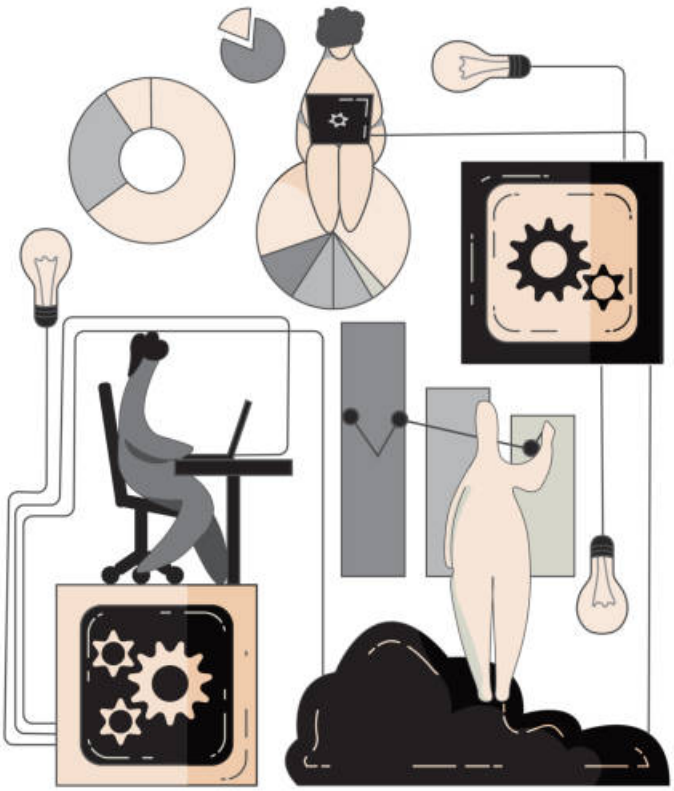
Algorithm Analysis

Ji Yong-Hyeon



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

December 10, 2023

Contents

Chapter 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

$$abcdefgh \longrightarrow defghabc.$$

1.1 A Juggling Algorithm

Algorithm 1: Juggling Algorithm

Data: Size n and a rotation count d
Result: Array $A[]$ rotated by d

```
g ← gcd(n, d)
for i ← 0 to g − 1 do
    temp ← A[i]
    j ← i
    while True do
        k ← j + d
        if k ≥ n then
            k ← k − n
        end
        if k == i then
            break
        end
        A[j] ← A[k]
        j ← k
    end
    A[j] ← temp
end
```

```c
// Function to calculate GCD
int gcd(int a, int b) {
    return b ? gcd(b, a%b) : a;
}

// Function to rotate the array
void jugglingRotation(int arr[], int n, int d) {
    d = d % n;  // In case d >= n
    int gcd_val = gcd(n, d);
    for (int i = 0; i < gcd_val; i++) {
        // Move i-th values of blocks
        int temp = arr[i];
        int j = i;

        while (1) {
            int k = j + d;
            if (k >= n) {
                k = k - n;
            }
            if (k == i) {
                break;
            }
            arr[j] = arr[k];
            j = k;
        }
        arr[j] = temp;
    }
}
```

1.2 The Block-Swap Algorithm

Algorithm 2: Block-Swap Alg.

Data: size n and a positions d
Result: Array $A[]$ rotated by d

Function Swap$(A[], start1, start2, blockSize)$**:**
  for $i \leftarrow 0$ **to** $blockSize - 1$ **do**
    $temp \leftarrow A[start1 + i]$
    $A[start1 + i] \leftarrow A[start2 + i]$
    $A[start2 + i] \leftarrow temp$
  end
if $d == 0 \vee d == n$ **then**
  return
end
$i \leftarrow d$
$p \leftarrow d$
$j \leftarrow n - d$
while $i \neq j$ **do**
  if $i > j$ **then**
    Swap$(A, p - i, p, j)$
    $i \leftarrow i - j$
  else
    Swap$(A, p - i, p + j - i, i)$
    $j \leftarrow j - i$
  end
end
Swap$(A, p - i, p, i)$

```c
void blockSwap(int A[], int start1, int start2, int blockSize) {
    int i, temp;
    for (i = 0; i < blockSize; i++) {
        temp = A[start1 + i];
        A[start1 + i] = A[start2 + i];
        A[start2 + i] = temp;
    }
}

void blockSwapRotate(int arr[], int n, int d) {
    int i, j, p;
    if (d == 0 || d == n) {
        return; // No rotation needed
    }

    i = p = d;
    j = n - d;
    while (i != j) {
        if (i > j) {
            blockSwap(arr, p - i, p, j);
            i -= j;
        } else {
            blockSwap(arr, p - i, p + j - i, i);
            j -= i;
        }
    }
    blockSwap(arr, p - i, p, i);
}
```

1.3 The Reversal Algorithm

Algorithm 3: Reversal Algorithm

Data: size n and a rotation count d
Result: Array $A[]$ rotated by d

Function `Reverse`$(A[], start, end)$**:**
  while $start < end$ **do**
    $temp \leftarrow A[start]$
    $A[start] \leftarrow A[end]$
    $A[end] \leftarrow temp$
    $start \leftarrow start + 1$
    $end \leftarrow end - 1$
  end
end
Function $RotateRotationA[], n, d$**:**
  `Reverse`$(A[], 0, d - 1)$
  `Reverse`$(A[], d, n - 1)$
  `Reverse`$(A[], 0, n - 1)$
end

```c
// Function to reverse the elements of the array from index 'start' to 'end'
void reverse(int arr[], int start, int end) {
    int temp;
    while (start < end) {
        temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

// Function to rotate the array 'arr' of size 'n' by 'd' elements to the left
void ReverseRotate(int arr[], int n, int d) {
    if (d == 0 || d == n) return; // No rotation needed
    d = d % n; // In case the rotation count is greater than array size
    reverse(arr, 0, d - 1);
    reverse(arr, d, n - 1);
    reverse(arr, 0, n - 1);
}
```

