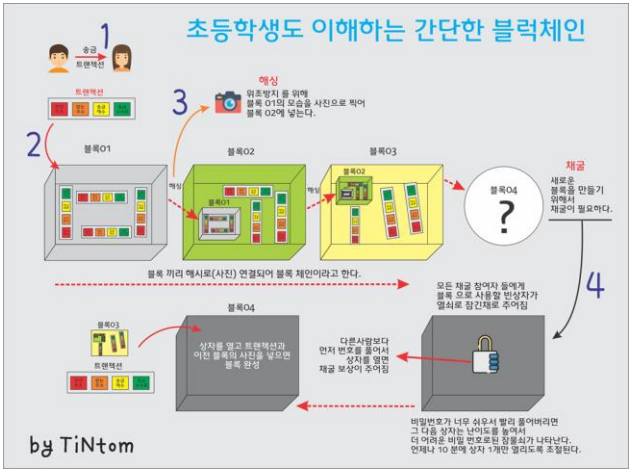
Introduction to data structures and algorithm

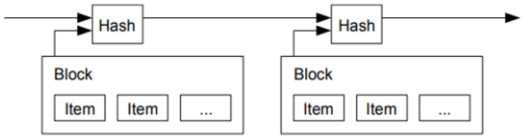
Hyerim Bae
Department of Industrial Engineering, Pusan National University
hrbae@pusan.ac.kr

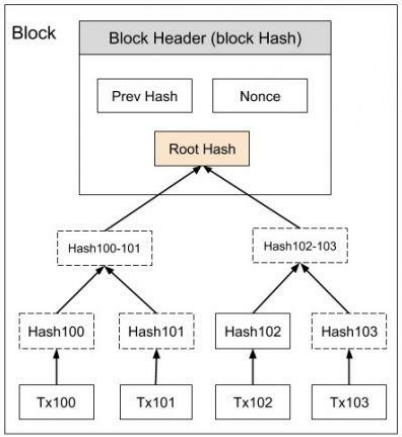
블록체인은 어떻게 구현되어 있을까요?

- 블록과 체인



Source: https://steemkr.com/kr/@tintom/2fgvq8





Transactions Hashed in a Merkle Tree

Source: https://needjarvis.tistory.com/634

Data and information

- Data
  - Set of values that are obtained by observation, measuring
  - Expression about facts or concept
  - Numeric, Characters
- Information
  - Assignments of meaning to data
  - Knowledge about decision

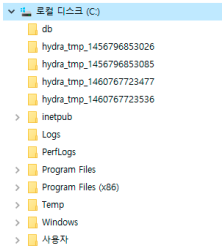
Data structure

- Definition
  - Data representation
  - Relation among data items
  - Sometimes it means ‘Algorithm’
- Data type
  - Numerical
    - Decimal
      - Unpacked decimal
      - Packed decimal
    - Binary
      - Integer
      - Float
  - Non-numerical
    - Character
      - BCD
      - EBCDIC
      - ASCII
    - Logical
    - Pointer

- Types of Data structure
  - Linear
    - 배열, 벡터, 리스트, 스택, 큐
  - Non-linear
    - 트리
    - 그래프
  - File structure
    - 순차파일
    - 색인파일
    - 직접파일

e-Board

Contacts



Directory

$$(x+a)^n = \sum_{k=0}^{n} \binom{n}{k} x^k a^{n-k}$$

Math

- Selecting data structure
  - Amount of data
  - Frequency of using data
  - Characteristics of data (Dynamic, or static?)
  - Capacity of memory
  - Access time
  - Easy of programming

Algorithm

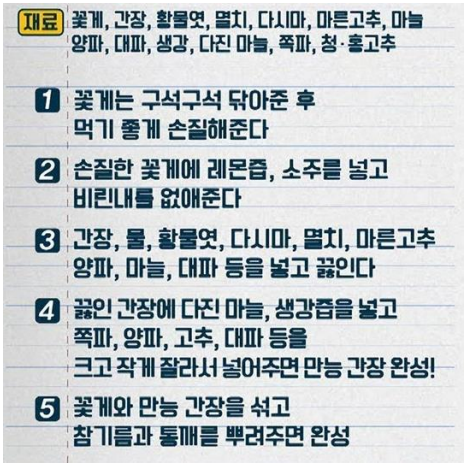
- 5 elements of algorithm
  - Input
  - Output
  - Definiteness
  - Finiteness, Terminate
  - Effectiveness
- Languages for algorithm
  - Natural language
  - Flow chart
  - Pseudo Code *
  - Programming language
    - Low level language: Assembler, machine language
    - High level language: Interpreter, Compiler

알고리즘(Algorithm)

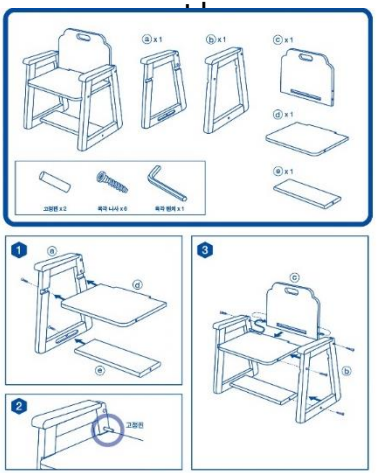
- **알고리즘이란**
  문제를 해결하기 위하여 특정한 형태의 입력(input)을 가지고 원하는 결과(output)을 얻어낼 수 있는 잘 정의된(well-defined) 절차

- **우리 생활에서 발견할 수 있는 알고리즘**

요리 레시피



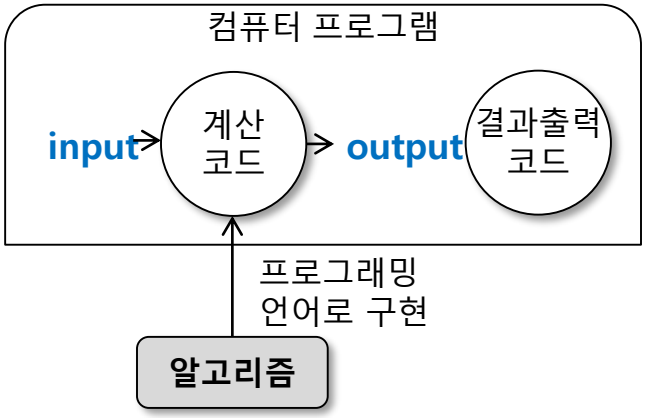
제품 조립설명



프로그램(Program)

- **알고리즘이란**
  문제를 해결하기 위하여 특정한 형태의 입력(input)을 가지고 원하는 결과(output)을 얻어낼 수 있는 잘 정의된(well-defined) 절차

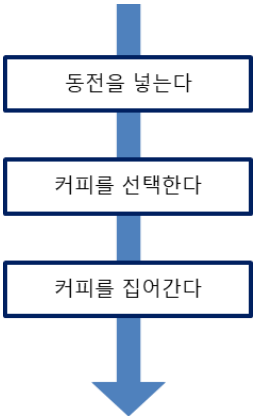
- **프로그램(Program)**
  알고리즘에 정의된 절차를 컴퓨터로 하여금 수행하도록 하기 위해 기술한 지시문



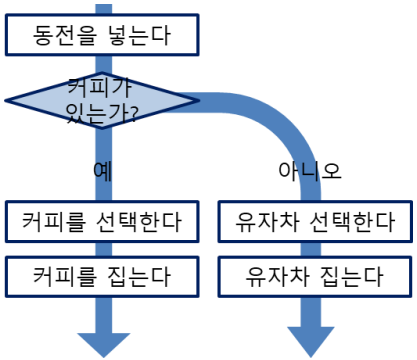
알고리즘의 구조

- **알고리즘을 구성하는 세가지 구조**
  알고리즘에 사용되는 절차는 세 가지 구조의 조합으로 구성되어 있다.

