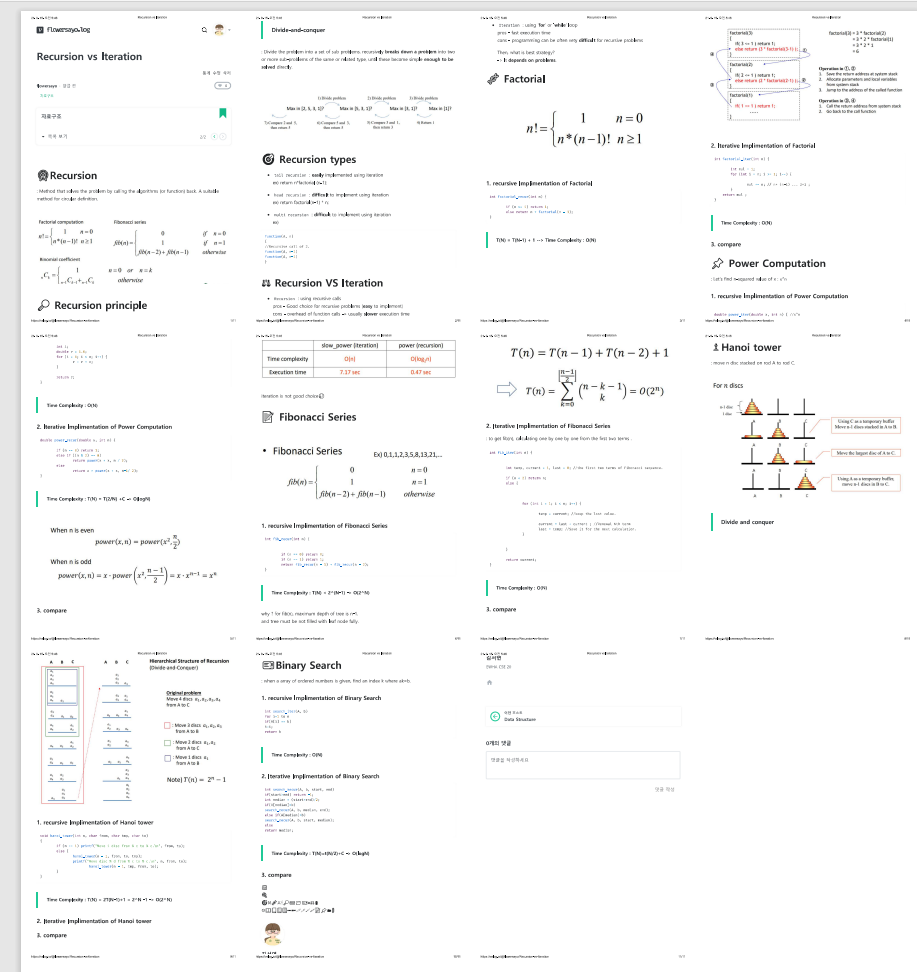
강의 들은 것 블로그에 정리한 내용입니다 : )

혹시 다음부터 아래 스크린샷 + 링크로 요약과제를 제출하는 것이 가능할까요?

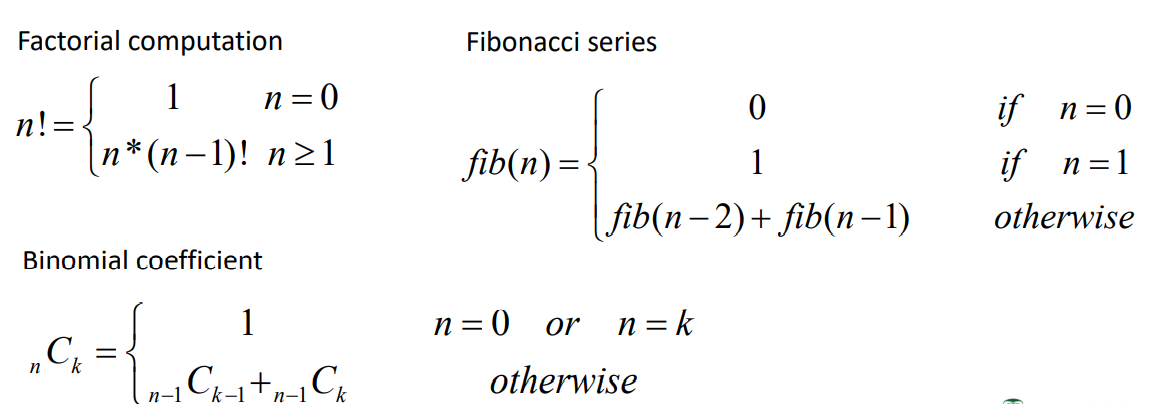
[Recursion vs Iteration (velog.io)](https://velog.io/@flowersayo/Recursion-vs-Iteration)

[Array Pointer (velog.io)](https://velog.io/@flowersayo/Array-Pointer)

