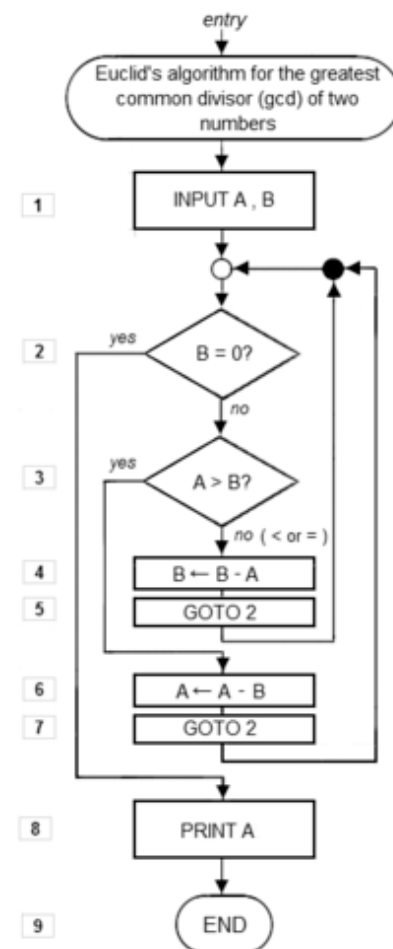


제 2 강. 알고리즘과 알고리즘의 성능

1. 알고리즘의 정의
2. 정렬과 검색 알고리즘
3. 알고리즘의 성능





알고리즘이 뭐지?
a step-by-step procedure
for calculations.

1. 알고리즘의 정의

알고리즘은 어떤 일을 하는 절차를 말한다.

컴퓨터에서는 프로그램이 수행할 작업을 말하며 형식적으로 정의하면 다음과 같다.

(Algorithms are used for calculation, data processing, and automated reasoning)

- 정의(Definition)

명령의 집합

(a finite set of instructions to accomplish a particular task)

- 조건(Criteria) – 알고리즘이 갖출 조건

(1) 입력이 있다 : zero or more inputs

(2) 출력이 있다 : at least one output

(3) 명확해야 한다 : clear and unambiguous instruction

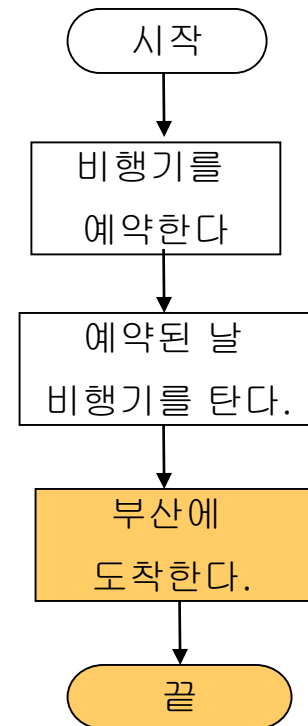
(4) 유한성 : terminates after a finite number of steps

알고리즘의 서술은 3가지 기능 (순서, 반복, 조건)으로 한다.

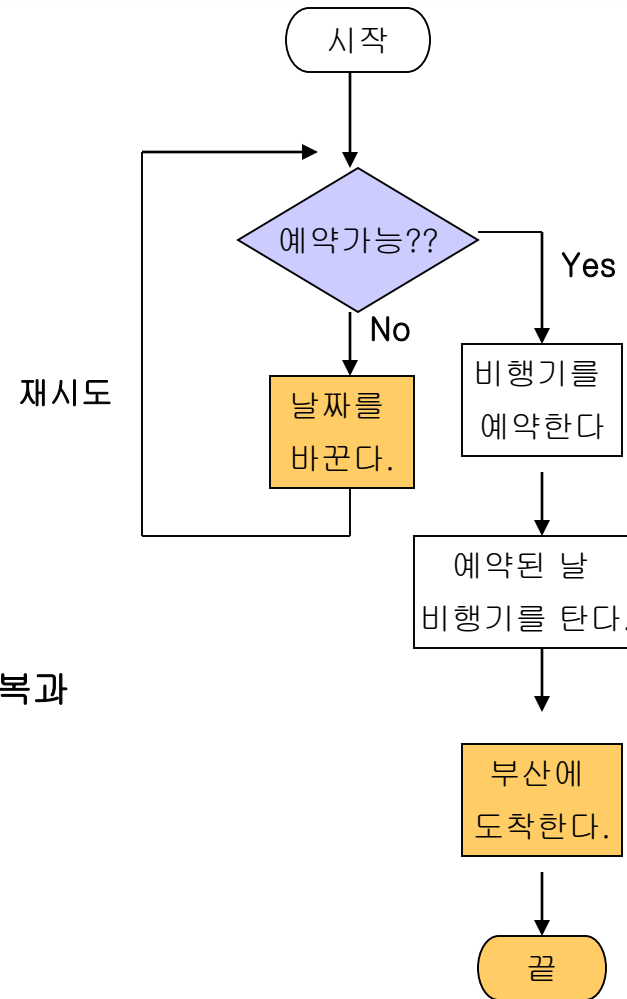
1.1 일상생활의 알고리즘의 예

- 알고리즘은 명령어들이 다음과 같이 3가지로 합성된다
 - 연속(sequence) - 명령어 다음에 명령어가 나온다
 - 반복(repetition) - 명령어가 반복이 된다.
 - 조건(condition) - 조건에 따라 명령의 수행이 결정된다.

