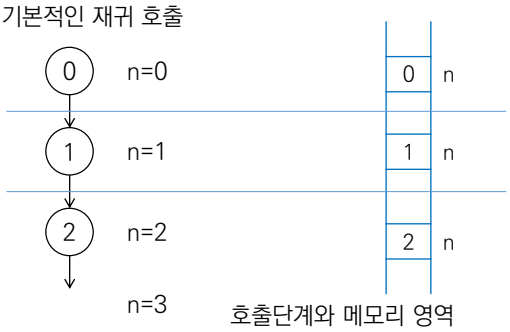
## 재귀함수

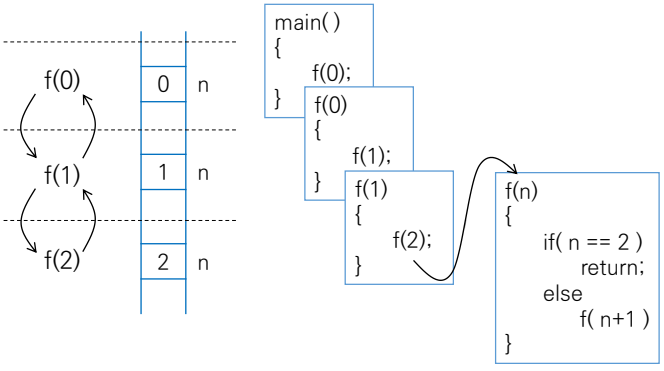
- 특징
  - 각 단계는 같은 함수이다.
  - 각 단계에서 사용하는 메모리 영역이 다르다.
  - 정해진 횟수만큼, 혹은 조건을 만족할 때 까지 호출을 반복한다.

```
f(int n)
{
    ...
    f(n+1);
}
```



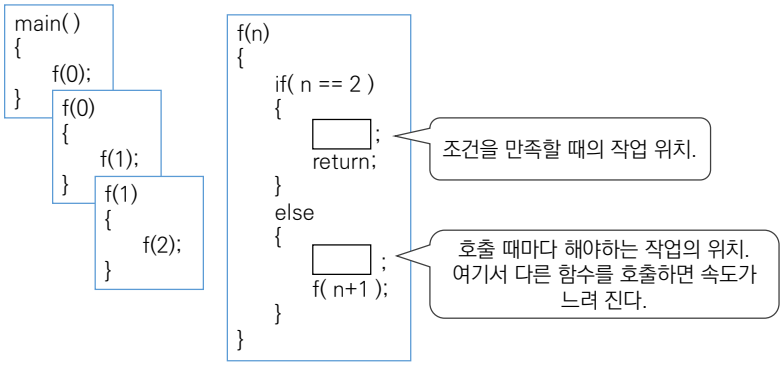

## 정해진 횟수만큼 호출하기

- 호출 횟수에 대한 정보는 인자로 전달.
- 마지막 호출에서 재귀 호출 중단 판정.



## 재귀함수의 구조

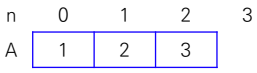
- 조건을 만족하면 더 이상 호출하지 않고  작업하기.
- 호출 횟수만큼 작업하기.




- 재귀호출을 이용해 배열 채우기.
  - 호출 단계를 배열 인덱스로 활용.
  - 호출 단계가 배열의 크기를 벗어나면 호출 중지.
  - 배열이 다 채워지면 내용 출력.

| n | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| A | 1 | 2 | 3 | |

- 재귀호출을 이용해 배열 채우기.
  - 호출 단계를 배열 인덱스로 활용.
  - 호출 단계가 배열의 크기를 벗어나면 호출 중지.
  - 배열이 다 채워지면 내용 출력.




```
f(n)
{
    if( n == 3 )
    {
        printArray( );
        return;
    }
    else
    {
        A = n+1;
        f( n+1 );
    }
}
```

## 연습

- N! (factorial)
  - 3! = 3 * 2!
  - f(n) = n  * f(n-1); // n > 0;
  - 0! = 1;

```
int f(int n)
{
    if( n < 2 )
        return 1;
    else
        return n * f(n-1);
}
```

## 같은 깊이에 2가지 상태를 갖는 재귀함수

- 배열의 각 자리에 0/1이 오는 모든 경우 만들기.

