(2) + 2 = 3 \\ D(4) &= D(2) + 2 + 2 = 5 \\ D(5) &= D(2) + 2 + 2 + 2 = 7 \\ &\vdots \\ D(k) &= D(2) + 2 \cdot (k-2), \end{aligned}$$

and so

$$D(n) = D(2) + 2 \cdot (n-2) = 1 + 2n - 4 = 2n - 3 = \Theta(n).$$

3.3.1 The Zero-One Principle

"How can we test if a comparison network sorts?"

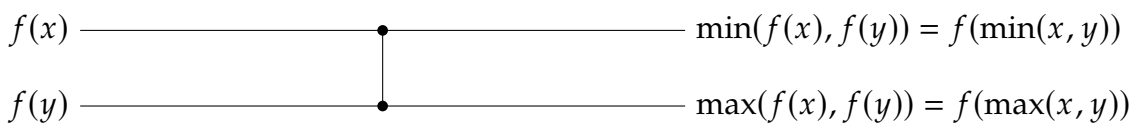
Lemma 3.1. *If a comparison network transforms*

$$a = \langle a_1, \ldots, a_n \rangle \quad \textit{into} \quad b = \langle b_1, \ldots, b_n \rangle,$$

then for any montonically increasing function f, it transforms

$$f(a) = \langle f(a_1), \ldots, f(a_n) \rangle \quad \textit{into} \quad f(b) = \langle f(b_1), \ldots, f(b_n) \rangle.$$

Proof. Since f is montotonically increasing function, we know



We use mathematical induction on the depth of each wire in a general comparison network:

(i) (Basis Step) Clearly

(ii) (Induction Step)

□

Example 3.3. Consider

Let

$$f(x) := \begin{cases} 0 & : x \leq 3 \\ 1 & : x > 1 \end{cases}$$

Then

$f(6) = 1$ —— $0 = f(2)$

$f(5) = 1$ —— $0 = f(3)$

$f(2) = 0$ —— $1 = f(4)$

$f(4) = 1$ —— $1 = f(5)$

$f(3) = 0$ —— $1 = f(6)$

Zero-One Principle

Theorem 3.2. *If a comparison network with n inputs sorts all 2^n possible sequences of 0's and 1's, then it sorts all sequences of arbitrary numbers correctly.*

Proof. Suppose that

$$\exists \text{sequence } \langle a_1, a_2, \ldots, a_n \rangle \text{ s.t. } a_i < a_j \text{ but } \langle \cdots, a_j, \cdots, a_i, \cdots \rangle.$$

We define a montonically increasing function f as

$$f(x) := \begin{cases} 0 & : x \leq a_i, \\ 1 & : x > a_i. \end{cases}$$

By Lemma, □

3.3.2 A Bitonic Sorting Network

Bitonic

Definition 3.3. A sequence is **bitonic** if it monotonically increases, then monotonically decreases, or it can be circularly shifted to become so.

Half-cleaner

Definition 3.4. A bitonic sorter is composed of severs stages, each of which is called a **half -cleaner**. Each half-cleaner is a comparison network of depth 1 in which input line i is compared with line $i + \frac{n}{2}$ for $i = 1, 2 \ldots, \frac{n}{2}$. (We assume that n is even.)

Example 3.4.

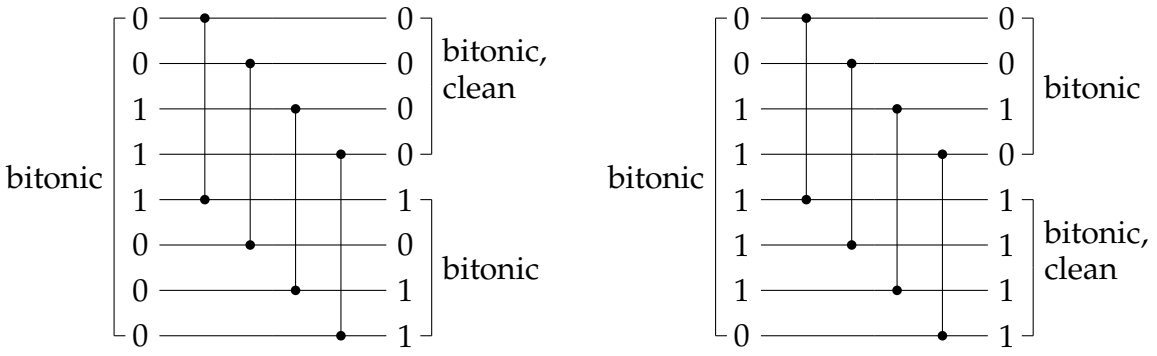


Figure 3.1: The comparison network Half-Cleaner[8].