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

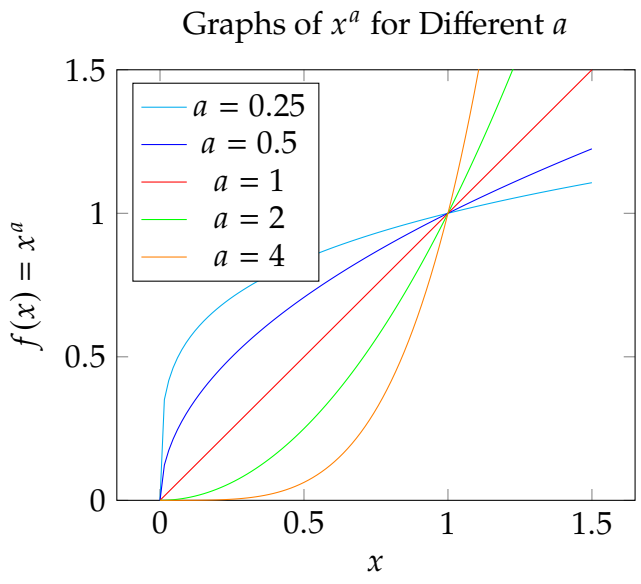
$$\Gamma_f := \left\{(x, y) \in \mathbb{R}^2 : y = f(x)\right\}.$$

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.

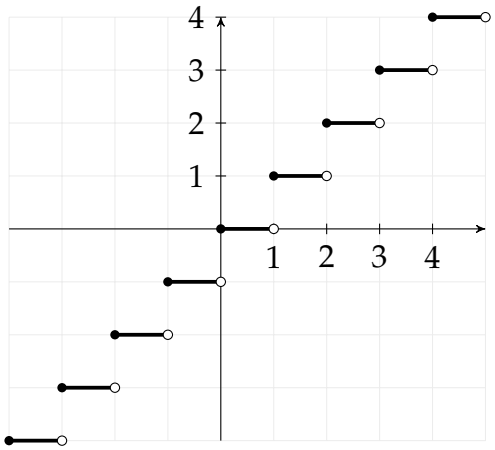


Figure 2.1: Graph of $f(x) = \lfloor x \rfloor$.

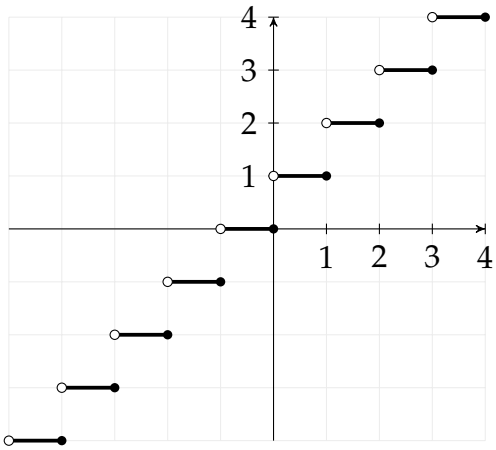


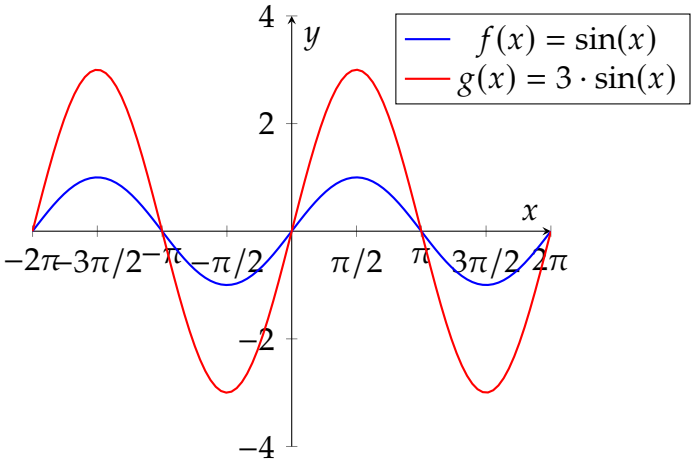
Figure 2.2: Graph of $f(x) = \lceil x \rceil$.

