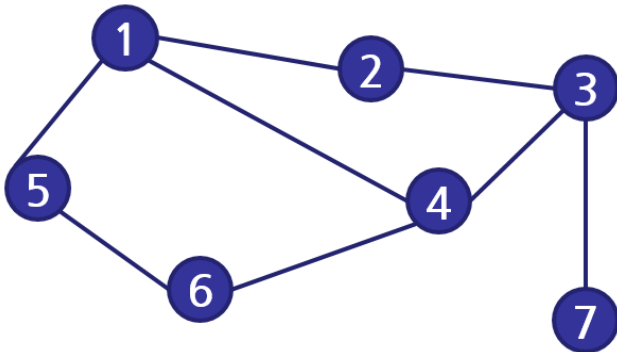


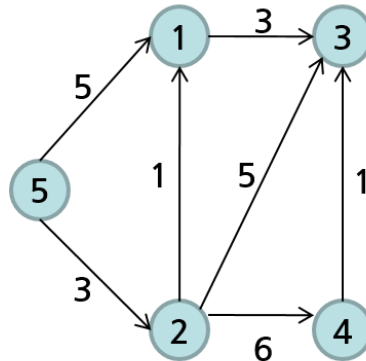
그래프(Graph) 용어

그래프는 서로 관계가 있는 개체들과 그 관계를 그림으로 나타낸 것입니다. 이때 개체를 정점(Vertex) 그 관계를 간선(Edge)으로 나타냅니다.

아래의 그림은 그래프의 예입니다.



[비가중치 무방향 그래프]



[가중치 방향 그래프]

왼쪽의 그래프는 정점이 7개이며, 간선이 8개인 그래프로 간선에 화살표 표시가 없으므로, 양방향으로 연결된 상태입니다. 1→2와 2→1이 서로 연결된 구조이며, 어떻게 이동을 하더라도 동일한 비용이 요구되는 비가중치 그래프입니다. 비가중치 그래프인 경우는 간선 위에 아무런 표시가 되어 있지 않습니다.

오른쪽의 그래프는 정점이 5개이며, 간선이 7개인 그래프로 간선에 화살표 표시가 있기 때문에 방향성을 갖고 연결된 상태입니다. 이를 방향 그래프라고 합니다. 그리고 간선 위에 숫자가 써있는데 이것은 가중치(Weight)를 의미합니다. 가중치란 정점에서 다른 연결된 정점으로 이동 시의 비용으로 해석할 수 있으며, 시간, 노력 등을 의미합니다.

정리해 보면 왼쪽의 그래프는 “비가중치 무방향 그래프”, 오른쪽은 “가중치 방향 그래프”가 되며, “비가중치 방향 그래프”와 “가중치 무방향 그래프”도 있을 수 있습니다.

참고로 모든 정점이 서로 서로 모두 연결되어 있는 그래프를 “완전 연결 그래프”라고 합니다.

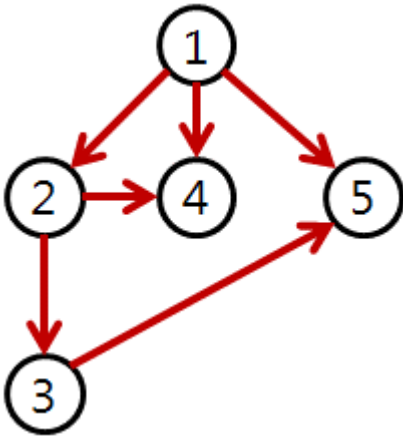
그래프 데이터 저장에 대한 이해

그래프의 저장에는 주로 2차원 배열 또는 Linked List를 사용하게 되는데, 알고리즘에서는 2차원 배열을 주로 사용합니다.

다음은 [보너스]라는 문제에서 사용되는 그래프로 정점이 5개인 방향성 그래프입니다. 그래프를 위한 입력 데이터가 그림이 아닌 정점의 개수, 간선의 수, 간선의 목록으로 주어졌습니다.

이때는, 정점의 개수만큼의 행, 열을 가진 2차원 배열을 준비하고, 행을 “출발지”, 열을 “도착지”로 하여 배열에 두 정점에 서로 관계가 있음을 표시합니다. 반대로 행을 “도착지”, 열을 “출발지”로 하여 사용할 수도 있지만, 대부분의 경우 행 번호를 “출발지”, “상위”와 같은 의미로 사용합니다.

입력 데이터를 이용한 그래프의 저장 배열을 보면 색으로 그 위치를 표시해 두었습니다. 각 간선의 위치를 배열에서 찾아 보시기 바랍니다.



[그래프]

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

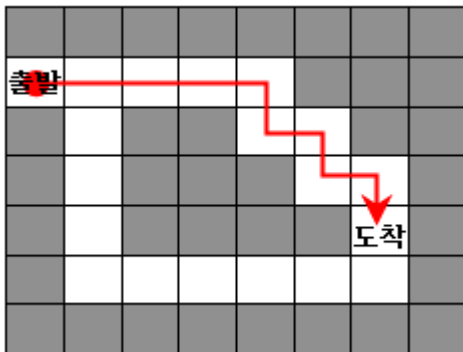
[입력 데이터]

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

[그래프 저장 배열]

다음은 그래프의 또 다른 형태로 표 형식의 그림으로 그래프가 주어진 것입니다. [미로 탈출 로봇]이라는 문제의 그래프인데 이때는 각 셀(Cell)이 정점이 되며 흰색으로 표시된 이동할 수 있는 경로(상, 하, 좌, 우)가 간선이 됩니다. 아래 그림에서 정점이 64개(행의 수 x 열의 수, 7x8=64개)이며, 정점의 이름이 좌표로 주어지게 됩니다.

출발 정점은 (2,1)이 되며, 도착 정점은 (5,7)이 됩니다. 그리고, 출발 정점과 연결된 정점이 (2,2)입니다. 정점 (2,2)는 (2,1), (2,3), (3,2)의 3개 정점과 간선을 가지고 있습니다.



[그래프]

	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1
2	0	0	0	0	0	1	1	1
3	1	0	1	1	0	0	1	1
4	1	0	1	1	1	0	0	1
5	1	0	1	1	1	1	0	1
6	1	0	0	0	0	0	0	1
7	1	1	1	1	1	1	1	1

[입력데이터 == 그래프 저장 배열]

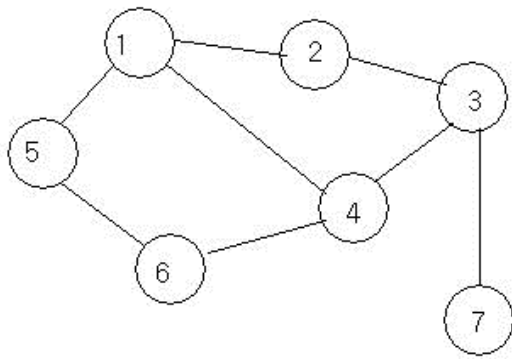
위에서는 연결된 간선이 0으로 표시되어 있지만, 입력데이터에 따라 연결된 간선의 표현이 다를 수 있습니다.

다만, 간선의 정보가 입력된 경우, 반드시 2차원 배열에 연결상태를 저장해야만 하는 것은 아닙니다. 대부분의 경우는 2차원 배열에 각 정점들 간의 연결 정보 표시하여 사용하면 좀 더 빠르고 편하게 연결 정보에 접근할 수 있지만, 문제에 따라, 입력되는 간선의 정보를 그대로 이용해야 하는 경우도 있습니다.

예를 들어 "깊이우선탐색 I", "너비우선탐색 I"이라는 문항은 DFS, BFS의 알고리즘을 그대로 적용하여 풀이하는 문항입니다. "깊이우선탐색 I"은 하나의 정점이 다른 여러 정점과 연결되어 있는 경우 그 여러 개의 정점 중 방문하지 않은 정점 중 하나를 방문하는 방식이고, "너비우선탐색"은 하나의 정점과 다른 여러 정점이 연결되어 있을 때 그 여러 개의 정점을 동일 시점에 빙글 돌면서 모두 방문하는 방식으로 방문하게 됩니다.

