알고리즘및실습

제1장 알고리즘 개요

1. 알고리즘

알고리즘(algorithm)이란 홀로 어떤 목적을 위해 수행하는 일련의 단계를 말한다. 컴퓨터공학에서는 어떤 문제(problem)에 대해 답을 찾아 주는 일련의 단계를 알고리즘이라 한다. 여기서 문제는 입력과 출력을 통해 정의한다. 예를 들어 검색 문제은 다음과 같이 정의할 수 있다.

 배열 검색 문제

- 입력. n개의 정수로 구성된 배열 A와 정수 v
- 출력. v가 배열에 있으면 `true` 없으면 `false`

이처럼 알고리즘은 어떤 입력이 주어졌을 때 기대한 값을 계산하거나 찾아주는 일련 단계이다.

알고리즘은 다음 3가지 특성을 가지고 있어야 한다.

- 특성 1. (definiteness). 알고리즘은 명확해야 한다.
- 특성 2. (finiteness). 알고리즘은 유한성(또는 종결성) 특성을 가지고 있어야 한다.
- 특성 3. (correctness). 알고리즘은 정확해야 한다.

특성 1은 수행 과정이 모호하지 않아야 한다는 것을 말한다. 또 알고리즘은 그것을 수행했을 때 궁극에는 종료해야 한다. 물론 종료하는데 매우 오래 걸릴 수 있다. 문제를 해결하지만 너무 많은 시간이 걸리는 알고리즘은 실제 활용하기 어렵다. 알고리즘은 입력이 주어졌을 때 기대하는 출력을 주어야 한다. 정확하지 않은 알고리즘은 사용하기 어렵다. 하지만 정확한 알고리즘이지만 시간이 너무 오래 걸리면 근사값을 찾아주는 알고리즘을 대신 사용할 수 있다. 따라서 이 특성들은 갖추길 희망하는 특성이지 반드시 갖추어야 하는 것은 아니며, 갖추지 못한다고 제시된 일련의 단계가 알고리즘이 아니라고 말할 수 없다.

정확성과 관련된 강건성(soundness)이라는 개념과 완전성(completeness)이라는 개념도 있다. 강건한 알고리즘이 제시하는 해는 항상 정확하다. 예를 들어 100이하의 정수에 대해 소수를 정확하게 판정해 주지만 100보다 큰 수에 대해서는 답을 주는 알고리즘은 강건한 알고리즘이다. 완전한 알고리즘은 정확하지 않은 해를 제시할 수 있지만 해가 있는 경우에는 항상 정확한 해를 제시한다. 예를 들어 모든 소수에 대해 항상 참을 주지만 일부 합성수에 대해서도 참을 주는 소수 판정 알고리즘은 완전한 알고리즘이다.

정확성은 부분적 정확성(partial correctness), 완벽 정확성(total correctness)으로 구분하며, 부분적으로 정확한 알고리즘이 제시하는 답은 항상 정확하지만 종료를 하지 않을 수 있다. 완벽하게 정확한 알고리즘은 항상 올바른 답을 주며 반드시 종료한다.

알고리즘을 일상 자연어로 서술할 수 있지만 이 서술로부터 코드를 작성하기 어렵고, 복잡한 알고리즘을 표현하기 힘들기 때문에 보통 알고리즘은 의사코드(pseudocode)를 이용하여 서술한다. 이 교재에서는 언어와 독립적인 의사코드를 통해 서술하며, 사용하는 의사코드는 파이썬과 유사하다.

이 교재에서 사용하는 의사코드는 기호를 다음과 같이 이용하며,

- $:=$: 대입
- $=, \neq, >, \geq, <, \leq$: 비교 (주의. $=$은 비교 연산자로 사용함)
- $++, --, +=, -=$: 기존 프로그래밍 언어에 있는 일부 연산자 그대로 사용한다.
- 삼항 선택 연산자는 사용하지 않고 파이썬 구조를 사용한다.

특별한 경우가 아니면 1색인을 사용한다.

1.1 알고리즘 성능 분석

주어진 문제를 해결하는 알고리즘은 다양할 수 있다. 다음과 같은 문제가 주어졌을 때,

중복 요소 존재 여부 찾기 문제

- 입력. n개의 정수로 구성된 배열 A
- 출력. 중복 요소가 있으면 `true` 없으면 `false`

이를 해결하는 알고리즘은 다음과 같이 다양하다.

알고리즘 1.1 중복 요소 존재 여부 전주조사 알고리즘

```
1: function containsDuplicate(A[])
2:     for i := 1 to n − 1 do
3:         for j := i + 1 to n do
4:             if A[i] = A[j] then return true
5:     return false
```

알고리즘 1.2 중복 요소 존재 여부 전주조사 알고리즘

```
1: procedure containsDuplicate(A[])
2:     sort(A)
3:     for i := 2 to n do
4:         if A[i] = A[i − 1] then return true
5:     return false
```

알고리즘 1.3 중복 요소 존재 여부 전주조사 알고리즘

```
1: procedure containsDuplicate(A[])
2:     S := empty set
3:     for i := 1 to n do
4:         if A[i] ∈ S then return true
5:         else S.put(A[i])
6:     return false
```

제시된 3개의 알고리즘은 모두 중복 요소 존재 여부 문제를 해결한다. 반복문 위주로 살펴보면 첫 번째 알고리즘은 이중 `for` 문으로 구성되어 있고, 나머지 두 알고리즘은 단일 `for` 문으로 구성되어 있다. 두 번째 알고리즘은

정렬하는 비용에 의해 알고리즘 성능이 좌우하며, 세 번째 알고리즘은 집합에서 요소를 찾는 비용과 집합에 요소를 삽입하는 비용에 좌우한다. 이들 알고리즘에 대한 자세한 분석은 뒤로 미룬다.

이처럼 한 문제를 해결하는 여러 알고리즘이 있을 때, 이 중에 어떤 알고리즘을 사용하는 것이 효과적인지 판단할 수 있어야 한다. 이를 위해 필요한 것이 성능 분석이다. 우리는 성능이 좋은 것을 선택하여 사용하고 싶다. 따라서 성능 분석의 주 목적은 같은 문제를 해결하는 두 알고리즘의 비교이다. 반면에 서로 다른 문제를 해결하는 두 알고리즘의 비교는 알고리즘을 비교하기 위한 목적이 아니라 두 문제의 어려운 정도를 비교하기 위한 것이다.

알고리즘의 성능이 나쁘면 용도가 전혀 없을 것 같지만 실제 성능이 나쁘지만 정확한 알고리즘은 우수한 알고리즘을 데스트할 때 활용할 수 있다. 특히, 테스트 데이터를 생성하기 위해 이와 같은 알고리즘을 많이 활용한다. 성능이 좋은 알고리즘을 개발하였지만 정확성을 증명하지 못한 경우에는 이 알고리즘의 정확성을 검증하기 위해 많은 테스트 데이터를 이용하여 테스트하는 것이 필요할 수 있다. 보통 테스트 데이터는 문제를 해결하는 정확한 알고리즘이 없으면 만들기 어렵기 때문에 많은 경우 성능이 나쁘지만 정확한 알고리즘을 이용하여 생성하는 경우가 많다.

