알고리즘 개요
빅O

한국기술교육대학교 컴퓨터공학부 김상진


```
while(!morning){
    alcohol++
    dance++
} #party
```

```
while(!sleep){
    think++
    solve++
} #cse-mode
```

교육목표

- 알고리즘 소개
- 알고리즘 성능 분석
  - 빅O

> - 알고리즘은 입력과 출력에 의해 정의되는 문제를 해결하는 일련의 절차를 말함
> - 한 문제를 해결하는 알고리즘은 다양할 수 있음
> - 한 문제를 해결하는 여러 알고리즘이 있을 때 어떤 것을 선택?
> - 알고리즘의 성능을 분석할 수 있어야 함
> - 알고리즘의 시간 복잡도와 공간 복잡도를 분석할 수 있어야 함
> - 시간 복잡도, 공간 복잡도를 분석할 때 사용하는 기본 기법은 빅O임
> - 빅O를 이해하고 알고리즘이 주어졌을 때 그것의 시간 복잡도와 공간 복잡도를 빅O를 이용하여 분석할 수 있어야 함

- 살펴보는 중요 알고리즘
  - Karatsuba, mergesort
- Algorithm Design Mantra:
  Can We Do Better?

알고리즘 개요 (1/2)

- 홀로 어떤 목적(solving computational problem)을 위해 수행하는 일련의 단계
- 이 수업에서는 어떤 문제에 대해 답을 찾아주는 일련의 단계
  - 문제는 입력과 출력을 통해 정의함
    - 예) 검색 문제
      - 입력. n개의 수로 구성된 배열 A와 정수 v
      - 출력. v가 배열에 있으면 true 없으면 false
- 알고리즘은 명확해야 함 (definiteness)
  - 수행 과정이 모호하지 않아야 함
- 알고리즘은 유한성(종결성) 특성을 가지고 있어야 함 (finiteness)
  - 알고리즘은 반드시 종료(매우 오래 걸릴 수 있음)해야 함
- 알고리즘은 정확해야 함 (correctness)
  - 주어진 입력에 대해 요구되는 출력을 주어야 함

알고리즘이 이 조건들을 갖추고 있지 못할 수 있음

알고리즘 개요 (2/2)

- 알고리즘의 서술
  - 예) 검색 문제에 대한 알고리즘 서술
    - 배열 A의 첫 번째 요소부터 차례로 v와 비교한다.
      비교하는 과정에서 일치된 값이 있으면 true를 출력하고,
      끝까지 비교하였지만 없으면 false를 출력한다.
    - 이와 같이 자연어로 표현할 수 있지만
      - 단점 1. 복잡한 알고리즘을 표현하기 힘듦
      - 단점 2. 이 서술로부터 코드를 작성하는 것이 힘듦
  - 언어와 독립적인 의사코드를 사용하여 표현함

알고리즘 성능 분석 (1/3)

- 한 문제를 해결하는 알고리즘은 다양할 수 있음
- 그러면 이 중에 어떤 알고리즘을 사용하는 것이 효과적인지 판단할 수 있어야 함
  - 다른 문제를 해결하는 알고리즘을 비교 분석하는 경우는 별로 없음
    - 문제의 어려운 정도를 분석할 때는 두 문제를 해결하는 가장 효과적인 알고리즘의 성능을 비교할 수 있음
- 성능 분석을 통해 성능이 우수한 알고리즘을 선택함
  - 성능이 나쁜 알고리즘은 무조건 쓸모가 없나? 아님
    - 성능이 나쁘지만 정확한 알고리즘은 우수한 알고리즘을 테스트할 때 활용할 수 있음
    - 개발해야 하는 응용에서 해결해야 하는 문제의 입력 크기의 범위가 제한적인 경우 어떤 알고리즘을 사용하여도 큰 차이가 없을 수 있음
      - 개발하기 편한 것, 개발자가 잘 알고 있는 것을 선호할 수 있음
- 알고리즘의 성능 분석은 크게 입력에 따라 소요되는 시간과 필요한 공간을 분석함. 전자를 시간 복잡도라 하고, 후자를 공간 복잡도라 함

알고리즘 성능 분석 (2/3)

- 알고리즘의 성능은 입력과 독립적일 수 있고, 입력에 따라 달라질 수 있음
  - 최선(best) 경우, 평균(average) 경우, 최악(worst) 경우 분석이 가능함
  - 최선의 경우 비교는 보통 의미가 없고, 평균의 경우는 분석이 어렵기 때문에 보통 최악 경우 분석을 사용함
- 성능 분석의 결과로 알고리즘을 카테고리화하고 싶은 것임
  - 알고리즘을 성능 수준별로 분류하기 위해 사용하는 것이 빅O임
- 알고리즘의 시간 복잡도를 측정하기 위해 실제 실행 시간을 측정하지 않음
  - 특히, 실행 시간을 측정하여 두 알고리즘을 비교하지 않음
    - 실행 시간을 측정하기 위해서는 알고리즘을 구현해야 함
    - 구현은 개발자에 따라 차이가 있을 수 있고, 실행 시간은 개발 환경, 언어, 실행 환경에 영향을 받음
    - 비교 결과를 통해 알고리즘을 카테고리화 하는 것이 힘듦
  - 참고. 입력 크기에 따라 실행 시간의 변화를 분석하는 것은 의미가 있음