텍스트, 실내, 스크린샷, 여러개이(가) 표시된 사진

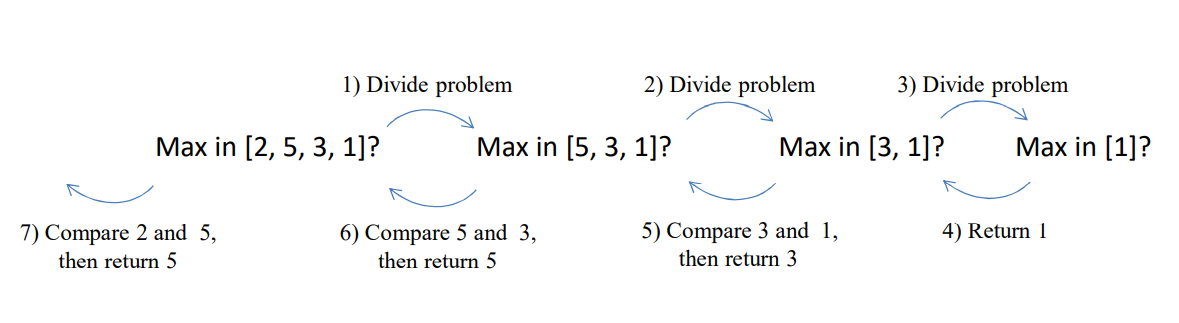
자동 생성된 설명

# 🎡Recursion

: Method that solves the problem by calling the algorithms (or function) back. A suitable method for circular definition.  


# 🔎 Recursion principle

### Divide-and-conquer

: Divide the problem into a set of sub problems. recursively **breaks down a problem** into two or more sub-problems of the same or related type, until these become simple **enough to be solved** directly.  


# 🎯 Recursion types

* tail recursion : **easily** implemented using iteration  
  ex) return n\*factorial (n-1);
* head recursion : **difficult** to implement using iteration  
  ex) return factorial(n-1) \* n;
* multi recursion : **difficult** to implement using iteration  
  ex)

function(A, n)

{

//Recursive call of 2.

function(A, n-1)

function(A, n-1)

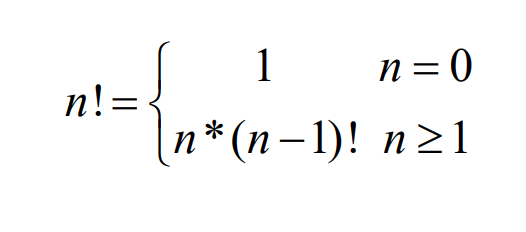
}

# ⚖ Recursion VS Iteration

* Recursion : using recursive calls  
  pros - Good choice for recursive problems (**easy** to implement)  
  cons - overhead of function calls -> usually **slower**execution time
* Iteration : using '**for**' or '**while**' loop  
  pros - fast execution time  
  cons - programming can be often vety **difficult** for recursive problems

Then, what is best strategy?  
-> It **depends on problems**.

# 🧬 Factorial



### 1. recursive Implimentation of Factorial

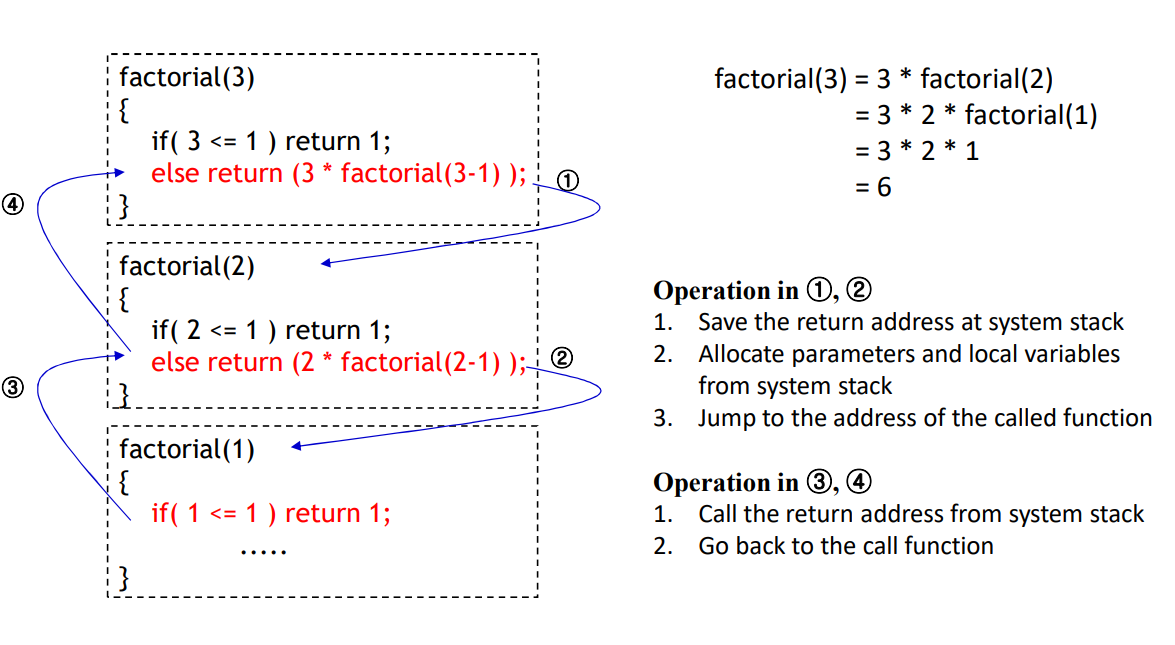
int factorial\_recur(int n) {

if (n <= 1) return 1;

else return n \* factorial(n - 1);

}

#### T(N) = T(N-1) + 1 --> Time Complexity : O(N)



### 2. Iterative Implimentation of Factorial

int factorial\_iter(int n) {

int mul = 1;

for (int i = n; i >= 1; i--) {

mul \*= n; // n\* (n-1) ... 2\*1 ;

}

return mul ;