성능이 나쁜 알고리즘을 실제 응용에서 사용하는 경우도 있다. 응용이 해결해야 하는 문제의 입력 크기 범위가 제한적이면 성능이 나쁜 알고리즘도 충분히 사용할 수 있다. 특히, 성능이 우수한 알고리즘을 만드는 것 자체가 어려울 수 있고, 그와 같은 알고리즘은 이해하기 어려울 수 있어 요구사항 변경에 따라 확장하거나 수정하기 힘들 수 있다. 이 때문에 응용 성능에 큰 영향이 없다면 성능은 다른 것과 비교하여 떨어지지만 누구나 이해하기 쉽고 유지보수하기 쉬운 알고리즘을 사용하는 것이 더 효과적일 수 있다.

알고리즘의 성능 분석은 크게 입력에 따라 소요되는 시간과 필요한 공간을 분석한다. 전자를 **시간 복잡도**라 하고, 후자를 **공간 복잡도**라 한다. 알고리즘의 성능은 입력과 독립적일 수 있고 입력에 따라 달라질 수 있다. 따라서 최선(best case), 평균(average case), **최악**(worst case) 경우를 각각 나누어 분석할 수 있다. 최선 경우의 비교는 보통 의미가 없고, 평균의 경우는 분석하기 복잡하기 때문에 일반적으로 성능 분석은 최악의 경우에 대한 분석을 이용하여 비교한다. 우리는 보통 최악의 경우에도 더 좋은 성능을 보이는 알고리즘이 더 우수한 알고리즘으로 해석한다. 더욱이 많은 경우 평균 비용과 최악의 경우 시간 복잡도가 같다. 하지만 실제 알고리즘을 적용하는 응용에서 사용하는 데이터를 고려하였을 때 최악의 경우가 일어나지 않거나 일어날 확률이 매우 낮을 수 있다. 따라서 분석 결과보다 실제 알고리즘이 보이는 성능은 더 우수할 수 있다.

공간 복잡도는 알고리즘의 입력을 나타내기 위해 필요한 공간과 알고리즘이 내부적으로 사용하는 메모리 공간(auxiliary space)을 분석하여 계산한다. 성능 분석은 같은 문제를 해결하는 알고리즘을 분석하는 것이기 때문에 알고리즘이 내부적으로 사용하는 메모리 공간이 중요하다.

알고리즘의 성능 분석을 통해 분석된 알고리즘의 성능 수준을 제시할 수 있어야 한다. 우리는 제시된 수준을 이용하여 같은 문제를 해결하는 두 알고리즘을 비교하게 된다. 빅O는 알고리즘이 어떤 특정 성능 수준에 해당하는지 나타낼 때 사용하는 표기법이자 분석 방법이다.

1.2 시간 복잡도

알고리즘의 시간 복잡도를 측정하기 위해 실제 실행 시간을 측정하지 않는다. 특히, 실행 시간을 측정하여 두 알고리즘을 비교하지 않는다. 그 이유는 다음과 같다.

- 실제 실행 시간을 측정하기 위해서는 구현을 해야 한다.
- 알고리즘의 구현은 개발자에 따라 차이가 있을 수 있고, 실행 시간은 개발 환경, 프로그래밍 언어, 실행 환경에 따라 차이가 있을 수 있다.
- 다양한 입력에 대해 비교해야 한다.

- 비교 결과를 통해 알고리즘을 성능 수준 별로 분류하는 것이 어렵다.

이 때문에 알고리즘의 시간 복잡도는 주어진 입력에 따라 필요한 기본 연산의 수를 파악하여, 이를 이용하여 분석한다. 여기서 기본 연산이란 산술 연산, 대입 연산, 비교 연산과 같이 더 이상 세분화되지 않는 연산을 말한다. 각 기본 연산의 성능이 같지 않지만 같은 문제를 해결하는 알고리즘을 비교하는 것이고, **점근적**(asymptotic) 분석이기 때문에 결국 비교에 핵심이 되는 연산은 다르지 않아 모든 종류의 기본 연산의 성능이 같다고 생각하고 분석하여도 문제가 되지 않는다. 특히, 같은 문제를 해결하는 알고리즘에서 핵심 역할을 하는 연산은 보통 같다. 여기서 점근적 분석이란 입력 크기가 충분히 큰 경우만 분석 결과가 의미가 있는 분석을 말한다. 그 이유는 입력 크기가 작으면 두 알고리즘의 성능 차이를 느낄 수 없고 클 경우에만 의미있게 느낄 수 있기 때문이다.

예를 들어 이전 절에서 언급한 검색 문제를 해결하는 다음 알고리즘을 분석하여 보자. 이 알고리즘에서 필요한

알고리즘 1.4 선형 검색 알고리즘

```
1: function SEARCH(A[], v)
2:     for i := 1 to n do
3:         if A[i] = v then return true
4:     return false
```

기본 연산을 파악하기 위해서는 구체적인 언어로 구현된 알고리즘이 필요하다. 하지만 구체적 구현 없이도 가정을 통해 충분히 분석할 수 있으며, 언어에 따라 나타나는 차이는 최종적으로 점근적 분석에 의해 무시할 수 있다.

위 알고리즘을 분석하기 위해 `for` 문의 수행 비용은 다음과 같다고 가정한다. 변수 `i`의 초기화, 반복마다 변수 `i`와 `n`의 비교, 변수 `i`의 증가가 필요하다. 이를 토대로 최선, 평균, 최악의 경우 필요한 기본 연산 수를 계산하면 다음과 같다.

- 최선의 경우($A[1]$에 v가 있는 경우) $\Longrightarrow$ 대입 연산: 1($i := 1$), 비교 연산($i < n$, $A[i] = v$): 2 $\Longrightarrow$ 3
- 평균의 경우: 여러 가정이 추가로 필요하다. n개의 값이 모두 다르고, i번째에 있는 요소를 검색할 확률이 같고, 있는 것을 검색할 확률과 없는 것을 검색할 확률이 같다고 가정하면 다음과 같이 분석할 수 있다.

$$0.5 \times \frac{3(1+2+\ldots+n)}{n} + 0.5 \times (3n+2) = \frac{9n+7}{4}$$

- 최악의 경우(v가 배열에 없는 경우) $\Longrightarrow$ 대입 연산: 1, 증가 연산: n, 비교 연산: $2n+1$ $\Longrightarrow$ $3n+2$

선형 검색 알고리즘의 공간 복잡도를 분석하여 보자. 이 알고리즘의 입력은 배열과 정수 하나이며, 내부적으로는 지역 변수 i를 사용한다. `for` 문에서 사용하는 배열의 크기는 언어에 따라 인자로 주어질 수 있고, 배열 자체가 용량 정보를 가지고 있을 수 있다. 입력에 따라 배열이 차지하는 공간은 변할 수 있지만 나머지 요소는 변하지 않는다.

알고리즘 수행에 필요한 연산 수가 적으면 더 효율적인 알고리즘이다. 정확하게는 알고리즘 A가 B보다 빠르다는 것은 최악의 경우 필요한 시간이 입력 크기가 증가함에 따라 A가 B보다 느리게 증가한다는 것을 의미한다. 공간 복잡도 측면에서는 더 적은 공간을 사용하는 알고리즘이 우수한 알고리즘이다.