다음 정점으로 이동하기 위해 연결된 여러 개의 정점 중 하나를 고르거나 빙글 돌 때 어떤 기준으로 고르거나 방향을 잡아야 하는 것일까?

“깊이우선탐색 1” 문항을 보면 “단, 한 정점에서 갈 수 있는 곳이 여러 곳일 먼저 입력된 정점부터 먼저 방문한다. FIFO 즉, 2 4, 2 3 순으로 입력되었다면 2번 정점에서 4번을 먼저 방문하고 나중에 3번을 방문하도록 작성해야 한다.”라고 되어 있습니다. 이때 2 4, 2 3은 입력 데이터로 제공된 간선의 정보 중 일부입니다. 또한 “주어지는 모든 그래프는 양방향 무가중치 그래프이다” 라는 조건이 붙어 있기 때문에 다음에 방문할 위치를 찾기 위해 간선 번호로 주어진 for문을 돌면서 두 숫자 중 왼쪽에 있는 경우와 오른쪽에 있는 경우에 대해 상대방 정점이 방문하지 않았다면 방문 표시를 합니다.



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

다음 “보물섬” 그림에서 정점과 간선은 무엇인가?

W	L	L	W	W	W	L
L	L	L	W	L	L	L
L	W	L	W	L	W	W
L	W	L	W	L	L	L
W	L	L	W	L	W	W

보물섬 지도는 왼쪽 그림과 같이 직사각형 모양이며 여러 칸으로 나뉘어져 있다. 각 칸은 육지(L)나 바다(W)로 표시되어 있다. 이 지도에서 이동은 상하좌우로 이웃한 육지로만 가능하며, 한 칸 이동하는데 한 시간이 걸린다.

이때, 정점 및 간선을 설명하시오.

Triangular matrix

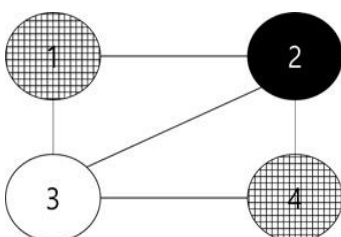
삼각형 모양으로 데이터가 제공되는 것으로 일부 필요한 데이터만 제공되는 경우, 또는 나머지 데이터를 채워 사용하게 되는 경우가 있습니다. 정점이 4개인 그래프의 데이터가 triangular matrix로 제공되는 예는 다음과 같습니다.

4 3
0
1 0
1 1 0
0 1 1 0

만일, 무방향 그래프에 대한 저장 데이터라면

필요한 경우 $a[i][j]$ 에 입력 받은 데이터를 $a[j][i] = a[i][j]$; 와 같이 복사해 사용해야 합니다.

문제풀이에 필요 없을 경우 복사 할 필요는 없지만, 복사해 두면 보다 유연하게 데이터에 접근할 수 있게 됩니다. -- “그래프 칠하기” 문항 참조



재귀 함수 호출을 이용한 다양한 경우의 수 작성

재귀 호출을 이용하여 경우의 수를 작성하기 전에 생각해야 할 것이 있습니다. 이것은 경우의 수에 대한 이해와 재귀호출을 왜 하는 것이며, 몇 번해야 할까에 대한 것입니다.

경우의 수를 사전에서 찾아보면 “어떤 일이 일어날 수 있는 경우의 가지 수”라고 되어 있습니다.

어떤 일을 문제 해결을 하는 것으로 바꾸어 작성해 보면 “문제를 해결하기 위해서 시도해 볼 수 있는 방법의 나열”이라고 생각할 수 있습니다. 즉, 여러분은 주어진 문제를 해결하기 위해 다양한 시도를 하게 되며, 그 시도에 대해서 어떻게 해보아야 하는지 고민하여 경우의 수 나열을 할 수 있어야 그 경우의 수를 만들기 위한 재귀 함수를 이용한 함수를 작성할 수 있습니다.

즉, 먼저 경우의 수를 나열하여 본 뒤 그 경우의 수 작성을 하기 위한 함수를 재귀 함수로 작성하는 것입니다.

경우의 수를 나열할 때에는 프로그램에서 사용해야 하므로 주로 숫자를 사용하여 표현합니다. 어떤 데이터가 있을 때 사용할지 말지에 대한 것이라면 0, 1 두 개의 숫자를 사용하고, 어떤 데이터를 사용할지에 대한 것이라면 1 ~ N (데이터 개수)의 숫자를 사용하여 표현하면 됩니다. 숫자로 표현하기 힘든 경우는 문자(알파벳)를 이용하여 표현할 수도 있습니다.

이해를 위해 디지털 회로와 같은 과정에서 사용하는 AND, OR 등의 Gate를 생각해 봅시다. 만일 AND Gate의 입력이 2가지 일 때 출력에 대한 진리 표를 만든다면 다음과 같이 작성이 될 것입니다.

입력		출력
A	B	
0	0	0
0	1	0
1	0	0
1	1	1

<AND Gate>

입력		출력
A	B	
0	0	0
0	1	1
1	0	1
1	1	1

<OR Gate>

입력		출력
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

<XOR Gate>

위의 진리 표에서 입력은 A, B 두 개이며 출력은 하나입니다.

디지털 회로에서는 입력 A와 B 및 출력이 전기가 흐름/흐르지 않음의 두 가지 상태이기 때문에 숫자 0, 1로 표현한 것입니다. 앞으로 우리도 어떤 상태의 종류에 따라 숫자의 사용 범위를 잘 정해야 합니다.

이것을 문제 해결과 연관하여 표현해 보면 두 개의 입력을 이용해 만든 4가지 경우의 수는 문제 해결을 위해 시도할 수 있는 방법이며, 출력은 해당 시도에 대한 결과값입니다. 여기에서 살펴 볼 것은 동일한 경우의 수라도 문제가 어떤 동작을 요구하는지에 따라 출력이 달라지게 된다는 것입니다. 입력이 모두 발생하는 경우 출력을 1로 한다는 것이 AND, 하나의 입력이라도 있으면 출력을 1로 한다는 것인 OR, 서로 다른 입력이 있는 경우 출력을 1로 한다는 것인 XOR입니다.

그리고, 최종적으로 그러한 결과 중에 원하는 것을 골라내는 것이 문제에서 원하는 답이 됩니다. 다음의 문제에 대한 답을 적어 봅시다.

- 1) AND Gate에서 입력이 2개인 경우 출력이 1인 것의 개수는?
- 2) OR Gate에서 입력이 2개인 경우 출력이 1인 것의 개수는?

3) XOR Gate에서 입력이 2개인 경우 출력이 1인 것의 개수는?

위의 문제에서 Gate 종류가 다르기 때문에 같은 질문인데도 답이 다릅니다.

알고리즘 문제에서도 동일한 경우에 수라도 서로 다른 결과값을 만들게 되는 다양한 조건이 있습니다. 시간, 이억 등의 합을 구하거나 데이터의 곱, 차이 값, 특정 상황을 만난 횟수를 세는 것을 이용하여 결과를 구하게 되는 것을 조건으로 생각하면 됩니다. 그 결과 중 최소값, 최대값, 횟수 세기 등 원하는 단 하나의 결과를 찾아 출력하는 것이 요구됩니다.

문제에서 요구되는 경우의 수를 직접 나열할 수 있어야 하며, 재귀함수를 이용하여 그 경우의 수를 작성하면 어떤 순서로 해당 경우의 수가 발생할지 미리 예측할 수 있어야 합니다. 그리고 처음 재귀 함수를 이용한 풀이를 학습할 때는 for문을 이용하여 어떻게 만들 수 있을까에 대해 생각해 보면 재귀로 작성할 때 도움이 될 수 있습니다. 어느 정도 연습 후에는 for문으로 어떻게 만들지 생각하지 않아도 자연스럽게 재귀로 작성하실 수 있을 것입니다.