알고리즘 성능 분석 (3/3)

- 시간 복잡도
  - 입력에 따라 필요한 기본 연산의 수를 계산함
    - 기본 연산: 산술연산, 대입, 비교 등
    - 각 기본 연산의 비용(예: 덧셈과 곱셈)은 다르지만 구분하지 않고 분석
      - 같은 문제를 해결하는 알고리즘은 보통 유사 연산 사용
- 공간 복잡도
  - 알고리즘의 입력을 나타내기 위해 필요한 공간과 알고리즘 내부적으로 사용하는 메모리 공간(auxiliary space)을 분석하여 계산함
  - 같은 문제의 알고리즘을 비교하기 때문에 비교에서는 알고리즘 내부적으로 사용하는 메모리 공간이 중요함
  - 참고. in-place 알고리즘
  - 참고. 인접 행렬 vs. 인접 리스트

검색

- 입력. n개의 수로 구성된 배열과 정수 v
- 출력. v가 배열에 있으면 true, 없으면 false

의사코드 작성할 때 색인은 1부터 표시

```
for i := 1 to n do
    if A[i] = v then return true
return false
```

최선, 최악 분석을 위해서는 언제가 최선인지, 최악인지 분석할 수 있어야 함

- 시간 복잡도 분석 (일반 for문 고려: for(int i=0; i<n; ++i)
  - 최선의 경우(best case): 대입연산 1(i=0), 비교연산 2(i<n, A[i]==v) $\Longrightarrow$ 3
  - 최악의 경우(worst case): 대입연산 1, 비교연산 $2n+1$, 증가연산: $n \Longrightarrow 3n+2$
  - 평균 경우(average case): 여러 가정(배열 내에 있는 값을 검색할 확률, 각 위치에 검색 값이 있을 확률 등)이 필요함
- 공간 복잡도 분석
  - 고정된 지역 변수(i, n)만 추가로 사용

최악의 경우 분석을 선호하는 이유
- 분석하기 쉬움
- 많은 경우 평균 비용과 최악 비용이 같음
- 고객 반응 입장에서는 …
  99번 빠르고 1번 느리면…

가장 많이 등장하는 수

- **입력.** n개의 수로 구성된 배열
- **출력.** 가장 많이 등장하는 수

```
M := −1
Mcount := 0
for i := 1 to n do
    V := A[i]
    count := 0
    for j := 1 to n do
        if A[i] = V then count += 1;
    if Mcount < count then
        Mcount := count
        M := V
return M
```

```
H := {} // hashmap with size>1.3n
for i := 1 to n
    if H.containsKey(A[i]) then
        count := H.get(A[i])
        H.put(A[i], count + 1)
    else H.put(A[i], 1)
M := −1
max := 0
for all (k, v) ∈ H do
    if v > max then
        M := k
        max := v
return M
```

공간 vs. 성능

- 해시맵, 트리맵 중 어느 것?
- 첫 번째 for 루프에서 M을 구할 수 있음

중복 요소 존재 여부

- **입력.** n개의 수로 구성된 배열
- **출력.** 중복 요소가 있으면 true 없으면 false
- 배열에 따라 종료 시점이 다름
  - 최악 경우 분석

```
for i := 1 to n − 1 do
    for j = i + 1 to n do
        if A[i] = A[j] then return true
return false
```

```
sort A
for i := 2 to n do
    if A[i] = A[i − 1] then return true
return false
```

```
S := {} // empty set
for i := 1 to n do
    if A[i] ∈ S then return true
    else S.put(A[i])
return false
```

정수 곱셈

- **입력.** 두 개의 n자리 정수 X, Y
- **출력.** 두 수의 곱 $X \times Y$

초등학교 Algorithm

			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

- Y의 각 자리수마다 n개의 곱셈 필요: n
  - 곱셈 과정에서 자리올림 때문에 추가 덧셈 연산 필요: $n \rightarrow 2n$
  - **참고.** 덧셈과 곱셈은 같은 비용의 연산은 아님
  - 비용: $\leq 2n^2$
- 결과를 합하는 과정: $\leq 2n^2$의 덧셈 ($2n \times n$ 크기 배열)
- 총 비용: $\leq 4n^2$

- 정확하게 분석하기 어렵기 때문에 상한을 이용하여 표시함

Algorithm Designer's Mantra

Perhaps the most important principle for the good algorithm designer is to refuse to be content.
Aho, Hopcraft, Ullman,
Analysis of Computer Algorithm, 1974

Can we do better?