}

#### Time Complexity : O(N)

### 3. compare

# 📌 Power Computation

: Let's find n-squared value of x :x^n

### 1. recursive Implimentation of Power Computation

double power\_iter(double x, int n) { //x^n

int i;

double r = 1.0;

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

r = r \* x;

}

return r;

}

#### Time Complexity : O(N)

### 2. Iterative Implimentation of Power Computation

double power\_recur(double x, int n) {

if (n == 0) return 1;

else if ((n % 2) == 0)

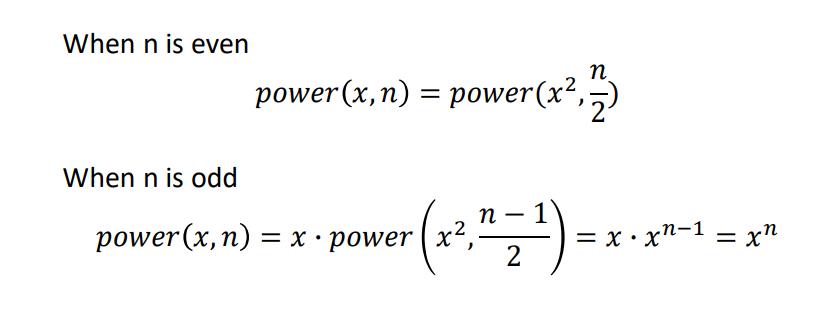
return power(x \* x, n / 2);

else

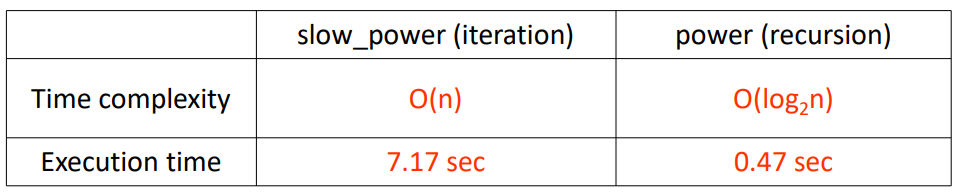
return x \* power(x \* x, n-1/ 2);

}

#### Time Complexity : T(N) = T(2/N) +C -> O(logN)

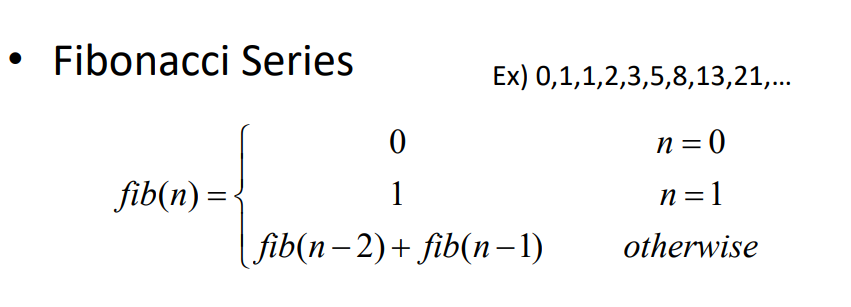


### 3. compare



iteration is not good choice😥

# 📝 Fibonacci Series



### 1. recursive Implimentation of Fibonacci Series

int fib\_recur(int n) {

if (n == 0) return 0;

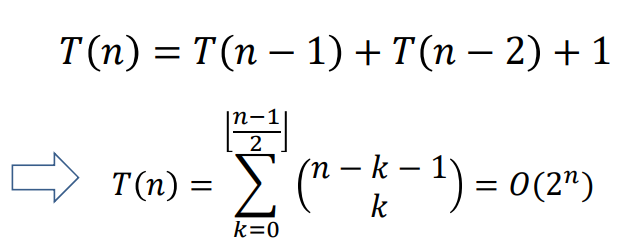
if (n == 1) return 1;

return fib\_recur(n - 1) + fib\_recur(n - 2);

}

#### Time Complexity : T(N) < 2^(N-1) -> O(2^N)

why ? for fib(n), maximum depth of tree is n-1.  
and tree must be not filled with leaf node fully.



### 2. Iterative Implimentation of Fibonacci Series

: to get fib(n), calculating one by one by one from the first two terms .

int fib\_iter(int n) {

int temp, current = 1, last = 0; //the first two terms of Fibonacci sequence.

if (n < 2) return n;

else {

for (int i = 1; i < n; i++) {

temp = current; //keep the last value.

current = last + current ; //Renewal Nth term

last = temp; //Save it for the next calculation.

}

}

return current;