예1) 서울에서 부산으로 가는 방법을 컴퓨터 알고리즘으로 기술하면 다음과 같다.



1.1 일상생활의 알고리즘의 예



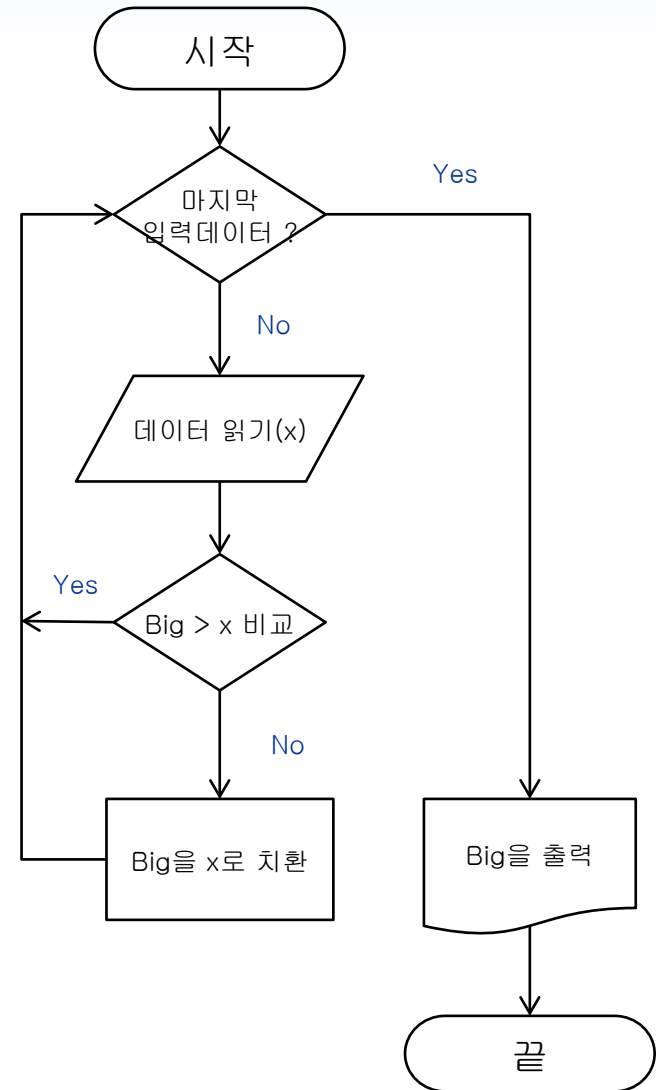
예2) 비행기 표가 없을 경우 반복과 조건이 포함된다.

1.2 컴퓨터의 알고리즘의 예

(컴퓨터 알고리즘의 예)

데이터 개수 n 개에서 큰 수를 찾는 알고리즘을 기술하여보자

데이터를 한 개씩 읽어서 가장 큰 수 라고 기억된 수와 비교하여 큰 수를 찾아 나간다.





2. 정렬과 검색 알고리즘

컴퓨터 분야의 문제 중에서 가장 방법이 많고 많이 쓰이는 알고리즘이

정렬(Sorting)과 검색(Searching)이다.

정렬은 흩어져있는 데이터를 키 값(주민등록번호, 학번 등)을 이용하여 순서대로 열거하는 알고리즘이다.

검색은 데이터(정렬이 안된 데이터 , 정렬이 된 데이터)에서 키 값에 해당되는 데이터를 찾는 알고리즘이다.


앞 장에서 학습한 바와 같이 정렬이 된 데이터에서 검색은 정렬이 안된 데이터에서 검색보다 쉽다. 그러나 항상 데이터를 정렬된 상태로 만들어 놓아야 하는 수고가 필요하다(새로운 데이터 1개 추가, 데이터 1개 삭제 작업 때).

2.1 정렬 알고리즘

정렬 : n 개의 데이터를 정렬한다.

선택정렬 : n 개의 데이터를 놓고 가장 작은 수를 골라 정렬될 장소에 이동한다.
작은 수는 다음 번에 다시 작은 수 찾을 때 방해가 되지않게 처리한다.
($n-1$)개 데이터로 매번 한 개씩 줄어든 데이터로 위의 과정을 반복한다.