2. 알고리즘의 예

2.1 정수 곱셈

정수 곱셈

- 입력. 두 개의 n자리 정수 X와 Y
- 출력. 두 수의 곱 X와 Y

초등학교부터 우리가 사용한 정수 곱셈 알고리즘을 생각하여 보자. 예를 들어 5,678과 1,234를 곱하는 과정은 다음과 같다.

			5	6	7	8
×			1	2	3	4
		2	2	7	1	2
	1	7	0	3	4	
1	1	3	5	6		
5	6	7	8			
7	0	0	6	6	5	2

위 과정에서 사용한 최악의 경우 필요한 기본 연산 수를 파악하여 보자. Y의 각 자리수마다 n개의 곱셈이 필요하다. 곱셈 과정에서 자리올림 때문에 추가 덧셈 연산이 필요하다. 따라서 각 자리수마다 최대 $2n^2$개의 기본 연산이 필요하다. 이렇게 각 자리수마다 곱셈을 진행한 결과를 모두 더하여야 한다. 최종 결과를 나타내는 정수의 최대 자리수는 $2n$이며, 각 자리수를 얻기 위해 최대 n개의 덧셈이 필요하다. 실제 자리올림 때문에 $n+1$개가 필요할 수 있지만 각 자리수마다 n개 덧셈이 필요하지 않기 때문에 대략적으로 최대 $2n^2$개의 덧셈이 필요하다고 결론을 내릴 수 있다. 그러면 초등학교 덧셈 알고리즘의 총 비용은 최대 $4n^2$이 필요하다.

우리가 계속 사용하고 이 알고리즘보다 더 빠른 덧셈 알고리즘이 존재한다. 항상 훌륭한 알고리즘 개발자는 존재하는 알고리즘보다 더 빠른 알고리즘이 있는지 고민을 해야 한다. 초등학교보다 더 빠른 Karatsuba라는 곱셈 알고리즘이 있다. 이 알고리즘은 3장에서 살펴볼 **분할 정복**(divide-and-conquer) 기법을 사용하고 있다.

분할 정복하는 곱셈 알고리즘의 기본적 개념은 다음과 같다. 주어진 n자리 정수를 각각 $n/2$자리 정수로 나누어 이들을 이용하여 다음 원리를 이용하여 곱셈을 진행한다.

$$(10^{n/2}a+b)(10^{n/2}c+d) = 10^n ac + 10^{n/2}(ad+bc) + bd$$

앞서 살펴본 예제를 이용하면 다음과 같다.

- $X = 5,678$, $Y = 1,234$
- $a = 56$, $b = 78$, $c = 12$, $d = 34$
- 단계 1. $ac = 672$
- 단계 2. $bd = 2,652$
- 단계 3. $(a+b)(c+d) = 6164$
- 단계 4. $(3) - (2) - (1) = 2,840$

- 단계 5. 결과 $= (1) \times 10^n + (4) \times 10^{n/2} + (2) = 7,006,652$

이와 같은 방법으로 곱셈을 하게 되면 2개의 n자리 정수를 3번의 $n/2$자리 곱샘과 덧셈을 통해 계산할 수 있다. 원래 $(a+b)(c+d)$를 직관적으로 계산하면 4번의 곱셈이 필요하지만 Karatsuba는 3번으로 줄이고 있다. 또 이것을 한번만 줄여 진행하는 것이 아니라 재귀적으로 $n/2$자리로 계속 바꾸어 곱셈을 진행한다. 이것이 Karatsuba 곱셈 알고리즘이다. 초등학교 곱셈 알고리즘과 Karatsuba 곱셈 알고리즘을 비교하면 Karatsuba 알고리즘이 더 효과적이다. 얼마만큼 효과적인지는 4장에서 자세히 분석한다. 이 절에서는 우리가 생각하고 있는 매우 빠른 최적의 알고리즘보다 더 좋은 알고리즘이 항상 존재할 수 있다는 것을 기억할 필요가 있다.

2.2 정렬 알고리즘

정렬 문제

- 입력. n개의 정수로 구성된 배열 A
- 출력. 같은 수로 구성된 오름차순으로 정렬된 배열

정렬 알고리즘은 성능에 따라 이차 시간(quadratic time)이 필요한 것과 의사 선형 시간(quasilinear time, near linear time)이 필요한 것으로 분류할 수 있다. 먼저 이차 시간이 필요한 알고리즘을 몇 가지 살펴보고, 의사 선형 시간이 필요한 합병 정렬 알고리즘을 살펴보고자 한다.

2.2.1 이차 시간 정렬 알고리즘

첫 번째 살펴볼 정렬 알고리즘은 다음에 제시된 선택 정렬(selection sort) 알고리즘이다.

알고리즘 1.5 선택 정렬 알고리즘

```
1: procedure SORT(A[])
2:     for i := 1 to n − 1 do
3:         minLoc := i
4:         for j := i + 1 to n do
5:             if A[minLoc] > A[j] then minLoc := j
6:         if minLoc ≠ i then SWAP(A[minLoc], A[i])
```

이 알고리즘은 매번 남아 있는 것 중 가장 작은 값을 찾아 그것을 최종 위치로 옮긴다. 따라서 입력과 무관하게 항상 반복 횟수가 동일하며, 이 반복 과정에서 항상 동일한 횟수의 비교가 이루어진다. 실제 이 횟수는 다음과 같다.

$$n-1+n-2+\cdots+1 = n(n-1)/2$$

하지만 `minLoc` $:= j$와 SWAP$(A[$`minLoc`$], A[i])$은 입력 데이터에 따라 이루어지는 횟수가 다르다. 그러면 어떤 경우가 최악의 경우인가?

거꾸로 정렬된 경우(예: $[8,7,6,5,4,3,2,1]$)와 나머지는 정렬되어 있고 가장 작은 수가 맨 뒤에 있는 경우(예: $[2,3,4,5,6,7,8,1]$)를 비교하여 보자. 제시된 거꾸로 정렬된 예가 입력으로 주어졌을 때 외부 **for** 문이 한번 반복하면 그 결과는 $[1,7,6,5,4,3,2,8]$이 된다. 따라서 4번 반복하면 정렬되며, 그 뒤로는 SWAP도 `minLoc` $:= j$도 실행되지 않는다. 반면에 두 번째 경우는 $n-1$번 반복할 때마다 SWAP이 이루어진다. 외부 **for** 루프가 반복할 때마다 SWAP이 발생하는 경우는 지금 제시된 형태가 유일한 것은 아니다. 예를 들어 $[5,4,6,3,7,8,2,1]$ 등도 매번 SWAP이 이루어진다.

두 번째 살펴볼 정렬 알고리즘은 다음에 제시된 삽입 정렬(insertion sort) 알고리즘이다.

알고리즘 1.6 삽입 정렬 알고리즘

```
1: procedure SORT(A[])
2:     for i := 2 to n do
3:         temp := A[i]
4:         j := i − 1
5:         while j >= 1 and temp < A[j] do
6:             A[j + 1] := A[j]
7:             −−j
8:         A[j + 1] := temp
```

이 알고리즘은 점진적으로 정렬되어 있는 구간을 하나씩 늘린다. 현재 4번째 위치에 있는 요소를 고려할 차례이면 앞 3개의 요소는 정렬되어 있다. 따라서 알고리즘은 거꾸로 정렬되어 있는 경우가 최악의 경우이다. 이때 반복하는 총 횟수는 다음과 같다.

$$1 + 2 + \cdots + n - 1 = n(n-1)/2$$

반대로 최선의 경우는 이미 정렬되어 있는 경우이다. 이 경우에는 반복하는 총 횟수는 $n - 1$이다.