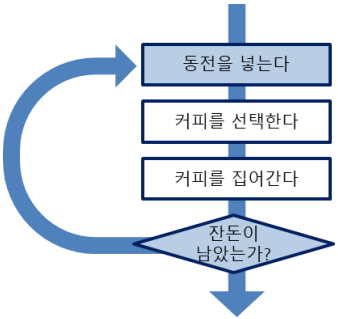
순차구조



선택구조

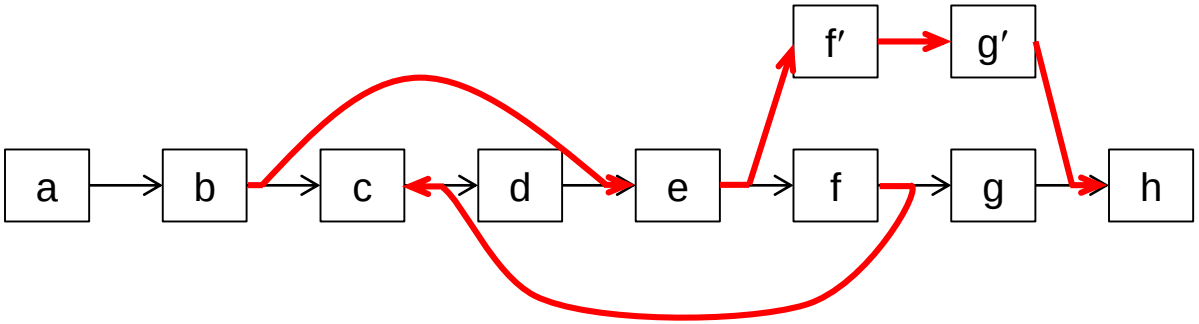


반복구조



결국 알고리즘은

- 복잡한 일도 작은 단위로 쪼개면 단순한 작업들의 흐름이 된다.
- 알고리즘은 순차+선택+반복을 조합하여 선택과 반복
- 복잡한 흐름은 순차적으로만은 표현할 수 없다.



Algorithm

- Languages for algorithm
  - Natural language
  - Flow chart
  - Pseudo Code *
  - Programming language
    - Low level language: Assembler, machine language
    - High level language: Interpreter, Compiler

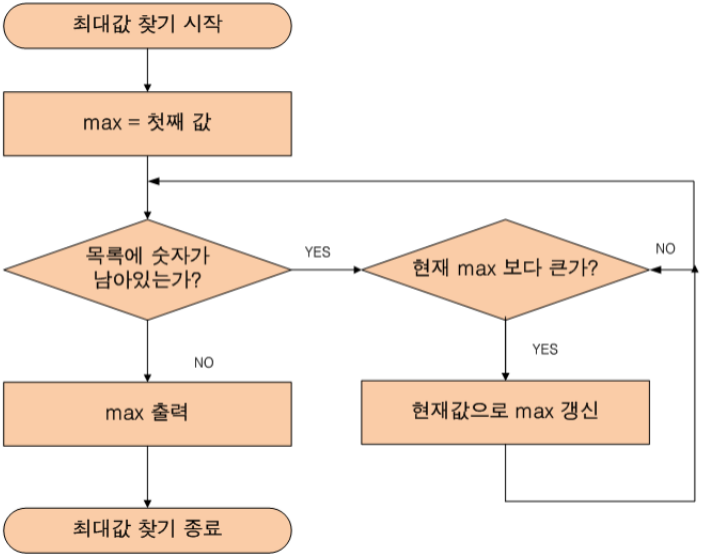
알고리즘의 체계적 표현법 - 순서도

- 알고리즘에 사용되는 처리와 절차를 도형 기호를 사용해 시각적으로 표현
- 직관적이고 표현이 쉽지만, 복잡한 경우 오히려 알고리즘의 이해를 방해

기호	의미
단말	알고리즘의 시작과 끝
처리	알고리즘의 시작과 끝
선택	비교, 판단을 통한 분기
	절차의 흐름방향 표시

순서도 예시

- 주어진 5개의 수에서 최대값을 찾는 알고리즘



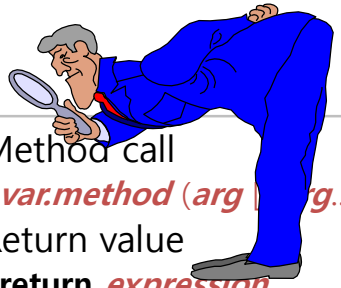
알고리즘의 체계적 표현법 - 유사코드

- 자연어보다는 구조적이지만, 실제 프로그래밍 언어보다는 덜 구체적인 표현방법
- 알고리즘의 핵심적 내용에 집중하고, 프로그램을 구현할 때 여러가지 문제들을 감출 수 있음.
- 예시: 유사코드의 작동원리

Pseudocode (§3.2)

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Pseudocode Details



- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***…])

  Input …

  Output …

- Method call

  var.method (***arg*** [, ***arg***…])
- Return value

  return ***expression***
- Expressions
  - ← Assignment (like = in Java)
  - = Equality testing (like == in Java)
  - n^2 Superscripts and other mathematical formatting allowed
- Array indexing
  - *A*[i] represents the *i*-th cell in the array
  - The cells of an *n*-celled array *A* are indexed from *A*[0] to *A*[n-1]

유사코드 예시

Algorithm *arrayMax*(*A*, *n*)
Input array *A* of *n* integers
Output maximum element of *A*

currentMax ← *A*[0]
for *i* ← 1 **to** *n* − 1 **do**
if *A*[*i*] > *currentMax* **then**
currentMax ← *A*[*i*]
return *currentMax*

3. A list of n distinct integers $a_1, a_2, \ldots, a_n$ is called a *mountain list* if the elements reading from left to right first increase and then decrease. The location of the *peak* is the value i where a_i is greatest. For example, the list $1, 2, 3, 7, 6$ is a mountain list with a peak at 4 since the greatest number occurs in the fourth position. We also consider a list of n increasing numbers to be a mountain list with a peak at n, and a list of n decreasing numbers to be a mountain list with a peak at 1.

(a) Give pseudocode for an algorithm based on linear search that takes as input a mountain list $a_1, a_2, \ldots, a_n$ and returns the location of the peak.

procedure LinearPeak($a_1, a_2, \ldots, a_n$: mountain list)
1. **for** $i := 1$ to $n - 1$
2. **if** [ ]
3. **return** i
4. **return** n

(b) (5 points) Give an algorithm based on binary search that takes as input a mountain list $a_1, a_2, \ldots, a_n$ and returns the location of the peak.

procedure BinaryPeak($a_1, a_2, \ldots, a_n$: mountain list)
1. $i := 1$
2. $j := n$
3. **while** $(i < j)$
4. $m := \lfloor \frac{i+j}{2} \rfloor$
5. **if** [ ]
6. $i := m + 1$
7. **else**
8. $j := m$
9. **return** i

알고리즘 예시: 탐색과 정렬

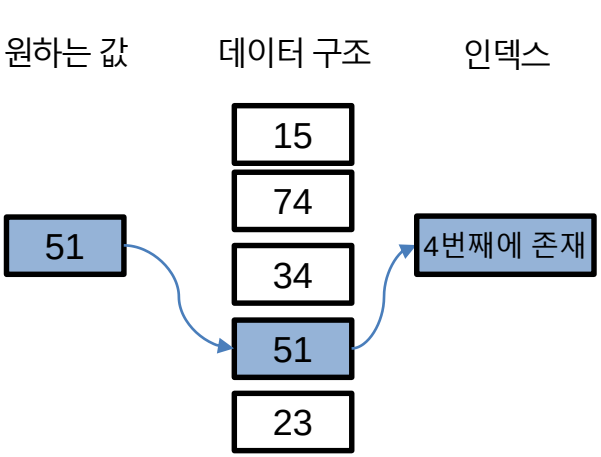
- 가장 기초적이고 대표적으로 활용되는 두 알고리즘: {탐색}과 {정렬}