/* 방법 1 알고리즘 : A 배열의 작은 원소를 차례로 B 배열에 이동하는 알고리즘 */

```
void selection_sort(int *A, int*B)
{
    int i, j, small, temp; //int A[10], B[10];

    small=0;                /* 가장 작은 값을 B[0]에 이동 */
    for(i=0; i<10; i++)
    { if(A[small] > A[i]) small=i; }
    B[0]=A[small]; A[small]=99;

    small=0;                /* 두번째 작은 값을 B[1]에 이동 */
    for(i=0; i<10; i++)
    { if(A[small] > A[i]) small=i; }
    B[1]=A[small]; A[small]=99;

    small=0;                /* 세번째 작은 값을 B[2]에 이동 */
    for(i=0; i<10; i++)
    { if(A[small] > A[i]) small=i; }
    B[2]=A[small]; A[small]=99;

    ...
    ...
}
```



```
#include <stdio.h>
void selection_sort(int *A, int*B);
int main()
{
    int A[10] = { 10,13,25,15,4,20,5,29,14,21 };
    int B[10] = { 0 };
    int i;
    selection_sort(A, B);
    for (i = 0; i < 10; i++) printf("%d ", B[i]);
    printf("\n");
}
```

```
void selection_sort(int *A, int *B)
{
    int i, small; //int A[10], B[10];

    small = 0; /* 가장 작은 값을 B[0]에 이동 */
    for (i = 0; i < 10; i++)
    {if (A[small] > A[i]) small = i;}
    B[0] = A[small]; A[small] = 99;

    small = 0; /* 두번째 작은 값을 B[1]에 이동 */
    for (i = 0; i < 10; i++)
    {if (A[small] > A[i]) small = i;}
    B[1] = A[small]; A[small] = 99;

    small = 0; /* 세번째 작은 값을 B[2]에 이동 */
    for (i = 0; i < 10; i++)
    {if (A[small] > A[i]) small = i;}
    B[2] = A[small]; A[small] = 99;
```

```
small = 0; /* 네번째 작은 값을 B[2]에 이동 */
for (i = 0; i < 10; i++)
{if (A[small] > A[i]) small = i;}
B[3] = A[small]; A[small] = 99;
```

```
small = 0; /* 다섯번째 작은 값을 B[2]에 이동 */
for (i = 0; i < 10; i++)
{if (A[small] > A[i]) small = i;}
B[4] = A[small]; A[small] = 99;
```

```
small = 0; /* 여섯번째 작은 값을 B[2]에 이동 */
for (i = 0; i < 10; i++)
{if (A[small] > A[i]) small = i;}
B[5] = A[small]; A[small] = 99;
```

```
small = 0; /* 일곱번째 작은 값을 B[2]에 이동 */
for (i = 0; i < 10; i++)
{if (A[small] > A[i]) small = i;}
B[6] = A[small]; A[small] = 99;
```

```
small = 0; /* 여덟번째 작은 값을 B[2]에 이동 */
for (i = 0; i < 10; i++)
{if (A[small] > A[i]) small = i;}
B[7] = A[small]; A[small] = 99;
```

```
small = 0; /* 아홉번째 작은 값을 B[2]에 이동 */
for (i = 0; i < 10; i++)
{if (A[small] > A[i]) small = i;}
B[8] = A[small]; A[small] = 99;
```

```
small = 0; /* 마지막 작은 값을 B[2]에 이동 */
for (i = 0; i < 10; i++)
{if (A[small] > A[i]) small = i;}
B[9] = A[small]; A[small] = 99;
```



- 선택 정렬 알고리즘 (개선) – 앞의 방법을 개선하여 배열 A로만 작업을 한다.

A[10]

10	13	25	15	4	20	5	29	14	21
----	----	----	----	---	----	---	----	----	----

A[10]

4	13	25	15	10	20	5	29	14	21
---	----	----	----	----	----	---	----	----	----

선택 정렬
(가장 작은 수를 선택하여 재 배치)



/* **방법 2 알고리즘** : A 배열의 작은 원소를 차례로 A 배열에 교환하는 알고리즘 */

```
void selection_sort(int *A, int*B)
{
    int i, j, small, temp; //int A[10], B[10];

    for(j=0; j<9; j++)
    {
        small=j; // small=0;
        for(i=(j+1); i<10; i++) //          for(i=1; i<10; i++)
        {
            if(A[small] > A[i]) small=i; }

        temp=A[small]; A[small]=A[j]; A[j]=temp;
    }
}
```

2.1 정렬 알고리즘

(선택정렬 알고리즘 작성)

선택 정렬 : n 개의 데이터를 정렬한다.