정수 곱셈

- **입력.** 두 개의 n자리 정수 X, Y
- **출력.** 두 수의 곱 $X \times Y$
  - $X = 5678, Y = 1234$
  - $a = 56, b = 78, c = 12, d = 34$
  - 단계 1. $ac = 672$
  - 단계 2. $bd = 2652$
  - 단계 3. $(a + b)(c + d) = 6164$
  - 단계 4. $(3) - (2) - (1) = 2840$

- $\left(10^{\frac{n}{2}}a + b\right)\left(10^{\frac{n}{2}}c + d\right) = 10^n ac + 10^{\frac{n}{2}}(\mathrm{ad} + \mathrm{bc}) + \mathrm{bd}$

Karatsuba Algorithm

- 분할 정복(divide-and-conquer)

	6	7	2	0	0	0	0
				2	6	5	2
+		2	8	4	0	0	0
	7	0	0	6	6	5	2

재귀 알고리즘

- $X = 10^{n/2}a + b, Y = 10^{n/2}c + d$
  - a, b, c, d: $n/2$ 자리수
  - $XY = \left(10^{\frac{n}{2}}a + b\right)\left(10^{\frac{n}{2}}c + d\right) = 10^n ac + 10^{\frac{n}{2}}(\mathrm{ad} + \mathrm{bc}) + \mathrm{bd}$
  - ac, ad, bc, bd를 재귀적으로 구함
- Gauss's Trick: $(a + b)(c + d) = ac + ad + bc + bd$
  - $ad + bc = (a + b)(c + d) - ac - bd$
  - 3개의 recursive multiplication만 필요 (추가적인 덧셈)
- 초등알고리즘 vs. Karatsuba 알고리즘: 어떤 알고리즘이 더 빠른가???
- 실제 Karatsuba가 더 효율적임
  - 하지만 우리가 곱셈할 때 사용하지 않을 것임 (인간 vs. 컴퓨터)

정렬 알고리즘

- 입력. n개의 수로 구성된 배열
- 출력. 같은 수로 구성된 오름차순으로 정렬된 배열
- 정렬 알고리즘은 성능에 따라 이차 시간(quadratic time, n^2)이 필요한 것과 의사 선형 시간(quasilinear time, near linear time, $n\log_2 n$)이 필요한 것으로 분류할 수 있음
  - 예) 이차 시간: 삽입 정렬, 선택 정렬, 버블 정렬 등
  - 예) 의사 선형 시간: 합병 정렬, 빠른 정렬, 힙 정렬 등

선택 정렬 (Selection Sort)

```
for i := 1 to n − 1 do
    minLoc := i
    for j := i + 1 to n do
        if A[minLoc] > A[j] then minLoc := j
    if minLoc ≠ i then
        swap(A[minLoc], A[i])
```

- 입력과 무관하게 항상 동일 수의 비교가 필요함
- 반복문의 반복 횟수:
  $n-1+n-2+\ \dots + 1 = n(n-1)/2$
- 최악의 경우는?

- 경우 1. [8 7 6 5 4 3 2 1]
  minLoc = j: $< \frac{n^2}{4}$, swap: $\frac{n}{2}$
- 경우 2. [2 3 4 5 6 7 8 1]
  minLoc = j: $n-1$, swap: $n-1$
  swap이 $n-1$번 필요한 경우는 이 외에도 많음

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

삽입 정렬 (Insertion Sort)

```
for i ≔ 2 to n do
    temp ≔ A[i]
    j ≔ i − 1
    while j ≥ 1 and temp < A[j] do
        A[j+1] ≔ A[j]
        --j
    A[j+1] ≔ temp
```

- 최악의 경우: 거꾸로 정렬되어 있는 경우
  - 반복문의 반복 횟수:
    $1 + \cdots + n - 2 + n - 1 = n(n-1)/2$
- 최선의 경우: 이미 정렬되어 있는 경우
  - 반복문의 반복 횟수: $n - 1$

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

버블 정렬 (Bubble Sort)

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

```
for i ≔ 1 to n − 1 do
    flag ≔ false
    for j ≔ n downto i + 1 do
        if A[j-1] > A[j] then
            swap(A[j-1], A[j])
            flag ≔ true
    if not flag then break
```