| 호출단계 → | 0 | 1 | 2 |
|---|---|---|---|
| A | 0/1 | 0/1 | 0/1 |

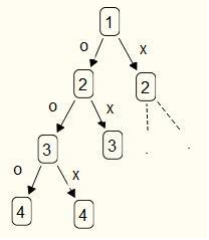
000
001
010
…
111

예 ) 부분 집합 만들기



## 

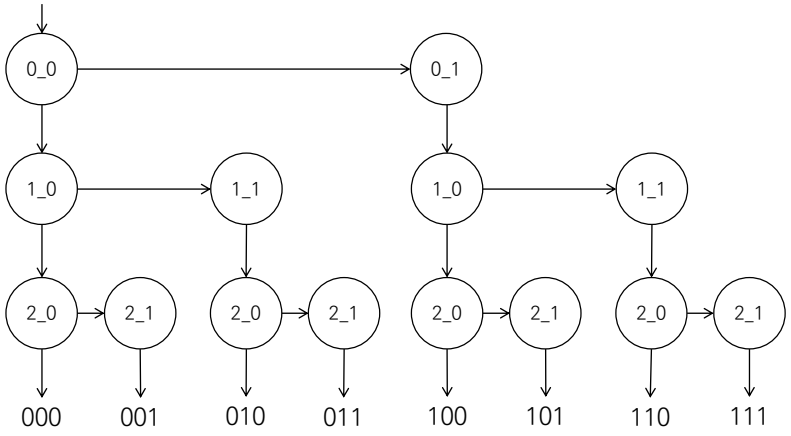
- 부분집합




o는 포함된 경우, x는 포함되지 않은 경우.

```
L[n] = 1;       // n번 요소를 포함시키고,
subset(n + 1); // 다음 요소 선택하는 함수 호출
L[n] = 0;       // n번 요소는 포함하지 않고,
subset(n + 1); // 다음 요소 선택하는 함수 호출
```

- 호출 단계
  - ‘호출 깊이_상태’로 표시된 호출 단계.




- 호출 깊이 k, 채울 배열의 크기 N, 배열 A
  - A[k]가 0, 1인 경우에 대해 각각 A[k+1] 결정 단계 호출.

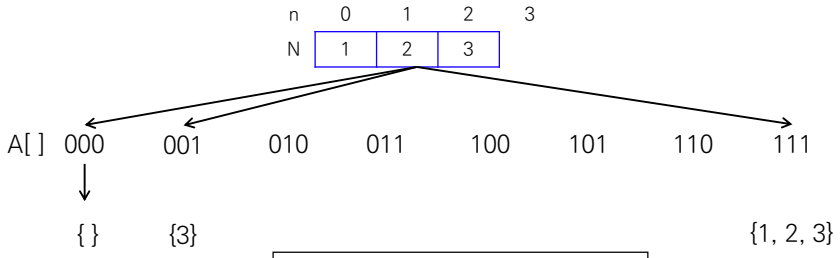
```
int A[N];

f( k, N)
{
    if( k==N )
        //배열 A 출력
    else
        A[k] = 0;
        f( k+1, N );
        A[k] = 1;
        f( k+1, N );
}
```

```
int A[N];

f( k, N)
{
    if( k==N )
        //배열 A 출력
    else
        for( i : 0 -> 1)
            A[k] = i;
            f( k+1, N );
}
```

그림에서 수평으로 나타낸 화살표는 for문으로 처리한다.

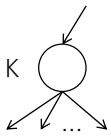
## 연습

- {1, 2, 3}의 모든 부분 집합.
  - 각 원소의 포함 여부를 저장하는 배열을  추가로 만듦.
  - 모든 원소의 포함여부가 결정되면 부분집합을 출력.




```
for i : 0 -> 2
    if(A[i] == 1)
        printf("%d", N[i]);
```

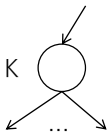
## 재귀 호출이 여러 번인 경우




항상 j번인 경우

```
    …
    else
        … // 현재 단계에서 처리할 일
        for i : 1 -> j
            f( K+1, i );
}
```

예 ) 1, 2, 3을 중복 사용해 3자리 숫자 만들기

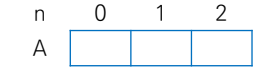


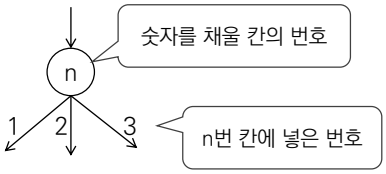

조건에 따라 달라지는 경우

```
    …
    else
        … // 현재 단계에서 처리할 일
        for i : 1 -> j
            if ( j가 유효 )
                f( K+1, i )
}
```

예 ) 그래프에서 인접 노드에 방문하기

- 1, 2, 3을 중복 사용해 3자리 숫자 만들기.