알고리즘으로 기술된 선택 정렬 방법

정렬되지 않은 데이터에서 가장 작은 수를 골라 정렬된 리스트의 (i 번째, $i=0,1,2,\dots$)에 붙인다.

```
for (i=0; i<n-1; i++) {
```

list[i] 부터 list[n-1] 까지 비교하여
가장 작은 수가 list[min]이라 하자;

list[i] 와 list[min]을 교환한다;

```
}
```

2.1 정렬 알고리즘

(선택정렬 알고리즘)

```
for (i=0; i<n-1; i++) {  
    list[i] 부터 list[n-1] 까지 비교하여  
    가장 작은 수가 list[min]이라 하자;  
    list[i] 와 list[min]을 교환한다;  
}
```



```
for (i=0; i<n-1; i++) /* n-1번 반복 */  
{  
    min = i;  
    for(j=i+1; j < n; j++) /* 작은 수 선택 */  
    { if(list[j] < list[min]) min = j; }  
    temp=list[min]; list[min]=list[i]; list[i]=temp;  
    /* list[min]과 list[i]를 교환 */  
}
```





2.2 검색 알고리즘

검색(Search) : n 개의 데이터를 검색한다.

가장 간단한 검색 알고리즘은 **선형 검색(Sequential Search)** 알고리즘이다.

n 개의 데이터를 놓고 한 개씩 검색하는 방법이다

(선형검색 프로그램을 작성해보자...)

데이터가 **정렬**이 된 상태에서는 **이진 검색(Binary Search)** 방법이 더 효율적이다.

이진 검색은 전체 데이터의 중간에 있는 데이터를 비교하면서

찾는 데이터가 앞/뒤 중 어디 있는지 알 수 있다

(찾아야 할 데이터 개수가 $\frac{1}{2}$ 로 줄어든다.)

2.2 검색 알고리즘

(이진검색 알고리즘)

* **65 찾기**

※ 배열이 정렬된 상태이므로
반씩 잘라가면서 큰 쪽을 택한다.

9	15	16	19	21	39	51	65	76	85	99
---	----	----	----	----	----	----	----	----	----	----

39와 65를 비교한다. 65가 39의 오른쪽에 있는 것을 알았으므로 오른쪽 ½에 대하여 검색한다.


비교

9	15	16	19	21	39	51	65	76	85	99
---	----	----	----	----	----	----	----	----	----	----

76와 65를 비교한다. 65가 76의 왼쪽에 있는 것을 알았으므로 왼쪽 ½에 대하여 검색한다.

비교

9	15	16	19	21	39	51	65	76	85	99
---	----	----	----	----	----	----	----	----	----	----



2.2 검색 알고리즘

(이진 검색 알고리즘의 작성)

- **left와 right의 표시**
 - left와 right는 탐색하고자 하는 배열의 왼쪽, 오른쪽 끝 지점을 가리킨다
 - 초기값으로 left=0, right=n-1로 설정
- **list의 중간 위치**
middle = (left+right)/2 설정
- **list[middle]과 searchnum을 비교한다**
 - 1) searchnum < list[middle]
right를 middle로 설정(right=middle-1)
 - 2) searchnum = list[middle]
middle을 반환(return middle)
 - 3) searchnum > list[middle]
left를 middle+1로 설정(left=middle+1)
- 위 과정을 middle값을 다시 계산하여 탐색을 계속한다

2.2 검색 알고리즘

(이진 검색 알고리즘의 작성)

```
/* 반복적인 알고리즘이다, 순환적으로도 작성할 수 있다. */

int binsearch(int list[],int searchnum, int left,int right)
{
    /*searchnum 에 대해 list [0]<=list[1]<= ...<=list[n-1]을 탐색.
    찾으면 그 위치를 반환하고 못 찾으면 -1을 반환한다.*/
    int middle;
    while(left <= right) {
        middle = (left + right) / 2;
        if (list[middle] < searchnum) left = middle+1;
        else if (list[middle] > searchnum) right = middle-1;
        else if (list[middle] == searchnum) return middle;
    }
    return -1; /* 찾지 못함 */
}
```



Question 선택정렬 (Question2.1)

1. 선택 정렬

① 다음 선택 정렬 알고리즘에서 8번 문장의 수행 횟수는?

(1) 55 (2) 45 (3) 9 (4) 10


② 다음 선택 정렬 알고리즘에서 7번 문장의 수행 횟수는?