예를 들어 설명하면 다음과 같습니다.

두 개의 주사위를 얻을 수 있는 두 수의 합의 빈도가 가장 높은 합의 수를 구하는 문제가 있다고 할 때, 두 개의 주사위를 던져서 얻을 수 있는 두 수를 경우의 수라고 할 수 있습니다. 경우의 수를 나열하여 보면 (1,2) (1,3) (1,4) ... (6,6) 과 같습니다. 일반적으로 어떤 사건의 주체가 되는 것의 개수만큼 for문이 반복되고 그 주체가 갖는 상태 변화를 반복횟수로 사용하게 되므로 두 개의 주사위 던지기에서는 주사위가 사건의 주체가 되고, 주사위를 던져서 나오는 값의 종류가 반복횟수로 사용됩니다. 여기에 주사위가 두 개이므로 for문이 중첩되어 사용됩니다.

즉, 첫 번째 주사위를 바깥쪽 for문 두 번째 for문을 안쪽 for문으로 하여 두 개의 for문을 이용하여 모든 경우의 수를 만들 수 있습니다. 첫 번째 주사위 1에 두 번째 주사위 1~6, 첫 번째 주사위 2에 두 번째 주사위 1~6 이 발생하는 순서로 경우의 수를 만들게 됩니다.

주사위의 개수가 정해진 경우는 for문을 그 주사위 개수만큼 사용하면 되지만, 주사위의 개수가 정해지지 않고 N 개로 입력되어 변경되는 경우는 재귀함수로 작성하여 경우의 수를 만들 수 있습니다.

for문을 이용하여 작성된 경우의 수를 재귀함수를 이용하여 동작하도록 만들 때는 for문의 개수가 depth, for문의 반복횟수가 재귀함수 호출횟수가 됩니다. 2개의 1~6 범위의 숫자를 눈으로 갖는 주사위의 경우 다음과 같이 재귀함수로 작성될 수 있습니다.

>> 코드

이번에는 재귀 호출을 이용해 경우의 수를 만들 때 “재귀 호출”의 의미를 살펴 보겠습니다. 재귀 호출을 하는 것은 어떤 문제 해결을 위해 시도를 한다는 것을 의미하고, 재귀 호출을 여러 번 하는 것은 그 문제 해결을 위해 시도해 볼 수 있는 방법이 여러 번 있다는 것을 의미합니다.

즉, 어떤 문제를 해결하기 위해 시도할 수 있는 3가지 방법이 있다면 재귀 호출을 3번 하면 되는 것이고, 4가지 방법이 있다면 재귀 호출을 2번하면 되는 것입니다.

문제에서 그래프의 이용

BFS나 DFS 문제에서 이러한 그래프는 사용될 수도 있고, 사용되지 않을 수도 있습니다. 그래프를 이용하여 데이터를 직접적으로 이동해가며 답을 찾을 경우도 있지만, 데이터가 어떤 답을 찾는데 보조 자료로 사용되는 경우도 있습니다. 많은 경우 BFS는 그래프를 직접 이동해 가면서 답을 찾지만, DFS의 경우는 그래프를 보조 자료로 사용하거나 사용하지 않는 경우도 많습니다.

예를 들어 보너스(직원들에게 보너스를 부여하는 문항)문항의 경우는 그래프가 직원들간의 상하 관계를 나타내며, 이 그래프를 순회(Traversal) 하는 것이 아니라 어떤 직원에게 어떤 보너스를 줄 수 있는지 없는지 판단용으로 그래프를 사용하는 것입니다. 또한, 그래프 칠하기(각 Node에 색을 부여하되 인접한 Node끼리 같은 색을 부여하지 않도록 하는 문항)에서는 어떤 Node에 특정 색을 부여할 수 있는지 없는지 판단용으로 그래프를 사용하는 것입니다.

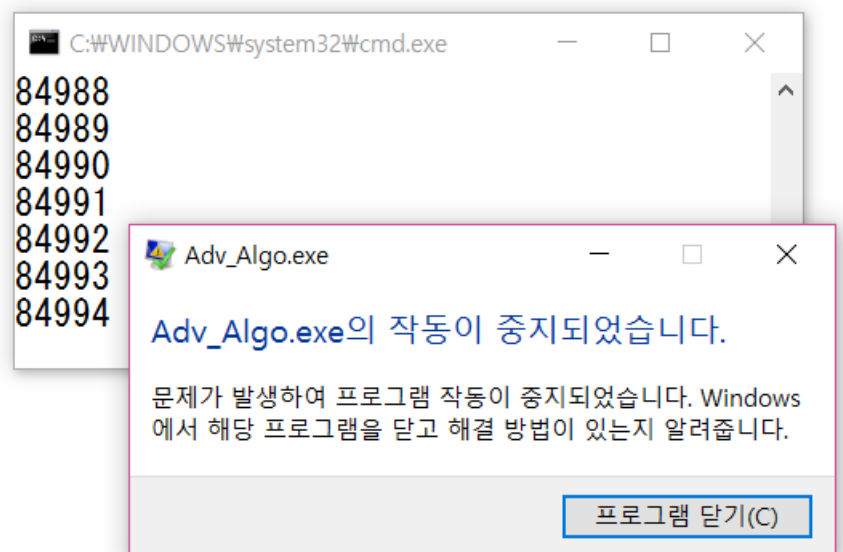
DFS 작성을 위한 재귀 함수의 이해

DFS 알고리즘을 이용할 때 Stack 또는 재귀 함수를 이용하여 구현할 수 있습니다. 본 강좌에서는 구현을 편하고, 이해하기 쉽게 하기 위해 재귀 함수를 이용하는 방법을 사용합니다.

재귀 함수란 함수 내에서 자신을 다시 호출하는 것으로 실행 속도가 빠르다거나 메모리를 절약하는 방법은 아닙니다. 잘못 사용하면 Stack Overflow가 발생할 수 있는 방법입니다. 이에 정확한 재귀 함수의 특징을 알고 적절하게 사용해야 합니다.

먼저, 재귀 함수의 형태를 살펴보겠습니다. 아래의 코드를 작성하여 실행하여 보면, 숫자가 한 줄에 하나씩 증가되면서 출력되고, 시간이 지나면 Run Time Error가 발생하는 것을 볼 수 있습니다. 이것은 함수 호출이 Stack을 사용하며, 재귀 함수의 호출이 멈추지 않아 Stack이 부족해졌기 때문에 발생한 것입니다.

```
1  #include <stdio.h>
2  int cnt;
3  void test( )
4  {
5      printf("%d\n", ++cnt);
6      test();
7  }
8  int main(void)
9  {
10     test();
11     return 0;
12 }
```



왜 계속 실행 되었을까요?

함수는 return문이나 함수의 닫기 괄호 '}'를 만나면 종료 됩니다.

그러나, 예시에서 작성된 test 함수는 닫기 괄호 '}' 이전에 test 함수 호출 문이 있어 닫기 괄호를 만날 수 없기 때문에 종료되지 않습니다.

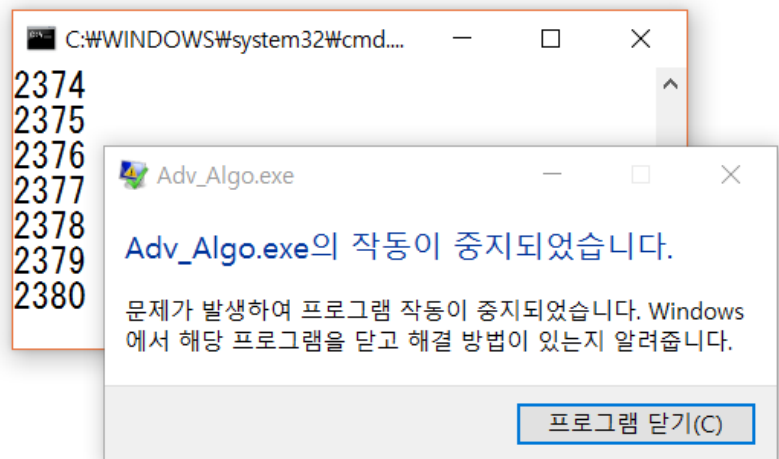
Stack 메모리 크기 설정이 다를 수 있기 때문에 예제를 실행 결과에 표시되는 숫자를 다를 수 있습니다. 기본 설정은 대부분 1MB 입니다. 설정에 대한 부분은 다른 파일을 참조하세요.