세 번째 살펴볼 정렬 알고리즘은 다음에 제시된 버블 정렬(bubble sort) 알고리즘이다.

알고리즘 1.7 버블 정렬 알고리즘

```
1: procedure SORT(A[])
2:     for i := 1 to n − 1 do
3:         flag := false
4:         for j := n downto i + 1 do
5:             if A[j − 1] > A[j] then
6:                 SWAP(A[j − 1], A[j])
7:                 flag := true
8:         if not flag then break
```

이 알고리즘은 선택 정렬처럼 매 반복마다 남아 있는 것 중 가장 작은 것을 최종 위치로 옮긴다. 하지만 선택 정렬과 달리 이 요소만 옮기는 것이 아니라 뒤에서부터 비교하면서 다른 요소도 부분적으로 이동을 한다. 따라서 반복이 진행될 때마다 결과에 더 빠르게 가까워진다. 이 알고리즘도 거꾸로 정렬되어 있는 경우가 최악의 경우이다. 이때 반복하는 총 횟수는 다음과 같다.

$$n - 1 + n - 2 + \cdots + 1 = n(n-1)/2$$

또 삽입 정렬과 마찬가지로 최선의 경우는 이미 정렬되어 있는 경우이다. 이 경우에는 반복하는 총 횟수는 $n - 1$이다.

2.2.2 합병 정렬 알고리즘

합병 정렬 알고리즘은 분할 정복 기법을 사용하는 정렬 알고리즘으로 재귀 알고리즘이다. 분할 정복은 문제를 작은 문제로 나누어 각 소문제를 해결한 후에 그것을 결합하여 큰 문제를 해결하는 알고리즘 설계 기법이다. 재귀 알고리즘의 시간 복잡도는 분석하기 쉽지 않다. 보통 재귀 호출이 얼마나 많이 이루어지는지 분석해야 하며, 비재귀 과정에서 일어나는 비용에 대한 분석도 필요하다. 이 알고리즘에서 비재귀적 부분에서 일어나는 일은 합병(merge)이다.

합병의 비용을 분석하여 보자. 3개의 **while** 문을 통해 총 반복하는 횟수는 합병하는 구간의 크기이다. 따라서

알고리즘 1.8 합병 정렬 알고리즘

```
1: procedure SORT(A[])
2:     MERGESORT(1, n, A)
1: procedure MERGESORT(lo, hi, A[])
2:     if lo < hi then
3:         mid := lo + (hi − lo)/2
4:         MERGESORT(1, mid − 1, A)
5:         MERGESORT(mid, hi, A)
6:         MERGE(lo, mid, hi, A)
1: procedure MERGE(lo, mid, hi, A[])
2:     left[] := A[lo ... mid − 1]
3:     right[] := A[mid ... hi]
4:     L, R, i := 1, 1, lo
5:     while L <= mid − 1 and R <= hi do
6:         if left[L] < right[R] then A[i++] := left[L++]
7:         elseA[i++] := right[R++]
8:     while L <= mid − 1 do
9:         A[i++] := left[L++]
10:    while R <= hi do
11:        A[i++] := right[R++]
```

$n/2$ 크기의 두 개의 배열을 합병할 때 반복하는 총 횟수는 n이다. 이 과정에서 최대 3번의 비교가 이루어지며, 한번의 대입이 이루어지고, 2개의 변수가 하나 증가한다. 또 3개의 변수의 초기화가 필요하므로 최대 $6n + 3$ 기본 연산이 필요하다. 물론 여기에 두 개의 $n/2$ 크기의 베열로 나누는 비용은 포함하지 않은 비용이다. $6n + 3$을 더 대략적으로 표현하기 위해 총 비용을 $9n$으로 표현하여도 정확한 것은 아니지만 틀린 것은 아니다.

처음 배열의 크기가 8이면, 두 개의 크기가 4인 배열로 나누어지고, 나누어진 배열은 다시 크기가 2인 배열로 최종적으로 1인 배열로 나누어진다. 따라서 재귀 호출 트리를 그리면 이 트리에 총 레벨 수는 $\log_2 n + 1$임을 알 수 있다. 루트 레벨이 0이라고 할 때 레벨 j에는 2^j개의 재귀 호출이 이루어지며, 각 재귀 호출의 배열 크기는 $n/2^j$이다. 따라서 레벨 j에서 이루어지는 총 연산 수는 다음과 같다.

$$2^j \times 9 \times \frac{n}{2^j} = 9n$$

즉, 각 레벨에서 소요되는 비용은 같다. 그러므로 총 $\log_2 n + 1$ 레벨이 있으므로 전체 비용은 $9n \log n + 9n$이다.

3. 빅O

빅O를 이용하여 알고리즘을 분석하는 방법은 다음과 같다.

- 단계 1. 알고리즘 서술이나 구현 결과를 토대로 최악의 경우 입력 크기 n에 따라 필요한 기본 연산 수를 파악하여 $T(n)$ 다항식을 정의한다.
- 단계 2. $T(n)$ 다항식에서 최고차 항을 제외한 나머지 항은 제거하고, 최고차 항의 계수도 제거한다.
- 단계 3. 결과 다항식이 해당 알고리즘의 빅O가 된다.

빅O 분석에서 가장 어려운 부분이 단계 1이다. 선형 검색 알고리즘은 비교적 정확하게 최악의 경우 필요한 기본 연산 수를 파악할 수 있다. 하지만 필요한 기본 연산 수를 파악하기 어려운 알고리즘도 많다.

이 방법에 따라 앞서 살펴본 몇 알고리즘의 시간 복잡도와 공간 복잡도는 표 1.1와 같다. 제시된 알고리즘의 시간 복잡도부터 간단히 살펴보자. 선형 검색 알고리즘은 1.2절에서 분석한 것처럼 최악의 경우 $3n + 2$ 연산이 필요하다.

<표 1.1> 각종 알고리즘의 빅O

알고리즘	시간 복잡도	공간복잡도
선형 검색 알고리즘	$O(n)$	$O(n)$
삽입 정렬 알고리즘	$O(n^2)$	$O(n)$
합병 정렬 알고리즘	$O(n \log n)$	$O(n \log n)$

따라서 이 알고리즘의 빅O는 $O(n)$이다. 보통 선형 검색 알고리즘처럼 단일 반복문으로 구성되어 있고, 반복 횟수가 입력 크기 n에 비례하며, 반복문 몸체의 비용이 일정하면 그것의 시간 복잡도는 $O(n)$이다.

삽입 정렬 알고리즘은 2.2.1절에서 분석한 바와 같이 최악의 경우 총 $n(n-1)/2$번 반복한다. **for** 루프는 3개의 대입 연산과 **while** 문으로 구성되어 있고, **while** 문은 2번의 비교, 대입, 증감 연산으로 구성되어 있다. 실제 **while** 문은 루프를 종료하기 위해 한 번의 비교가 더 필요하다. 따라서 간단히 분석하기 위해 각 반복이 한 번씩 더 반복한다고 하면 총 소요되는 비용은 다음과 같다.

$$T(n) \leq 4n(n+2)/2 + 3(n-1) = 2n^2 + 7nn - 3$$

따라서 삽입 정렬 알고리즘의 시간 복잡도 빅O는 $O(n^2)$이다. 이처럼 보통 이중 반복문으로 구성된 알고리즘의 시간 복잡도는 $O(n^2)$이다. 하지만 이중 반복문이지만 $O(n^2)$인 아닌 경우도 있으므로 너무 단순하게 분석할 경우 잘못 분석할 수 있다.