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 4: 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 5: 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
```

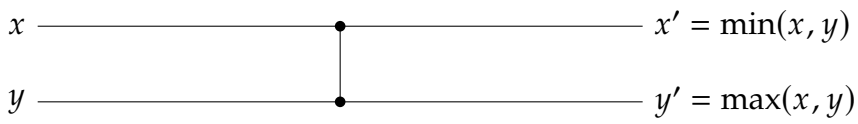
```
* [i j = 01 07] 02 92 54 99 05 47 20 66 66 72
* [i j = 01 06] 02 92 54 99 05 20 47 66 66 72
* [i j = 01 05] 02 92 54 99 05 20 47 66 66 72
* [i j = 01 04] 02 92 54 05 99 20 47 66 66 72
* [i j = 01 03] 02 92 05 54 99 20 47 66 66 72
* [i j = 01 02] 02 05 92 54 99 20 47 66 66 72
* [i j = 02 09] 02 05 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
* [i j = 03 09] 02 05 20 92 54 99 47 66 66 72
* [i j = 03 08] 02 05 20 92 54 99 47 66 66 72
* [i j = 03 07] 02 05 20 92 54 99 47 66 66 72
* [i j = 03 06] 02 05 20 92 54 47 99 66 66 72
* [i j = 03 05] 02 05 20 92 47 54 99 66 66 72
* [i j = 03 04] 02 05 20 47 92 54 99 66 66 72
* [i j = 04 09] 02 05 20 47 92 54 99 66 66 72
* [i j = 04 08] 02 05 20 47 92 54 99 66 66 72
* [i j = 04 07] 02 05 20 47 92 54 66 99 66 72
* [i j = 04 06] 02 05 20 47 92 54 66 99 66 72
* [i j = 04 05] 02 05 20 47 54 92 66 99 66 72
* [i j = 05 09] 02 05 20 47 54 92 66 99 66 72
* [i j = 05 08] 02 05 20 47 54 92 66 66 99 72
* [i j = 05 07] 02 05 20 47 54 92 66 66 99 72
* [i j = 05 06] 02 05 20 47 54 66 92 66 99 72
* [i j = 06 09] 02 05 20 47 54 66 92 66 72 99
* [i j = 06 08] 02 05 20 47 54 66 92 66 72 99
* [i j = 06 07] 02 05 20 47 54 66 66 92 72 99
* [i j = 07 09] 02 05 20 47 54 66 66 92 72 99
* [i j = 07 08] 02 05 20 47 54 66 66 72 92 99
* [i j = 08 09] 02 05 20 47 54 66 66 72 92 99
*
* Output: 02 05 20 47 54 66 66 72 92 99
**********************************************/
```

3.3 Sorting Networks

Comparator

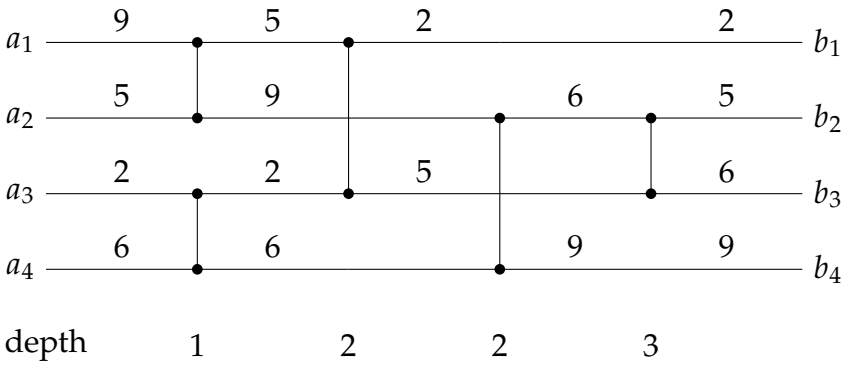
Definition 3.1. A **comparator** is a device with two inputs, x and y, and two outputs, x' and y', that performs the following function:

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



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.

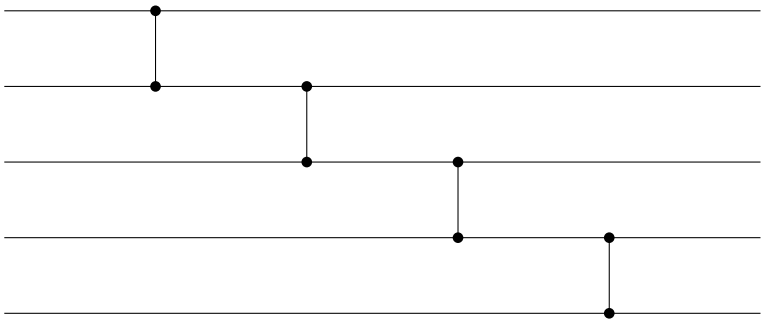
(2) If a comparator has two input wires with depths d_x and d_y, then its output wires have depth $\max(d_x, d_y) + 1$.

Remark 3.2.

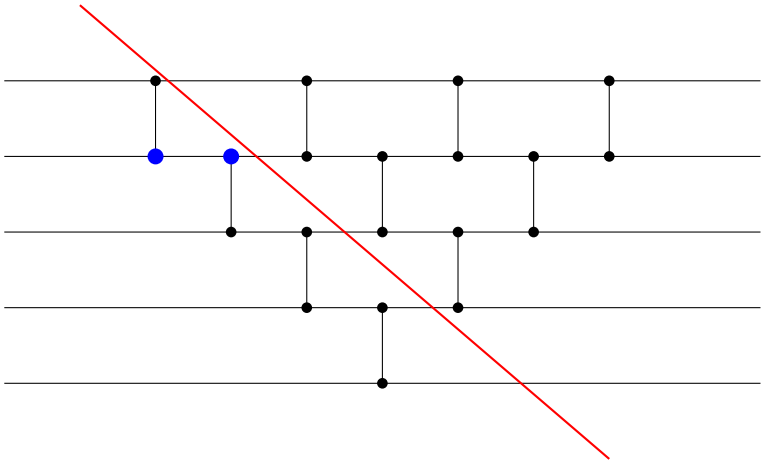
(1) **Depth of a Comparator** := depth of its output wire.

(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?"

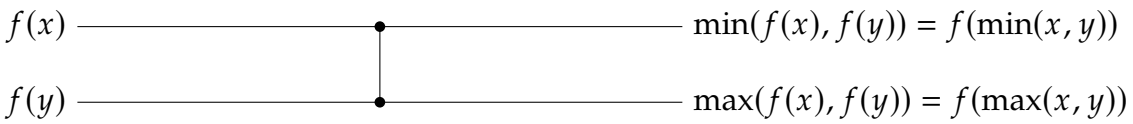
Lemma 3.1. *If a comparison network transforms*

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

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) Consider a wire at depth 0, an input wire a_i belonging to the input sequence a. When $f(a)$ is applied to the network, the input wire carries $f(a_i)$.