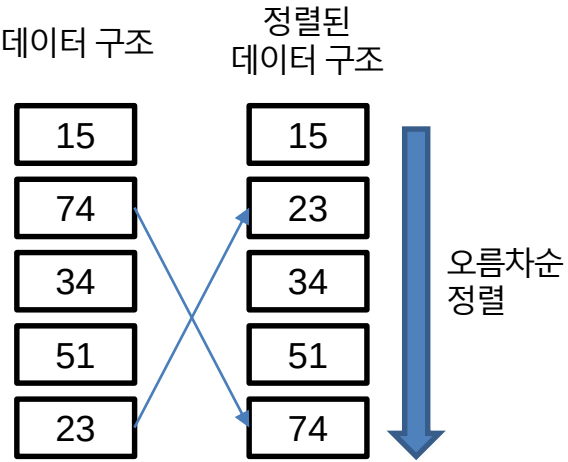
탐색

주어진 데이터 구조 안에 원하는 값이 있는지 찾아내는 문제

정렬

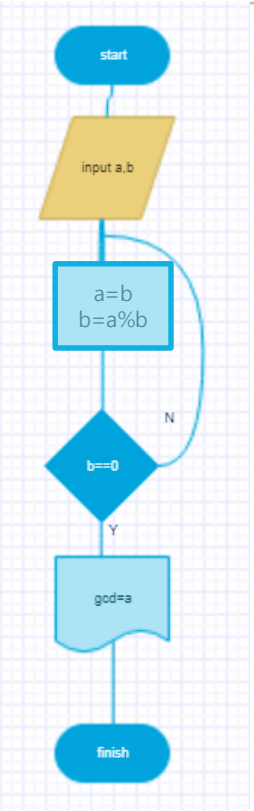
주어진 데이터를 일정한 기준에 따라 정렬하는 문제



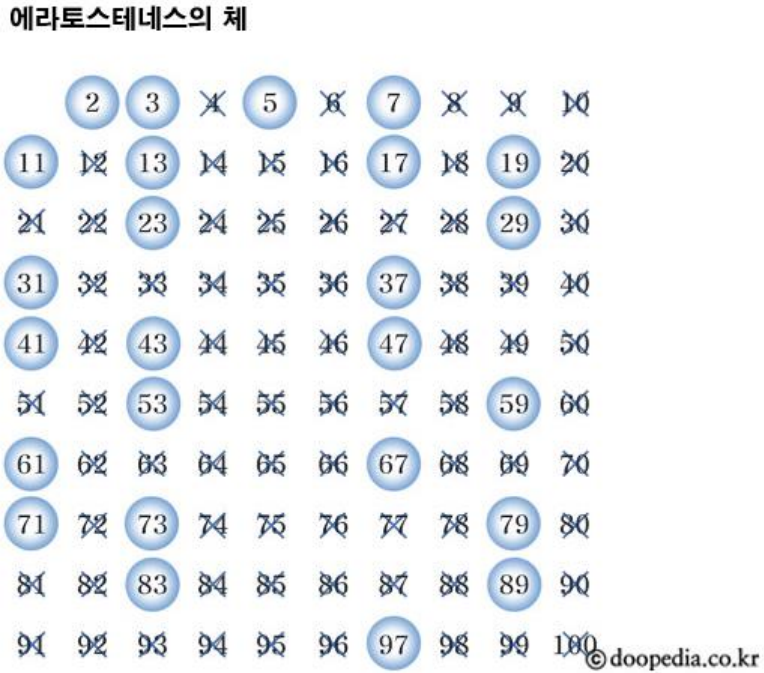


최대공약수 구하기 - 컨셉

- 12 와 30의 최대 공약수는?

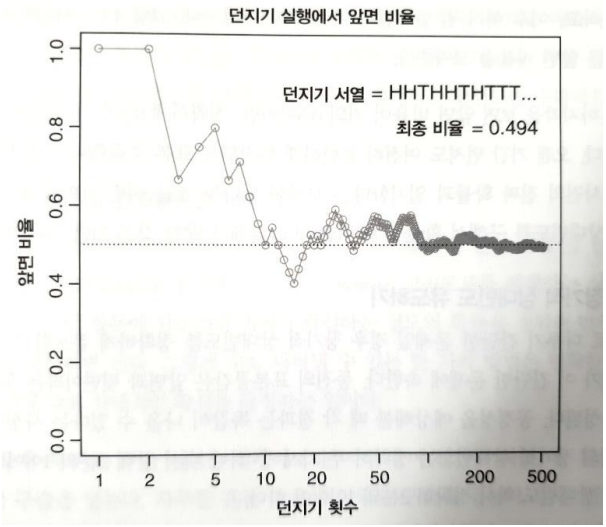


심화: 에라토스테네스의 체



동전 던지기에서 앞면이 나올 확률

- 1회, 2회, 3회, 4회, ...

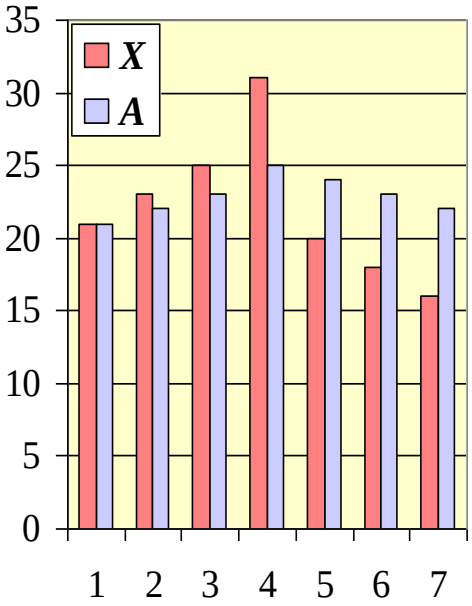


배열의 평균배열 (Prefix Average) 구하기

- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

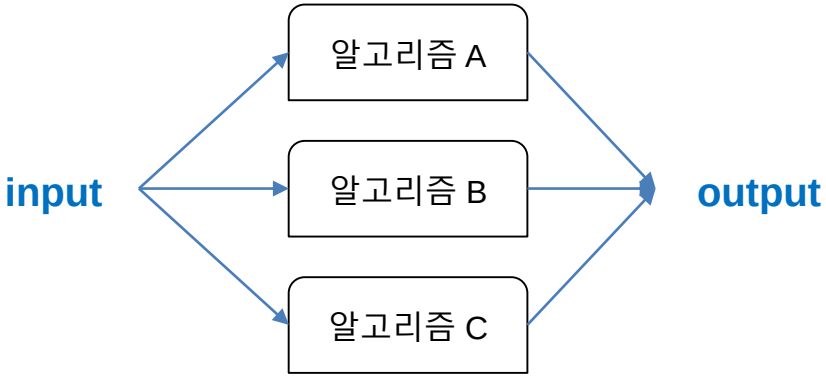
$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

- Computing the array A of prefix averages of another array X has applications to financial analysis



좋은 알고리즘이란?

- **문제해결을 위해 효율적인 알고리즘 설계 필요**
  같은 문제에 대해 서로 다른 알고리즘이 존재 가능



- **좋은 알고리즘이란?**
  다른 알고리즘에 비해 효율적인 알고리즘이란 어때야 할까?
  알고리즘의 성능 분석이 가능할까?

알고리즘의 성능

- 알고리즘의 효율성을 평가하는 두가지 척도 존재 = {시간, 공간}

- **시간(Time)**
  - 알고리즘의 명령문을 실행하는데 시간(time)이 소요됨
  - 알고리즘이 얼마나 빠르게 문제를 해결하는가?
  - 알고리즘의 실행시간에 무엇이 영향을 미치는가?
- **공간(Space)**
  - 데이터구조는 컴퓨터의 메모리공간 (memory space)를 차지
  - 어떤 종류의 데이터 구조가 사용될 수 있는가?
  - 데이터 구조가 실행시간에 어떻게 영향을 끼치는가?

