Chapter 4. Algorithms

An *algorithm* is a step-by-step method of solving some problem.

컴퓨터를 이용하여 주어진 문제를 해결하는 방법은 여러 가지가 있을 수 있다. Algorithm 이란 특정한 일을 수행하는 명령어들의 유한 집합을 의미하며, 여러 algorithm 중에서 가장 효율적인 algorithm 을 찾는 것이 중요하다. 수학에서 문제를 풀기 위하여 정의나 정리들을 이용하는 것과 같이 컴퓨터에서는 algorithm 을 사용한다.

4.1 Introduction

Properties of algorithms

(1) Input / output (입력/출력)
(2) Precision (명확성): 각 단계는 정확하게 기술되어야 한다.
(3) Determinism (결정성): 단계별로 결과는 입력 값에만 영향을 받아야 하며, 각 단계가 실행된 후에는 결과가 확정되어야 한다.
(4) Finiteness (유한성): 유한번의 명령이 수행된 후에는 반드시 끝나야 한다.
(5) Correctness (정확성): 주어진 문제를 정확하게 해결해야 한다.
(6) Generality (일반성): 같은 유형의 문제에 모두 적용될 수 있어야 한다.
(7) Effectiveness (효율성): 정확하면서도 효율적이어야 한다.

Representation of algorithms – pseudocode

Algorithm 은 순서도(flow chart), 유사코드(pseudocode), 언어(language, such as C++ and Java) 등의 여러 방법으로 표현될 수 있는데, 누구나 이해할 수 있도록 명확하게 기술되어야 하는 것이 중요하다. 유사코드는 다음과 같은 표현식을 사용한다.

(1) assignment operator: $x = y$
(2) if (condition) action
(3) arithmetic operators: $+ \; - \; * \; /$
(4) relational operators: $==, -=, <, >, \leq, \geq$
(5) logical operators: $\wedge, \vee, \sim$
(6) while (condition) action
(7) for var = init to limit, action

Algorithm 4.1: Finding the maximum of 3 numbers

```
def max3(a, b, c):
        large = a
        if b > large:
                large = b
        if c > large:
                large = c
        return large
```

- Algorithm 이 명확한지를 판별하는 한가지 방법으로, 실험적인 입력 값을 사용할 수 있다. Algorithm 의 변수 값을 추적하여, algorithm 구성에 사용한 논리가 적절한 가를 해 가는 방법이다.

- Tracing of algorithm 4.1: Choose $a = 6, \; b = 1, \; c = 9$

Algorithm 4.2: Finding the maximum value in a sequence of numbers

```
def max_sequence(s, n):
        # s ... list of length 'n'
        large = s[0]
        for i in range(1,n):
                if s[i] > large:
                        large = s[i]
        return large
```

- Input 's' is a list (in Python): *e.g.*, $s = [2, 11, 7, 4, 3, 9]$
- Index in Python starts from 0 as in C/C++, not from 1 as in FORTRAN and MATLAB.

4.2 Examples of algorithms

Algorithm 4.3: Text search: This algorithm searches for an occurrence of the pattern p in the text t. It returns the smallest index i such that p occurs in t starting at index i. If p does not occur in t, it returns 0.

```
def text_search(p, m, t, n):
        # p: pattern (in string) to search, m: length of p,
        # t: text (in string) to be examined, n: length of t
        for i in range(n-m+1):
                j = 0
                while t[i+j] == p[j]:
                        j = j + 1
                        if j == m:
                                return i
        return 0
```

```
def text_search2(p, m, t, n):
        for i in range(n-m+1):
                if t[i:i+m] == p:
                        return i
        return 0
```

- Inputs 'p (pattern)' and 't(text)' are strings: *e.g.*, t = 'discrete mathematics' and p = 'mat'.
- Tracing of algorithm 4.3: Choose p = "001" and t = "010001"

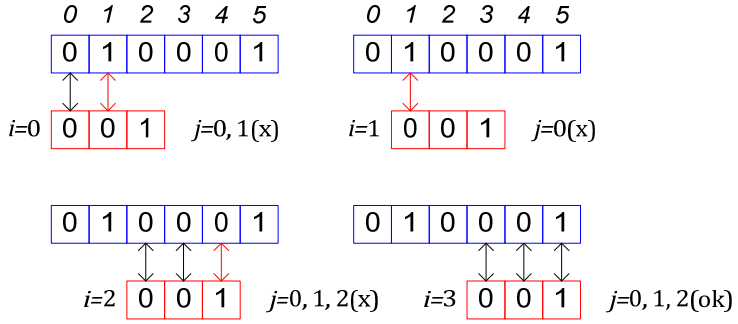

Figure 4-1 Graphical representation of tracing routine for algorithm 4.3

Sorting 정렬

To sort a sequence is to put it in some specified (ascending 오름차순 or descending 내림차순) order. Many sorting algorithms have been devised. Which algorithm is preferred in a particular application depends on factors such as the size of the data and how the data are represented. The insertion sort (삽입정렬) algorithm is known to be one of the fastest algorithms for sorting small sequences (≤ 50).