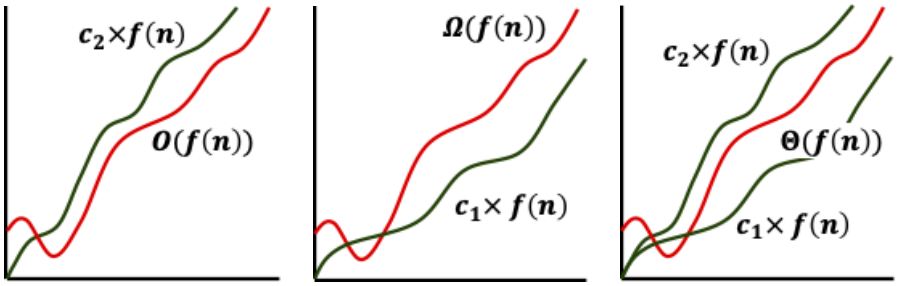
합병 정렬 알고리즘은 2.2.2에서 분석한 바와 같이 최악의 경우 총 $9n \log n + 9n$ 비용이 필요하므로 시간 복잡도는 $O(n \log n)$이다. 앞서 언급한 바와 같이 재귀 알고리즘의 시간 복잡도를 분석하는 것은 간단하지 않다. 하지만 4장에서 설명하는 도사 정리(master theroem)를 적용할 수 있는 재귀 알고리즘은 간단히 분석할 수 있다.

이제 제시된 알고리즘의 공간 복잡도를 분석하여 보자. 선형 검색 알고리즘은 입력 배열 외에 상수 공간만 사용하기 때문에 공간 복잡도의 빅O는 $O(n)$이다. 삽입 정렬 알고리즘의 공간 복잡도의 빅O도 입력 배열 외에 상수 공간만 사용하므로 $O(n)$이다. 합병 정렬은 알고리즘 1.8에 제시된 것처럼 MERGE를 호출할 때마다 `left`와 `right` 배열을 새로 만들어야 하기 추가적으로 많은 공간을 사용한다. 이 공간을 절약하는 방법이 있지만 알고리즘 1.8에 제시된 것처럼 재귀 호출마다 `left`와 `right` 배열을 새롭게 확보한다고 가정하고 분석하여 보자. 이 경우 재귀 호출 트리의 각 레벨마다 추가로 필요한 공간은 n에 비례하며, 레벨 수는 $\log n$에 비례하므로 추가로 총 필요한 공간은 $O(n \log n)$이다. 알고리즘 입력의 크기는 $O(n)$이므로 합병 정렬의 공간 복잡도는 $O(n \log n)$이다.

빅O 분석은 이처럼 최악의 경우에 대한 분석이며, 대략적 분석이고, 점근적 분석이다. 입력 크기가 충분히 클 경우에만 의미가 있는 점근적 분석을 목표로 하기 때문에 대략적으로 분석하여도 비교에 있어 문제가 없다. 이 때문에 단계 2에서 최고차 항의 계수와 나머지 항을 제거하는 것이며, 단계 1에서 아주 정확하게 분석하지 못하여도 비교에 큰 문제가 되지 않을 수 있다. 보통 상수 계수는 프로그래밍 언어, 실행 환경 등에 의해 바뀔 수 있는 요소이며, 이들 계수를 정확하게 분석하지 못하여도 비교 결과가 보통 바뀌지 않으며, 비교 능력에 영향을 주지 않는다. 우리는 입력 크기 n이 증가됨에 따라 알고리즘 수행 시간의 증가 비율을 비교하고 싶은 것이다. 증가 속도가 느린 것이 성능이 더 우수한 알고리즘이 되는 것이다. 하지만 최고차 항의 계수와 낮은 차수 항이 항상 무의미한 것은 아니다. 같은 수준의 알고리즘을 비교할 때는 제거한 계수와 항까지 이용하여 비교해야 한다.

정의 1.1 (빅O)**.** 입력 크기 n에 대해 최악의 경우 알고리즘의 수행 시간을 나타내는 함수 $T(n)$이 궁극에는 $cf(n)$보다 항상 증가 속도가 느리면 이 알고리즘의 빅O는 $O(f(n))$이라 한다.

$T(n)$이 $f(n)$보다 성능이 같거나 우수하면 $O(f(n))$이다. 따라서 선형 검색 알고리즘은 최악의 경우 n과 같거나 빠른 알고리즘이다. 알고리즘 A의 빅O가 $O(n^2)$이면 $k >= 2$에 대해 A는 $O(n^k)$에 포함된다. 따라서 선형 검색 알고리즘의 빅O는 $O(n^2)$이라고 표현하여도 틀린 것이 아니다. 우리는 알고리즘을 설명할 때 그것의 성능을 더 정확하게 표현하고 싶기 때문에 가장 엄격(tight)한 상한(upper bound)으로 설명하고 싶다. 따라서 선형 검색 알고



(1) 빅O (2) 빅Ω (3) 빅Θ

<그림 1.1> 빅O, 빅Ω, 빅Θ

리즘의 가장 엄격한 상한이 $O(n)$이면 이 알고리즘은 $O(n^2)$, $O(n^3)$ 등에 속하지만 A의 빅O는 $O(n)$으로 표현한다. 하지만 종종 알고리즘의 엄격한 상한을 구하기 어려운 경우도 많다. 따라서 알고리즘의 빅O가 제시되었을 때 이 알고리즘은 제시된 수준보다 훨씬 효과적인 알고리즘일 수 있다.

$T(n) \in O(f(n))$임을 보이기 위해서는 $n \geq n_0$에 대해 $T(n) \leq cf(n)$이 항상 만족하는 c와 n_0을 제시하면 된다. $f(n)$에 상수를 곱한 결과 함수는 특정 시점 이후에는 항상 $T(n)$보다 빠르게 증가한다는 것을 보이면 된다. 여기서 $f(n)$은 n, $n \log n$, n^2처럼 성능 분류 대표 함수이다. 성능 분류 대표 함수와 알고리즘의 $T(n)$을 비교하여 알고리즘이 어느 수준의 알고리즘인지 분류하게 된다. 하지만 앞서 설명한 것처럼 이를 통해 빅O를 제시하지 않고, 주어진 알고리즘의 $T(n)$을 구한 후에 최고차 항을 제외한 나머지 항은 제거하고, 최고차 항의 계수도 제거하면 그것이 해당 알고리즘의 빅O가 된다.

3.1 빅O, 빅Ω, 빅Θ, 리틀o

보통 알고리즘의 성능 분석 결과를 나타낼 때 가장 많이 사용하는 것이 빅O이다. 하지만 빅Θ로 성능을 표현할 수 있다면 빅O 대신에 빅Θ를 많이 사용한다. 빅O는 알고리즘 성능의 상한을 나타내는 반면에 빅Ω는 하한(lower bound)를 나타낸다. 상한과 하한이 머리속에 생각하는 것과 반대적 개념일 수 있다. 상한과 하한을 생각할 때 위가 나쁜 것이고, 아래가 좋은 것이다. 즉, 빅O는 이보다는 나쁠 수 없다는 것을 말하며, 빅Ω는 이보다는 좋을 수 없다는 것을 말한다.