- 컴퓨터 저장 용량의 증가에 따라 공간 보다는 시간의 중요성이 커지고 있음 → 해당 챕터에서는 **{시간}**에 대해 집중

프로그램 구현을 통한 시간측정

- 같은 문제를 해결하기 위한 서로 다른 두 알고리즘을 어떻게 비교할까?

- **가장 간단한 방법 → 프로그램 구현을 통한 시간측정**
  - 두 알고리즘을 프로그래밍 언어(Java, Python)로 구현
  - 두 프로그램의 실행시간을 측정

System.currentTimeMillis()

- **프로그램 구현을 통한 비교 방식의 문제점**
  - 알고리즘 자체가 아닌 여러가지 요소에 의해 영향을 받을 수 있음
    - 프로그래밍 스타일
    - 특정한 데이터 타입
    - 컴퓨터의 성능
    - 백그라운드 프로그램(공정한 시간 비교를 방해)

시간복잡도 분석을 통한 성능평가

- **알고리즘 분석**
  - {컴퓨터, 데이터, 프로그래밍 언어} 등의 외적인 요소를 배제
  - 수학적 기법을 통해 알고리즘의 성능을 객관적으로 평가

- **알고리즘 분석 방법**
  - 알고리즘 수행에 소요되는 연산의 개수를 파악
  - 연산의 개수를 알고리즘 입력크기{n}에 비례하는 시간복잡도 함수{f(n)}로 표현
  - 시간복잡도 함수를 통한 알고리즘 간의 효율성 비교

알고리즘 내부의 기본 연산

- 할당

```
sum = 0;
```

- 사칙연산 (+, -, *, /)

```
sum = sum + i;
```

- 비교

```
if (sum > 10)
```

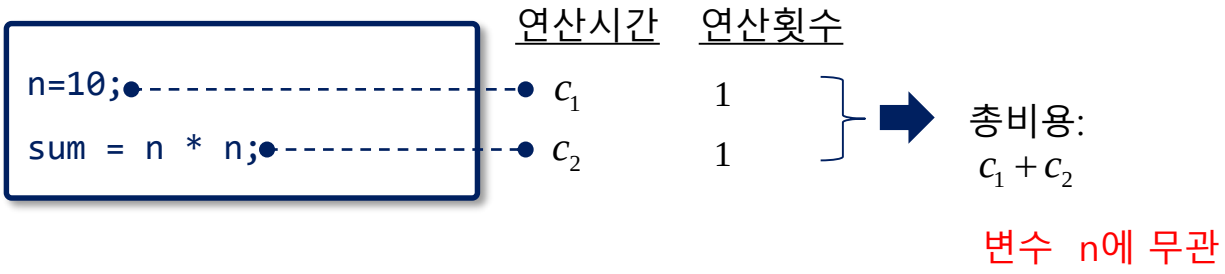
각각의 기본연산을 위해 상수의 연산시간이 소요됨

시간복잡도 계산을 위한 일반적 룰

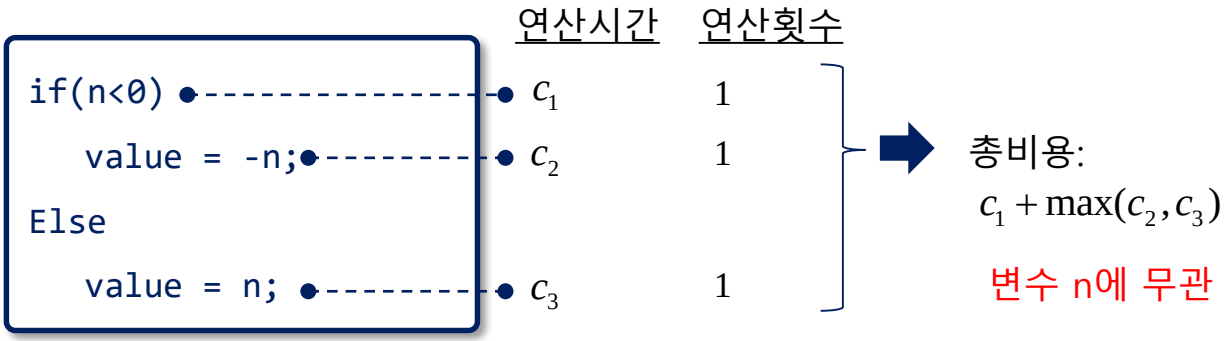
- **일반적 실행문**
  - 상수의 연산횟수를 더한다
- **조건문**
  - 두 조건에서 실행되는 실행문의 최대치보다 클 수 없다.
- **반복문**
  - 반복문 안에서 실행되는 연산의 횟수만큼 더한다
- **중첩된 반복문**
  - 반복이 중첩된만큼 곱하여 연산의 횟수를 계산한다

시간복잡도 계산 예시

- **일반 실행문**

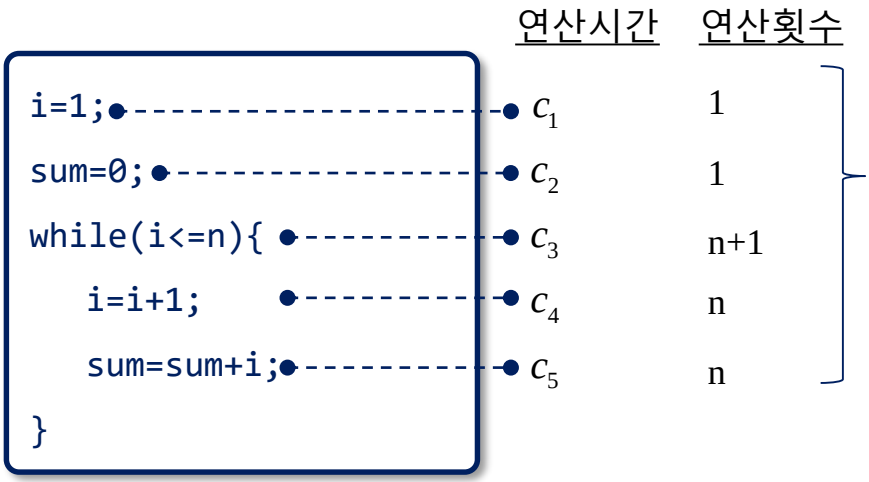


- **조건문이 포함된 경우**



시간복잡도 계산 예시

- **반복문 포함된 경우**



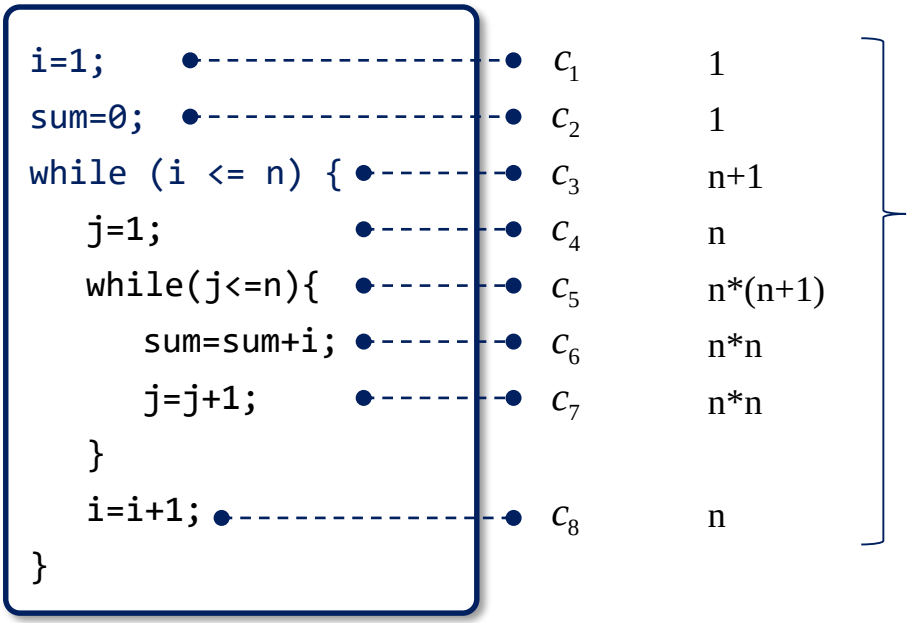
➡ 총비용:

$$c_1 + c_2 + (n+1)\cdot c_3 + n\cdot c_4 + n\cdot c_5$$

변수 n에 비례

시간복잡도 계산 예시

- **중첩된 반복문**



변수 n의 제곱에 비례

총비용: $c_1 + c_2 + (n+1)c_3 + nc_4 + n(n+1)c_5 + n^2c_6 + n^2c_7 + nc_8$

점근적 분석

- **크기가 작은 문제**
  - 알고리즘의 효율성이 중요하지 않다
  - 비효율적인 알고리즘도 무방

- **크기가 충분히 큰 문제**
  - 알고리즘의 효율성이 중요하다
  - 비효율적인 알고리즘은 치명적

- **점근적 분석**
  - 알고리즘의 시간복잡도 입력크기 n에 따라 얼마나 빠르게 증가하는지 파악하여 알고리즘을 비교하는 방식 필요.
  - 입력의 크기가 충분히 큰 경우에 대한 분석을 점근적 분석이라 한다

- 점근석 분석을 위한 시간복잡도의 표현방식 → **빅오표기법**

빅오 표기법

- O(f(n))
  - 기껏해야 f(n)의 비율로 증가하는 모든 함수들의 모임

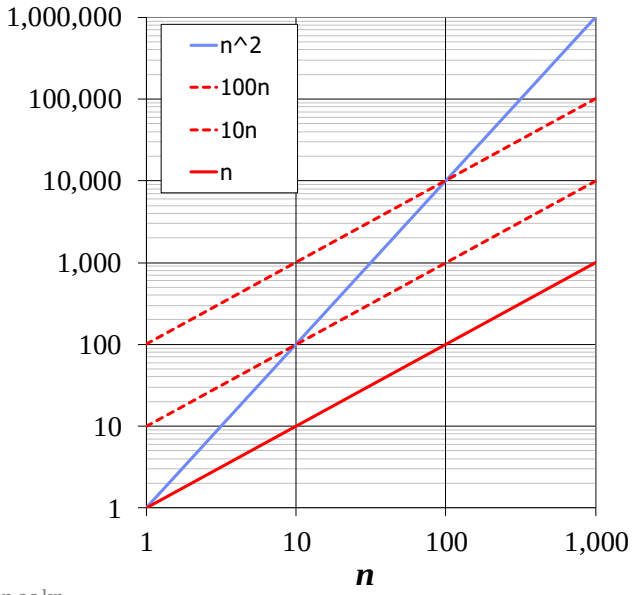
- 수학적 정의
  - 만약 어떤 알고리즘 A의 시간복잡도(f(n))보다 항상 클 수 있도록 만드는 c*g(n) 상수 c와 문제의 입력사이즈 n_0가 존재하면 알고리즘 A는 O(g(n))을 따른다고 말한다.

- 예시
  - 만약 어떤 알고리즘의 시간 복잡도가 $n^2–3*n+10$로 계산되었다고 하자. 이 알고리즘이 $O(n^2)$을 따르는가?

    → 위 시간복잡도 함수에 대해 다음을 만족시키는 c와 n_0 존재.
    $3*n^2 > n^2–3*n+10$ for all $n \geq 2$
    따라서 위 알고리즘은 $O(n^2)$을 따른다.

빅오 예

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant



빅오에 대한 규칙

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

빅오 표기법의 의미

- 입력자료의 크기가 큰 경우, 차수가 큰 항이 전체의 값을 주도
- 시간 복잡도 함수에서 불필요한 정보를 제거하여 알고리즘 분석을 쉽게 할 목적으로 도입
- 다시 말하면 $O(f(n))$은 최고차항의 차수가 $f(n)$과 일치하거나 더 작은 함수의 집합이라고 볼 수 있음.
- 예시
  - $5n^2+4n = O(n^2)$
  - $4n = O(n^2)$
  - $2n\log n = O(n^2)$

빅오표기법으로 표시된 차수

O(1) Time requirement is **constant**, and it is independent of the problem's size.

O($\log_2 n$) Time requirement for a **logarithmic** algorithm increases increases slowly as the problem size increases.

O(n) Time requirement for a **linear** algorithm increases directly with the size of the problem.

O($n*\log_2 n$) Time requirement for a **$n*\log_2 n$** algorithm increases more rapidly than a linear algorithm.

O(n^2) Time requirement for a **quadratic** algorithm increases rapidly with the size of the problem.

O(n^3) Time requirement for a c**ubic** algorithm increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm.

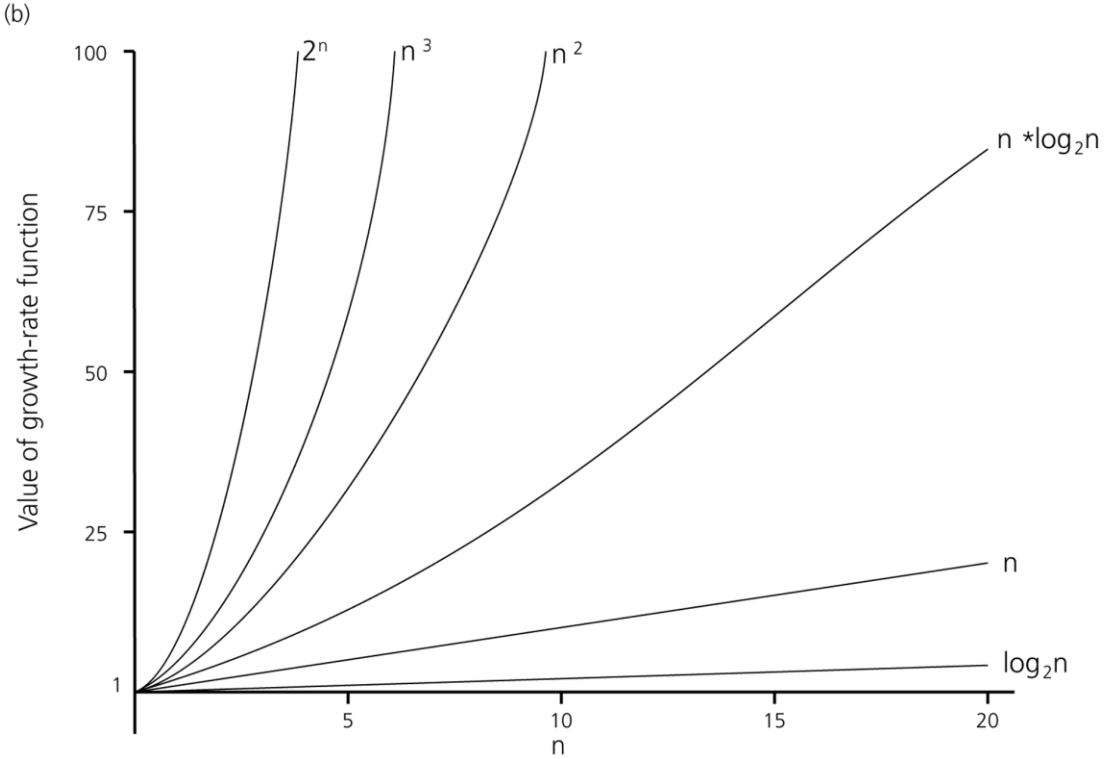
O(2^n) As the size of the problem increases, the time requirement for an **exponential** algorithm increases too rapidly to be practical.