Suppose we want to sort a sequence of data, $s_1, s_2, \ldots, s_n$ in ascending order. The basic idea of the insert sorting is as follows:

(1) Compare data s_1 and s_2 to output a sorted sequence s_1, s_2.
(2) Assume that the part of the sequence $s_1, s_2, \ldots, s_i$ is in ascending order.
(3) Given the next data s_{i+1}, insert s_{i+1} in $s_1, s_2, \ldots, s_i$ so that $s_1, s_2, \ldots, s_{i+1}$ is in ascending order.

For example, if we have a sorted sequence $\{8, 13, 20, 27\}$ and a new data $s_5 = 16$, we compare s_5 and each element in the sorted sequence to find the proper place to put s_5.

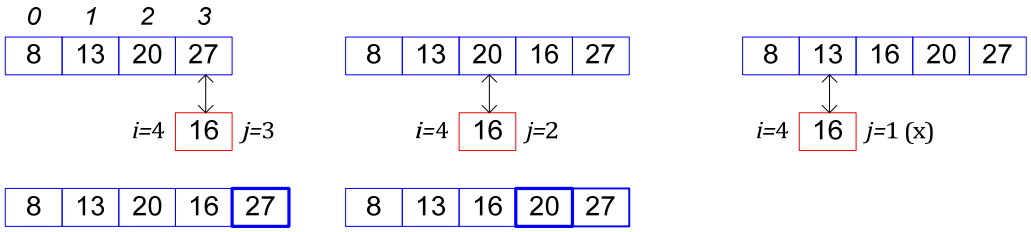

Figure 4-2 Demonstration of the insertion sorting

Algorithm 4.4: Insertion sort: This algorithm sort the sequence $s_1, s_2, \ldots, s_n$ in ascending order.

```
def insertion_sort(s, n):
    # s: a list containing data to be sorted, n: number of data in 's'
    for i in range(1, n):
        val = s[i]              # data to be compared
        j = i - 1               # j is s-index
        while (j>=0) and (val <= s[j]):      # if s[i] < s[j], insert s[i] prior to s[j]
            s[j+1] = s[j]
            j = j - 1
        s[j+1] = val
    return s
```

Algorithm 4.5: Bubble sort: This algorithm sort the sequence $s_1, s_2, \ldots, s_n$ in ascending order.

Given a sequence of data, $s_1, s_2, \ldots, s_n$, the bubble sort successively compare two adjacent data and change position if necessary.

(1) Starting from first two data s_1 and s_2, sort them. Then, compare s_2 and s_3 and so forth, until the last two are sorted. After the first loop, the largest element is in the last position.
(2) Repeat the 1st step for the $(n-1)$ data.

```
def bubble_sort(s, n):
    # Outer-loop for 'n-1' times, while the inner loop compares one- pair of data at a time.
    for i in range(n-1):
        for j in range(n-i-1):
            if s[j] > s[j+1]:
                s[j], s[j+1] = s[j+1], s[j]
    return s
```

Complexity of algorithms

Algorithm 은 정확한 출력 값을 계산해야 함은 물론 이를 효율적으로 처리해야 한다. Algorithm 의 효율성은 연산하는데 필요한 시간(time: number of steps, 연산량)과 기억장소의 크기(space: *e.g.* the number of variables)로 평가한다.

특히 시간은 동일한 문제를 해결하는 들의 효율성을 비교하는 중요한 척도가 된다. 예를 들어, 한 algorithm 이 하나의 문제를 풀기 위하여 n steps 을 필요로 하는데 비하여, 다른 algorithm 은 n^2 steps 을 필요로 한다면 우리는 자연스럽게 첫 번째 algorithm 을 선호하게 될 것이다. Algorithm 4.4 에서는 $for-loop$가 항상 $(n-1)$번 반복 수행되지만, 특정한 i 값에서 $while-loop$가 수행되는 횟수는 입력에 따라 달라진다. 따라서, 값이 고정 되더라도, algorithm 4.4 의 연산량은 입력에 따라 변화한다. 이러한 경우에 우리는 두 가지 경우를 고려할 수 있다: best and worst cases.

(1) best-case time: When the input sequence is already sorted, then the $while-loop$ will never be executed.

(2) worst-case time: When the input sequence is sorted in decreasing order, then the $while-loop$ will execute $(i-1)$ times for the ith iteration of $for-loop$.

이에 비하여, algorithm 4.5 에서 요구하는 연산량은 항상 일정하다. 즉, 비교 연산을 기준으로 할 때, 첫 번째 $for-loop$에서 $(n-1)$번, 두 번째 $for-loop$에서 $(n-2)$번 등, 총 $(n-1)(n-2)/2$번의 비교 연산이 수행 됨을 알 수 있다.

4.2 Analysis of algorithms

Algorithm 분석은 그 algorithm 을 실행하는 데 필요한 연산량을 측정하는 것이다.