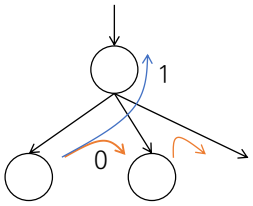




```
f(n)
{
    if( n == 3)
        printA( );
        return;
     else
        for i : 1 -> 3
            A[n] = i;
            f( n+1 );
}
```



- 원하는 조건을 찾으면 중단하는 경우

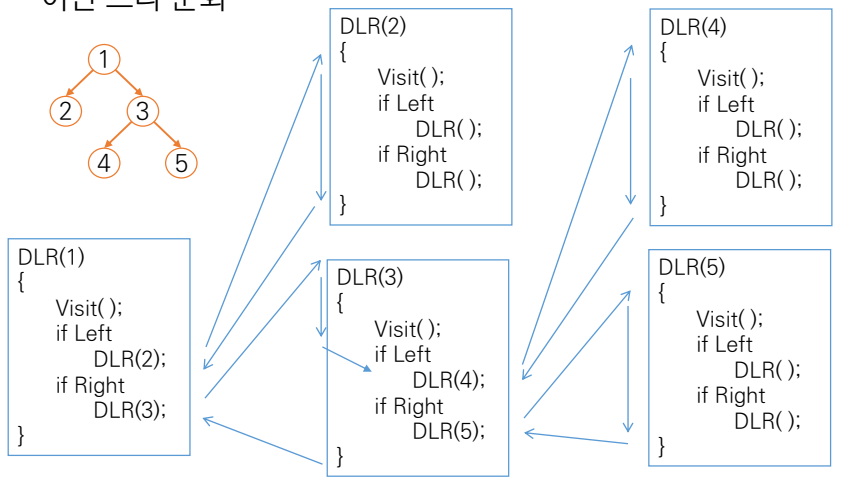
```
if ( 답을 찾은 중단 조건 )
    …
    return 1;
else
    … // 현재 단계에서 처리할 일
    for i : 0 -> j
        r = f( K+1, i );
        if( r == 1)
            return 1;
}
```




예 ) 부분 집합의 합이 23인 경우가 있으면 1, 없으면 0을 출력하라.
{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

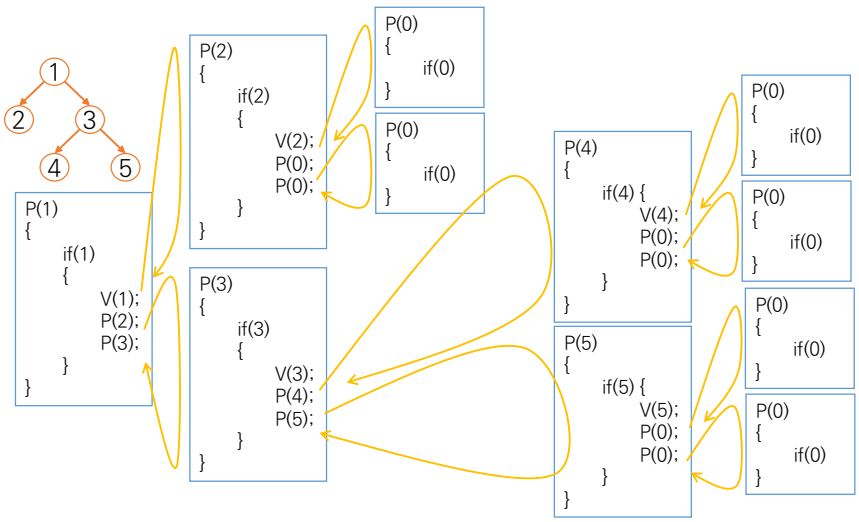
# 재귀 응용

- 이진 트리 순회






- 이진 트리 순회

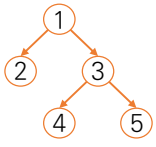



- 포화이진트리의 저장 방법

## 이진트리

- 트리 정보 저장
  - 부모 번호를 인덱스로 자식 번호를 저장.

| 부모 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 자식1 | 2 | 0 | 4 | 0 | 0 |
| 자식2 | 3 | 0 | 5 | 0 | 0 |

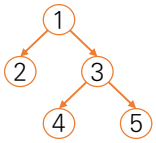



  - 자식 번호를 인덱스로 부모 번호를 저장.

| 자식 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 부모 | 0 | 1 | 1 | 3 | 3 |

## 연습

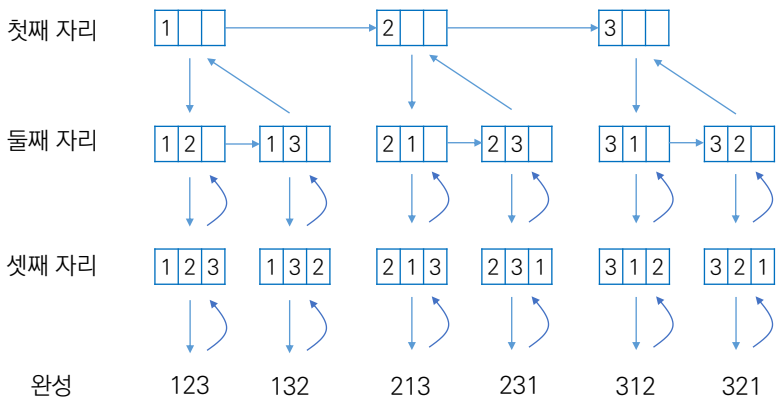
- 배열을 순회하고 방문한 노드의 개수를 출력하시오.