빅오표기법으로 표시된 차수

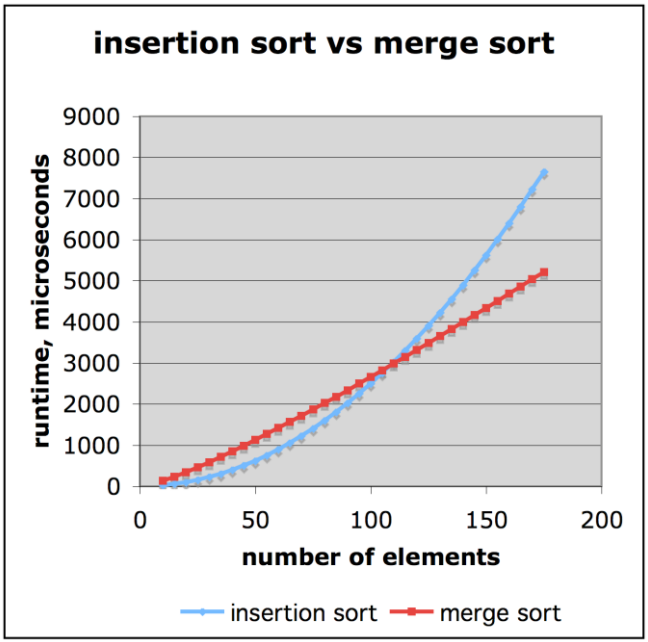
(a)

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

그래프로 표현


(b)
100
75
50
25
1
Value of growth-rate function
2^n
n^3
n^2
$n * \log_2 n$
n
$\log_2 n$
5
10
15
20
n

시간 비교



insertion sort is

$n^2 / 4$

merge sort is

$2 n \lg n$

Sort a million items?

insertion sort takes roughly 70 hours

while

merge sort takes roughly 40 seconds

if runtime is...	time for n + 1	time for 2 n	time for 4 n
c lg n	c lg (n + 1)	c (lg n + 1)	c(lg n + 2)
c n	c (n + 1)	2c n	4c n
c n lg n	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
c n^2	~ c n^2 + 2c n	**4c n^2**	16c n^2
c n^3	~ c n^3 + 3c n^2	8c n^3	64c n^3
c 2^n	c 2^{n+1}	c 2^{2n}	c 2^{4n}

실행시간	최대 문제 크기(n)		
	1초	1분	1시간
$400n$	2,500	150,000	9,000,000
$20n \lceil \log n \rceil$	4,096	166,666	7,826,087
$2n^2$	707	5,477	42,426
n^4	31	88	244
2^n	19	25	31

빅오표기법의 수학적 성질

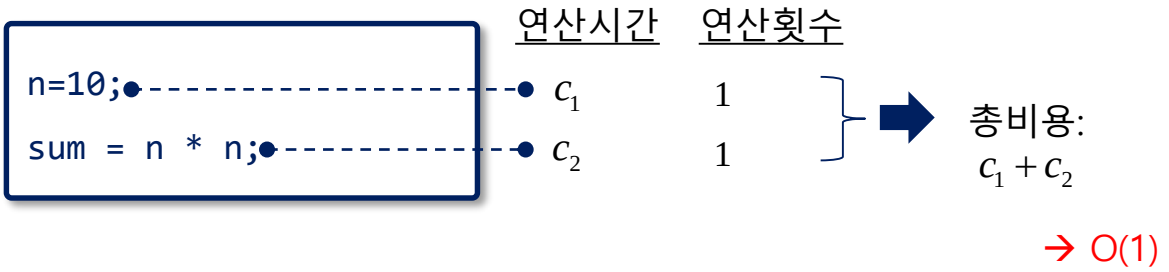
- 언제나 작은 차수는 무시할 수 있다.
  - 만약 알고리즘의 시간복잡도가 $n^3+4n^2+3^n$ 라면 $O(n^3+4n^2+3^n)$ 대신 $O(n^3)$ 라고 표시할 수 있다.
  - 즉 가장 높은 차수만을 사용하여 표시하면 된다.

- 다항식 앞에 곱하는 상수항은 무시할 수 있다.
  - 만약 알고리즘이 $O(5n^3)$이라면 이 알고리즘은 또한 $O(n^3)$이다.

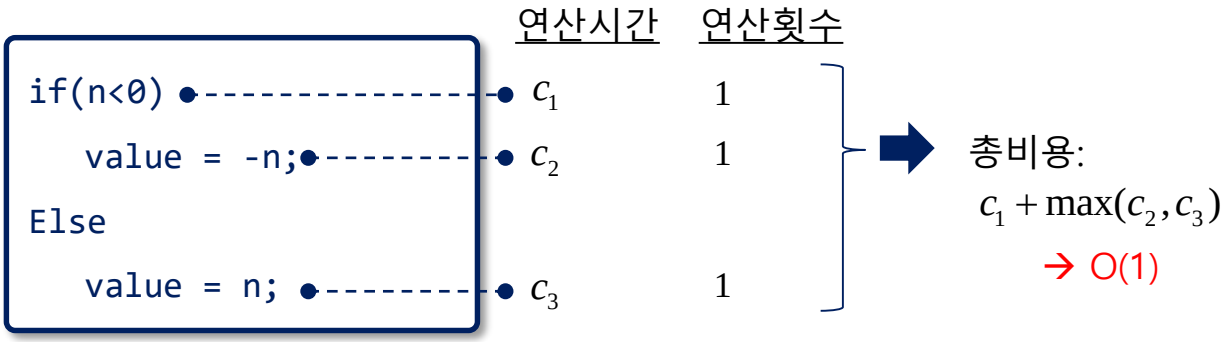
- $O(f(n)) + O(g(n)) = O(f(n)+g(n))$
  - 서로 다른 빅오 표기법으로 표시된 복잡도 함수를 합칠 수 있다.
  - 예시: $O(n^3) + O(4n) = O(n^3+4n) = O(n^3)$