Lemma 3.3. *Let the input to a half-cleaner is a bitonic sequence of 0's and 1's. Then the output satisfies the following properties:*

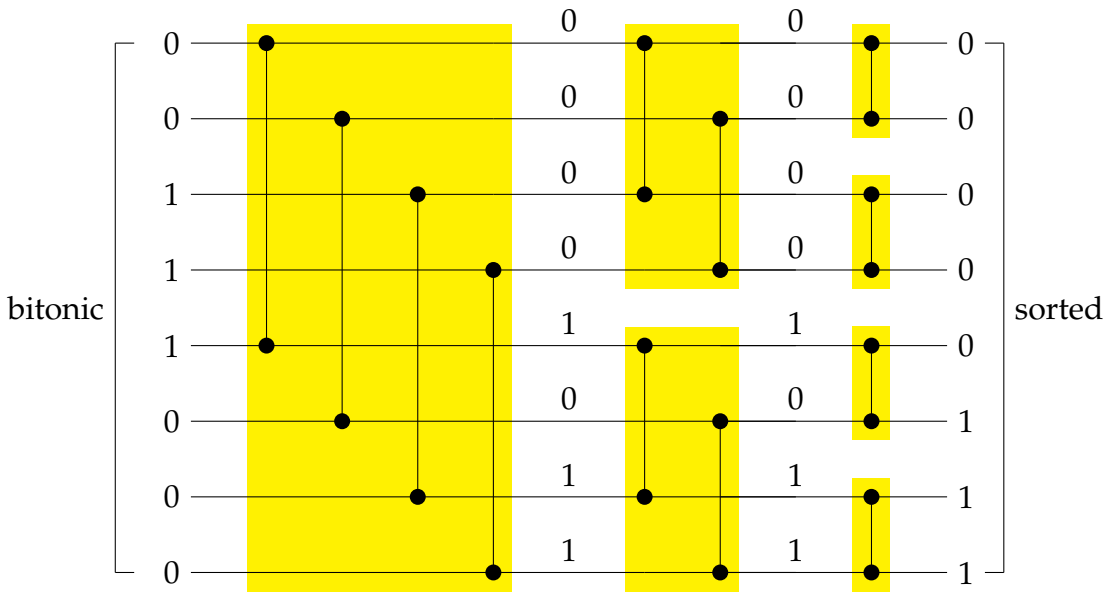
(1) both the top half and the bottom half are bitonic;

(2) every element in the top half is at least as small as every element of the bottom half, and