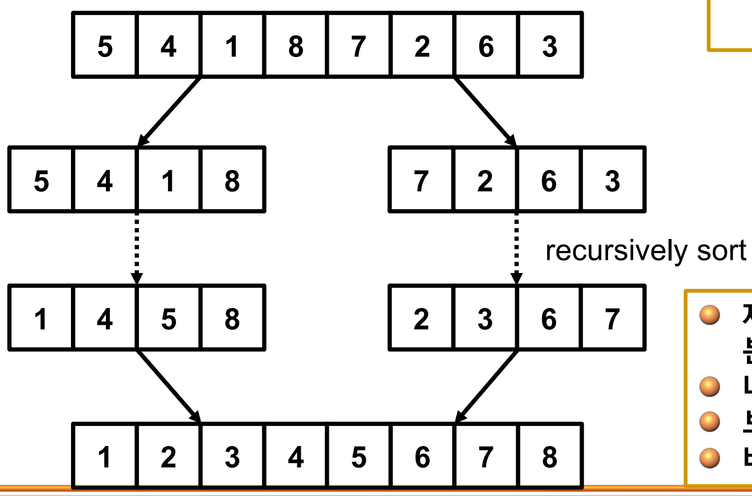
- 최악의 경우: 거꾸로 정렬되어 있는 경우
  - 반복문의 반복 횟수:
    $n - 1 + n - 2 \pm \cdots + 1 = n(n-1)/2$
- 최선의 경우: 이미 정렬되어 있는 경우
  - 반복문의 반복 횟수: $n - 1$

합병 정렬 (1/3)

canonical divide-and-conquer algorithm

- 분할 정복(divide-and-conquer) 방식의 알고리즘
- 재귀 알고리즘
  - 단계 1. 재귀적으로 왼쪽 절반을 정렬
  - 단계 2. 재귀적으로 오른쪽 절반을 정렬
  - 단계 3. 두 반을 합병

- 재귀 알고리즘은 기저 사례에 대한 고려 필요 (재귀의 중단)

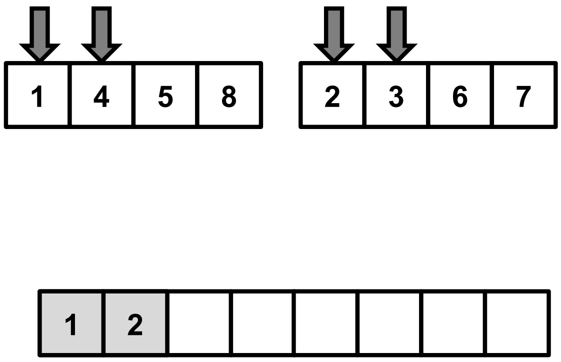


- 재귀 알고리즘의 시간 복잡도는 분석하기 쉽지 않음
- 나중에 도사 정리 배우면 간단
- 보통 재귀호출이 얼마나 많이?
- 비재귀 과정에서 일어나는 비용은?

합병 정렬 (2/3)

- merge

$L \coloneqq 1$
$R \coloneqq 1$
for $k \coloneqq 1$ ***to*** n ***do***
    if $A[L] < B[R]$ ***then***
        $C[k] \coloneqq A[L]$
        $++L$
    else
        $C[k] \coloneqq B[R]$
        $++R$
(move remaining B or A to C)



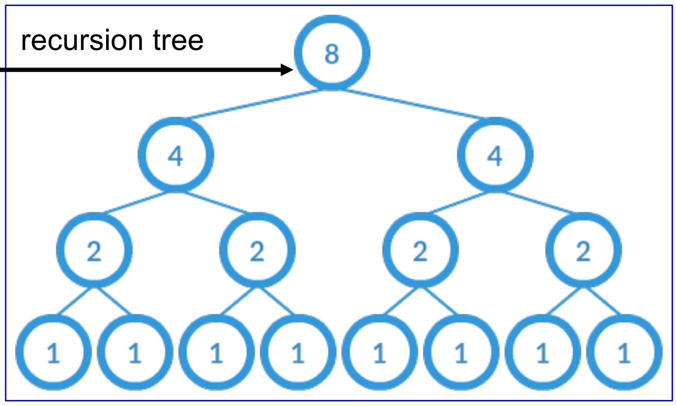
- 크기가 $n/2$인 2개의 배열에 대한 merge 수행 시간 $\leq 4n + 2 \leq 6n$
  - 4: 비교, 대입, 증가(2, *k*와 *L* 또는 *R*)

- L, R. k: 초기화 3
- L<n/2, R<n/2, A[L]<B[R]: 비교 3
- C[k] ≔ A[L] 또는 C[k] ≔ B[R]: 대입 1
- ++k, ++L 또는 ++R: 증가 2
- $6n + 3 \leq 9n$
- $6n$, $9n$이 중요한 것은 아님. 중요한 것은 선형이라는 것임

합병 정렬 (3/3)

- 전체 비용: $\le 6n\log_2 n + 6n$
  - Why? 재귀 트리의 깊이: $\log_2 n + 1$
    - 레벨 j: 2^j개의 재귀 호출이 이루어지고, 배열의 크기는 $n/2^j$
    - 레벨 j에서 연산 수: $\le 2^j \times 6 \times \left(\frac{n}{2^j}\right) = 6n$