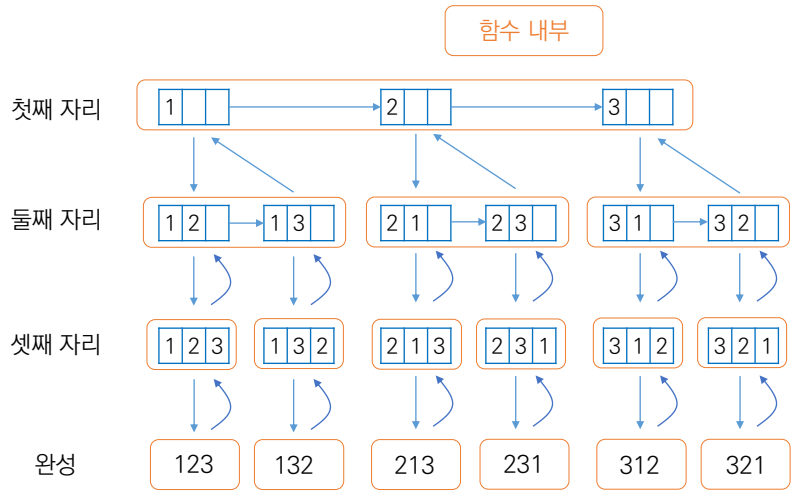

```
P(1)
{
    if(1)
    {
        cnt++;
        P(2);
        P(3);
    }
}
```

## 순열

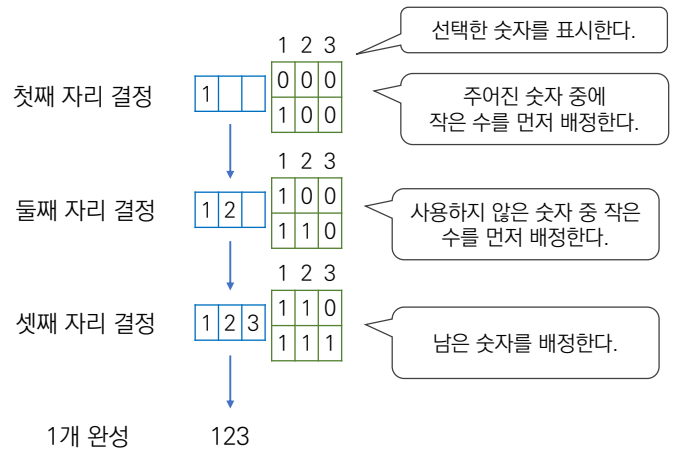
- 1, 2, 3으로 3자리 수 만들기



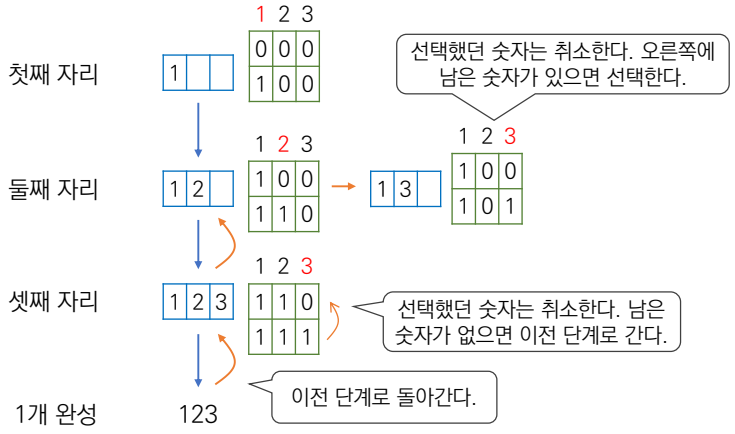
▪ 1, 2, 3으로 3자리 수 만들기



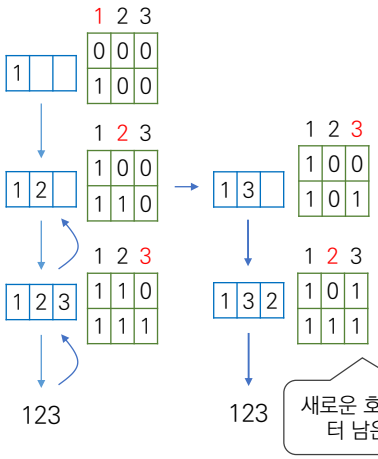

▪ 1, 2, 3으로 3자리 수 만들기



▪ 1, 2, 3으로 3자리 수 만들기 (계속)




▪ 1, 2, 3으로 3자리 수 만들기 (계속)




n : 사용할 숫자 개수
k : 선택한 숫자를 넣을 자리
i ; 사용하지 않은 숫자를 찾는 순서

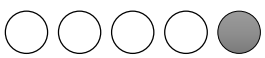
```
for (int i = 0; i < n; i++)
{
    if (used[i] == 0)
    {
        used[i] = 1;
        p[k] = a[i];
        perm(n, k + 1);
        used[i] = 0;
    }
}
```

## 연습

- {1, 2, 3, 4}의 원소를 사용해 순열 만들기.