다른 특성을 하나 더 살펴 보겠습니다.

함수의 parameter 및 지역 변수 선언도 Stack을 사용하기 때문에 배열을 재귀 함수 내부에 선언하거나, 너무 많은 지역 변수를 선언하는 것은 좋지 않습니다.

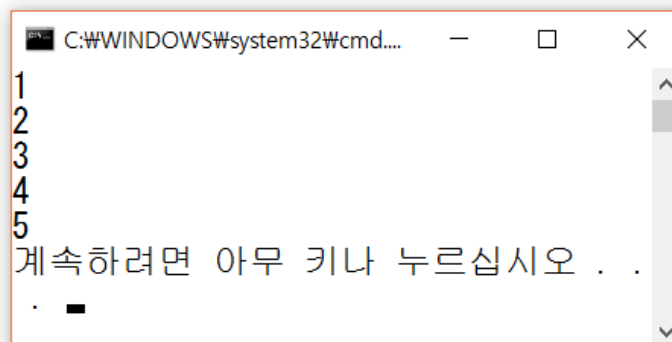
앞의 예제에서 test 함수 내부에 배열을 하나 선언하고 실행하면 실행 횟수가 많이 줄은 것을 볼 수 있습니다.

```
1 #include <stdio.h>
2 int cnt;
3 void test( )
4 {
5     int a[100]={0};
6     printf("%d\n", ++cnt);
7     test( );
8 }
9 int main(void)
10 {
11     test();
12     return 0;
13 }
```



재귀 함수 내부의 코드 작성은 되도록 간단하고 명쾌하게 작성하도록 하며, 불필요한 지역 변수의 사용은 자제해야 합니다. 이번에는 정상적으로 종료하는 재귀 함수를 작성해 보겠습니다.

```
1 #include <stdio.h>
2 int N = 5;
3 void test(int L)
4 {
5     if (L>N) return;
6     printf("%d\n", L);
7     test(L+1);
8 }
9 int main(void)
10 {
11     test(1);
12     return 0;
13 }
```



위의 예제에서는 인수로 받은 숫자를 출력한 뒤에 + 1 하여 재귀 함수를 호출하고 있으며, 인수로 받은 숫자가

N의 값보다 크면 종료합니다. 실행 화면을 보면 N이 5이기 때문에 1부터 5까지만 출력된 것을 볼 수 있습니다.

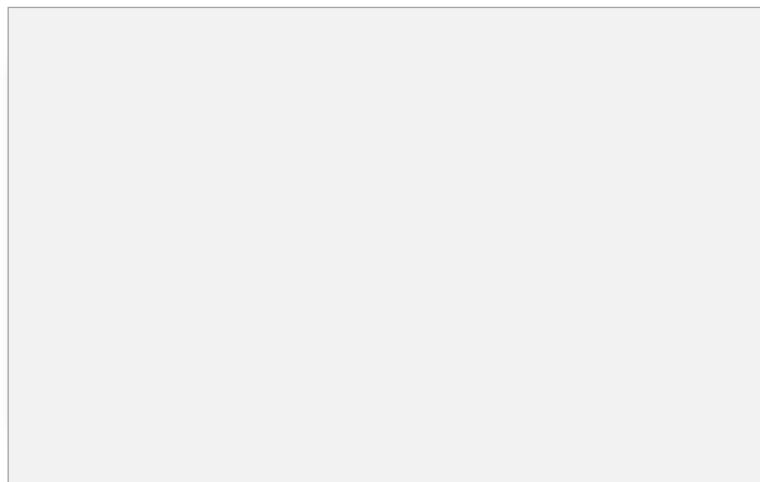
종료 조건은 재귀 함수 호출 전에 기술해야 하며, 가장 위쪽에 위치하는 것이 분석 시 편하기 때문에 가장 먼저 종료 조건을 기술하고, 처리 내용을 다음에 기술하였습니다. 만일, 종료 조건을 출력 문 뒤로 옮기면 1 ~ 5를 출력하기 위해서는 조건이 $L \geq N$ 으로 수정되어야 합니다. 숫자를 출력한 뒤에 조건을 검사하기 때문입니다. $L > N$ 을 사용하면 1 ~ 6이 출력됩니다.

```
void test(int L)
{
    printf("%d\n", L);
    if (L >= N) return;
    test(L+1);
}
```

앞으로 재귀 함수를 이용하여 DFS를 작성할 경우 종료 조건을 위쪽에 재귀 호출을 아래쪽에 작성할 것입니다. DFS는 대부분 검색을 위한 것이므로 데이터 검색에 대한 부분을 작성해야 하는데 이것은 종료 조건의 앞 또는 뒤쪽에 작성할 것입니다. 위의 test 함수 구조를 반드시 기억해 두시기 바랍니다.

다음 test 함수는 어떻게 동작할지 그 결과를 예측해 본 뒤, 실행하여 결과를 확인하고 동작을 이해하여 보세요.

```
2 int N = 5;
3 void test(int L)
4 {
5     if (L > N) return;
6     printf("%d ", L);
7     test(L+1);
8     printf("%d ", L);
9 }
10
11 int main(void) {
12     test(1);
13     return 0;
14 }
```

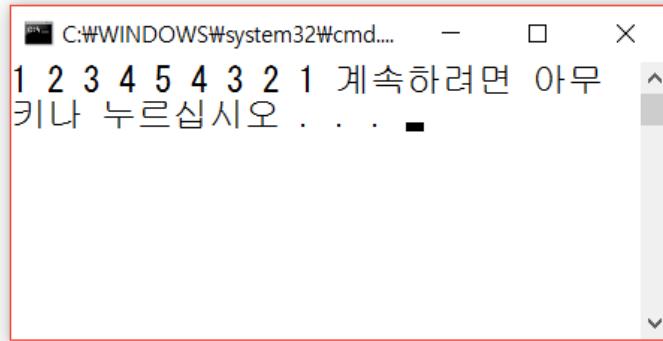


위의 코드를 다음 결과 화면과 같이 한 줄에 출력되며, '5'가 한 번만 출력되도록 코드를 수정해 보시기 바랍니다.


```

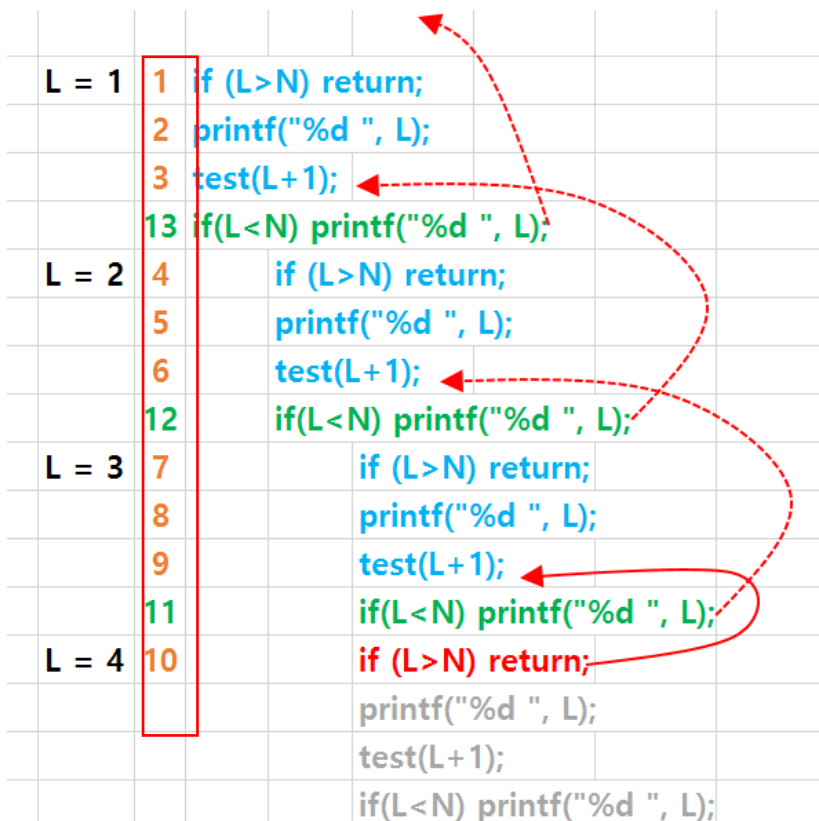
1  #include <stdio.h>
2  int N = 5;
3  void test(int L)
4  {
5
6
7
8
9  }
10 int main(void)
11 {
12     test(1);
13     return 0;
14 }