(1) 55 (2) 45 (3) 9 (4) 10

데이터 : `int data[]={20, 1, 50, 55, 34, 21, 4, 66, 71, 8};`

```
1. void selection_sort(int data[])
2. { int i, j, t, min;
3.   for(i=0; i<9; i++)
4.   {
5.       min = i;
6.       for(j=i+1; j<10; j++)
7.           { if (d[min]>d[j]) min = j;}      /* 7번 */
8.       t=d[i]; d[i]=d[min]; d[min]=t;      /* 8번 */
9.   }
10. }
```

(참고사이트) <http://www.pythontutor.com/c.html#mode=edit>



Question 이진검색 (Question2.2)

2. 이진 검색알고리즘 `binsearch(list, 15, 0, 10)` 호출시 3번 문장의 수행 횟수는?

- ① 1 ② 2 ③ 3 ④ 4

데이터 : `int list[]={9, 15, 16, 19, 21, 39, 51, 55, 76, 85, 99};`

`int binsearch(int list[], int searchnum, int left, int right)`

`{ /*searchnum 에 대해 list [0]<=list[1]<= ...<=list[n-1]을 탐색. 찾으면 그 위치를 반환하고 못 찾으면 -1을 반환한다.*/`

```
1.   int middle;
2.   while(left <= right) {
3.       middle = (left + right) / 2;
4.       if (list[middle] < searchnum) left = middle+1;
5.       else if (list[middle] > searchnum) right = middle-1;
6.       else if (list[middle] == searchnum) return middle;
7.   }
8.   return -1; /* 찾지 못함 */
}
```



3. 알고리즘의 성능

- 성능(performance)
 - 성능 분석
 - : 컴퓨터에 상관없이 시간과 공간의 추산에 초점을 둔다
 - : 복잡도 이론 (Complexity Theory)
 - 성능 측정
 - : 컴퓨터에 의존적인 시간을 알아 내는 것이다.
 - 컴퓨터에서 직접 실행하여 정확한 시간을 측정한다.
- 공간 복잡도(Space Complexity)
 - : 프로그램을 실행시켜 완료하는데 필요로 하는 기억 장소의 크기
- 시간 복잡도(Time Complexity)
 - : 프로그램을 실행시켜 완료하는 데 필요한 컴퓨터 시간의 양을 의미한다



3.1 공간 복잡도(Space Complexity)

- 고정된 공간 영역

프로그램이 실행되는 데이터의 크기와 관계없이 일정한 공간을 필요로 한다.

예) 프로그램 명령어가 차지하는 공간

- 변하는 공간 영역

필요 공간이 데이터의 개수에 따라 변한다. 데이터의 개수를

n 이라고 하면 프로그램에 따라 $2 \times n$, $3 \times n$, $n \times n$ 의 영역이 필요하다.

▣ 컴퓨터 기억 장소에 제한이 있기 때문에 경우

공간 복잡도는 중요한 요소가 된다. (기억장치가 발달하여 의미가 줄어듦 !)



- 필요한 공간의 예 - n개의 데이터를 더하는 프로그램 예


`list[], n, tempsum, i => n+3 개의 데이터 공간`

```
float sum(float list[], int n)
{
    float tempsum = 0;
    int i;
    for(i = 0; i < n; i++)
        tempsum += list[i];
    return tempsum;
}
```

- 필요한 공간

프로그램 사이즈 = 고정

데이터 사이즈 = $n + 3$



3.2 시간 복잡도(Time complexity)

프로그램 P 에 의하여 실행되는 시간 t 는 기계에 따라 달라진다.

펜티엄 1GHz 기계와 펜티엄 2GHz 기계에서 같은 프로그램을 실행시킨다면?

절대적인 시간 t 보다는 연산의 개수 $T(P)$ 를 사용하자!

$T(P)$ = count the number of operations the program performs

3.2 시간 복잡도(Time complexity)

• 프로그램 수행 시간 분석 방법

다음 프로그램은 반복하여 수행되는 프로그램이다

수행 시간이 컴퓨터마다 다르므로 수행 시간에 대한 추정을 해보자

```
1 : float sum(int n) {  
2 :     int count=0;  
3 :     int i;  
4 :     for(i = 0; i < n; i++)  
5 :         count+=2;  
6 :     count += 3;  
7 :     return count;  
8 : }
```

실행 순서 :
(n이 2일 때)
2-4-5-4-5-4-6-7
(n이 3일 때)
2-4-5-4-5-4-5-4-6-7


n=0 일 경우 :

for loop은 실행이 되지 않으므로 전체 실행되는
문장은 4 = 2번문, 4번문, 6번문, 7번문

n=0 이 아닐 경우 :

전체 실행되는 문장은

$2 \times n + 4 = 2\text{번문} + 4\text{번문}, 5\text{번문을 } n\text{번 수행}$
 $+ 4\text{번문}, 6\text{번문}, 7\text{번문}$



3.2 시간 복잡도(Time complexity)