Level 0:




```
mergeSort(A, lo, hi):
    If lo < hi then
        mid := lo + (hi-lo)/2
        mergeSort(A, lo, mid)
        mergeSort(A, mid+1, hi)
        merge(A, lo, mid, hi)
```

- 재귀호출 트리의 높이는 $\log_2 n$에 비례
- 각 레벨에서 소요 비용은 같음.
  나누어서 처리하지만 전체적으로 참여
- 각 레벨에서 소요 비용은 n에 비례

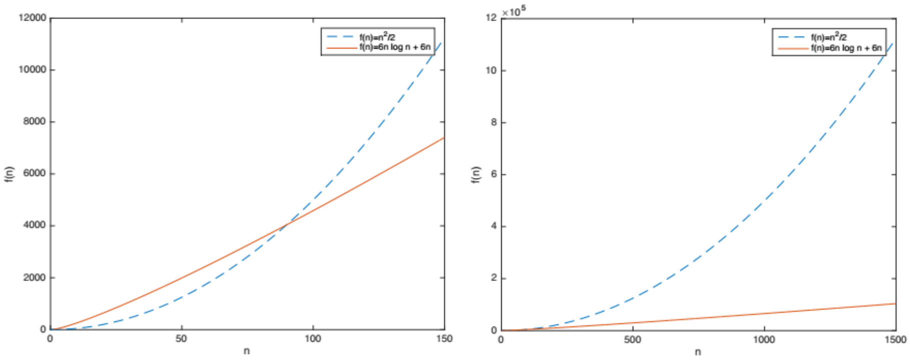
빅O 분석

$6n\log_2 n + 6n$

- 원리 1. 최악의 경우 분석
- 원리 2. 큰 그림 분석
  - 상수 계수(constant factors)와 낮은 차수 항(low-order terms)은 무시
    - 전자는 시스템 의존적 요소, 후자는 큰 입력에 대해서는 의미가 없음
    - Why? 간단함
    - Why? 상수는 프로그래밍 언어, 실행 환경 등에 의해 결정되는 요소임
    - Why? 이들을 무시해도 예측 능력, 비교 능력은 유지됨
  - 상수 계수, 낮은 차수 항이 항상 무의미한 것은 아님
    - 같은 수준의 알고리즘을 비교할 때는 필요함
- 원리 3. 점근적(asymptotic) 분석
  - 입력 크기 n이 증가됨에 따라 알고리즘 수행 시간의 증가 비율에 초점
  - 입력 크기가 작으면 시간 복잡도가 큰 의미가 없음

예) 합병 정렬 vs. 삽입 정렬

- $6n\log_2 n + 6n$ vs. $\frac{1}{2}n^2$



(a) Small values of n (b) Medium values of n

- n이 작으면 오히려 n^2이 $n\log_2 n$보다 우수
  - 입력 크기가 작으면 삽입 정렬이 합병 정렬보다 우수
- n이 클 경우에만 상수 계수와 낮은 차수 항들을 무시할 수 있음 (큰 그림, 점근적 분석)

빠른 알고리즘이란?

- 빠른 알고리즘: 최악의 경우 수행시간이 입력 크기에 따라 느리게 증가하는 알고리즘
  - 누가 더 느리게 증가할까?
    - 증가 비율(rate of growth)이 중요
  - 기준점은 선형 시간(입력 크기)임

시간복잡도 분석 예 (1/2)

- 반복문의 형태를 잘 관찰할 필요가 있음

- 입력: n개 정수로 구성된 배열, 정수 v
- 출력: 정수 v가 배열에 존재하면 true 없으면 false

```
for i := 1 to n do
    if A[i] = v then return true
return false
```

- 최악의 경우: v가 배열에 없는 경우
- 총 비용: n
- $O(n)$

- 입력: n개 정수로 구성된 배열 A와 B, 정수 v
- 출력: 정수 v가 A 또는 B에 존재하면 true 없으면 false

```
for i := 1 to n do
    if A[i] = v return true
for i := 1 to n do
    if B[i] = v return true
return false
```

```
for i := 1 to n do
    if A[i] = v return true
    if B[i] = v return true
return false
```

- 최악의 경우: v가 A와 B 배열에 모두 없는 경우
- 총 비용: $2n$
- $O(n)$

시간복잡도 분석 예 (2/2)

- 입력: n개 정수로 구성된 배열 A와 B
- 출력: A와 B에 공통으로 어떤 정수 v가 존재하면 true 없으면 false

```
for i := 1 to n do
    for j := 1 to n do
        if A[i] = B[j] return true
return true
```