*(3) at least one half is **clean**(all 0's or all 1's).*

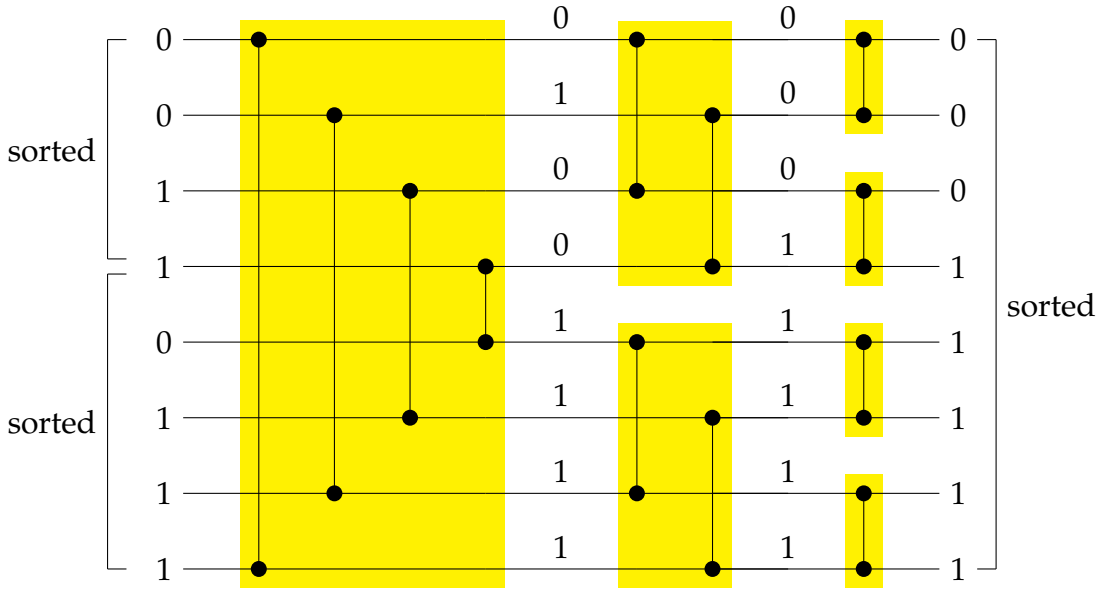
Proof. content... □

Example 3.5.



3.3.3 Merging Network

Example 3.6.



Chapter 4

Summary

4.1 Sorting of Numbers

- Input: A sequence of n numbers $[a_1, a_2, \ldots, a_n]$.
- Output: A permutation (reordering) $[a'_1, a'_2, \ldots, a'_n]$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

4.2 The Divide-and-Conquer Approach

- Divide the problem into a number of subproblems.
- Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, solve the subproblems in a straightforward manner.
- Combine the solutions to the subproblems into the solution for the original problem.

4.3 Merge-Sort Algorithm

- Divide: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
- Conquer: Sort the two subsequences recursively using merge sort.
- Combine: Merge the two sorted subsequences to produce the sorted answer.

4.4 Merge Procedure

[Details of the merge procedure]

4.5 Example Merge Procedure

[Details of the example merge procedure]

4.6 Correctness of Merge Procedure

- Loop invariant: [Description of the loop invariant]

4.7 Merging - Worst Case Example

- Symmetrically sized inputs (e.g., two times four elements): 67775558.
- We compare 6 with all 5s and 8 (4 comparisons).
- We compare 8 with all 7s (3 comparisons).
- Generalized for n elements: worst case requires $n - 1$ comparisons.

4.8 Analysis of Merge-Sort Regarding Comparisons

- When $n \geq 2$ for merge-sort steps:
  - Divide: Just compute q as the average of p and $r \Rightarrow$ no comparisons.
  - Conquer: Recursively solve two subproblems each of size $n/2 \Rightarrow 2T(n/2)$.
  - Combine: MERGE on an n-element subarray requires cn comparisons $\Rightarrow$ $cn = \Theta(n)$.

4.9 Merge-Sort Recurrences

- Recurrence regarding comparisons: [Details of the recurrence]
- Time complexity: [Details of the time complexity]

4.10 Figures

Figure 4.1: Example figure caption.

4.11 Introduction to Merging

Merging is the process of combining two adjacent sorted sequences into a single sorted sequence. The operation takes two sequences of sizes m and n and produces a sorted sequence of $m + n$ elements.

4.11.1 Example of Merging

$$\begin{aligned} \text{Input Sequences:} \quad & 4, 91 \quad \text{and} \quad 3, 92 \\ \text{Merged Sequence:} \quad & 3, 4, 91, 92 \end{aligned}$$

4.12 Stability in Sorting Algorithms

Stability is a property of a sorting algorithm which maintains the relative order of equal elements from the input in the output.

4.12.1 Stability Example

4.12.2 Importance of Stability

The stability of a sorting algorithm is crucial in scenarios where the relative ordering of equivalent elements carries significance, such as in radix sort or when multiple keys are used for sorting.

4.13 Case Study: Radix Sort

4.13.1 Radix Sort Process

4.13.2 Implications of Stability in Radix Sort

Stability ensures that radix sort maintains the relative ordering of elements with equal keys, which is essential for the correctness of the algorithm.

Theorem 4.1. *The merge-sort algorithm is stable under the condition that the merge operation is implemented in a stable manner.*

4.14 Highlights

- Worst-case running time: $\Theta(n^2)$
- Expected running time: $\Theta(n \log n)$
- Constants hidden in $\Theta(n \log n)$ are small.
- Sorts in place.