다음 프로그램은 행렬에 있는 데이터 값을 합산하는 프로그램이다.
수행 시간이 컴퓨터마다 다르므로 수행 시간에 대한 추정을 해보자.

```
void add(int a[][M_SIZE],int b[][M_SIZE],
         int c[][M_SIZE],int rows,int cols)
{
    int i, j;
    for(i = 0; i < rows; i++)
        for(j = 0; j < cols; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```

전체 수행되는 횟수 = $rows + 1 +$
 $(rows) * (cols + cols + 1))$
 $= 2 \cdot (rows) \cdot (cols) + 2 \cdot (rows) + 1$

- rows와 cols가 n으로 같다면
 $= 2 \cdot n^2 + 2 \cdot n + 1$

3.2 시간 복잡도(Time complexity)

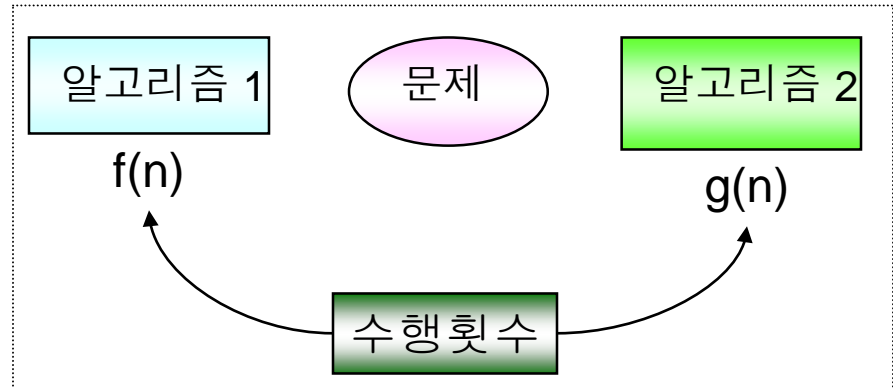
- 행렬의 덧셈

행렬의 덧셈을 테이블로 계산 횟수 합산

문 장	s/e	횟 수	전 체 횟수
void add(int a[], [Max_SIZE]...)			
{	0	0	0
int i,j;	0	0	0
for(i=0;i<rows;i++)	1	rows+1	rows+1
for(j=0;j<cols;j++)	1	rows x (cols+1)	rows x cols+rows
c[i][j] = a[i][j] + b[i][j];	1	rows x cols	rows x cols
}	0	0	0
전 체		2rows x cols + 2rows + 1	

3.3 알고리즘 분석의 표현법

- 알고리즘의 수행시간을 비교할 때 입력 데이터 개수 n 에 대한 함수로 결정된다.
- 같은 작업을 하는 알고리즘이 어느 것이 빠른가?



결정하는 요소 : 시간 복잡도(time complexity)

1) 입력의 크기(input size)

- 입력 데이터 n 의 크기에 따라 결정된다.

$T(n) = ?$

2) 입력 데이터의 형태

- 가능한 입력의 형태에 따라

평균 시간 : $A(n) = ?$

최악의 경우 : $W(n) = ?$

- 보통은 “최악의 경우”에 관심을 갖는다.
- 최악의 경우는 “상한시간”이 관심 대상이다.

3.3 알고리즘 분석의 표현법

(O-표기법)

- 알고리즘의 수행시간을 기본이 되는 명령어가 입력데이터 n 에 대하여 수행되는 횟수로 나타내면 가장 알기 쉽다
즉, 어떤 알고리즘이 비교할 때 수행시간을 직접 재는 것보다 입력 데이터 개수 n 에 대한 함수로 결정한다
- **Big “O”**

정의) 알고리즘 A의 필요한 연산함수 $f(n)$ 의 수행 복잡도는 다음을 만족하면 $O(g(n))$ 이라고 정의한다.

=> 양의 상수 c 와 n_0 이 존재하여, 적당한 값 n_0 보다 큰 n 값에 대하여 $f(n) \leq c \cdot g(n)$ 을 만족한다. 즉, 큰 n 값들에 대하여 $g(n)$ 함수 값은 항상 $f(n)$ 함수 값 보다 크거나 같은 상한 값 함수이다. $f(n)$ 함수는 $g(n)$ 보다 더 작게 증가한다는 의미이다..

예) $f(n) = 25 \cdot n$, $g(n) = n^2$

if let $c = 1$,

$25 \cdot n \leq n^2$ for all $n \geq 25$ ($n_0 = 25$)

$\therefore f(n) = 25 \cdot n \leq n^2 = O(n^2)$

-> 적당한 c 값과 적당한 n_0 보다 큰 모든 n 값에 대하여 항상 $f(n) \leq c \cdot g(n)$ 성립시킬 수 있으므로 $f(n)$ 의 수행 복잡도는 $O(g(n))$ 이다.

$f(n)=O(g(n))$ 라고 쓴다

n	$f(n) = 25 \cdot n$	$g(n) = n^2$
1	25	1
2	50	4
...
25	625	625
26	650	676
...

3.3 알고리즘 분석의 표현법


(O-표기법)

- $f(n) = O(g(n))$ 이라고 표기하면
모든 n , $n \geq n_0$ 에 대하여 $g(n)$ 은 상한함수(upper bound)이다.
그러나 어느 정도 상한인지는 알수없다.
 $n = O(n^2)$, $n = O(n^{2.5})$
 $n = O(n^3)$, $n = O(2^n)$
보통은 $g(n)$ 은 $f(n)$ 보다 크면서 가장 차이가 적은 함수를 사용한다.
- $f(n) = O(g(n)) \Leftrightarrow O(g(n)) = f(n)$

정리) if $f(n) = a_m n^m + \dots + a_1 n + a_0$, then $f(n) = O(n^m)$

증명)

$$\begin{aligned} f(n) &\leq |a_k| \cdot n^k + |a_{k-1}| \cdot n^{k-1} + \dots + |a_1| \cdot n + |a_0| \\ &= \{|a_k| + |a_{k-1}|/n + \dots + |a_1|/n^{k-1} + |a_0|/n^k\} \cdot n^k \\ &\leq \{|a_k| + |a_{k-1}| + \dots + |a_1| + |a_0|\} \cdot n^k \\ &= c \cdot n^k \quad (c = |a_k| + |a_{k-1}| + \dots + |a_1| + |a_0|) \\ &= O(n^k) \end{aligned}$$