정의 1.2 (빅Ω)**.** 입력 크기 n에 대해 최악의 경우 알고리즘의 수행 시간을 나타내는 함수 $T(n)$이 궁극에는 $cf(n)$보다 항상 증가 속도가 빠르면 이 알고리즘의 빅Ω는 $\Omega(f(n))$이라 한다.

$T(n) = \Omega(f(n))$임을 보이기 위해서는 $n \geq n_0$에 대해 $T(n) \geq cf(n)$이 항상 만족하는 c와 n_0을 제시하면 된다. $f(n)$에 상수를 곱한 결과 함수는 특정 시점 이후에는 항상 $T(n)$보다 느리게 증가한다는 것을 보이면 된다. 예를 들어 알고리즘 A의 빅Ω가 $\Omega(n^2)$이면 알고리즘 A는 최악의 경우 n^2과 같거나 느린(성능이 나쁜) 알고리즘이라는 것을 말한다. 알고리즘 A의 빅Ω가 $\Omega(n^k)$이면 $j <= k$에 대해 A는 $\Omega(n^j)$에 포함된다. 이 때문에 항상 모든 알고리즘은 $\Omega(1)$이다.

빅Ω의 경우 $T(n)$을 대략적으로 분석할 수 없다. 빅O의 경우 실제 $T(n)$의 최고차항이 일차이지만 정확하게 분석하기 어려워 $T(n)$의 최고차항이 이차가 되어 알고리즘의 빅O를 $O(n^2)$으로 분석하더라도 이 빅O가 틀린 것은 아니다. 하지만 빅Ω의 경우 이 알고리즘을 $\Omega(n^2)$으로 분석하는 것은 잘못된 것이다. 알고리즘 A의 최악의 경우 빅O는 이 알고리즘의 평균 경우와 최선 경우 빅O로 제시하여도 틀리지 않지만 최악의 경우 빅Ω는 평균 경우와 최선 경우 빅Ω로 제시하였을 때 틀릴 수 있다.

정의 1.3 (빅Θ)**.** 입력 크기 n에 대해 최악의 경우 알고리즘의 수행 시간을 나타내는 함수 $T(n)$이 궁극에는 $c_1 f(n)$보다 항상 증가 속도가 빠르고 $c_2 f(n)$보다 항상 증가 속도가 느리면 이 알고리즘의 빅Θ는 $\Theta(f(n))$이라 한다.

빅Θ는 특정 알고리즘의 빅O와 빅Ω가 같은 경우를 말한다. 즉, 알고리즘 A의 빅Θ가 $\Theta(n^2)$이면 이 알고리즘의 성능은 정확하게 n^2과 같다는 것을 말한다. 빅Θ는 빅Ω와 마찬가지로 $T(n)$를 엄격하게 계산할 수 없으면 알고리즘의 빅Θ를 구하는 것이 어려울 수 있다.

작은o는 주어진 복잡도보다 무조건 빠르다는 것을 나타낸다. 즉, 빅O는 해당 복잡도보다 같거나 좋다는 것을 나타내지만 작은o는 여기서 같거나 부분이 빠지게 된다. $T(n) = n^2$이면 이 알고리즘의 빅O는 $O(n^2)$이지만 이 알고리즘은 $o(n^2)$ 속하지 않는다.

3.2 입력 크기

빅O 분석을 할 때 입력 크기에 대한 함수를 정의해야 한다. 이때 알고리즘이 n개의 데이터를 처리하면 입력 크기는 n이라고 생각하면 된다. 예를 들어 선형 검색에서 검색하는 배열의 크기가 입력의 크기가 된다. 하지만 정확한 입력 크기는 입력을 표현하기 위해 필요한 문자 수이다.

입력을 표현하기 위한 문자 수는 비트 수로 계산할 수 있고, 해당 데이터를 컴퓨터로 표현하기 위해 타이핑해야 하는 키보드의 수로 계산할 수 있다. 예를 들어 검색 문제의 입력은 크기가 m인 정수 배열과 검색할 값이다. 32비트 정수를 사용한다고 가정하면 입력 크기 $n = 32(m+1)$이다. 선형 검색 알고리즘의 $T(m) = 3m + 2$이고 $m = n/32 - 1$이므로 $T(n) = \frac{3}{32}n - 1$이다. 따라서 검색 문제에서 입력 크기를 정확하게 계산하여 분석을 하는 것과 기존처럼 대충 분석을 하는 것과 결과는 같다.

소수 판별 문제는 다음과 같이 정의된다.

소수 판별 문제

- 입력: 양의 정수 x
- 출력. x가 소수이면 `true` 아니면 `false`

이 문제를 해결하는 다음 알고리즘의 빅O를 분석하여 보자.

알고리즘 1.9 소수 판별 알고리즘

```
1: function ISPRIME(x)
2:     for i := 2 to SQRT(x) do
3:         if x%i = 0 then return false
4:     return true
```

이 알고리즘은 x가 소수일 때가 최악의 경우이며, $\sqrt{x} - 1$만큼 나머지 연산과 비교 연산을 수행한다. 따라서 이 알고리즘은 $2(x^{1/2} - 1)$번 비용이 소요되지만 이 알고리즘의 빅O는 $O(n^{1/2})$가 아니다. 이 알고리즘의 입력 크기는 1이 아니며, 정수가 클수록 필요한 비용은 증가한다. 정수 x를 표현하기 위해 필요한 비트 수 n은 약 $\log_2 x$ 비트이다. 즉, $x \approx 2^n$이다. 따라서 $T(n) = (2^n)^{1/2} = 2^{n/2}$이다. 그러므로 이 알고리즘의 빅$O$는 $O(2^n)$이다. 이처럼 정확한 입력 크기를 적용하여 분석하지 않으면 빅O를 올바르게 분석하지 못하는 경우가 있다. 보통 주어진 정수 값에 따라 비용이 달라지는 경우에는 정확한 입력 크기를 고려하여 분석해야 한다.

3.3 표준 복잡도

빅O 알고리즘을 분석할 때, 우리는 주어진 알고리즘이 다음 표준 복잡도 중에 어느 복잡도 수준에 해당하는지 알고 싶은 것이다. 즉, 빅O를 결정하기 위해 $n \geq n_0$에 대해 $T(n) \leq cf(n)$이 항상 만족하는 c와 n_0을 제시하면 되는데, 이때 사용하는 $f(n)$는 보통 다음 중 하나를 이용하게 된다. 위키피디아(`https://en.wikipedia.org/wiki/`

Time_complexity)에 보면 더 세분화되어 있지만 학부 수준에서는 다음 정도만 알고 있으면 된다.