```



TIP) L과 N 값이 같을 경우 두 개의 printf 문 중에서 하나를 실행하지 않도록 조건을 사용하시면 됩니다.

위의 질문에 대한 답은 아래와 같으며, N이 3인 경우 test 함수의 실행을 순서대로 적어 보면 다음과 같습니다. 그림에 숫자를 이용하여 실행 순서를 적어 두었으며, 하늘색 - 빨강색 - 초록색 코드 순서로 실행됩니다.



[문제] 다음과 같이 출력되도록 위의 test함수를 수정하세요. main에서 test함수를 2번 호출하는 것이 아닙니다.

```

2 int N = 5;
3 void test(int L) {
4
5
6
7
8 }

```

[문제] 재귀 함수를 사용하여 다음과 같이 동작하는 test함수를 작성하세요. main 함수는 수정하지 않습니다.

```

2 int N = 5;
3 void test(int L) {
4
5
6
7
8
9
10
11 }

```

이제, 재귀 호출을 이용하여 다양한 경우의 수를 만드는 다양한 예를 만들고 분석해 보도록 하겠습니다.

원하는 값을 구하기 위해 여러 개의 데이터가 있을 때 각 데이터의 사용/미사용 선택, 사용 순서 유지/변경, 중복 사용/미사용 등에 대한 것을 바탕으로 다양한 문제 해결을 위한 방법의 경우의 수를 만드는 것입니다.

1. N개 데이터에서 R(0~N의 수)개 데이터 선택 사용, 순서 유지, 중복 미사용 (조합)

결과를 얻기 위해 어떤 데이터를 사용한다 사용하지 않는다는 두 번의 시도를 할 수 있다면, 각 데이터의 사용/미사용에 대한 것을 재귀로 두 번 호출합니다. 이때, 데이터의 사용 순서에 따라 결과가 달라지지 않는다면 데이터를 사용 순서를 앞에서 뒤로 유지하면서 사용하도록 제한하면 되기 때문에 중복 사용에 대한 확인을 별도로 할 필요가 없습니다.

아래의 경우의 수를 이용할 때 0을 사용, 1을 미사용으로 보면, 1번은 3개의 데이터를 모두 사용, 2번은 첫 번째, 두 번째 데이터의 사용, 3번은 첫 번째, 세 번째 데이터의 사용으로 해석할 수 있습니다.

```

C:\WINDOWS\system32\cmd...
1 : 0 0 0
2 : 0 0 1
3 : 0 1 0
4 : 0 1 1
5 : 1 0 0
6 : 1 0 1
7 : 1 1 0
8 : 1 1 1
계속하려면 아무 키나 누르십시오 . . .

```

0을 사용, 1을 미사용으로 볼 때 경우의 수의 분석

데이터를 사용하지 않는 경우: 111

한 개 데이터를 사용하는 경우: 011, 101, 110

두 개 데이터를 사용하는 경우: 001, 010, 100

모든 데이터를 사용하는 경우: 000

재귀 함수로 작성하기 전에 왜 재귀함수로 작성하는지에 대한 이해를 돕기 위해 for문을 이용하여 다음과 같은 경우의 수를 만들어 보도록 하겠습니다. 3개의 0, 1을 이용하여 만들 수 있는 8가지의 경우의 수를 for문을 이용하여 출력한 것입니다.

아래의 MakeCases 함수를 보면 세 개의 입력 i, j, k 각각은 0부터 2보다 작을 동안 1씩 증가하면서 동작하고 있습니다. 즉, 0과 1일 때 동자하므로 2번 동작합니다. 이 경우는 "문제 해결을 위해 시도할 수 있는 방법이 두 가지이다"라고 볼 수 있습니다.

```

11 void MakeCases(void) {
12     int i, j, k;
13     for(i=0; i<2; i++) {
14         for(j=0; j<2; j++) {
15             for(k=0; k<2; k++) {
16                 printf("%2d : %d %d %d\n", ++cnt, i, j, k);
17             }
18         }
19     }
20 }

```

만일, 필요한 상태만 증가한다면 변수를 사용하여 다음과 같이 수정할 수 있습니다. 세 개의 입력 i, j, k 각각이 1~N까지 1씩 증가하면서 동작하므로, 이 경우는 "문제 해결을 위해 시도할 수 있는 방법이 N가지이다"라고 볼 수 있습니다.

```

2 int N = 3;
3 void MakeCases(void) {
4     int i, j, k, cnt = 0;
5     for(i=1; i<=N; i++) {
6         for(j=1; j<=N; j++) {
7             for(k=1; k<=N; k++) {
8                 printf("%d : %d %d %d\n", ++cnt, i, j, k);
9             }
10        }
11    }
12 }

```

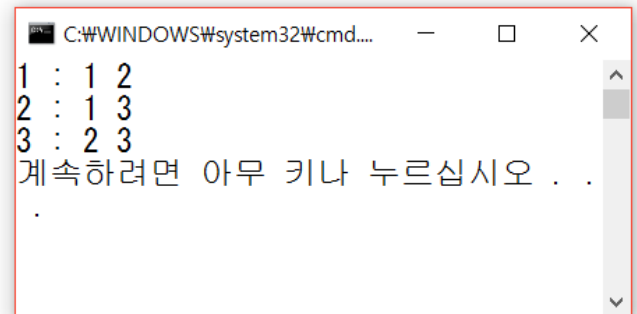
위와 같은 for문을 이용할 경우 구현은 단순하지만 시도할 수 있는 경우의 수가 아닌, **입력의 개수가 달라질 경우 for문의 중첩이 달라져야 하는 문제점**이 있습니다. 즉, 입력의 개수 == for문의 중첩수가 됩니다.

다음은 N개의 데이터에서 2개의 데이터를 뽑는 경우에 대한 함수입니다. (중복 없이, 순차적으로만 사용)

```

2 int N = 3;
3 void MakeCases(void) {
4     int i, j, cnt = 0;
5     for(i=1; i<=N-1; i++) {
6         for(j=i+1; j<=N; j++) {
7             printf("%d : %d %d\n", ++cnt, i, j);
8         }
9     }
10 }

```



```

C:\WINDOWS\system32\cmd...
1 : 1 2
2 : 1 3
3 : 2 3
계속하려면 아무 키나 누르십시오 . . .