시간복잡도 계산 예시

- **일반 실행문**

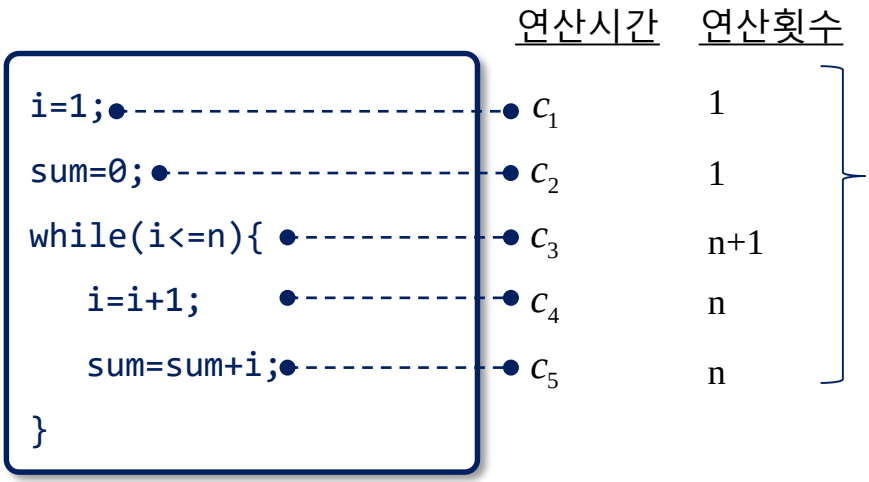


- **조건문이 포함된 경우**



시간복잡도 계산 예시

- **반복문 포함된 경우**



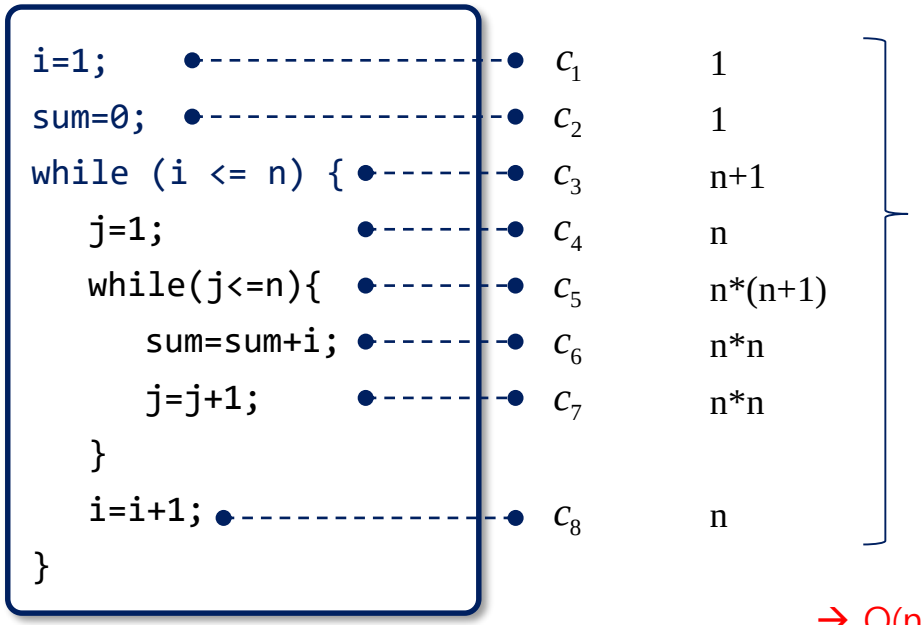
➡ 총비용:

$$c_1 + c_2 + (n+1)\cdot c_3 + n\cdot c_4 + n\cdot c_5$$

→ O(n)

시간복잡도 계산 예시

- **중첩된 반복문**



→ $O(n^2)$

총비용: $c_1 + c_2 + (n+1)c_3 + nc_4 + n(n+1)c_5 + n^2c_6 + n^2c_7 + nc_8$

시간복잡도의 최상, 최악, 평균적 케이스

- 알고리즘은 입력사이즈 (n)의 문제라 할지라도 서로 다른 시간복잡도를 가질 수 있음 → **{Worst, Best, Average Case}**

- **최악의 경우 {Worst case}**
  - 문제를 해결하는데 가장 많은 시간을 소모
  - 시간복잡도의 상한선 (Upper limit)
  - 보통 알고리즘 복잡도 계산에 많이 사용됨

- **최선의 경우 {Best case}**
  - 문제를 해결하는데 최소한의 시간을 소모

- **평균적인 경우 {Average case}**
  - 입력사이즈 n의 문제를 해결하는데 들어가는 평균시간 계산
  - 입력사이즈 n을 기준으로 가능한 데이터의 모든 경우의 수를 따져봐야 함으로 보통 매우 복잡

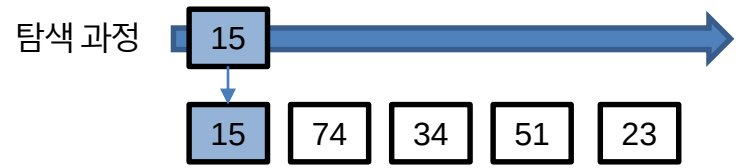
선형탐색의 시간복잡도

- 선형탐색 알고리즘

```
LinearSearch(Array[n],x):
      for i=1 to i=n
       if a[i]==x
           return i;
      return -1
```

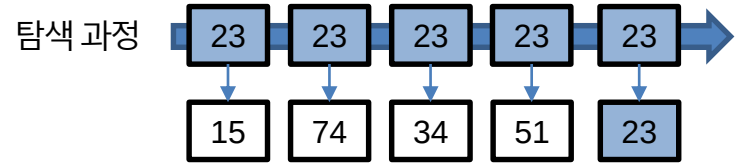
15	74	34	51	23

1) 찾는 값이 15일 때



1번만에 탐색 종료 → 최상의 경우

2) 찾는 값이 23일 때



5번만에 탐색 종료 → 최악의 경우

선형탐색의 복잡도

- 선형탐색 알고리즘

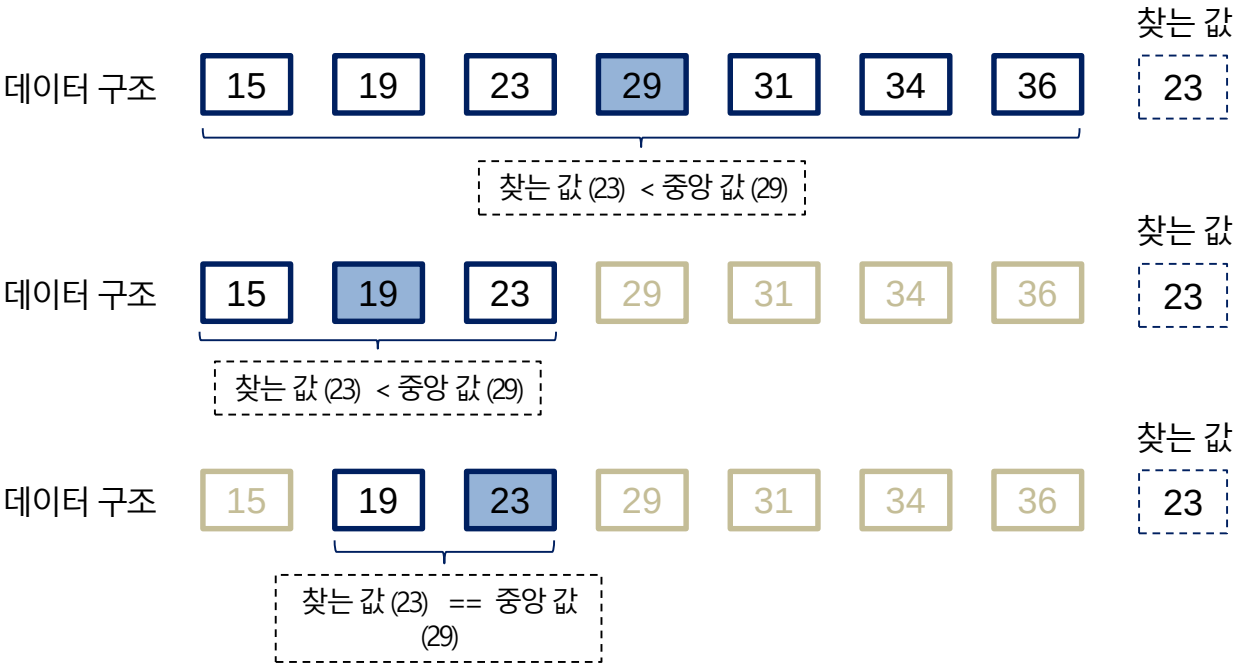
```
LinearSearch(Array[n],x):
      for i=1 to i=n
       if a[i]==x
           return i;
      return -1
```

- **최상의 경우**
  x가 Array의 첫번째에 위치하는 경우
  →O(1)
- **최악의 경우**
  x가 Array의 마지막에 위치하는 경우
  → O(n)
- **평균적 경우**
  x가 Array 내에서 위치할 확률 1/n
  이를 이용해 다음과 같이 계산

$$\frac{\sum_{i=1}^{n} i}{n} = \frac{(n^2+n)/2}{n}$$ → O(n)

이진탐색의 시간복잡도

- 전략: 중앙값과 비교를 통해 탐색범위를 절반씩 추려 나감



이진탐색의 시간복잡도

```
int binarySearch(int a[], int size, int x) {
   int low =0;
   int high = size -1;
   int mid;      // mid will be the index of
              // target when it's found.
   while (low <= high) {
       mid = (low + high)/2;
       if (a[mid] < x)
         low = mid + 1;
       else if (a[mid] > x)
           high  = mid - 1;
     else
           return mid;
   }
   return -1;
}
```

Best Case	→ 1회
Worst Case	→ $\log_2 n + 1$회
$n=2^5=32$개	→ 5+1=6회
$n=2^6=64$개	→ 6+1=7회
$n=2^7=128$개	→ 7+1=8회