- 중첩 반복문
- 최악의 경우: A와 B 배열에 공통 요소가 없는 경우
- 총 비용: n^2
- $O(n^2)$

- 입력: n개 정수로 구성된 배열
- 출력: A에 중복 요소가 있으면 true 없으면 false

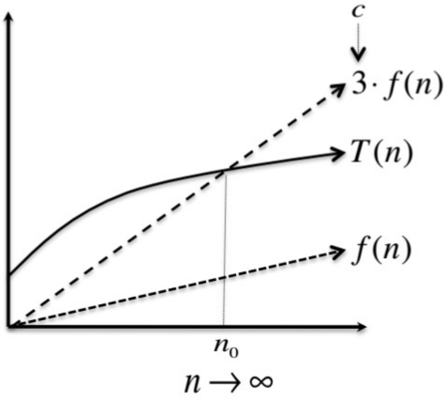
```
for i := 1 to n − 1 do
    for j := 1 + 1 to n do
        if A[i] = A[j] then return true
return false
```

- 최악의 경우: 총 $n-1, n-2, \ldots, 1$개의 비교가 필요함
- 총 비용: $\frac{n(n-1)}{2}$
- $O(n^2)$

Big-O

- 입력 크기 n에 대해 최악의 경우 알고리즘 수행 시간을 나타내는 함수 $T(n)$이 $c \times f(n)$보다 빠르면(증가속도가 느리면, 우수하면) 이 알고리즘의 빅O는 $O(f(n))$이라 함

빅O: upper bound



- 여기서 $f(n)$는 $n, n^2, \log n$과 같은 성능 분류 대표 함수임
- 빅O는 점근적 접근

- $f(n)$과 같거나 나쁠 수 없음
- $f(n)$과 같거나 좋음(우수함)
- tight한 upper bound로 설명하고 싶음

  - $T(n) \in O(f(n))$ iff there exist positive constants c and n_0 s.t.
  $$T(n) \le c \cdot f(n)$$
  for all $n \ge n_0$.

Big-O

- $\forall n \ge n_0,\ \ T(n) \le c \cdot f(n)$
- Claim. If $T(n) = a_k n^k + \cdots + a_1 n + a_0$ then $T(n) = O(n^k)$.
  - 증명) $n_0 = 1, c = |a_k| + \cdots + |a_1| + |a_0|$
- Claim. For every $k \ge 1$, n^k is not $O(n^{k-1})$.
  - 증명) 모순 증명
    - 성립한다면 모든 $n \ge n_0$에 대해 $n^k \le c \cdot n^{k-1}$를 만족하는 c, n_0가 존재해야 함
    - 이것의 의미: 모든 $n \ge n_0$에 대해 $n \le c$임
      - $n = c + 1$이면 $n \le c$가 성립할 수 없음
- $T(n) = O(n^k)$이면 $m > k$에 대해 $T(n) = O(n^m)$임

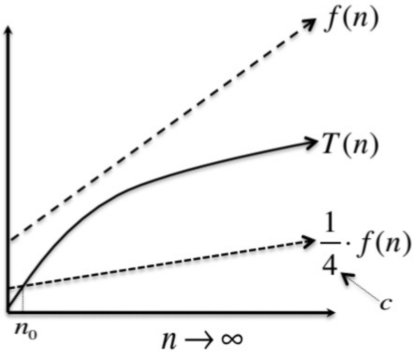
Big-Omega

- **Big-Omega 정의:** $T(n) \in \Omega(f(n))$ **iff there exist positive constants** c **and** n_0 **s.t.**

$$T(n) \geq c \cdot f(n)$$

for all $n \geq n_0$.

빅O: lower bound



- $f(n)$과 같거나 좋을 수 없음
- $f(n)$과 같거나 나쁨

- 입력 크기 n에 대해 최악의 경우 알고리즘 수행 시간을 나타내는 함수 $T(n)$이 $c \times f(n)$보다 느리면(증가속도가 빠르면) 이 알고리즘의 빅Ω는 $\Omega(f(n))$이라 함
- 빅Ω는 대략적 분석을 할 수 없음. $\Omega(n^2)$으로 분석하였는데 실제 $\Omega(n)$이면 분석이 틀린 것임

Big-Theta

- **Big-Theta의 정의:** $T(n) = \Theta(f(n))$ **iff there exist positive constants** c_1, c_2 **and** n_0 **s.t.**

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$$

for all $n \geq n_0$.

  - $T(n) = \Theta(f(n))$ **iff** $T(n) = O(f(n))$ **and** $T(n) = \Omega(f(n))$.