## 조합

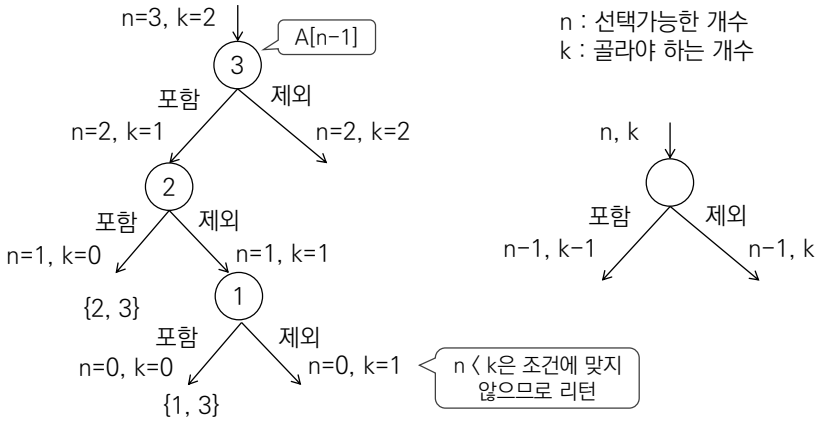
- n개에서 k개를 고르는 경우의 수 : nCk
  - {1, 2, 3}에서 두 개의 숫자를 고르는 경우의 수 : ${}_3C_2$
    - {1, 2}, {1, 3}, {2, 3}
- 두 경우로 나눠 생각할 수 있다.



특정원소를 포함하는 경우와 포함하지 않는 경우로 나눠서 생각할 수 있음.

  - 특정 원소를 선택하면 남은 숫자도 하나 줄고(n-1), 골라야 하는 개수도 줄어듬(k-1).
  - 특정 원소를 제외하면 남은 숫자는 하나 줄고(n-1), 골라야 하는 개수는 그대로(k).
  - ${}_nC_k = {}_{n-1}C_{k-1} + {}_{n-1}C_k$

- n개에서 k개를 고르는 경우의 수 : nCk
  - A[] = {1, 2, 3}에서 두 개의 숫자를 고르는 경우의 수 : ${}_3C_2$



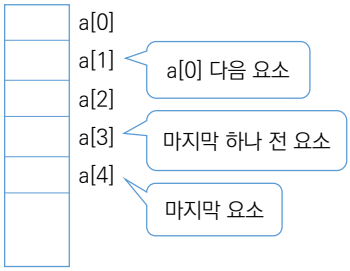

## 연습

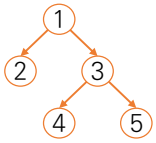
- {1, 2, 3}에서 2개를 고르는 조합 만들기.

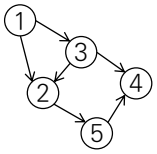
```
void ncr(int n, int k)
{
        if (k == 0)
        {
                for (int i = 0; i < 2; i++)
                printf("%d ", C[i]);
                printf("\n");
        }
        else if (n < k)
            return;
        else
        {
                C[k - 1] = A[n - 1];
                ncr(n - 1, k - 1);
                ncr(n - 1, k);
        }
}
```

## 자료구조

- 선형과 비선형 자료구조.

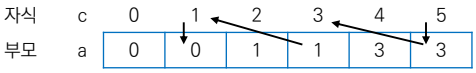


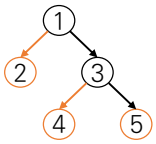




## 이진 트리

- 조상 노드 찾기



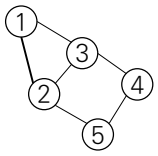




5번 노드의 조상 찾기

```
c = 5;
while( a[c] != 0 ) // 루트인지 확인
{
    c = a[c];
    printf("%d ",c);
}
```

## 인접행렬

- 무향 그래프




| M | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 |
| 2 | 1 | | | | |
| 3 | 1 | | | | |
| 4 | 0 | | | | |
| 5 | 0 | | | | |

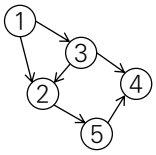
인접은 1, 아닌 경우 0

그래프는 간선 양쪽의 노드 번호로 표현 가능.