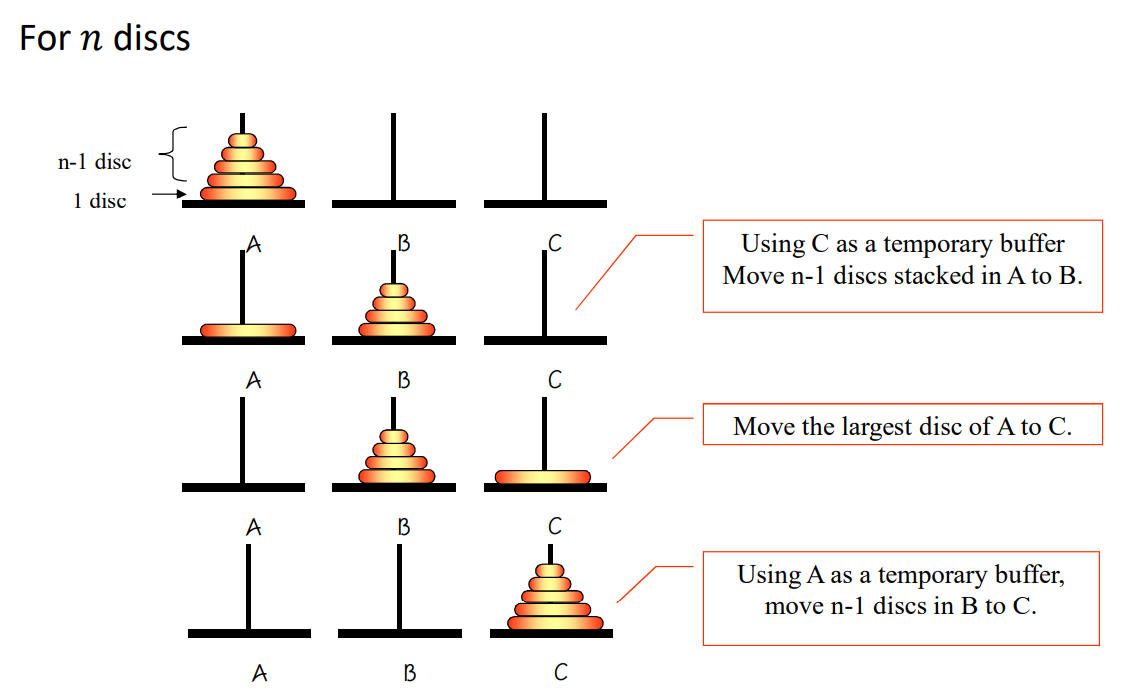
}

#### Time Complexity : O(N)

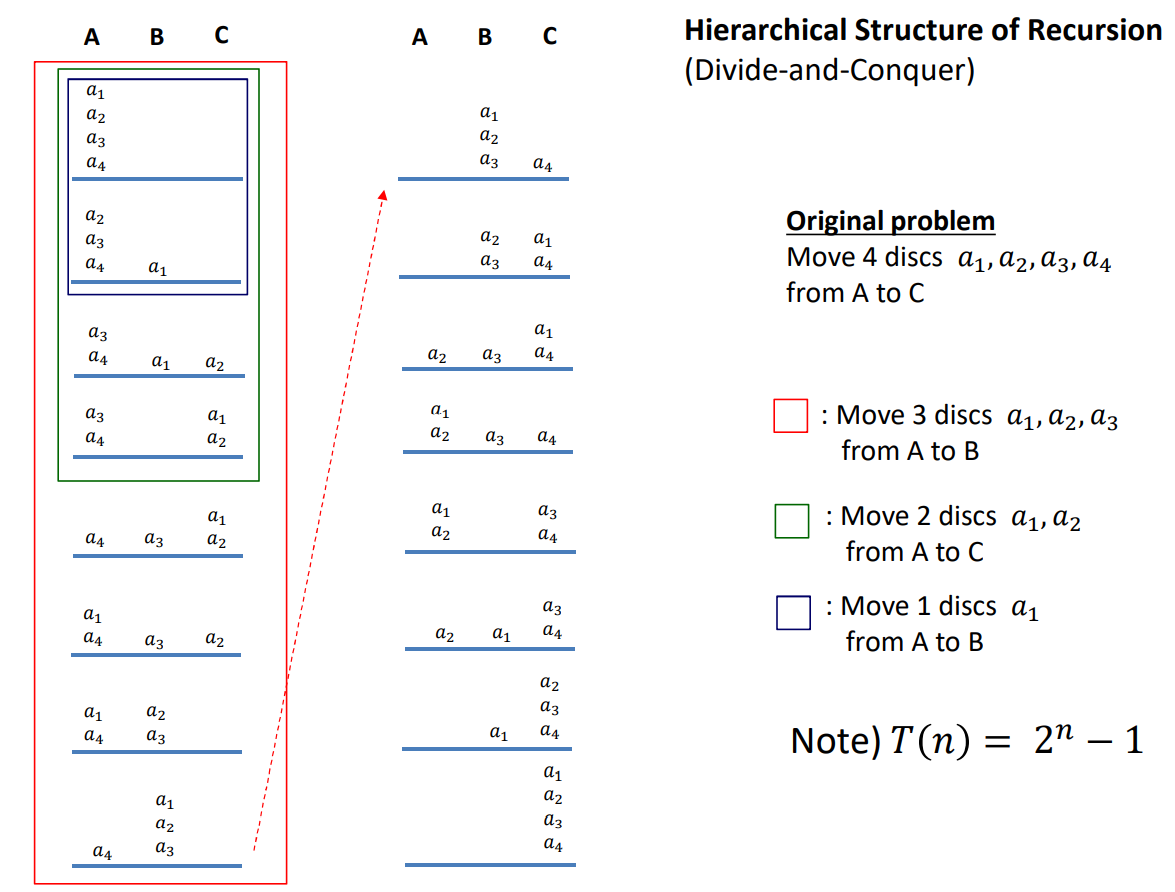
### 3. compare

# ♟Hanoi tower

: move n disc stacked on rod A to rod C.