```

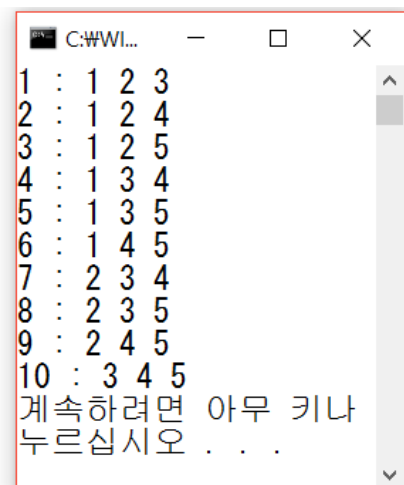
데이터의 개수가 바뀌더라도 "2개"라는 뽑는 데이터 개수가 정해져 있고 이것을 for문으로 작성했기 때문에 함수가 바뀌지 않습니다. N을 4로 변경하고, 2개를 뽑을 때의 경우의 수는 1 2, 1 3, 1 4, 2 3, 2 4, 3 4 의 6가지 입니다.

그러나 N개 데이터에서 3개의 데이터를 뽑아야 한다면 for문이 3중 중첩 되어야 합니다.

```

2 int N = 5;
3 void MakeCases(void) {
4     int i, j, k, cnt = 0;
5     for(k=1; k<=N-2; k++) {
6         for(i=k+1; i<=N-1; i++) {
7             for(j=i+1; j<=N; j++) {
8                 printf("%d : %d %d %d\n", ++cnt, k, i, j);
9             }
10        }
11    }
12 }

```



```

C:\W...
1 : 1 2 3
2 : 1 2 4
3 : 1 2 5
4 : 1 3 4
5 : 1 3 5
6 : 1 4 5
7 : 2 3 4
8 : 2 3 5
9 : 2 4 5
10 : 3 4 5
계속하려면 아무 키나 누르십시오 . . .

```

위와 같이 for문을 이용하여 경우의 수를 만들 경우 데이터의 개수의 변동 또는 사용할 데이터 수에 따라 for문의 증가 또는 감소 사용이 필요할 수 있습니다. 재귀 함수를 이용할 경우 이러한 문제를 해결할 수 있습니다.

재귀 함수를 작성하기 전에 DFS 알고리즘 구현 및 Debugging에 큰 도움을 줄 수 있는 배열 및 그 배열의 내용을 출력하는 함수를 살펴 보도록 하겠습니다. 재귀 함수 호출을 이용할 경우 Debugger를 이용한 Debugging이 힘들기 때문에 상태를 저장할 수 있는 도구가 필요합니다. 사용 상태 정보를 저장할 배열과 그 상태를 출력할 함수를 다음과 같이 작성합니다.

상태 정보가 사용/미사용과 같이 두 가지 상태인 경우 0을 사용, 1을 미사용으로 저장하고, 여러 가지의 상태 종류가 있다면 1부터 N까지의 숫자로 상태 정보를 저장하게 됩니다.

```

2  int list[10];
3  int cnt;
4  void printList(int L) {
5      int i;
6      printf("%2d : ", ++cnt);
7      for(i=1; i<L; i++)
8          printf("%d ", list[i]);
9      printf("\n");
10 }

```

list: 사용(1)/미사용(0)과 같은 문제해결을 위한 시도의
번호를 저장할 배열

cnt: 일련번호 역할을 할 정수형 변수

각 입력에 대해서 사용/미사용의 상태를 실행하는 MakeCases 함수와 동일한 동작을 하는 재귀 함수를 작성해 보도록 하겠습니다.

for문의 중첩 사용 횟수를 재귀 함수 호출의 연속 횟수로 보고 작성하고, for문의 반복변수의 범위를 재귀 함수 내에서 재귀 함수를 호출하는 횟수로 변경하면 됩니다.

```

11 void MakeCases(void) {
12     int i, j, k;
13     for(i=0; i<2; i++) {
14         for(j=0; j<2; j++) {
15             for(k=0; k<2; k++) {
16                 printf("%2d : %d %d %d\n", ++cnt, i, j, k);
17             }
18         }
19     }
20 }

```

재귀 함수 코드는 및 실행 결과는 다음과 같습니다.

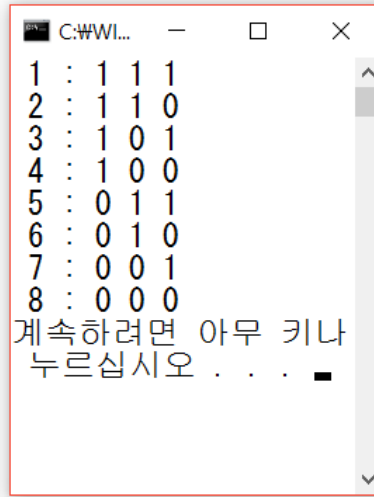
함수 이름을 DFS라고 하였으며, 인수는 재귀 함수의 동작을 멈추도록 하기 위해 사용하는 것으로 최대 깊이 (Depth)까지 재귀 함수 호출이 연속으로 이루어지면 멈추도록 하기 위해 사용합니다. 예를 들어 N개 숫자 데이터를 배열의 1~N에 입력 받아 각각의 숫자를 사용/미사용을 하면서 사용하는 경우에 대한 합을 구하는 경우를 가정하여 봅시다. 이 경우 인수 L을 각 숫자의 index로 사용하여 숫자에 접근하고, 1~ N까지가 사용하는 index 번호이고, N보다 크게 되면 사용을 할 수 없습니다. 이를 위해 main에서 L에 1을 넣어 호출하며, DFS 함수 내에서 $L > N$ 인 경우 연속 재귀 호출을 종료하여 1~N까지의 L을 사용하도록 합니다.

DFS 함수 내부에서 재귀 호출은 두 번하며, 한번은 데이터의 사용을 다른 한 번은 데이터의 미사용을 의미합니다. 이런 경우 2^N 가지 경우의 수가 만들어 집니다. 만일, N이 10이라면 1024개의 경우의 수가 N이 20이라면 2^{20} 개의 경우의 수가 만들어 집니다.

```

11 int N = 3;
12 void DFS(int L) {
13     if (L>N) {
14         printList(L); return;
15     }
16     list[L] = 1;
17     DFS(L+1);
18     list[L] = 0;
19     DFS(L+1);
20 }
21 int main(void) {
22     DFS(1);
23     return 0;
24 }

```



재귀 함수를 이용하여 작성한 위의 코드는 for문을 이용했을 때와 달리 N을 다른 숫자로 변경하여 추가 코드 작성 없이 N에 해당하는 경우의 수를 만들 수 있습니다.

[문제] 위의 DFS 함수를 이용하고 N=4인 경우 발생하게 되는 경우의 수 및 경우를 나열하여 이해하였는지 확인하시기 바랍니다. (코드를 실행하여 확인하지 말고 생각해서 답하시기 바랍니다.)

코드를 조금 수정하여 경우의 수 계산에 대해서 알아 보도록 하겠습니다.

```

11 int N = 3;
12 void DFS(int L) {
13     if (L>N) {
14         printList(L); return;
15     }
16     list[L] = 1;
17     DFS(L+1);
18     list[L] = 2;
19     DFS(L+1);
20     list[L] = 3;
21     DFS(L+1);
22 }
23 int main(void) {
24     DFS(1);
25     return 0;
26 }

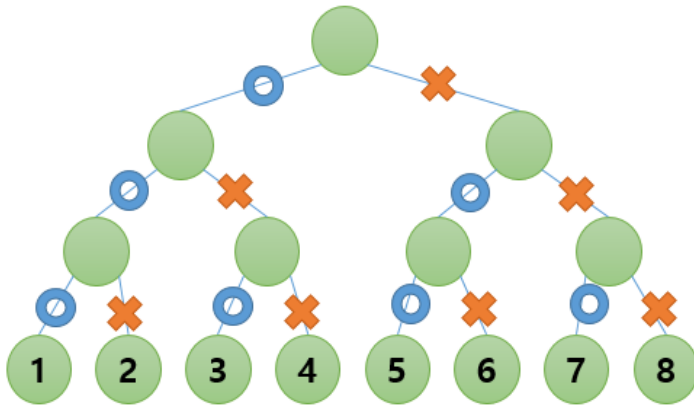
```

1 : 1 1 1	
2 : 1 1 2	15 : 2 2 3
3 : 1 1 3	16 : 2 3 1
4 : 1 2 1	17 : 2 3 2
5 : 1 2 2	18 : 2 3 3
6 : 1 2 3	19 : 3 1 1
7 : 1 3 1	20 : 3 1 2
8 : 1 3 2	21 : 3 1 3
9 : 1 3 3	22 : 3 2 1
10 : 2 1 1	23 : 3 2 2
11 : 2 1 2	24 : 3 2 3
12 : 2 1 3	25 : 3 3 1
13 : 2 2 1	26 : 3 3 2
14 : 2 2 2	27 : 3 3 3

함수 내에서 재귀 호출을 3번 수행한 결과 27가지 경우의 수가 만들어 지는 것을 볼 수 있습니다.