- 상수 시간(constant time): $O(1)$, 예) 스택 자료구조에서 PUSH 연산
- 로그 시간(logarithmic time): $O(\log n)$, 예) 이진 검색
- 선형 시간(linear time): $O(n)$, 예) 선형 검색
- 의사 선형 시간(quasilinear time): $O(n \log n)$, 예) 합병 정렬, 힙 정렬, 퀵 정렬
- 이차 시간(quadratic time): $O(n^2)$, 예) 삽입 정렬
- 삼차 시간(cubic time): $O(n^3)$, 예) 일반 $n \times n$ 행렬 곱셈
- 다차 시간(polynomial time): $O(n^k)$
- 이차 시간(exponential time): $O(2^n)$, 에) 알고리즘 1.9의 소수 판별 알고리즘
- 계승 시간(factorial time): $O(n!)$, 예) 전수 조사로 외판원 문제를 해결할 경우

3.4 다중 변수 빅O

지금까지 살펴본 빅O는 $O(n)$, $O(n \log n)$처럼 하나의 변수만 이용하여 표현하고 있다. 하지만 알고리즘을 분석하다 보면 $O(mn)$, $O(m+n)$처럼 여러 개의 변수를 이용하여 빅O을 표현해야 할 때도 있다. 보통 그래프 관련 알고리즘[1]의 경우 그래프를 구성하는 노드의 수와 간선의 수를 모두 이용하여 성능을 표현한다.

다중 변수를 사용하여 빅O를 표현하면 그것의 성능이 지난 절에서 제시한 표준 복잡도와 잘 연결되지 않아 얼마나 우수한 알고리즘인지 인식이 잘 안될 수 있다. 다중 변수를 사용하면 실제 각 변수들의 관계와 입력을 표현하는 방법에 의해 성능 수준이 결정되기 때문에 제시된 빅O만 보고 그것의 성능 수준을 단정할 수 없다. 예를 들어 0-1 배낭 문제를 동적 프로그래밍으로 해결하는 알고리즘[2]의 복잡도는 $O(nW)$인데, 이 알고리즘의 성능은 다차 시간이 아니다.

앞서 언급한 바와 같이 같은 문제를 해결하는 알고리즘을 비교하는 것이 궁극의 목적이며, 같은 문제의 경우에는 고려하는 입력 크기가 같고 내부적으로 유사한 기본 연산을 사용하기 때문에 주어진 복잡도를 이용하여 서로 비교하는 것은 문제가 되지 않는다. 같은 문제를 해결하는 알고리즘의 성능이 각각 $O(m+n)$, $O(mn)$이면 전자가 후자보다 성능이 좋은 알고리즘인 것은 명확하다.

그래프 관련 알고리즘에서 노드의 수가 n이고 간선의 수가 m이면 m의 범위를 n을 이용하여 나타낼 수 있기 때문에 두 개의 변수로 나타낸 빅O를 하나의 변수를 이용한 빅O로 바꾸어 표현할 수 있다. 예를 들어 무방향 연결 그래프에서 m의 범위는 $n-1$부터 $n(n-1)/2$이므로 해당 그래프에서 어떤 문제를 해결하는 알고리즘의 성능이 $O(mn)$이면 $O(n^3)$이라고 표현할 수 있다. 빅O는 최악의 경우 분석이기 때문에 이 분석 자체가 틀린 것은 아니지만 문제에서 다루는 그래프에 따라 알고리즘의 성능을 엄밀(tight)하게 표현하지 못할 수 있는 문제가 있다. 즉, 알고리즘의 성능에 영향을 주는 변수가 여러 개 있을 때, 이들을 모두 사용하여 빅O를 표현하는 것이 성능에 대한 더 많은 정보를 주기 때문에 더 선호한다.

예를 들어 그래프 알고리즘이 m과 n을 이용하여 성능을 표현할 수 있다면 희소 그래프와 밀집 그래프에서 성능이 어떻게 되는지 더 직관적으로 이해할 수 있다. 좀 더 쉽고 정확하게 비교하기 위해 비교하고자 하는 두 알고리즘의 복잡도를 일부러 어떤 형식이 되도록 유도할 수 있다. 예를 들어 그래프 관련 알고리즘에서 빅O가 $O(m \log m)$일 때, m이 최대 n^2보다 작다면 빅O를 $O(m \log n)$으로 표현할 수 있다. 물론 $O(n^2 \log m)$, $O(n^2 \log n)$으로 표현할

[1] 그래프 관련 알고리즘은 주로 노트 7과 9에서 학습함
[2] 동적프로그래밍을 이용한 0-1 배낭 문제는 10장에서 학습함

수 있지만 $m > n$이므로 두 개를 모두 사용하는 표현 중 $O(m \log n)$을 가장 많이 사용한다. 이와 같은 기준으로 표현해야 유사 알고리즘을 비교할 때 더 정확한 비교가 가능하다.

0-1 배낭 문제를 해결하는 알고리즘의 시간 복잡도 $O(nW)$에 대해 좀 더 부연 설명하면 다음과 같다. 빅O 분석은 입력 크기가 커짐에 따라 수행 시간의 증가 비율에 초점을 두고 있다. 예를 들어 $O(n)$이면 입력 크기가 증가함에 따라 수행 시간도 크기에 비례하여 증가한다는 것을 의미한다. 그런데 $O(nW)$의 경우 두 변수 중 하나가 고정된 경우 알고리즘의 수행시간 증가 속도가 달라진다. 0-1 배낭 문제의 경우 W가 고정된 상태에서 n이 증가하면 수행 시간은 선형적으로 증가하지만 n이 고정된 상태에서 W가 증가하면 기하급수적으로 증가한다.

4. 알고리즘의 빅O 분석 Tip

보통 비재귀적 알고리즘은 그 알고리즘을 구성하는 반복문에 의해 시간 복잡도가 결정된다. 반면에 재귀 알고리즘은 재귀 호출 트리를 그려보고, 이 트리를 기본적으로 분석해야 한다. 이 분석에서 트리의 높이와 노드의 최대 자식 수가 중요하다. 트리 높이가 입력 크기 n에 대해 n이고, 각 노드에서 최대 k개 재귀 호출이 이루어지며, 비재귀 과정에서 비용이 $O(1)$이면 이 알고리즘의 시간 복잡도는 $O(k^n)$이다. 이것이 성립하는 이유는 재귀 호출 트리의 각 노드에서 소요되는 시간의 총합이 알고리즘의 수행 시간이 되기 때문이다.

각 노드마다 항상 k개 재귀 호출이 이루어지면 전체 재귀 호출 수는 k^n이다. 하지만 각 노드마다 이루어지는 재귀 호출 수가 가변적이지만 최대 k개이면 전제 재귀 호출 수는 k^n보다 적다. 실제 전체 재귀 호출 수를 식으로 표현할 수 있으면 그것을 이용할 수 있지만 식을 유도하는 것이 어려울 수 있다. 빅O는 점근적 분석이며, 큰 그림 분석이므로 이 예를 $O(k^n)$로 분석하는 것이 틀린 것은 아니다. 앞서 설명한 바와 같이 엄격한 상한을 찾는 것이 어려운 경우가 많다. 재귀 알고리즘에서 재귀 호출의 입력 크기가 항상 같고 같은 수의 재귀 호출이 이루어지면 4장에서 설명하는 도사 정리를 적용하여 기계적으로 빅O를 분석할 수 있다.