### Divide and conquer



### 1. recursive Implimentation of Hanoi tower

void hanoi\_tower(int n, char from, char tmp, char to)

{

if (n == 1) printf("Move 1 disc from % c to % c.\n", from, to);

else {

hanoi\_tower(n - 1, from, to, tmp);

printf("Move disc % d from % c to % c.\n", n, from, to);

hanoi\_tower(n - 1, tmp, from, to);

}

}

#### Time Complexity : T(N) = 2T(N-1)+1 = 2^N -1 -> O(2^N)

### 2. Iterative Implimentation of Hanoi tower

### 3. compare

# 🎫Binary Search

: when a array of ordered numbers is given, find an index k where ak=b.

### 1. recursive Implimentation of Binary Search

int search\_iter(A, b)

for i=1 to n

if(A[i] == b)

k=i;

return k

#### Time Complexity : O(N)

### 2. Iterative Implimentation of Binary Search

int search\_recur(A, b, start, end)

if(start>end) return -1;

int median = (start+end)/2;

if(A[median]<b)

search\_recur(A, b, median, end);

else if(A[median]>b)

search\_recur(A, b, start, median);

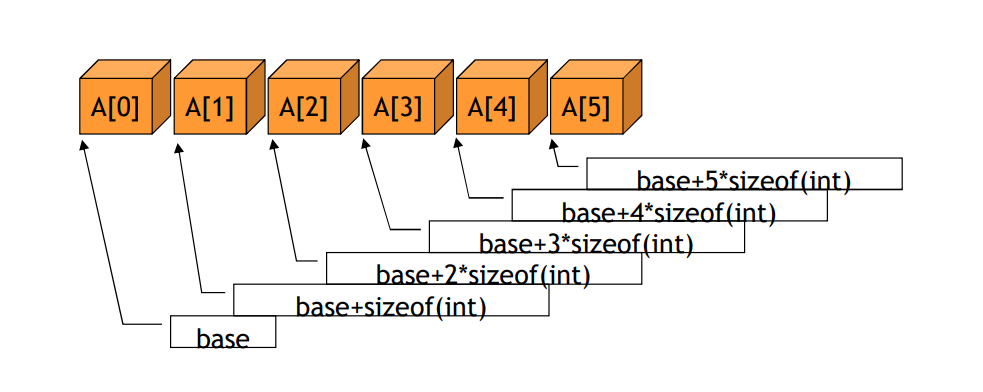
else

return median;

#### Time Complexity : T(N)=t(N/2)+C -> O(logN)

**🎈 Array**

: systematic arrangement of multiple variable with same data type sequentially.



**🧷 Array VS Structure**

* Array : group data of the **same**type.
* Structure : group data of **different**types.  
  thus, comparing between structure variables can't be implemented  
  ex ) person p1 > person p2 (x)

**📥 Self-Referential Structrue**

: Structrue that has one or more **pointers to itself** in the field. Is often used in linked lists or trees.

typedef struct ListNode {

char data[10];

struct ListNode \*link;

} ListNode;

**📰 Structrue Array**

#define MAX\_STUDENTS 200

#define MAX\_NAME 100

typedef struct {

int month;

int date;

} BirthdayType;

typedef struct {

char name[MAX\_NAME];

BirthdayType birthday; // another structure

} StudentType;

StudentType students[MAX\_STUDENTS];

void main()

{

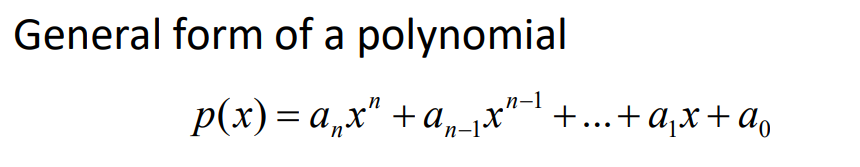
strcpy(students[0].name, “HongGilDong”);

students[0].birthday.month = 10; //Approach Hierarchically

students[0].birthday.date = 28;

}

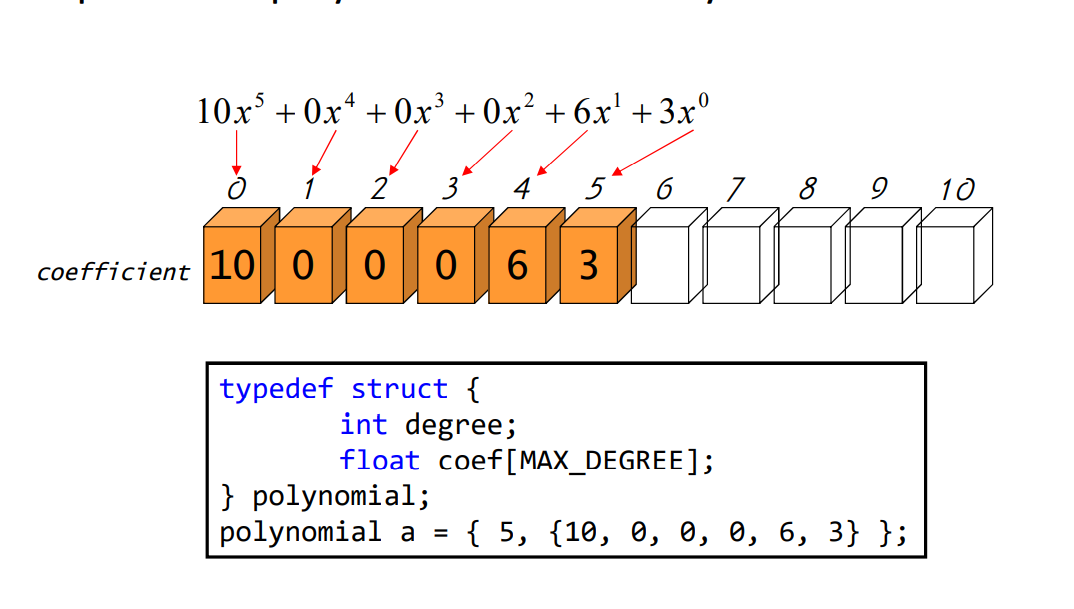
**📐Applications of Arrays: Polynomials**

: Let's see how Addition operation in polynomial works using structure.  