이것을 바탕으로 재귀 호출에 따른 경우의 수를 구하는 식을 도출해 보면 **(재귀호출 수)^N**로 볼 수 있습니다. 즉, 재귀 호출의 횟수가 늘어나거나 N의 수가 커지게 되면 경우의 수도 많이 증가하게 되어 불필요한 재귀 호출의 동작을 막는 것이 중요하며, 이것을 "가지치기"한다 라고 합니다.

Tree의 Level을 데이터 개수(N)로 생각하고, 자식 Node 개수를 재귀 함수 호출 개수로 보면 됩니다. 예를 들어 N=3 이고, 재귀 호출이 2번 있는 경우의 그림은 다음과 같습니다.



즉, N이 3인 경우 DFS(4)가 8번 호출되는
것이며 이것이 경우의 수가 됩니다.

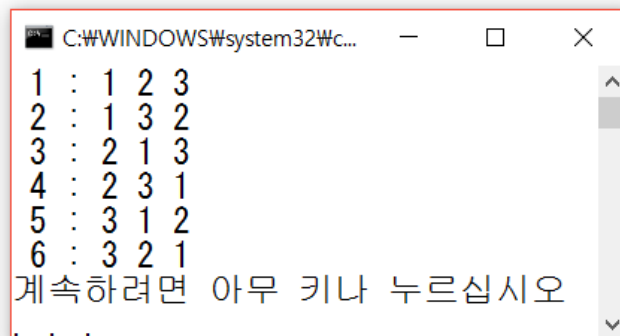
2. N개 데이터 모두 사용, 순서 변경, 중복 미사용 (순열)

예를 들어 N=3인 경우 아래와 같이 6가지 경우가 발생하며 visit 배열(방문 확인용 배열)을 사용하여 사용된 번호를 다시 사용되지 않도록 하고, L은 숫자의 index가 아니라 사용된 숫자 개수를 세는 용도로 사용합니다. N개의 숫자를 사용할 것이므로 몇 개의 숫자를 사용했는지 세어 나가다가 N개를 모두 사용했을 때 멈추기 위함입니다. 이때, 어떤 데이터를 사용 했는지는 앞서 준비한 list 배열에 저장합니다.

```

11 int N = 3;
12 int visit[10];
13 void DFS(int L) {
14     int i;
15     if (L>N) {
16         printList(L); return;
17     }
18     for(i=1;i<=N; i++) {
19         if (visit[i] == 0) {
20             visit[i] = 1; list[L] = i;
21             DFS(L+1);
22             visit[i] = 0; list[L] = 0;
23         }
24     }
25 }

```



코드를 보면 for문이 사용되었으며, 1부터 N까지 1씩 증가되며 사용되지 않은 번호를 찾아 사용을 표시($visit[i] = 1$) 하고 $DFS(L+1)$ 을 호출합니다. $DFS(L+1)$ 의 수행 뒤에는 사용 표시를 해지($visit[i] = 0$) 합니다.

1

2

3

L=1에서 for문이 1~3순서로 반복하기 때문에 1번을 사용하게 됩니다

1

2

3

L=2에서는 앞서 L=1에서 1번을 사용 중이기 때문에 2번을 사용하게 됩니다

1

2

3

L=3에서는 앞서 1, 2번을 사용 중이기 때문에 3번을 사용하게 됩니다

위의 경우가 가장 먼저 만들어지는 경우이며, 이후에 $L=4$ 에서 return 된 후, $L=3$ 은 $visit[3] = 0$ 을 수행하고, i 의 값이 N 보다 크게 되어 for문을 종료하고 $L=2$ 로 return 됩니다. $L=2$ 에서는 $visit[2] = 0$ 을 수행하고 3번을 사용하게 되며, $L=3$ 를 호출하고 $L=3$ 은 앞서 사용이 해제된 2번을 사용하게 됩니다. 이런 방식으로 총 6가지의 경우가 만들어 집니다. $N=4$ 인 경우 24가지의 조합이 발생되며, 이를 토대로 경우의 수에 대해 식으로 작성해 보면 **N!** 이 됩니다.

[문제] $N=4$ 인 경우 발생하는 24가지 조합을 나열하여 보시기 바랍니다. (코드를 실행하지 말고 실행을 손파일링 하시기 바랍니다.)

3. 일정 수치까지 중복하여 사용

일정 금액을 가지고 있고 그 금액에 맞추어 투자를 하거나 물건을 사는 등의 동작을 할 경우 “몇 개” 또는 다른 의미로 사용할 수 있도록 데이터를 만들어야 합니다. 이런 경우 아래와 같이 사용할 수 있는 비용에 해당되는 M 변수를 사용하고 main에서는 $DFS(1, M)$ 을 호출하고 각 Level에서 사용된 비용만큼 빼서 다음 Level로 보냅니다. 그러다가 남은 비용(R)이 0이되면 금액을 모두 사용한 것이므로 그 때 어떤 이익이나 다른 비용을 계산하여 원하는 결과를 찾습니다. 다음은 M 이 10인 경우의 코드와 실행결과 입니다. 실행결과에서 1번의 경우는 2를 두 번, 3을 두 번 사용하는 경우이고, 2번의 경우는 2를 5번 사용하는 경우입니다. 총 13가지 경우가 발생하는 것을 볼 수 있습니다.

```

12 int N = 3;
13 int M = 10;
14 void DFS(int L, int R) {
15     int i;
16     if (R == 0) {
17         printList(L, R); return;
18     }
19     if (L > N) return;
20     for(i=0; i<=R; i++) {
21         list[L] = i + '0';
22         DFS(L+1, R-(i*L));
23         list[L] = 0;
24     }
25 }

```

```

C:\WINDOWS...
1 ( 0) : 0 2 2
2 ( 0) : 0 5
3 ( 0) : 1 0 3
4 ( 0) : 1 3 1
5 ( 0) : 2 1 2
6 ( 0) : 2 4
7 ( 0) : 3 2 1
8 ( 0) : 4 0 2
9 ( 0) : 4 3
10 ( 0) : 5 1 1
11 ( 0) : 6 2
12 ( 0) : 7 0 1
13 ( 0) : 8 1
14 ( 0) :
계속하려면 아무 키나

```

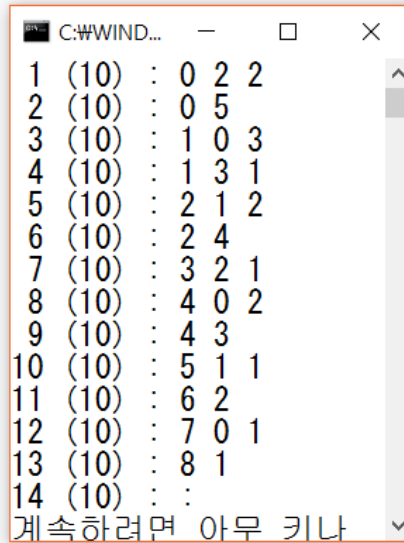
경우의 수는 사용하는 단위와 처음에 정해진 M 의 크기에 따라 달라집니다. 이때 사용되지 않는 것도 있고, 여러