(ii) (Induction Step) **Induction Step:** Consider a wire at depth d, where $d \geq 1$. The wire is the output of a comparator at depth d, and the input wires to this comparator are at a depth strictly less than d. By the inductive hypothesis, if the input wires to the comparator carry values a_i and a_j when the input sequence a is applied, then they carry $f(a_i)$ and $f(a_j)$ when the input sequence $f(a)$ is applied. By our earlier claim, the output wires of this comparator carry $f(\min(a_i, a_j))$ and $f(\max(a_i, a_j))$. Since they carry $\min(a_i, a_j)$ and $\max(a_i, a_j)$ when the input sequence is a, the lemma is proved.

□

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, if we give the input $\langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$, then in the output we will have $f(a_i)$ after $f(a_j)$. But this results in an unsorted 0-1 sequence. A contradiction.

$$\begin{array}{cc} \langle a_1, a_2, \ldots, a_n \rangle & \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle \\ (a_i < a_j) & \\ \downarrow & \downarrow \\ \langle \ldots, a_j, \ldots, a_i, \ldots \rangle & \langle \ldots, f(a_j), \ldots, f(a_i), \ldots \rangle \\ & = 1 \quad = 0 \end{array}$$

□

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 3.3.

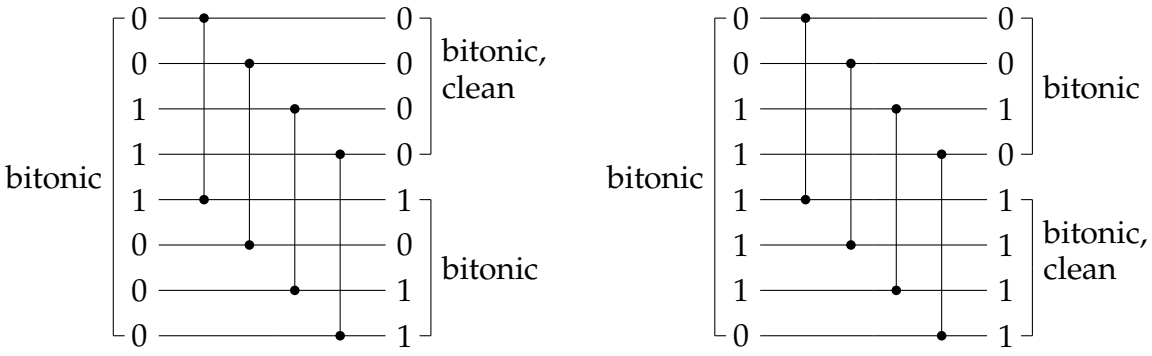


Figure 3.1: The comparison network Half-Cleaner[8].

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:*

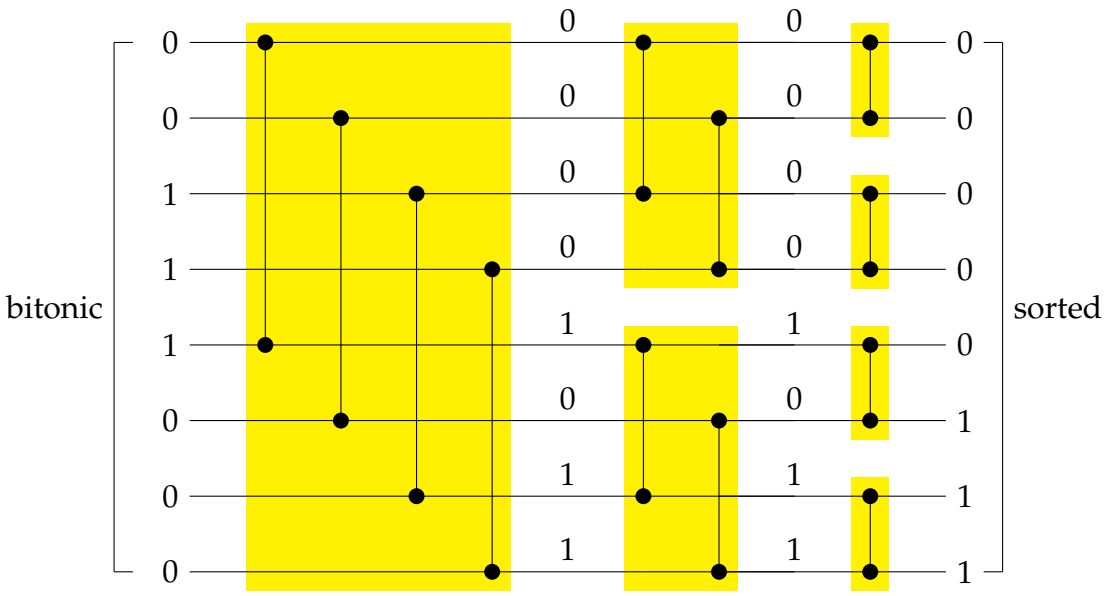
(1) both the top half and the bottom half are bitonic;

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

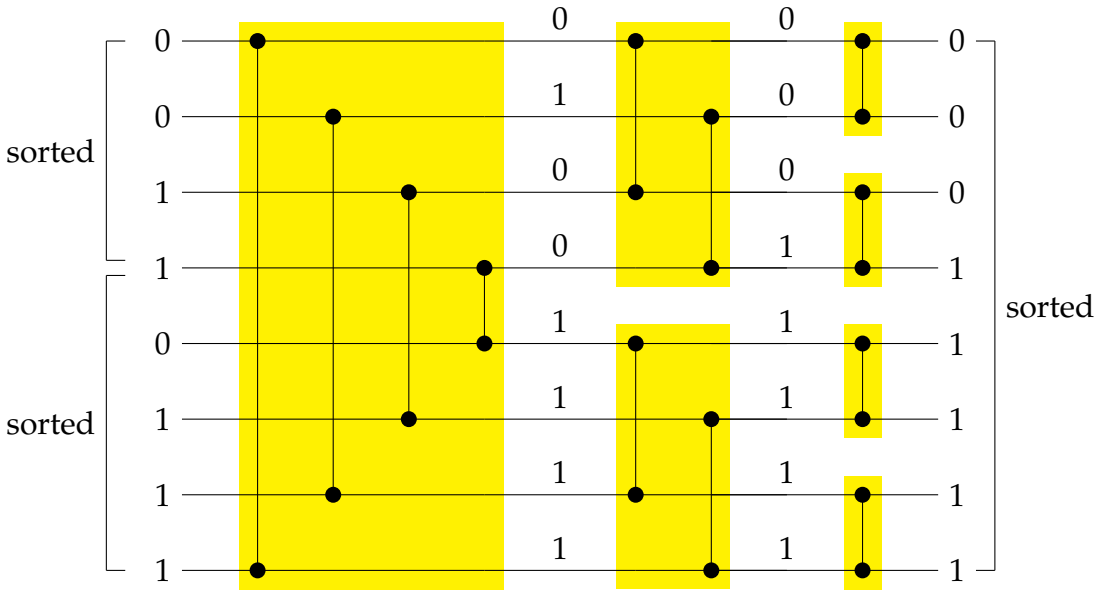
*(3) at least one half is **clean**(all 0's or all 1's).*

Proof. content... □

Example 3.4.



Example 3.5 (Merging Network)**.**



Chapter 4

Heap and Merge Sort

4.1 Heap-sort

- A Heap is a nearly complete binary tree, where each node contains some value.
- Heap-Property: the value of some node N is greater than or equal to the values of both children of N.
- Height of node = number of edges on the longest simple path from the node down to a leaf.
- Height of heap = height of root = $\Theta(\log n)$, where n is the number of nodes in the tree.

4.1.1 Heap as Datastructure

Consider an array:

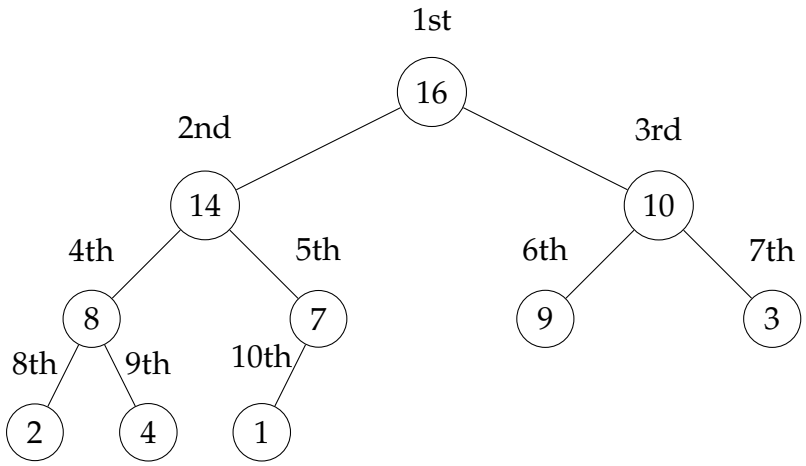
index	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
value	16	14	10	8	7	9	3	2	4	1

Interpretation of arrays as trees:

PARENT(i)
return $\left\lfloor \frac{i}{2} \right\rfloor$

LEFT(i)
return $2i$

RIGHT(i)
return $2i + 1$



There are Max-heaps and Min-heaps. The height is $\lfloor \log_2 n \rfloor$.

4.1.2 MAX-HEAPIFY

Algorithm 6: Max Heapify

Input: An array A and an index i
Output: A modified array where the subtree rooted at i satisfies the max-heap property

Function `MaxHeapify`(A, i)**:**
  $l \leftarrow LEFT(i)$
  $r \leftarrow RIGHT(i)$
  if $l \leq length[A]$ *and* $A[l] > A[i]$ **then**
    $largest \leftarrow l$
  else
    $largest \leftarrow i$
  end
  if $r \leq length[A]$ *and* $A[r] > A[largest]$ **then**
    $largest \leftarrow r$
  end
  if $largest \neq i$ **then**
    swap $A[i]$ and $A[largest]$
    `MaxHeapify`$(A, largest)$
  end