4.15 Divide and Conquer Strategy with Quicksort

- **Divide:** Partition $A[p \ldots r]$ into two (possibly empty) subarrays $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ such that each element in the first subarray is $\leq A[q]$ and $A[q]$ is $\leq$ each element in the second subarray.
- **Conquer:** Sort the two subarrays by recursive calls to QUICKSORT.
- **Combine:** No work is needed to combine the subarrays because they are sorted in place.

4.16 Creation of Partitions (Divide Step)

4.16.1 Loop invariant

1. All entries in $A[p \dots i]$ are $\leq$ pivot.
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot.
3. $A[r]$ = pivot.

4.16.2 Time for partitioning

- $\Theta(n)$ to partition an n-element subarray.

4.17 Correctness of Partition

- **Initialization:** Before the first iteration of the loop, the loop invariant is trivially satisfied as the subarrays are empty.
- **Maintenance:** During each iteration, if $A[j] \leq$ pivot, we swap $A[j]$ with $A[i+1]$ and increment both i and j. If $A[j] >$ pivot, we only increment j.
- **Termination:** When $j = r-1$, the array is partitioned into $A[p \dots i] \leq$ pivot, $A[i+1 \dots r-1] >$ pivot, and $A[r]$ = pivot.

4.18 Performance Cases

4.18.1 Worst-Case Scenario

- Occurs when the subarrays are unbalanced, with 0 elements in one and $n-1$ in the other.
- Recurrence: $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.

4.18.2 Best-Case Scenario

- Occurs when the subarrays are balanced each time, with $\leq n/2$ elements.
- Recurrence: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$.

4.18.3 Average-Case Analysis

- Average running time is much closer to the best case.
- If partitioning always produces a 9-to-1 split, the recurrence is $T(n) \leq T(9n/10) + T(n/10) + \Theta(n) = O(n \log n)$.

4.19 Randomized Version of Quicksort

- To ensure performance, randomization is added to quicksort.
- The algorithm assumes all input permutations are equally likely.

4.20 Detailed Average Case Analysis

- Dominant cost is partitioning, which is called at most n times.
- Total work done is $O(n + X)$, where X is the total number of comparisons.

4.21 Figures

Figure 4.2: Detailed recursion tree of quicksort.

4.22 Decision-Tree for Insertion Sort on 3 Elements

The decision tree model provides an abstraction of comparison sorts, representing the sequence of comparisons that a sorting algorithm makes over all possible inputs of a given size.

4.23 Algorithm Analysis

4.23.1 Decision Tree

- A decision tree is an abstraction of any comparison sort.
- It represents the comparisons made by a specific sorting algorithm on inputs of a given size.
- This model abstracts away everything else such as control and data movement, focusing only on the comparisons.

4.23.2 Decision Tree Properties

- There are at least $n!$ leaves on the decision tree since every permutation of the input array appears at least once.
- The length of the longest path from root to leaf depends on the algorithm:
  - Insertion sort: $\Theta(n^2)$
  - Merge sort: $\Theta(n \log n)$

4.23.3 Lower Bound on the Height of Decision Trees

Theorem 4.2. *Any decision tree that sorts n elements has a height of $\Omega(n \log n)$.*

Proof. Consider the height h of the decision tree. There must be at least $n!$ leaves to account for every permutation of the input. Taking logarithms, we have $h \geq \log(n!)$. By Stirling's approximation, we know that $\log(n!)$ is $\Omega(n \log n)$, which implies the height h must also be $\Omega(n \log n)$. □

4.24 Figures

Figure 4.3: Decision tree for insertion sort on 3 elements.

4.25 Polynomial-Time Algorithms

Polynomial-time algorithms are significant in the study of computational complexity. They represent algorithms whose worst-case running time is a polynomial function of the size of the input.

- Polynomial-time algorithms: On inputs of size n, their worst-case running time is $O(n^k)$ for some constant k.
- The class $\mathcal{P}$ represents the set of problems solvable by a deterministic Turing machine in polynomial time. The "P" stands for "polynomial time."
- Problems in $\mathcal{P}$ are considered "easy" or "tractable" in terms of computation, as they can be solved efficiently.