**1. Polynomial Representation in Arrays (1)**

: Store all terms of a polynomial in an array. one polynomial in **single structure**.

– Pros: **Simplified** polynomial operations  
– Cons: It causes **wasteful** space, when most of the coefficients  
are zero.



#include <iostream>

#define MAX(a,b) (a>b ? a:b)

#define MAX\_DEGREE 101

using namespace std;

typedef struct {

int degree; //차수

float coef[MAX\_DEGREE]; // 항들의 계수

}polynomial;

polynomial add(polynomial A, polynomial B) {

polynomial C;

int Apos = 0, Bpos = 0, Cpos = 0;

int degree\_A = A.degree;

int degree\_B = B.degree;

C.degree = MAX(A.degree, B.degree); //C의 차수 = A,B중 최고차항

while (Apos <= A.degree && Bpos <= B.degree) {

if (degree\_A > degree\_B) // A의 차수가 더 크다면

{

C.coef[Cpos++] = A.coef[Apos++]; //A항을 C항에 옮긴다.

degree\_A--;

}

else if (degree\_A > degree\_B) //차수가 같다면

{

C.coef[Cpos++] = A.coef[Apos++] + B.coef[Bpos++];

degree\_A--;

degree\_B--;

}

else // B의 차수가 더 크다면

{

C.coef[Cpos++] = B.coef[Bpos++]; //A항을 C항에 옮긴다.

degree\_B--;

}

}

return C;

}

int main() {

polynomial A = { 5,{3,6,0,0,0,10} };

polynomial B = {4,{7,0,5,0,1} };

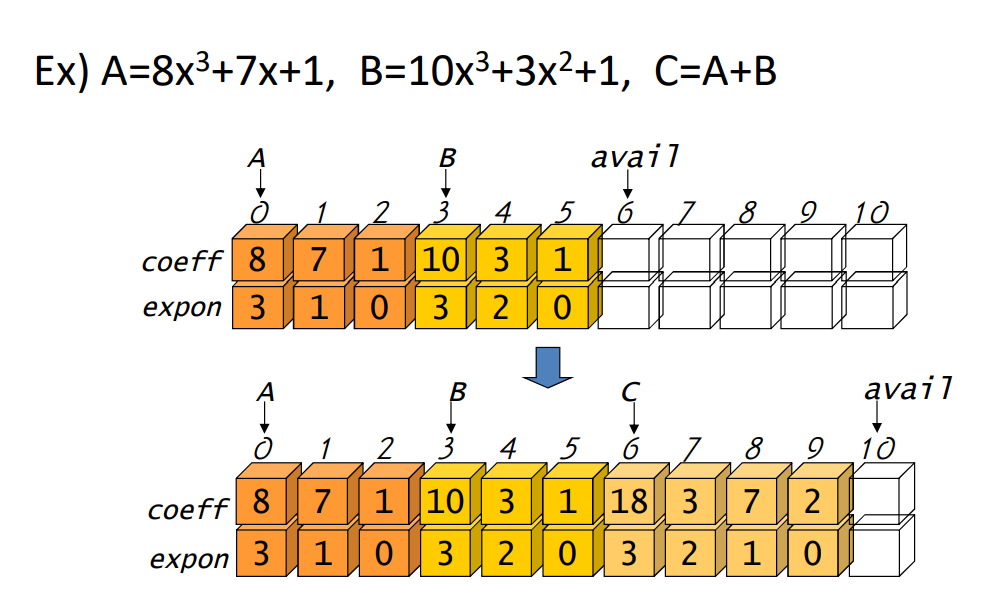
polynomial C = add(A, B);

}

**2. Polynomial Representation in Arrays (2)**

: Store **only non-zero terms** of a polynomial in an array.  
multiple polynomial in **single array**.

• Pros: **Efficient use of memory** space  
• Cons: Polynomial operations are **complex**



#include <iostream>

#define MAX\_TERMS 101

using namespace std;

struct {

float coef;

int expon;

}terms[MAX\_TERMS] = { {8,3},{7,1},{1,0},{10,3},{3,2},{1,0} };

//위 문장의 의미? 구조체 배열을 만든다?

int avail = 6; //결과를 넣을 인덱스 배열의 시작점

//compare two integers

char compare(int a, int b) {

if (a > b) return '>';

else if (a == b) return '=';

else return '<';

}

void attach(float coef, int expon) { //새로운 항을 새로운 다항식 C (결과)에 추가한다.

if (avail > MAX\_TERMS) {

fprintf(stderr, "too many terms\n");

exit(1);

}

//avail이 오른쪽으로 한칸씩 움직이면서 새로운 항을 추가함.

terms[avail].coef = coef; //계수넣기

terms[avail++].expon = expon; //지수 넣고 인덱스 뒤로 이동

}

void add(int As, int Ae, int Bs, int Be, int\* Cs, int\* Ce) {

float tempcoef; //덧셈 저장용

\*Cs = avail;

while (As <= Ae && Bs <= Be) {

switch (compare(terms[As].expon, terms[Bs].expon))

{

case '>': //A항 차수 > B항 차수

attach(terms[As].coef, terms[As].expon);

As++; // A를 다음 항으로

break;

case '=': //A항 차수 = B항 차수

tempcoef = terms[As].coef + terms[Bs].coef; //A항 계수 + B항계수 합친것

if (tempcoef) //만일 더했을때 계수가 0이라면 새로운 다항식 C에 추가할필요가 없음. (없는항)

attach(tempcoef, terms[As].expon); //2번째인자는 terms[Bs].expon 여도 상관X 어짜피 차수 같으니..