Example 4.1 Given a set X of n elements, each of which are labeled to either 'red' or 'black', suppose we want to find the number of subsets of X that contains at least one red element.

$|X| = n$이면, $|P(X)| = 2^n$이므로, 검사 algorithm 은 2^n 단위의 연산량을 요구한다고 볼 수 있다.

Algorithm 의 연산량은 일반적으로 입력의 크기 n에 비례하여, n의 함수, 특히 다항식으로 표현된다. Table 4.1 은 입력 크기 n에 따른 연산량과 연산 시간을 보여주고 있다 (단, 하나의 연산에 10^{-6} 시간이 소요된다고 가정한다).

예를 들어, 연산량은 입력의 크기 n을 변수로 하여 다음과 같이 표현될 수 있다:

$$t(n) = 60n^2 + 5n + 1$$

위에서와 같이 연산량이 n의 다항식으로 표현되는 경우, n의 증가와 함께, n^2의 증가가 월등히 빠르기 때문에 나머지 부분을 무시하고 "연산량이 $60n^2$에 비례한다"라고 말할 수 있다. We say that $t(n)$ is of order n^2 and write $t(n) = \Theta(n^2)$ (big-Oh 라 읽는다).

Table 4.1 입력 크기 n에 따라 연산량과 연산 시간 (10^{-6} [sec/step]을 가정)

time \ n	5	10	50	10^2	10^3
1	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}
$\log_2 n$	2×10^{-6}	3×10^{-6}	6×10^{-6}	7×10^{-6}	10^{-5}
n	5×10^{-6}	10^{-5}	5×10^{-5}	10^{-4}	10^{-3}
$n \cdot \log_2 n$	10^{-5}	3×10^{-5}	3×10^{-4}	7×10^{-4}	10^{-2}
n^2	3×10^{-5}	10^{-4}	3×10^{-3}	10^{-2}	1
n^3	10^{-4}	10^{-3}	10^{-1}	1	16.7 min
2^n	3×10^{-5}	10^{-3}	36 yrs	4×10^{16} yrs	3×10^{287} yrs
$n!$	10^{-4}	3.6	9.7×10^{50} yrs	?	?

Definition Let f and g be function defined on N. We write

(4.1) $f(n) = O(g(n))$

and say that $f(n)$ is of order $g(n)$ or $f(n)$ is *big oh* of $g(n)$, if there exists a constant $C > 0$ such that

(4.2) $|f(n)| \le C|g(n)|$

for all but finitely many positive integers n.

Table 4.1 에서 알 수 있듯이, algorithm 의 연산량은 다음과 같은 관계를 갖는다.

$$O(1) < O(\log_2 n) < O(n) < O(n \cdot \log_2 n) < O(n^2) < O(n^3) < \cdots < O(2^n) < O(n!)$$

일반적으로 $O(n^2)$ 이상의 연산량을 요구하는 algorithm 은 n값의 증가와 함께 연산량이 급격히 증가하여 컴퓨터에서 수행이 거의 불가능하다고 볼 수 있다.

Theorem 4.1 Let

(4.6) $p(n) = a_k n^k + \cdots + a_1 n + a_0$

be a polynomial in n of degree k, where each a_i is non-negative. Then, $p(n) = O(n^k)$.

Example 4.2 (a) $t_a(n) = 2n + 3\log_2 n$:$O(n)$

(b) $t_b(n) = \log_b n$:$O(\log_b n)$

(c) $t_c(n) = 1 + 2 + \cdots + n$:$O(n^2)$

(d) $t_d(n) = 1^k + 2^k + \cdots + n^k,\ k \in Z^+$:$O(n^{k+1})$

Example 4.3 Consider the number of times the statement $x = x + 1$ is executed in the following algorithm:

```
x = 0
for i = 1 to n
    for j = 1 to i
        x = x + 1
```

Example 4.4 Consider the number of times the statement $x = x + 1$ is executed in the following algorithm:

```
x = 0
i = n
while (i ≥ 1) {
    x = x + 1
    i = ⌊i/2⌋
}
```

Algorithm 4.6: Given two $(n \times n)$ matrices A and B, let $\mathrm{C} = \mathrm{AB}$. Then

(4.7) $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$, for $1 \le i, j \le n$

```
input: A, B, n
matrix_product(A, B, n) {
    for i = 1 to n {
        for j = 1 to n {
            c_ij = 0
            for k = 1 to n
                c_ij = c_ij + a_ik * b_kj
        }
    }
    return 0
}
```

4.3 Recursive algorithms 재귀적 알고리즘

A recursive function is a function that invokes itself. A recursive algorithm is an algorithm that contains a recursive function.

Example 4.5 Let $f(n) = n!$ defined on N. Then, since $n! = n \cdot (n-1)!$, we have

$$f(n) = nf(n-1)$$

which is a recursive function.

Algorithm 4.7: Computing n factorial

```
input: n
factorial(n) {
    if (n ≤ 0) return 1
    return n * factorial(n − 1)
}
```