3.3 알고리즘 분석의 표현법

(O-표기법)

[연습] O-표기법에 대한 연습을 해보자

예 1) 어떤 알고리즘의 수행 시간 $f(n) = 3n + 4$ 이면

$3n + 4 \leq 4n$ 이므로 $O(n)$ 이라고 표기할 수 있다.

예 2) $f(n) = 2n^2 + 5n + 3$ 은 $O(n^2)$ 이다.


예 3) $f(n) = 0.01n^3 + 20n^2 + 2n + 3$ 은 $O(n^3)$ 이다.

예 4) $f(n) = 3n + 4$ 는 $O(n)$ 이고, $O(n^2)$ 이며, $O(n^3)$ 라고 표기할 수 있지만

일반적으로는 $O(n)$ 으로 표기한다.

예 5) $f(n) = 2n^2 + 5n + 3$ 은 $O(n^2)$ 이지만 $O(n)$ 라고 쓰면 틀린다. (X)

예 6) $f(n) = 0.001 \cdot n \log n + 5n + 3$ 은 $O(n \log n)$ 이다.



3.3 알고리즘 분석의 표현법

(O-표기법)

[참고] 상한함수 외에 하한함수와 동등함수에 대한 표기가 있다.

Omega : 어떤 알고리즘의 수행시간에 대한 **하한(lower bound)** 함수이다.

정의) $f(n) = \Omega(g(n))$

양의 상수 c 와 n_0 이 존재하여, 적당한 값 n_0 보다 큰 n 값에 대하여 $f(n) \geq c \cdot g(n)$ 이다. 즉, 큰 n 값들에 대하여, $g(n)$ 함수 값은 항상 $f(n)$ 함수 값 보다 같거나 작은 하한 값 함수이다. $f(n)$ 함수는 $g(n)$ 보다는 더 빨리 증가한다.

Theta : 어떤 알고리즘의 수행시간에 대한 **상한과 하한이 같을 때** 표시하는 함수이다

정의) $f(n) = \Theta(g(n))$

양의 상수 c_1, c_2 와 n_0 이 존재하여, 적당한 값 n_0 보다 큰 n 값에 대하여 $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ 이다. 즉, 큰 n 값들에 대하여 $g(n)$ 함수 값은 항상 $f(n)$ 함수 값과 비슷하게 증가한다는 의미이다. O 표기법 보다는 더 증가 범위가 더 정확하다. 상한과 하한이 같기 때문에 알고리즘 복잡도가 정확히 어떤 함수인지 알 수 있다.

- more precise than both the “big oh” and omega notations
- $g(n)$ is both an upper and lower bound on $f(n)$

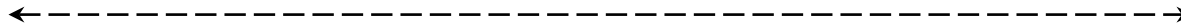
3.3 알고리즘 분석의 표현법

(알고리즘 $O(f(n))$ 함수와 함수의 값)

– class of time complexities

- $O(1)$: constant
- $O(\log_2 n)$: logarithmic
- $O(n)$: linear
- $O(n \cdot \log_2 n)$: log-linear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic

다항식 시간 복잡도
(polynomial time)



- $O(2^n)$: exponential
- $O(n!)$: factorial

지수 시간복잡도
(exponential time)




3.3 알고리즘 분석의 표현법

(알고리즘 $O(f(n))$ 함수와 함수의 값)

컴퓨터로 해결이 가능한 함수들

		n의 값					
함수	함수이름	1	2	4	8	16	32
1	constant	1	1	1	1	1	1
$\log n$	logarithmic	0	1	2	3	4	5
n	linear	1	2	4	8	16	32
$n \log n$	log linear	0	2	8	24	64	160
n^2	quadratic	1	4	16	64	256	1024
n^3	cubic	1	8	64	512	4096	32768
2^n	exponential	2	4	16	256	65536	4294967296
$n!$	factorial	1	2	24	40326	20922789888000	$26313X 10^{33}$

컴퓨터로 해결이 불가능한 함수들 – 매우 빠르게 증가한다.




Question O-표기법 (Question2.3)

1. (O-표기법 구하기) 다음 함수를 O-표기법으로 표현하여라.

- 1) $5n^4 + 6n + 3$
- 2) $n \log n + 2^n + 7$
- 3) 10
- 4) $n \log n + 1000n$

2. (O-표기법 함수간의 크기) - 다음 함수를 O-표기법으로 표시했을 때 Order가 작은 것부터 오름차순으로 순서를 정하라.

n $n \log n$ $n^2 \log n$ $n!$ 2^n n^2 $n^{1/2}$ $\log n$



Question 알고리즘 복잡도 (Question2.4)

❖ (알고리즘 복잡도 연습)

① 다음 알고리즘의 복잡도는? 가장 많이 수행되는 (*) 문장의 실행횟수를 n 에 관한 함수로 구해 본다.