1 2 1 3 2 3 3 4 2 5 4 5

```
scanf("%d %d", &n1, &n2);
M[n1][n2] = 1;
M[n2][n1] = 1;
```

- 방향성 그래프




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 |
| 2 | 0 | | | | |
| 3 | 0 | | | | |
| 4 | 0 | | | | |
| 5 | 0 | | | | |

도착

출발

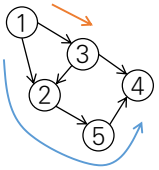
출발 -> 도착 노드로 주어지는 경우

1 2 1 3 3 2 3 4 2 5 4 5

```
scanf("%d %d", &n1, &n2);
M[n1][n2] = 1;
```

## 탐색

- 깊이 우선 탐색 (DFS)
  - 2개 이상의 선택이 가능할 때, 정해진 방향에 따라 다음 노드 선택.
  - 더 이상 갈 수 없으면 가장 가까운 이전 갈림길에서 다른 방향 선택.
    - 지나온 경로를 저장해야 함.



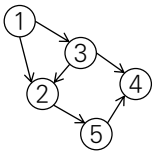

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 |

1 -> 2 -> 5 -> 4 -> 3



## 

- 재귀를 사용한 DFS
  - 재귀의 각 단계가 방문중인 노드 번호를 저장.
  - 방문한 노드에서 방문하지 않은 인접 노드 중 번호가 작은 곳으로 이동.

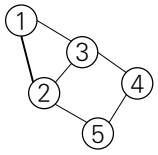



```
DFS(n)
    V[n] = 1                    // 방문 표시
    visit(n)                    // 노드에 대해 처리할 일
    for i : 1 -> N
        if( M[n][i] == 1 && V[i] == 0 )
            DFS(i) // 인접하고 방문하지 않은 노드로 이동
```

- 반복 구조의 DFS
  - 지나온 노드를 스택에 저장.

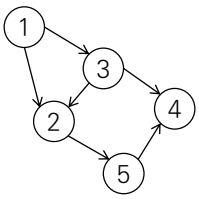
## 연습

- 다음 그래프를 DFS로 탐색하고 방문 순서를 출력하시오.




5 6
1 2 1 3 3 2 3 4 2 5 4 5

- BFS (너비 우선 탐색)
  - 시작 정점부터 거쳐가는 간선의 수가 같은 순서로 방문.
  - 다음 방문할 곳을 큐에 저장.

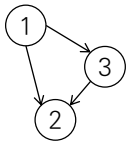



```
BFS(s)
    enQ(s)                  // 시작점 enQ
    V[s] = 1                // 방문 표시
    visit(s)                // 노드에 대해 처리할 일
    while( is_not_emptyQ())
        n = deQ()
        for i : 1 -> N
            if( A[n][i] == 1 && V[i] == 0 )
                enQ(i)
                V[i] = 1
```

  - V[i] = V[n] + 1로 바꾸면 시작 부터의 거리를 알 수 있다.

## 위상 정렬

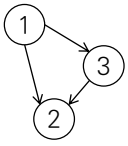
- 앞의 1, 3이 처리되어야 2번을 처리할 수 있는 경우.
  - 정점의 진입 차수를 활용.
  - 진입 차수가 0인 정점부터 시작.
  - 정점을 처리할 때 인접 정정에 처리되었음을 알림.
    - 인접 정점의 진입 차수를 하나 줄임.
    - 진입 차수가 0이 되면 다음 번에 처리할 차례가 됨.




| A | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
| 2 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 |

| I | 0 | 2 | 1 |
|---|---|---|---|

A : 인접행렬
I : 진입 차수




A : 인접행렬
I : 진입 차수

```
Sort( )
    for i : 0 -> N          // 진입차수가 0이면 enQ
        if( I[i] == 0 )
            enQ(i)
    while(is_not_emptyQ())
        n = deQ( )
        do(n)               // 노드 n에서 해야할 일
        for i : 1 -> N
            if( A[n][i] == 1 )
                I[i]—   // n의 인접노드 진입차수 감소
                if( I[i] == 0 )  // 진입차수가 0이면 enQ
                    enQ(i)
```

| Q | 1 | | | | | | | |
|---|---|---|---|---|---|---|---|---|

| I | 0 | 2 | 1 |
|---|---|---|---|

| Q | ~~1~~ | 3 | | | | | | |
|---|---|---|---|---|---|---|---|---|

| I | 0 | 2->1 | 1->0 |
|---|---|---|---|

| Q | ~~1~~ | ~~3~~ | 2 | | | | | |
|---|---|---|---|---|---|---|---|---|

| I | 0 | 1->0 | 0 |
|---|---|---|---|

| Q | ~~1~~ | ~~3~~ | 2 | | | | | |
|---|---|---|---|---|---|---|---|---|

| I | 0 | 0 | 0 |
|---|---|---|---|

## 미로

- 배열에 저장한 미로
  - 1-벽, 0-통로, S-출발, G-도착
  - 도착 가능 여부 판단, 최단 거리 구하기, 경로의 수 구하기 등.