```c
void MaxHeapify(int arr[], int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < n && arr[l] > arr[largest])
        largest = l;

    if (r < n && arr[r] > arr[largest])
        largest = r;

    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        printHeap(arr, n); // Print the heap after each swap
        MaxHeapify(arr, n, largest);
    }
}
```

```
Original Heap:
[3 2 15 5 4 45 6 7 51 8 ]
--------------------------
[3 2 15 5 8 45 6 7 51 4 ]
[3 2 15 51 8 45 6 7 5 4 ]
[3 2 45 51 8 15 6 7 5 4 ]
[3 51 45 2 8 15 6 7 5 4 ]
[3 51 45 7 8 15 6 2 5 4 ]
[51 3 45 7 8 15 6 2 5 4 ]
[51 8 45 7 3 15 6 2 5 4 ]
[51 8 45 7 4 15 6 2 5 3 ]
Max-Heap:
[51 8 45 7 4 15 6 2 5 3 ]
```

MAX-HEAPIFY is used to maintain the max-heap property.

- Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
- Assume left and right subtrees of i are max-heaps.
- After MAX-HEAPIFY, the subtree rooted at i is a max-heap.
- Compare $A[i]$, A[LEFT(i)], and A[RIGHT(i)].

– If necessary, swap $A[i]$ with the larger of the two children to preserve heap property.

– Continue this process of comparing and swapping down the heap, until the subtree rooted at i is a max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

- Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
- Assume left and right subtrees of i are max-heaps.
- After MAX-HEAPIFY, subtree rooted at i is a max-heap.

Time / Comparisons: $O(\lg n)$.

4.1.3 Creating a Max-Heap using MAX-HEAPIFY

Algorithm 7: Build Max Heap

Input: An array A
Output: A max-heap formed from array A

```
1 Function BuildMaxHeap(A, n):
2     n ← length[A]
3     for i ← ⌊n/2⌋ − 1 to 0 do
4         MaxHeapify(A, n, i)
5     end
```

```c
void BuildMaxHeap(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        MaxHeapify(arr, n, i);
        printHeap(arr, n);
    }
}
```