4.26 Determinism and Nondeterminism

The concepts of determinism and nondeterminism are central to understanding computational complexity classes such as P and NP.

- In a deterministic computation, each step follows uniquely from the preceding step. Given the current state and the next input symbol, the next state is determined.
- In a nondeterministic machine, multiple choices may exist for the next state at any point. The machine has the flexibility to proceed along various possible paths during computation.
- The computation of a nondeterministic Turing machine is a tree whose branches correspond to different possible computation paths. If any branch leads to the accept state, the machine accepts its input.
- The class NP represents the set of problems solvable by a nondeterministic Turing machine in polynomial time. "NP" stands for "nondeterministic polynomial time."

4.27 The Class NP

The class NP consists of problems that are "verifiable" in polynomial time.

- If given a "certificate" of a solution, it could be verified that the certificate is correct in polynomial time relative to the size of the input to the problem.
- Example: For the satisfiability problem, a certificate would be an assignment of values to variables. It is possible to check in polynomial time that this assignment satisfies the formula in conjunctive normal form.

4.28 The Satisfiability Problem

The satisfiability problem is a central problem in computational complexity.

- It asks whether there is an assignment of variables that satisfies a given Boolean formula.
- The satisfiability problem is in NP because a given assignment can be verified efficiently.

4.29 The (k-CNF) Satisfiability Problem

- This is a specific type of satisfiability problem where the formula is in conjunctive normal form with k literals per clause.
- The complexity of this problem is significant in the study of NP-completeness.

4.30 P vs. NP Question

- The open question is whether P is a proper subset of NP.
- To date, no polynomial-time algorithm has been discovered for an NP-complete problem, nor has it been proven that no such algorithm can exist.
- The P vs. NP question has been one of the most profound and perplexing open research problems in theoretical computer science since it was first posed in 1971.

4.31 Figures

Figure 4.4: Computation tree of a nondeterministic Turing machine.

4.32 NP-Complete Languages

4.32.1 Decision Problems

Decision problems ask a yes-no question about the input. The class NP-complete consists of those languages for which the question can be answered in polynomial time by a nondeterministic Turing machine, and the answer can be verified in polynomial time by a deterministic Turing machine.

4.33 Polynomial Reducibility

A language A is polynomial-time reducible to language B (denoted as $A \leq_p B$) if there exists a polynomial-time computable function f such that for every string w, w is in A if and only if $f(w)$ is in B. This notion of reducibility is used to relate the complexities of different decision problems.

4.34 Function Reducing A to B

Given a function f that reduces A to B, if B can be decided quickly (in polynomial time), then so can A, since we can apply f to the instance of A and then solve the instance of B in polynomial time.

4.35 Decidability

Definition 4.1 (Decidability)**.** A language A is **decidable** if and only if there exists a Turing machine M that accepts input w if w is in A, and rejects w if w is not in A, for all w in Σ^*. Such a Turing machine is called a **decider**.

4.36 The Satisfiability Problem

The Satisfiability Problem (SAT) is one of the most well-known NP-complete problems. In 1971, Stephen Cook proved that SAT is NP-complete by showing that every problem in NP is polynomial-time reducible to SAT.

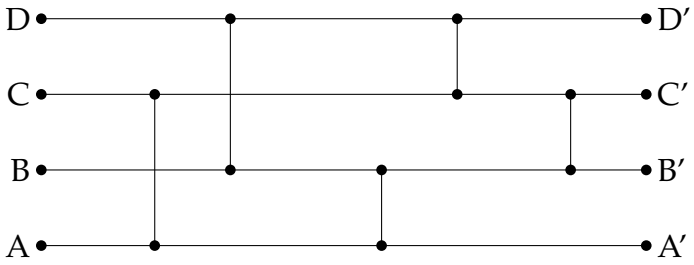
4.37 Figures

Chapter 5

Further Efficient Algorithms

Definition 5.1.

Theorem 5.1.



Chapter 6

Turing Machine

6.1 Deterministic Turing Machine, The class P

6.2 Non-deterministic Turing Machine, The class NP

Chapter 7

NP Problem

7.1 Decidability (The Halting Problem)

7.2 NP-completeness Theory