- $f(n)$과 같음

- $T(n) = \frac{1}{2}n^2 + 3n$일 때 다음 중 성립하는 것은?
  - $T(n) = O(n)$
  - $T(n) = \Omega(n)$
  - $T(n) = \Theta(n^2)$
  - $T(n) = O(n^3)$

- $O(f(n))$: $f(n)$과 같거나 좋음
- $\Omega(f(n))$: $f(n)$과 같거나 나쁨
- $\Theta(f(n))$: $f(n)$과 같음

- 증명하는 방법: n_0와 c 제시
- 예) $n_0 = 1, c_1 = \frac{1}{2}, c_2 = 4$를 이용하면 $\Theta(n^2)$임을 보일 수 있음

- 알고리즘이 $\Theta(f(n))$이면 $O(f(n))$와 $\Omega(f(n))$이 성립
- 알고리즘이 $\Theta(f(n))$이 아니면 $O(f(n))$와 $\Omega(f(n))$이 같을 수 없음

Little-o Notation

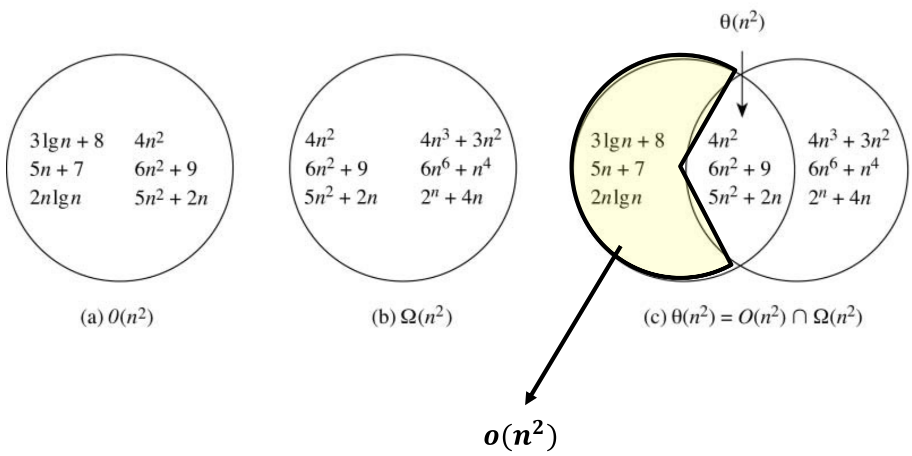
- Little-o 정의: $T(n) \in o(f(n))$ iff **for every positive** constants $c > 0$ there exits a n_0 s.t.

$$T(n) \leq c \cdot f(n)$$

  for all $n \geq n_0$.
- **Claim.** For all $k \geq 1, n^{k-1} \in o(n^k)$.
- Big O vs. Little o
  - 큰 *O*: 하나의 상수 $c > 0$에 대해서만 성립하면 됨
  - 작은 *o*: 모든 실수 $c > 0$에 대해 성립해야 함

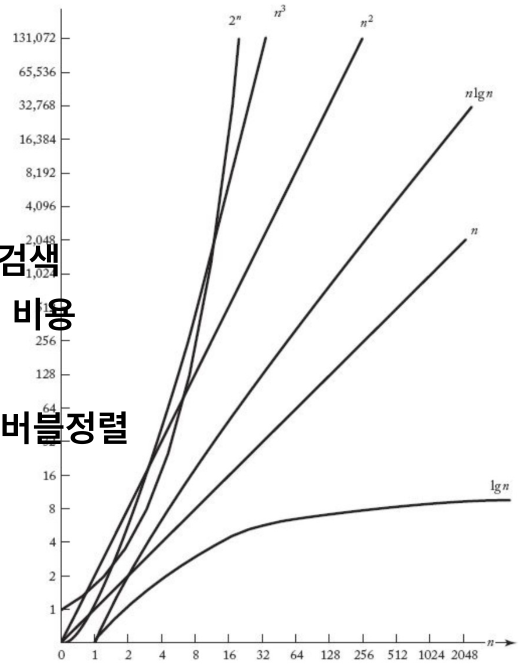
각 분석의 비교

- 요약
  - $O(f(n))$: $f(n)$보다 더 느릴 수 없는 알고리즘 (증가속도가 느린)
  - $\Theta(f(n))$: $f(n)$과 성능이 같은 알고리즘 (증가속도가 같은)
  - $\Omega(f(n))$: $f(n)$보다 빠를 수 없는 알고리즘 (증가속도가 빠른)



Common Order

- $O(1)$: 상수시간(constant time) 비용. 입력 크기에 전혀 영향을 받지 않는 경우
  - 예) 배열 끝에 요소 저장하기
- $O(\log n)$: 로그시간(logarithmic time) 비용. 한 번에 처리해야 하는 양이 반씩 줄어드는 경우
  - 예) 이진 검색
- $O(n)$: 선형시간(linear time) 비용. 예) 선형 검색
- $O(n\log n)$: 의사선형시간(quasilinear time) 비용
  - 예) 합병 정렬, 퀵정렬
- $O(n^2)$: 이차시간(quadratic time) 비용. 예) 버블정렬
- 다차시간(polynomial time) 비용
- $O(2^n)$: 지수시간(exponential time) 비용
- $O(n!)$: 계승시간(factorial time) 비용