```
Original Array:
[3 2 15 5 4 45 6 7 51 8 ]
--------------------------
[3 2 15 5 8 45 6 7 51 4 ]
[3 2 15 5 8 45 6 7 51 4 ]
[3 2 15 51 8 45 6 7 5 4 ]
[3 2 15 51 8 45 6 7 5 4 ]
[3 2 45 51 8 15 6 7 5 4 ]
[3 2 45 51 8 15 6 7 5 4 ]
[3 51 45 2 8 15 6 7 5 4 ]
[3 51 45 7 8 15 6 2 5 4 ]
[3 51 45 7 8 15 6 2 5 4 ]
[51 3 45 7 8 15 6 2 5 4 ]
[51 8 45 7 3 15 6 2 5 4 ]
[51 8 45 7 4 15 6 2 5 3 ]
[51 8 45 7 4 15 6 2 5 3 ]
Max-Heap:
[51 8 45 7 4 15 6 2 5 3 ]
```

4.1.4 Build-Max-Heap Correctness

- **Loop invariant:** At the start of every iteration of for loop, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.
- **Initialization:** By Exercise 6.1-7, we know that each node $\lfloor \frac{n}{2} \rfloor + 1$ to n is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor \frac{n}{2} \rfloor$ before the first iteration of the for loop, the invariant is initially true.
- **Maintenance:** Children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps. Correctly assuming that $i+1, i+2, \ldots, n$ are all roots of max-heaps, MAX-HEAPIFY makes node i a max-heap root. Decrementing i reestablishes the loop invariant at each iteration.
- **Termination:** When $i = 0$, the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

4.1.5 Complexity Build-Max-Heap

- Obviously $O(n \log n)$.
- Tighter Analysis:
  - Number of nodes of height h: $\leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$
  - Time/Comparisons:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

Then

$$\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2},\ |x| < 1 \implies O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \cdot \frac{1/2}{(1-1/2)^2} \right) = O(n).$$

4.1.6 Heapsort Algorithm

Heapsort(A, n)
```
BUILD-MAX-HEAP(A, n)
for i = n downto 2
exchange A[1] with A[i]
MAX-HEAPIFY(A, 1, i - 1)
```
Complexity:

- BUILD-MAX-HEAP: $O(n)$
- for loop: $n-1$ times / exchange elements: $O(1)$
- MAX-HEAPIFY: $O(\log n)$

Total time: $O(n \log n)$

4.2 Merge Sort

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 $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

4.2.1 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, however, just solve the subproblems in a straightforward manner.
- **Combine** the solutions to the subproblems into the solution for the original problem.

4.2.2 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.

Algorithm 8: Merge procedure of the Merge-Sort Algorithm

```
1  Function MERGE(A, p, q, r):
2      n1 ← q − p + 1;
3      n2 ← r − q;
4      Create arrays L[1..n1 + 1] and R[1..n2 + 1];
5      for i ← 1 to n1 do
6          L[i] ← A[p + i − 1];
7      end
8      for j ← 1 to n2 do
9          R[j] ← A[q + j];
10     end
11     L[n1 + 1], R[n2 + 1] ← ∞, ∞;
12     i, j ← 1, 1;
13     for k ← p to r do
14         if L[i] ≤ R[j] then
15             A[k] ← L[i]; i ← i + 1;
16         else
17             A[k] ← R[j]; j ← j + 1;
18         end
19     end
```

```c
void Merge(int A[], int p, int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;

    // Create L[] and R[] as temporary arrays
    int L[n1 + 1], R[n2 + 1];

    // Copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = A[p + i];
    for (int j = 0; j < n2; j++)
        R[j] = A[q + 1 + j];

    // Sentinel values
    int INT_MAX = 10000;
    L[n1] = INT_MAX;
    R[n2] = INT_MAX;

    int i = 0; // Initial index of first subarray
    int j = 0; // Initial index of second subarray

    // Initial index of merged subarray
    for (int k = p; k <= r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i];
            i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}
```

4.2.3 Merge-sort Algorithm

Algorithm 9: Merge-Sort algorithm procedure