버블정렬의 시간복잡도

- 버블 정렬에서 worst-case의 시간 복잡도는 ?
  - Swap의 발생 여부에 따라 조기 종료하는 방식이 아닌 알고리즘의 경우,
  - 단위 연산: $S[i]$와 $S[j]$의 값을 교환하는 회수
  - 최악의 경우: if 조건 문이 항상 true여서 교환이 매번 발생함
    → 입력 배열이 거꾸로 정렬되어 있을 경우

```
public static void bubble_sort(int n, int[] S) {
   for(int i = n - 1; i > 0; i--)
      for(int j = 0; j < i; j++)
         if(S[j] > S[j + 1])
→           exchange S[j] with S[j + 1];
}
```

Worst Case

$$W(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i)$$
$$= (n-1) + (n-2) + \ldots + 1$$
$$= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

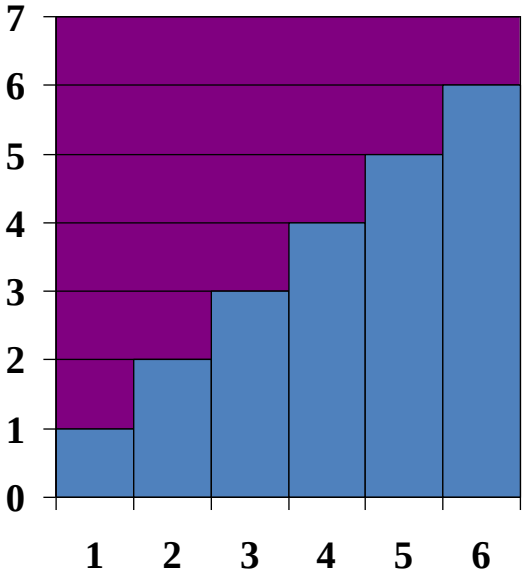
Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

```
Algorithm prefixAverages1(X, n)
    Input array X of n integers
    Output array A of prefix averages of X        #operations
    A ← new array of n integers                   n
    for i ← 0 to n – 1 do                         n
        s ← X[0]                                  n
        for j ← 1 to i do                         1 + 2 + …+ (n – 1)
            s ← s + X[j]                          1 + 2 + …+ (n – 1)
        A[i] ← s / (i + 1)                        n
    return A                                      1
```

알고리즘 평가해보기

- The running time of ***prefixAverages1*** is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm ***prefixAverages1*** runs in $O(n^2)$ time



Prefix Averages (Linear)

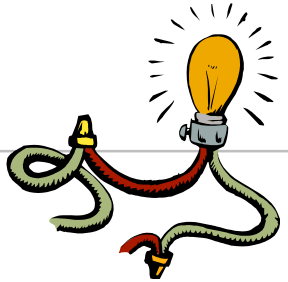
- The following algorithm computes prefix averages in linear time by keeping a running sum

Algorithm *prefixAverages2*(*X*, *n*)
Input array *X* of *n* integers
Output array *A* of prefix averages of *X* #operations
A ← new array of *n* integers *n*
s ← 0 1
for *i* ← 0 **to** *n* − 1 **do** *n*
s ← *s* + *X*[*i*] *n*
A[*i*] ← *s* / (*i* + 1) *n*
return *A* 1

i	0	1	2	3	4	5	6	7	8	9
X[i]	1	2	3	4	5	6	7	8	9	10
s	1	3	6	10	15	21	28	36	45	55
A[i]	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0	5.5

- Algorithm ***prefixAverages2*** runs in ***O*(*n*)** time

빅오의 친척들



Big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

빅오의 친척들

- **$5n^2$ is $\Omega(n^2)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

- **$5n^2$ is $\Omega(n)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$

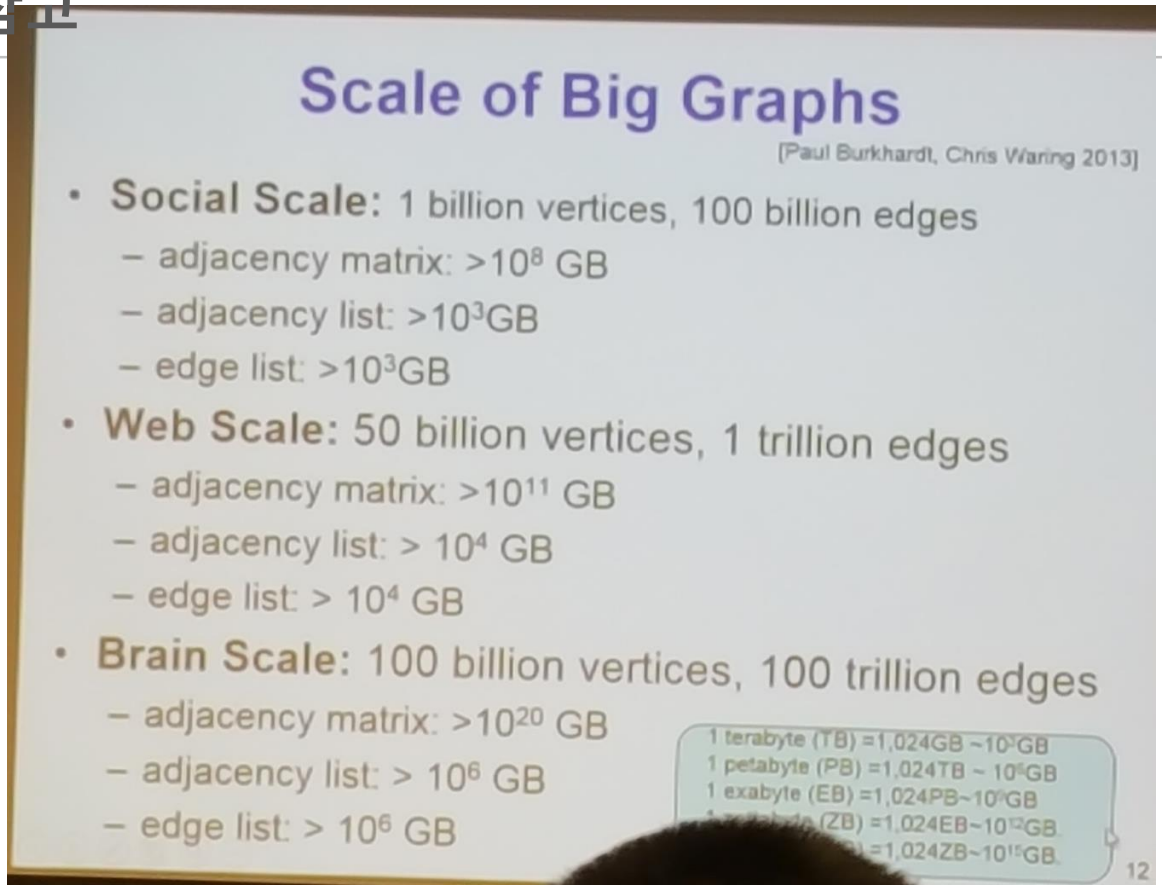
  let $c = 1$ and $n_0 = 1$

- **$5n^2$ is $\Theta(n^2)$**

  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \bullet g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

참고


Scale of Big Graphs
[Paul Burkhardt, Chris Waring 2013]
• Social Scale: 1 billion vertices, 100 billion edges
– adjacency matrix: >10^8 GB
– adjacency list: >10^3GB
– edge list: >10^3GB
• Web Scale: 50 billion vertices, 1 trillion edges
– adjacency matrix: >10^11 GB
– adjacency list: > 10^4 GB
– edge list: > 10^4 GB
• Brain Scale: 100 billion vertices, 100 trillion edges
– adjacency matrix: >10^20 GB
– adjacency list: > 10^6 GB
– edge list: > 10^6 GB
1 terabyte (TB) =1,024GB ~10^3GB
1 petabyte (PB) =1,024TB ~ 10^6GB
1 exabyte (EB) =1,024PB~10^9GB
(ZB) =1,024EB~10^12GB
=1,024ZB~10^15GB
12