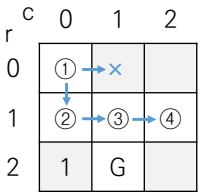
| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | S | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | G |
| 2 | 1 | 0 | 1 | 0 | 1 |
| 3 | 1 | 0 | 0 | 0 | 0 |
| 4 | 1 | 0 | 1 | 1 | 0 |

column

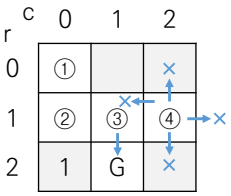
row



## 미로 찾기



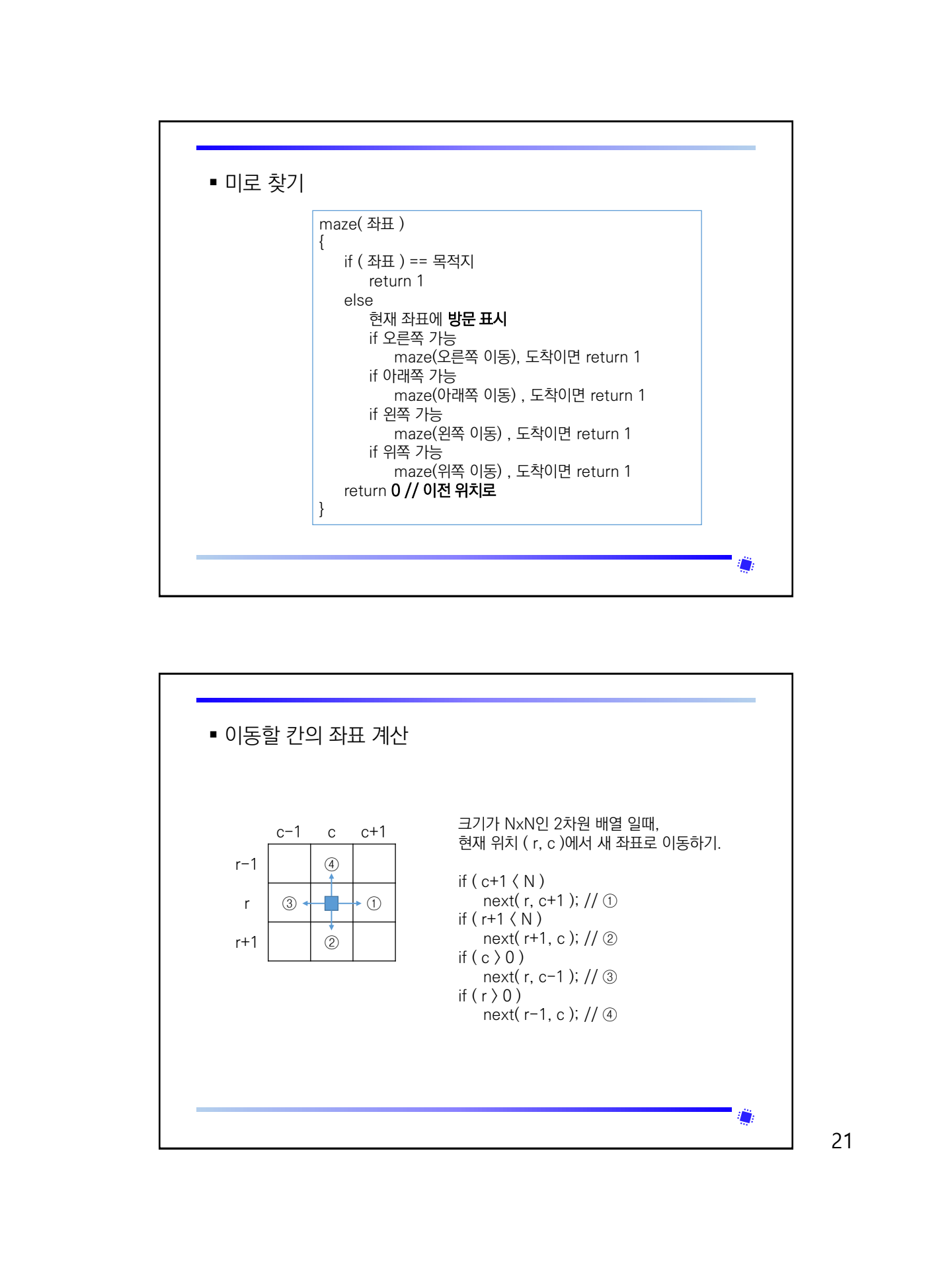

| | r | c | |
|---|---|---|---|
| ① | 0 | 0 | 오른쪽 불가. 아래 가능. |
| ② | 1 | 0 | 오른쪽 가능. |
| ③ | 1 | 1 | 오른쪽 가능. |