```
/* Initial call: Merge-Sort(A, 1, n)                    */
Function Merge-Sort(A, p, r):
    if p < r then
        q ← ⌊(p + r)/2⌋
        Merge-Sort(A, p, q)
        Merge-Sort(A, q+1, r)
        Merge(A, p, q, r)
    end
```

4.3 Stability in Merge Sort

Stability in a sorting algorithm ensures that the relative order of equal elements is preserved in the sorted output. Formally, this can be expressed as follows:

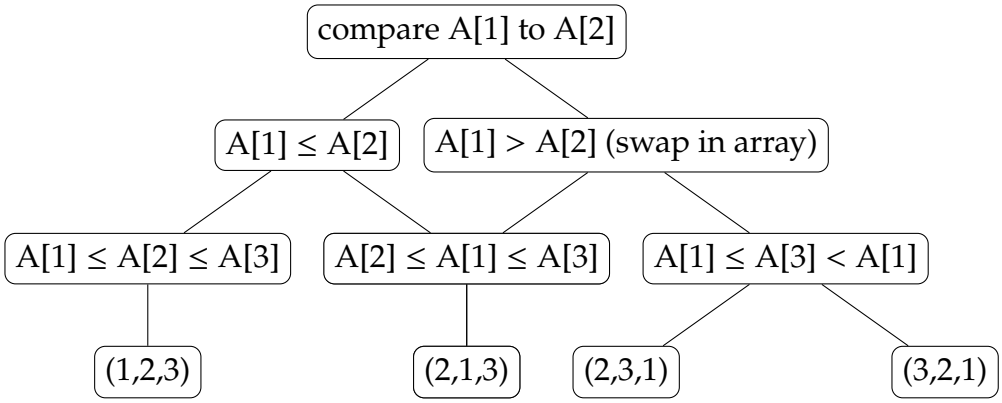
Let $A = [a_1, a_2, \ldots, a_n]$ be a list of elements to be sorted, and let f denote the sorting function. The sorting algorithm is said to be stable if for every pair of indices i, j with $1 \leq i < j \leq n$ and $a_i = a_j$, it holds that $f(A)_k = a_i$ and $f(A)_{k+1} = a_j$ for some k. Here, $f(A)$ represents the sorted list.

- A is the original list.
- $f(A)$ is the sorted list.
- If a_i and a_j are equal and a_i appears before a_j in A, then a_i should also appear before a_j in $f(A)$.

This definition succinctly captures the essence of stability in sorting algorithms.

4.4 Lower Bound for Sorting

Decision-tree for insertion sort on 3 elements:



4.4.1 Decision tree

How many leaves on the decision tree? There are at least $n!$ leaves, because every permutation appears at least once.

Q: What is the length of the longest path from root to leaf?
A: Depends on the algorithm. Examples

- Insertion sort: $\Theta(n^2)$
- Merge sort: $\Theta(n \log n)$

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

Proof. Take logs: $h \geq \lg(n!)$. Use Stirling's approximation: $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

$h \geq \lg\left(\sqrt{2\pi n}\right) + n \lg\left(\frac{n}{e}\right)$
$= n \lg n - n \lg e$
$= \Omega(n \lg n)$. □

Chapter 5

Complexity Theory

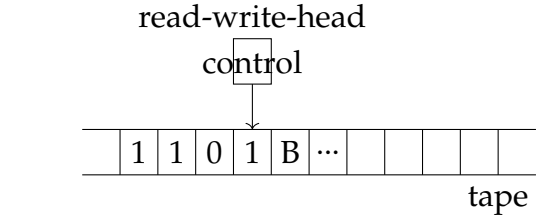
5.1 Turing Machines

- TM uses an infinite tape.
- TM has a tape head.
- Initially the tape contains only the input string and is blank everywhere else.
- TM continues computing until it decides to produce an output.
- "accepting state"
- The computation time of a TM on an input corresponds with the number of the computation steps carried out.
- Algorithms can be described by TMs.
- Deterministic and nondeterministic TM

5.2 Deterministic Turing Machine

Def. A $k(k \geq 1)$-tape deterministic Turing machine TM is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, B, q_f)$, where Q, Σ, Γ are all finite sets and Q is the set of states. Γ is the tape alphabet. B is the blank, a symbol in Γ. Σ is the input alphabet not containing the blank symbol B and is a subset of Γ. $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, S\}^k$ is the transition function (the partial function). $q_0 \in Q$ is the start state, and $q_f \in Q$ is the accept state

In Case of $k = 1$:



contains assignments (instructions) given by the transition function δ.

The transition function δ of a Turing machine is defined as follows:

$$\delta(q,(a_1,a_2,\ldots,a_k)) = (q',(a_1',a_2',\ldots,a_k'),(m_1,m_2,\ldots,m_k))$$

with $a_1',a_2',\ldots,a_k' \in \Gamma$ and $m_1,m_2,\ldots,m_k \in \{L,R,S\}$

5.2.1 Definition

A TM-computation is a finite sequence $B = \langle K_0,\ldots,K_n\rangle$ of TM-configurations, such that $\forall i < n$ each K_i yields K_{i+1}. K_n is the halting configuration if there is no succeeding configuration of K_n.

If K_n is a halting configuration, then the computation B needs n computation steps.

A TM-computation $B = \langle K_0,K_1,\ldots,K_n\rangle$ is complete, if K_n is a halting configuration.

A complete computation $B = \langle K_0,K_1,\ldots,K_n\rangle$ is an accepting computation, if the state of K_n is the accepting state q_f.

Let $X \in \Sigma^*$ be any word. A Turing machine TM accepts X, if $\exists$ an accepting computation for X.

The set $L(TM) = \{X \in \Sigma^* | TM \text{ accepts } X\}$ is called the language accepted by TM.

5.3 Deterministic Polynomial Computability - The class $\mathcal{P}$

- 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 that can be solved by a deterministic Turing machine (algorithm) 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.

5.4 Nondeterministic Turing machine

Def. A $k(k \geq 1)$-tape nondeterministic Turing machine NT is a 7-tuple $(Q,\Sigma,\Gamma,\delta,q_0,B,q_f)$, where Q,Σ,Γ are all finite sets and Q is the set of states. Γ is the tape alphabet. B is the blank, a symbol in Γ. Σ is the input alphabet not containing the blank symbol B and is a subset of Γ. $P \subseteq Q \times \Gamma^k \times Q \times \Gamma^k \times \{L,R,S\}^k$ is a transition relation. $q_0 \in Q$ is the start state, and $q_f \in Q$ is the accept state.

5.4.1 Determinism and Nondeterminism

In a deterministic computation, each step of a computation follows in a unique way from the preceding step. When the machine is in a given state and reads the next input symbol, we know what the next state will be—it is determined.

In a nondeterministic machine, multiple choices may exist for the next state at any given point. Throughout a computation, the machine has the flexibility to proceed along various possibilities.

5.4.2 Nondeterministic Turing Machine

The computation of a nondeterministic Turing machine is a tree whose branches correspond to different possible computation paths for the machine.

If any branch of the computation leads to the accept state, the machine accepts its input.

5.5 The class NP

Def. The class $\mathcal{NP}$ is the set of all languages which are acceptable (recognizable) by a polynomial time nondeterministic Turing machine.

Formally,

$\mathcal{NP} = \{S \subseteq \{0,1\}^* \mid \exists$ a polynomial time nondeterministic Turing machine NT for which $L(\text{NT}) = S\}$

5.5.1 Nondeterministic Polynomial Computability - The class $\mathcal{NP}$

Also we can say: The class $\mathcal{NP}$ represents the set of problems that can be solved by a nondeterministic Turing machine in polynomial time. The "NP" stands for "nondeterministic polynomial time."

5.5.2 Nondeterministic Polynomial Computability

The class $\mathcal{NP}$ consists of those problems that are "verifiable" in polynomial time i.e., if we were somehow given a "certificate" of a solution, then we could verify that the certificate is correct in polynomial time in the size of the input to the problem.

5.5.3 The class $\mathcal{NP}$