As++; Bs++;

break;

case '<': //A항 차수 < B항 차수

attach(terms[Bs].coef, terms[Bs].expon);

Bs++; // B를 다음 항으로

break;

}

}

//Q. 항들이 어떻게 남아있을 수 있냐?

//A. while문 조건이 As <= Ae && Bs <= Be 이기때문.

//A에 남아있는 항들을 복사 붙여넣기

for (; As <= Ae; As++) {

attach(terms[As].coef, terms[As].expon);

}

for (; Bs <= Be; Bs++) {

attach(terms[Bs].coef, terms[Bs].expon);

}

\*Ce = avail - 1; //avail은 현재 비어있는 공간을 가리키고 있으므로 -1을 해주어야 C가 끝나는 지점의 index가 됨.

}

int main() {

int Cs, Ce; //결과 다항식의 시작 & 끝 인덱스를 담을 변수

add(0, 2, 3, 5, &Cs, &Ce); //A다항식의 시작과 끝, B다항식의 시작과 끝.

//주소를 넘겨주어야 실질적으로 값을 변경할 수 있음.

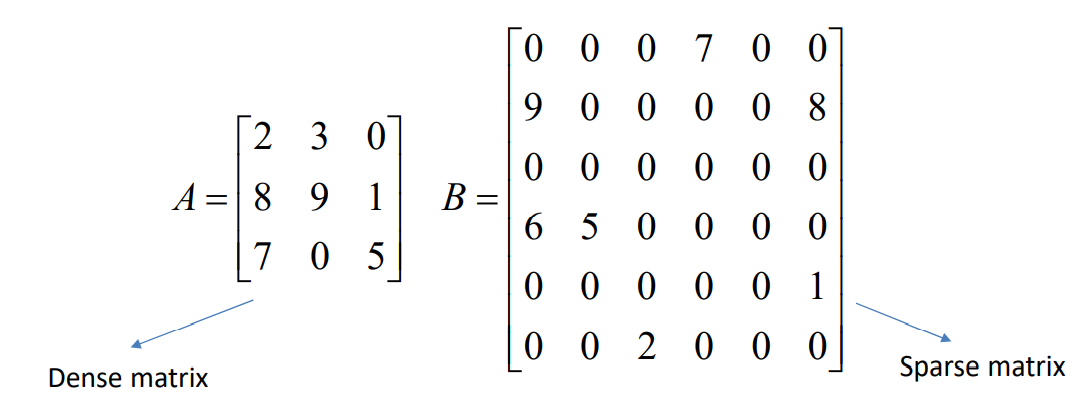
for (int i = Cs; i <= Ce; i++) {

cout << terms[i].coef<<' ';

}

}

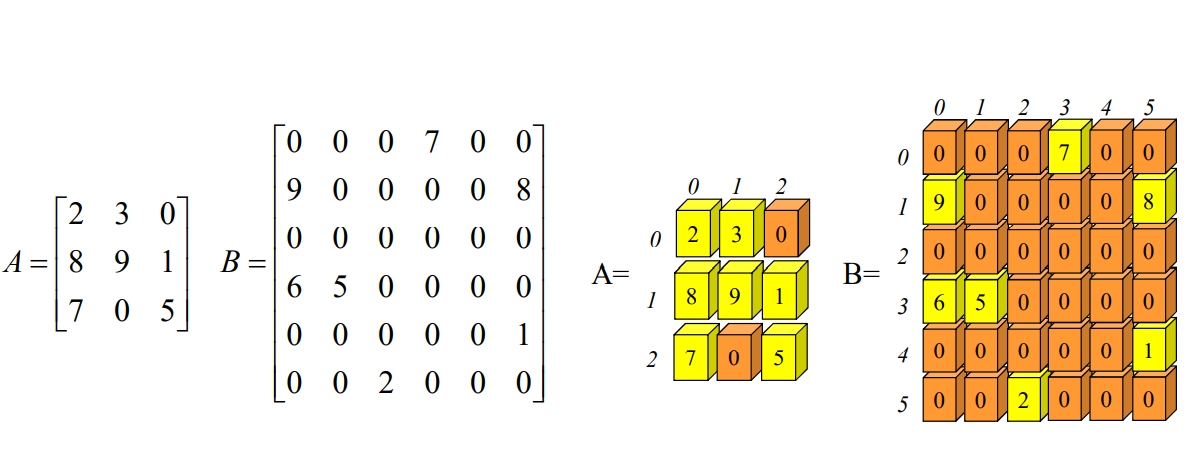
**📓 Sparse Matrix**

: Matrix where**most terms are zero**  


**1. Sparse Matrix Representation (1)**

: store all elements in a 2D array

* Pros: Matrix operations can be implemented **simply**.
* Cons: Memory is **wasted** when most terms are zero