| | r | c | |
|---|---|---|---|
| ④ | 1 | 2 | 오른쪽/아래 불가. 왼쪽 **이미 방문**. 위쪽 불가. |
| ③ | 1 | 1 | **이전 위치**로 되돌아감. 아래 가능. |
| G | 2 | 1 | 도착 |

▪ 미로 찾기

```
maze( 좌표 )
{
    if ( 좌표 ) == 목적지
        return 1
    else
        현재 좌표에 방문 표시
        if 오른쪽 가능
            maze(오른쪽 이동), 도착이면 return 1
        if 아래쪽 가능
            maze(아래쪽 이동) , 도착이면 return 1
        if 왼쪽 가능
            maze(왼쪽 이동) , 도착이면 return 1
        if 위쪽 가능
            maze(위쪽 이동) , 도착이면 return 1
    return 0 // 이전 위치로
}
```



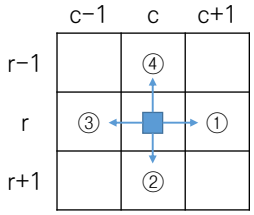
▪ 이동할 칸의 좌표 계산




크기가 NxN인 2차원 배열 일때,
현재 위치 ( r, c )에서 새 좌표로 이동하기.

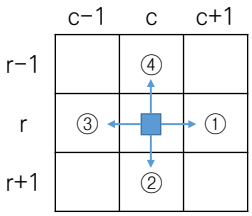
```
if ( c+1 〈 N )
    next( r, c+1 ); // ①
if ( r+1 〈 N )
    next( r+1, c ); // ②
if ( c 〉 0 )
    next( r, c−1 ); // ③
if ( r 〉 0 )
    next( r−1, c ); // ④
```

- 이동할 수 있는 칸의 좌표



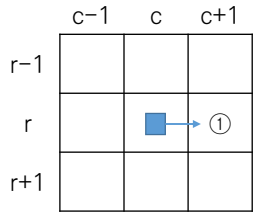

```
if( map[r][c] == 1) // 벽
    return;
else if( map[r][c] == 2 ) // 이미 방문한 곳
    return;
if ( c+1 < N )
    next( r, c+1 ); // ①
if ( r+1 < N )
    next( r+1, c ); // ②
if ( c > 0 )
    next( r, c-1 ); // ③
if ( r > 0 )
    next( r-1, c ); // ④
```

- 이동할 수 있는 칸의 좌표
  - 배열과 반복문을 이용한 이동 좌표 계산



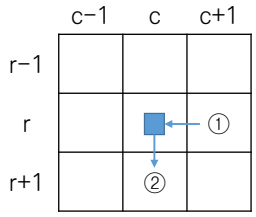

```
int dr[] = { 0, 1, 0, -1 };
int dc[] = { 1, 0, -1, 0};

...
for( i = 0 ; i < 4 ; i++ )
{
    nr = r + dr[i];
    nc = c + dc[i];
    if(( nr >= 0 )&&( nr < N )&&( nc >=0 )&&( nc < N ))
    {
        next( nr, nc );
    }
}
```

- DFS-재귀 호출
  - 각 호출 단계가 현재 좌표와 이동 방향을 기억한다.



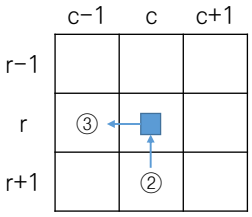

①로 이동

```
if ( c+1 < N )
    next( r, c+1 );
```




①에서 되돌아옴.
②로 이동.

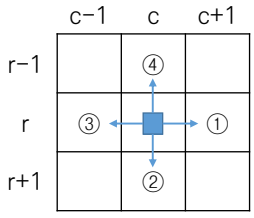
```
if ( r+1 < N )
    next( r+1, c );
```




②에서 되돌아옴.
③으로 이동.

```
if ( c > 0 )
    next( r, c-1 );
```

- DFS-반복
  - 현재 좌표와 이동 방향을 기억할 공간이 필요하다.




방법 1.
현재 좌표와 이동 방향을 기록한다.

## 연습

- 도착지까지 지나는 최소 칸 수는?

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | S | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | G |
| 2 | 1 | 0 | 1 | 0 | 1 |
| 3 | 1 | 0 | 0 | 0 | 0 |
| 4 | 1 | 0 | 1 | 1 | 0 |