Ex. For satisfiability problem, a certificate would be an assignment of values to variables. We could check in polynomial time that this assignment satisfies the formula in conjunctive normal form (See the following slides).

The (k-CNF) Satisfiability Problem Boolean logic (Boolean algebra)

- Let $v = \{p_1, p_2, \ldots, p_m\}$ be a set of logical variables.
- A truth assignment for v is a function $t : v \to \{T, F\}$.
- If $t(p) = T$ we say that p is "true" with respect to t; If $t(p) = F$ we say that p is "false".
- If $p \in v$, then p and $\neg p$ are called "literals" over v.

5.6 The Satisfiability Problem

- The literal p is true with respect to t if and only if the variable p is true with respect to t.
- The literal $\neg p$ is true with respect to t if and only if the variable p is false.
- A clause over v is a finite disjunction $q_{i_1} \vee q_{i_2} \vee \ldots \vee q_{i_k}$, where each q_{i_j} $(j = 1, \ldots, k)$ is a literal.
- A clause is satisfied by a truth assignment $\leftrightarrow$ at least one of its members is true with respect to the assignment.
- An expression (formula) F is in conjunctive normal form (CNF) if it is a finite conjunction of clauses:

$$F = \bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{n_i} q_{ij} \right) = (q_{11} \vee q_{12} \vee \ldots \vee q_{1n_1}) \wedge \ldots \wedge (q_{m1} \vee q_{m2} \vee \ldots \vee q_{mn_m})$$

- A formula F is satisfiable $\leftrightarrow$ there exists some truth assignment for v that simultaneously satisfies all clauses in F.

Definition:

The satisfiability problem is specified as follows: Given a formula F in conjunctive normal form over a set v of variables, is there a truth assignment that satisfies F? (Is F satisfiable?)

Language E

Def. E is the set of satisfiable formulas in conjunctive normal form.

Theorem 5.1. *The satisfiability problem is in NP.*

Placeholder for Korean text translation. □

5.7 The class P and The class NP

Inclusion of P in NP

Any deterministic Turing machine is automatically a nondeterministic Turing machine. So the class $\mathcal{P}$ is a subset of the class $\mathcal{NP}$.

Examples include the traveling salesman problem, knapsack, satisfiability problem, clique problem, etc.

Open Questions

- The open question is whether or not $\mathcal{P}$ is a proper subset of $\mathcal{NP}$.
- No polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them.

P vs NP Problem

- This so-called $\mathcal{P} \neq \mathcal{NP}$ question has been one of the deepest, most perplexing open research problems in theoretical computer science since it was first posed in 1971.

Chapter 6

Summary

6.1 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 $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

6.2 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.

6.3 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.

6.4 Merge Procedure

[Details of the merge procedure]

6.5 Example Merge Procedure

[Details of the example merge procedure]

6.6 Correctness of Merge Procedure

- Loop invariant: [Description of the loop invariant]

6.7 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.

6.8 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 $\Rightarrow$ $cn = \Theta(n)$.

6.9 Merge-Sort Recurrences

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

6.10 Figures

Figure 6.1: Example figure caption.

6.11 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.

6.11.1 Example of Merging

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

6.12 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.

6.12.1 Stability Example

6.12.2 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.

6.13 Case Study: Radix Sort

6.13.1 Radix Sort Process

6.13.2 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 6.1. *The merge-sort algorithm is stable under the condition that the merge operation is implemented in a stable manner.*

6.14 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.

6.15 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.

6.16 Creation of Partitions (Divide Step)

6.16.1 Loop invariant

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

6.16.2 Time for partitioning

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

6.17 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 $A[p \ldots i] \leq$ pivot, $A[i+1 \ldots r-1] >$ pivot, and $A[r]$ = pivot.

6.18 Performance Cases

6.18.1 Worst-Case Scenario

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

6.18.2 Best-Case Scenario

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

6.18.3 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 $T(n) \leq T(9n/10) + T(n/10) + \Theta(n) = O(n \log n)$.

6.19 Randomized Version of Quicksort

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

6.20 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.

6.21 Figures

Figure 6.2: Detailed recursion tree of quicksort.

6.22 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.

6.23 Algorithm Analysis

6.23.1 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.

6.23.2 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: $\Theta(n \log n)$

6.23.3 Lower Bound on the Height of Decision Trees

Theorem 6.2. *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 $\Omega(n \log n)$. □

6.24 Figures

Figure 6.3: Decision tree for insertion sort on 3 elements.

6.25 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.

6.26 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."

6.27 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.

6.28 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.

6.29 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.

6.30 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.

6.31 Figures

Figure 6.4: Computation tree of a nondeterministic Turing machine.

6.32 NP-Complete Languages

6.32.1 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.

6.33 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.

6.34 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.

6.35 Decidability

Definition 6.1 (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**.

6.36 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.

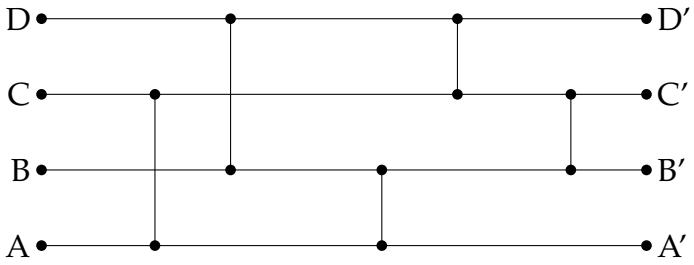
6.37 Figures

Chapter 7

Further Efficient Algorithms

Definition 7.1.

Theorem 7.1.



Chapter 8

Turing Machine

8.1 Deterministic Turing Machine, The class P

8.2 Non-deterministic Turing Machine, The class NP

Chapter 9

NP Problem

9.1 Decidability (The Halting Problem)

9.2 NP-completeness Theory