```
/* algorithm 1*/
#include <stdio.h>
int main()
{
    int i,j,m=0;
    int n=10;
    for(i=0; i<n; i++)
    {
        m=m+1; (*)
    }
    printf(" result = %d ", m);
}
```



Question (계속 ...)

② 다음 알고리즘의 복잡도는? 가장 많이 수행되는 (*) 문장의 실행횟수를 n 에 관한 함수로 구해 본다.

```
/* algorithm 2*/
#include <stdio.h>
int main()
{
    int i,j,m=0;
    int n=10;
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
        {
            m=m+1; } (*)
    }
    printf(" result = %d ", m);
}
```



Question (계속 ...)

③ 다음 알고리즘의 복잡도는? 가장 많이 수행되는 (*) 문장의 실행횟수를 n 에 관한 함수로 구해 본다.

```
/* algorithm 3*/
#include <stdio.h>
int main()
{
    int i,j,m=0;
    int n=10;
    for(i=0; i<n; i++)
    {
        for(j=0; j<i; j++)
        {
            m=m+1; } (*)
    }
    printf(" result = %d ", m);
}
```



Question (계속 ...)

④ 다음 알고리즘의 복잡도는? 가장 많이 수행되는 (*) 문장의 실행횟수를 n에 관한 함수로 구해 본다.

/* algorithm 4 */

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int i,j,k, m=0; int n=10;
```

```
    for(i=0; i<n; i++)
```

```
    {
```

```
        for(j=0; j<n; j++)
```

```
        {
```

```
            for(k=0; k<j; k++)
```

```
                m=m+1; (*)
```

```
        }
```

```
    }
```

```
    printf(" result = %d ", m);
```

```
}
```



Question (계속 ...)

⑤ 다음 알고리즘의 복잡도는? 가장 많이 수행되는 (*) 문장의 실행횟수를 n에 관한 함수로 구해 본다.

```
/* algorithm 5*/  
#include <stdio.h>  
int main()  
{  
    int i, m=0;  
    int n=10;  
    for(i=1; i<=n; i=i*2)  
    {  
        m=m+1; (*)  
    }  
    printf(" result = %d ", m);  
}
```



Review

- 알고리즘은 컴퓨터가 일을 하는 과정을 기술하는 방법이다.

일상생활의 알고리즘 : 사람의 일 처리

컴퓨터의 알고리즘 : 컴퓨터의 장점인 연산, 비교, 판단에 관한 알고리즘

- 정렬과 검색 알고리즘

알고리즘에서 가장 기본이 되는 알고리즘은 정렬과 검색 알고리즘이다.

- 알고리즘의 성능 표기법

- 알고리즘이 수행되는 시간을 입력 데이터의 개수 n 에 대한 함수로 표현할 수 있다.
 - O-표기법은 알고리즘이 수행되는 시간의 상한 함수이다.
 - 문제에 대하여 $O(f(n))$, $O(g(n))$ 알고리즘이 있다면 $f(n) > g(n)$ 인 경우 $O(g(n))$ 알고리즘이 효율적이다.
 - $O(f(n))$ 의 $f(n)$ 이 지수 함수이거나 factorial 함수이면 문제는 컴퓨터로 해결이 어렵다.
-