빅O 분석과 입력 크기

- 입력의 크기
  - 예) 리스트에 특정 요소 존재 여부: 리스트의 크기
  - 예) 주어진 양의 정수가 소수인지 판단: ???
    - 정수가 커질수록 소요 비용은 증가함
    - 정수가 커질수록 그것을 표현하는데 소요되는 비트 수는 증가함
  - 빅O에서 입력의 크기는 정확하게는 그 입력을 컴퓨터에 표현하기 위한 문자 수임 (그 입력을 작성하기 위해 타이핑해야 하는 키보드 수)
    - 정수 x를 표현하기 위한 비트 수: $n \approx \log_2 x \rightarrow x \approx 2^n$
    - 최악의 경우 시간복잡도: $T(x) = x^{1/2} \rightarrow T(n) = 2^{n/2} = O(2^n)$

```
bool isPrime(x):
    for i := 2 to √n do
        if x % i = 0 then return false
    return true
// log₂ x
```

```
find(A[], v):
    for i := 1 to k do
        if A[i] = v then return true
    return false
// n = k log₂ L
```

- $T(k) = 3k + 2 = \frac{3n}{\log_2 L} + 2$
- $O(n)$

소인수 분해 알고리즘

- 입력. 양의 정수 N
- 출력. N의 소인수

```
M := []
k := 1
d := 2
while N > 1 do
    while N % d = 0 do
        N := N/d
        M[k++] := d
    ++d
return M
```

- 최악의 경우: N이 소수인 경우
- while 문의 반복 횟수: $N-1$
- 그러면 $O(n)$???

- N의 커질수록 N를 저장하기 위해 필요한 공간이 증가함
- 입력이 차지하는 비트 수가 k일 때 최악의 경우 필요한 반복 횟수 $2^k - 2 \Rightarrow O(2^n)$

다중 변수 빅O

- 하나의 변수가 아니라 여러 개의 변수를 이용하여 빅O를 표현하는 경우가 종종 있음
  - 예) 그래프 알고리즘에서 노드 수가 n이고 간선 수가 m일 때, 인접 리스트를 이용한 BFS, DFS의 시간 복잡도는 $O(n+m)$
- 다중 변수 빅O는 표준 복잡도와 잘 연결되지 않음
- 같은 문제를 해결하는 알고리즘의 비교가 목적이므로 다중 변수를 이용한 시간 복잡도가 주어졌을 때, 어느 것이 더 우수한지는 쉽게 판단할 수 있음
  - 예) $O(m+n)$이 $O(mn)$보다 우수함
- 다중 변수 빅O에서 두 변수 간 어떤 관계가 성립하면 단일 변수 빅O로 바꾸어 분석할 수 있음
  - 무방향 연결 그래프에서 간선 수 m의 범위는 $n-1$에서 $n(n-1)/2$이므로 $O(mn)$을 $O(n^3)$으로 분석할 수 있음
- 하지만 각 변수가 알고리즘에 어떻게 영향을 주는지 알 수 있기 때문에 다중 변수 빅O를 그대로 사용하는 것이 바람직함

이동 평균 계산하기

- 입력. n개의 정수와 구간 M
- 출력. M 구간 평균값의 리스트
  - 예) 입력: 1 5 4 3 2, 2 → 출력: 3.0 4.5 3.5 2.5

- 다중 변수 빅O를 설명하기 위한 예는 아님

```
M := []
k := 1
for i := M to n do
    sum := 0
    for j := 1 to M do
        sum += A[i − j + 1]
    M[k++] := sum / M
return M
```

- $(n-M+1) \times M$
- $O(nM)$

- M이 $n/2$일 때 가장 많은 연산이 필요함
- M의 범위는 1부터 n이므로 엄밀한 상한은 아니지만 $O(n^2)$으로 분석할 수 있음

```
M := []
k := 1
sum := 0
for i := 1 to M − 1 do
    sum += A[i]
for i := M to n do
    sum += A[i]
    M[k++] := sum / M
    sum −= A[i − M + 1]
return M
```

- $O(n)$

- Sliding window 알고리즘

공간 복잡도 분석

- 재귀 호출을 이용하는 알고리즘은 재귀 호출의 깊이만큼 함수 스택 공간을 사용함
  - 이 공간이 공간 복잡도에 포함해야 함
  - 예)

```
factorial(n):
    if n = 1 then return 1
    else return n * factorial(n − 1)
```

  - 항상 재귀 호출의 깊이가 n이며, 스택 공간의 크기는 매개 변수 n 하나이므로 공간 복잡도는 $O(n)$임
- 알고리즘을 수행하는 동안 사용한 전체 공간이 아니라 가장 많은 공간을 사용한 순간의 공간 크기를 고려하는 것임