4.1 알고리즘 빅O 분석 결과를 해석할 때 주의사항

빅O는 입력 크기가 작으면 의미가 없다. 최고차 항의 계수를 포함하여 계수를 제외하고 나머지 항들을 모두 제거하였기 때문에 더 우수한 성능 수준으로 분류된 알고리즘의 실행 속도가 실제 더 느릴 수 있다. 빅O는 보통 최악의 경우 성능을 나타낸다. 하지만 입력 데이터에 따라 성능 편차가 크고 알고리즘을 적용한 환경에서 최악의 경우가 전혀 나타나지 않을 수 있다. 빅O는 점근적 분석이기 때문에 제시된 알고리즘의 빅O가 엄밀한 상한이 아닐 수 있다. 보통 성능 분석이 어려운 경우 엄밀하지 않을 확률이 높다.

다중변수 빅O는 성능에 여러 요소가 복합적으로 작용하는 것을 의미하기 때문에 각 요소가 어떤 영향을 주는지 알 수 있도록 하기 위해 모든 요소가 나타나도록 빅O를 보통 제시한다. 다중변수 빅O는 그 자체만으로는 성능 수준을 단정할 수 없다.

5. 공간 복잡도

공간 복잡도는 주어진 입력의 크기와 알고리즘 내부적으로 사용하는 메모리 공간에 의해 정의된다. 같은 문제의 알고리즘을 비교하기 때문에 비교에서는 알고리즘 내부적으로 사용하는 메모리 공간만 중요하다. 재귀 알고리즘은 재귀 호출의 깊이만큼 함수 스택 공간을 사용한다. 따라서 (재귀 호출의 최대 깊이 × 함수 스택 공간의 크기)를 공간 복잡도에 포함해야 한다.

이 알고리즘은 항상 재귀 호출의 깊이가 n이며, 스택 공간의 크기는 매개 변수 n 하나이므로 공간복잡도는 $O(n)$

알고리즘 1.10 계승 알고리즘

```
1: function FACTORIAL(n)
2:     if n = 1 then return 1
3:     else return n × FACTORIAL(n − 1)
```

이다.

알고리즘을 수행하는 동안 사용한 전체 공간이 아니라 가장 많은 공간을 사용한 순간의 공간 크기를 고려하여 공간 복잡도를 분석한다. 그래프의 경우 사용하는 표현 방법(인접 리스트 또는 인접 행렬)에 따라 사용하는 공간이 다르다. 인접 리스트는 간선에 비례하며, 인접 행렬은 무조건 $O(n^2)$이 필요하다. 보통 in-place로 문제를 해결한다는 것은 추가 공간을 사용하지 않는다는 것을 말한다. 합병 정렬은 추가 공간을 사용하지만 퀵 정렬, 힙 정렬은 추가 공간을 사용하지 않는다.

퀴즈

1. 알고리즘이 입력에 따라 성능이 달라질 수 있는 경우, 이 알고리즘의 성능을 최선의 경우, 평균의 경우, 최악의 경우로 구분하여 분석할 수 있다. 하지만 보통 우리는 최악의 경우를 분석한다. 최악의 경우를 분석하는 이유로 적절하지 않은 것을 모두 고루시오.

   ① 최선의 경우는 최악의 경우와 고려하여 큰 차이가 있을 수 있으며, 최선의 경우에만 좋다는 것이 큰 의미가 없기 때문에 사용하지 않는다.

   ② 평균의 경우는 최악의 경우보다 분석하기 쉽지만 가장 느렸을 때 성능이 평가(알고리즘을 사용하는 측이 느끼는 성능)에 중요하기 때문이다.

   ③ 평균의 경우와 최악의 경우 성능이 같은 경우가 많다.

   ④ 최악의 경우는 자주 발생할 수 있으며, 가장 오래 걸리는 시간을 나타내기 때문에 알고리즘의 종료되는 시점을 예측할 수 있다.

2. 다음 다중 변수 시간 복잡도 중 가장 우수한 알고리즘은?

   ① $O(m+n)$

   ② $O(m \log n)$

   ③ $O(mn)$

   ④ $O(n \log m)$

3. 양의 정수(32비트 정수) X와 0에서 9사이에 정수 k가 주어졌을 때, 정수 X의 각 자릿수가 k인 자릿수의 개수를 계산하는 다음 알고리즘의 빅O는?

```
1: function COUNTDIGITFREQUENCY(X, k)
2:     count := 0
3:     while X > 0 do
4:         if X%10 = k then ++count
5:         X := X/10
6:     return count
```

   ① $O(2^n)$

   ② $O(n)$

   ③ $O(1)$

   ④ $O(\log n)$

4. $T(n) = 3n^3 + n + 10$일 때 다음 중 성립하는 않는 것은?

   ① $O(n^4)$

   ② $\Omega(n^2)$

   ③ $O(n^2)$

④ $\Theta(n^3)$

5. 다음 중 필요한 비교 연산을 고려하였을 때, 항상 고정된 성능을 보이는 정렬 알고리즘, 최선의 경우 가장 성능이 좋은 정렬 알고리즘, 최악의 경우 성능이 가장 좋은 정렬 알고리즘 순으로 나열된 것은?

   ① 버블 정렬, 삽입 정렬, 합병 정렬
   ② 선택 정렬, 합병 정렬, 버블 정렬
   ③ 선택 정렬, 버블 정렬, 합병 정렬
   ④ 합병 정렬, 삽입 정렬, 버블 정렬

연습문제

1. 알고리즘 1.1, 1.2, 1.3의 시간 복잡도와 공간 복잡도를 제시하시오. 이때 알고리즘 1.3의 경우 사용하는 집합 자료구조가 해싱 기반인 경우와 균형 이진 검색 트리 기반인 경우로 나누어 분석 결과를 제시하시오.

2. n개의 수로 구성된 배열에서 가장 많이 등장하는 수를 찾으시오. 구현한 알고리즘의 시간 복잡도와 공간 복잡도를 함께 제시하시오.

3. 주어진 문자열에서 길이가 3인 좋은 부분 문자열의 개수를 찾으시오. 좋은 문자열이란 반복된 문자가 없는 문자열을 말한다. 예를 들어 "xyz"는 좋은 문자열이지만 "aab"는 좋은 문자열이 아니다. 힌트. sliding window 알고리즘을 사용하면 효과적으로 구현이 가능하다.

4. 알고리즘 1.8에 제시된 합병 정렬의 합병 알고리즘을 그대로 구현하면 $O(n \log n)$ 추가 공간을 사용하게 된다. 이 추가 공간을 가장 적게 사용하는 방법을 제시하시오.

5. 자바와 파이썬 표준 라이브러리의 정렬 함수는 Timsort라는 정렬 알고리즘을 사용한다. Timsort는 삽입 정렬과 합병 정렬을 함께 사용한다. 기본적 아이디어는 배열을 작은 크기로 나누고 각 나누어진 분할은 삽입 정렬로 정렬한다. 그다음 이 분할을 합병 정렬의 합병 알고리즘을 이용하여 합병하여 최종 정렬 결과를 도출한다. 보통 Timsort에서 사용하는 분할의 크기는 32 또는 64이다. 이 문제에서는 분할의 크기를 8로 설정하여 Timsort를 구현한다. 예를 들어 크기가 32인 배열이 입력으로 들어오면 크기가 8인 4개의 분할로 나누어 각 분할을 삽입 정렬로 정렬한다. 그다음 첫 두 분할을 합병하고, 세 번째와 네 번째 분할을 합병한 다음 크기가 16인 2개의 분할을 합병하여 최종 정렬 결과를 만든다.