#include <stdio.h>

#define ROWS 3

#define COLS 3

// Addition

void sparse\_matrix\_add1(int A[ROWS][COLS], int B[ROWS][COLS], int C[ROWS][COLS]) // C=A+B

{

int r,c;

for(r=0;r<ROWS;r++)

for(c=0;c<COLS;c++)

C[r][c] = A[r][c] + B[r][c];

}

main()

{

int array1[ROWS][COLS] = { { 2,3,0 },{ 8,9,1 },{ 7,0,5 } };

int array2[ROWS][COLS] = { { 1,0,0 },{ 1,0,0 },{ 1,0,0 } };

int array3[ROWS][COLS];

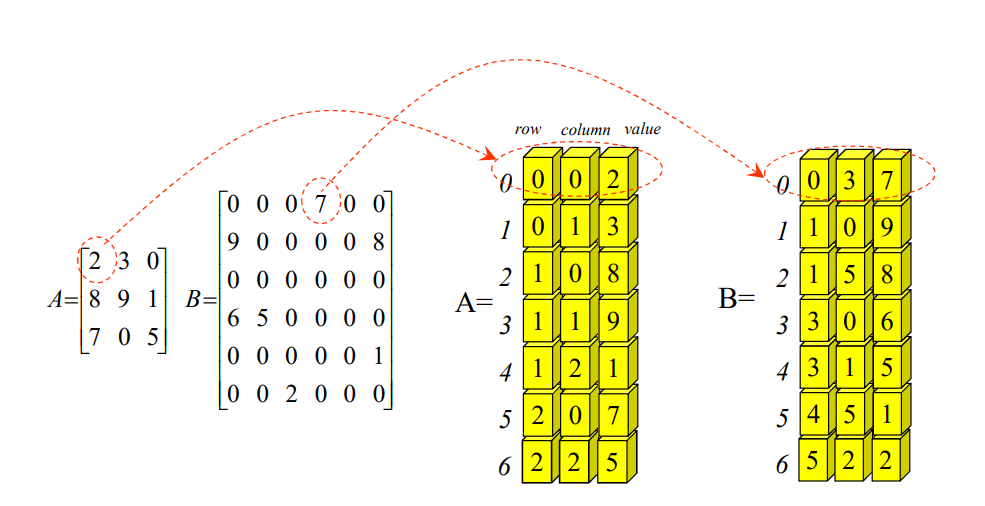
sparse\_matrix\_add1(array1,array2,array3);

}

**2. Sparse Matrix Representation (2)**

: store only non-zero elements

* Pros: Memory is saved for sparse matrices
* Cons: Complex implementation of matrix operations



#include <iostream>

#define ROWS 3

#define COLS 3

#define MAX\_TERMS 10

using namespace std;

typedef struct {

int row; //행

int col; //열

int value; //값

}element;

typedef struct SparseMatrix{

element data[MAX\_TERMS];

int rows; //매트릭스 행 크기

int cols; //매트릭스 열 크기

int terms; //element의 수 -> max랑 같은거 아닌가

}SparseMatirx;

SparseMatrix sparse\_matrix\_add2(SparseMatrix a, SparseMatrix b) {

SparseMatirx c;

int ca = 0, cb = 0, cc = 0; //배열 a,b,c 의 data의 인덱스를 각각 가리키는 변수.

// 배열 a와 배열 b가 같은 사이즈인지 확인한다.

if (a.rows != b.rows || a.cols != a.rows) {

fprintf(stderr, "size가 같지 않아 덧셈 불가능");

exit(1);

}

//배열 덧셈 결과인 C의 행,열 크기 세팅

c.rows = a.rows; //a==b이므로 b여도 상관 X

c.cols = a.cols;

c.terms = 0;

//rows cols가 일치하면 c에 추가한다.

while (ca < a.terms && cb << b.terms) {

// 각각의 item의 index를 계산한다. ->어떤항을 먼저 삽입할지 알기위함.

int inda = a.data[ca].row \* a.cols + a.data[ca].col; //배열c에 a가 들어갈 index.

//만약에 ca가 가리키는 값이 2행 3열이라면 c에서 2\*3+3 번째에 위치하게됨.

int indb = b.data[ca].row \* b.cols + b.data[ca].col; //배열c에 b가 들어갈 index.

if (inda < indb) { //a가 더 먼저나온다면 (c배열에는 윗줄부터 차례로 넣어야하므로 큰 값이 나중에 채워짐)

c.data[cc++] = a.data[ca++];

}

// a==b라면

else if (inda == indb) {

if (a.data->value + b.data->value != 0) {

//c.data =a.data+b.data 가 불가능하기 때문에 각각 고려해서 대입해주어야함.

c.data[cc].row = a.data[ca].row; //b여도 상관 X

c.data[cc].col = a.data[ca].col; //b여도 상관 X

c.data[cc].value = a.data[ca++].value + b.data[cb++].value; //c는 a값과 b값을 더한값

}

else

ca++; cb++; //더했을때 항이 0이므로 그냥 패스하고 다음항.

} //b가 더 먼저 나온다면

else

c.data[cc++] = b.data[cb++];

}

}

int main() {

SparseMatirx m1 = { {{ 1,1,5 },{ 2,2,9 }}, 3,3,2 };

SparseMatrix m2 = { {{ 0,0,5 },{ 2,2,9 }}, 3,3,2 };

SparseMatrix m3;

m3 = sparse\_matrix\_add2(m1, m2);