개 사용되는 경우도 있으며, 한 가지 종류만 사용되거나 다양한 종류가 사용되는 경우도 있는 것을 잘 보아 주시기 바랍니다. 그리고 위의 코드에서는 R(남은 비용)을 계산하였지만, 반대로 사용된 비용이 M과 같은지 확인하는 방법으로 코드를 작성할 수도 있습니다. 다음의 코드와 결과를 확인해 보시기 바랍니다.

```

12 int N = 3;
13 int M = 10;
14 void DFS(int L, int U) {
15     int i;
16     if (U == M) {
17         printList(L, U); return;
18     }
19     if (L > N) return;
20     for(i=0; i<=M-U; i++) {
21         list[L] = i + '0';
22         DFS(L+1, U+(i*L));
23         list[L] = 0;
24     }
25 }

```



```

C:\WIND...
1 (10) : 0 2 2
2 (10) : 0 5
3 (10) : 1 0 3
4 (10) : 1 3 1
5 (10) : 2 1 2
6 (10) : 2 4
7 (10) : 3 2 1
8 (10) : 4 0 2
9 (10) : 4 3
10 (10) : 5 1 1
11 (10) : 6 2
12 (10) : 7 0 1
13 (10) : 8 1
14 (10) :
계속하려면 아무 키나

```

4. 데이터를 여러 번 반복하여 사용하는 경우

동일한 데이터를 여러 번 사용해야 하는 경우 해당 데이터를 얼마나 사용할 수 있는지에 대한 최소/최대 횟수 범위를 확인하여 이것을 반복문의 기준으로 사용합니다. 예를 들어 2, 3, 5를 중복으로 사용할 수 있는 상태에서 10을 만들 수 있는 경우의 수를 만드는 경우에 대한 프로그램과 그 결과입니다.

```

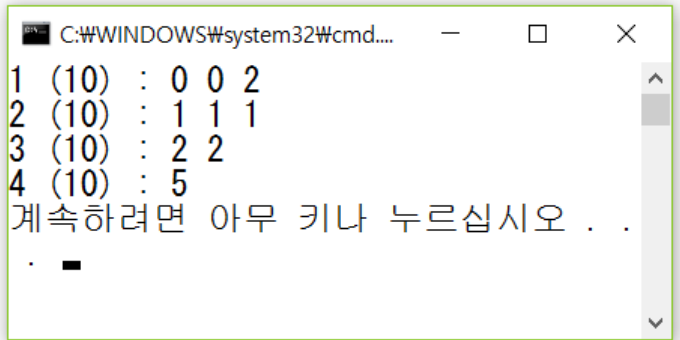
1 #include <stdio.h>
2 int a[] = {0, 2, 3, 5};
3 int M = 10;
4 int list[5];
5 int cnt;
6 void printList(int L, int R) {
7     int i;
8     printf("%d (%2d) : ", ++cnt, M-R);
9     for(i=1; i<L; i++) {
10         printf("%d ", list[i]);
11     }
12     printf("\n");
13 }

```

```

14 void DFS(int L, int R) {
15     int i;
16     if (R==0) {
17         printList(L, R);
18         return;
19     }
20     if(L>3 || a[L]>R || a[L]>M) return;
21
22     for(i=0; a[L]*i<=R; i++) {
23         list[L] = i;
24         DFS(L+1, R-a[L]*i);
25         list[L] = 0;
26     }
27 }
28 int main(void) {
29     DFS(1, M);
30     return 0;
31 }

```



```

C:\WINDOWS\system32\cmd...
1 (10) : 0 0 2
2 (10) : 1 1 1
3 (10) : 2 2
4 (10) : 5
계속하려면 아무 키나 누르십시오 . . .

```

이 외에도 아주 다양한 방법으로 경우의 수를 만들게 되며, 이것을 생각해 내는 것이 DFS를 활용하는 것입니다.

BFS(너비 우선 탐색) 문제 풀이 방법

Queue 만들기

```

struct st {
    int y; // 좌표의 행번호
    int x; // 좌표의 열번호
    int v; // 좌표의 값 // 한 정점을 여러 번 방문할 수 있는 경우 사용 금지!!
};

// Queue의 요소 개수는 정점의 최대수, 만일 여러 번 방문이라면 (정점의 최대수 x 시작점수)로 한다
struct st Queue[MAX*MAX];
int wp, rp; // wp : write 삽입 위치, rp : read 삭제(읽기) 위치

void In_Queue(int y, int x, int v) {
    Queue[wp].y = y;
    Queue[wp].x = x;
    Queue[wp].v = v;
    wp++;
}

struct st Out_Queue(void) {
    return Queue[rp++];
}

```

```
1. wp = rp = 0; // Queue 초기화 작업
```

방문표시 - 입력 배열을 직접 사용/수정, 입력 배열을 복사해서 사용/수정

입력 배열 및 별도의 값 배열(v)을 사용/수정 (2가지 사용) - 정점을 여러 번 방문 시

```
while(wp>rp) { }
```

3.1 Queue에서 내용을 꺼내 out 변수에 놓는다. // struct st out; out = Out_Queue();

3.2 out과 연결된 정점을 찾아 도착점인지 확인한다.

3.3 만일, 연결된 정점이 도착점이라면 원하는 값을 가지고 종료하고, 도착점이 아니라면 Queue에 넣고, 방문 표시 한다.

예1) 저글링 방사능 오염

입력	(1, 1)부터 사용, %1d -> int a[MAX][MAX], %s -> char a[MAX][MAX]
시작점	1개, (sy, sx 에 입력 받음)
도착점	1개지만, 어디일지는 모르는 상황, "더 이상 저글링을 오염 시킬 수 없을 때" <pre>while(wp>rp) { // 저글링 오염 시키는 부분 } // while문의 뒤 !요기! (도착점!!)</pre>
연결점	감염된 저글링의 4방향에 있는 저글링 a[ny][nx] == (1/'1') 을 만족하는 정점
방문표시	int 배열인 경우 - 입력 배열에 바로 초 대입 char 배열인 경우 - '1'이 아닌 다른 값으로 입력 배열을 바꾸고, Queue의 v 사용
Special	남은 저글링의 수를 출력해야 하므로, 입력 시 저글링의 수를 세어 두었다가, 오염시키면서 1개씩 감소 하는 방법을 사용함

장기판 배열 초기화 필요 (입력데이터 별도 없음)

입력	장기판 배열 초기화 필요 (입력데이터 별도 없음)
시작점	1개, (my, mx 에 입력 받음)
도착점	1개, (jy, jx에 입력 받음)
연결점	8방향 중 방문하지 않았던 좌표
방문표시	방문 표시를 하지 않을 경우 동일 좌표가 계속 Queue에 쌓이기 때문에 Queue의 크기를 초과 할 가능성 높음 따라서, 말이 방문했던 좌표에 대해 초기값과 다른 값으로 대입하여, 입력 배열을 다시 방문하지 않도록 함

예3) 토마토

토마토를 익혀 가는 방법이 날자 별로 동시에 익어야 함

예4) 보물섬

최단거리 이동, 돌아가지 않음

입력	어느 점을 시작점으로 사용해도 된다. char 배열
시작점	모든 'L'이 시작점이며, 동시에 동작하지 않음 (In_Queue를 동시에 하지 않음), BFS를 여러 번 호출 <pre> max = 0; for(i=1; i<=Y; i++) { for(j=1; j<=X; j++) { if (a[i][j] == 'L') { initVisit(); t = BFS(i, j); if (max<t) max = t; // 결과값 구하는 것 } } } </pre>
도착점	최장거리, while(wp>rp) { }을 끝냈을 때의 시간
연결점	out 좌표를 기준으로 이웃(상, 하, 좌, 우)에 있으면서, 'L' 땅인 것 아래의 방법은 동작을 하지만 비효율적이다. <pre> if (a[ny][nx] == 'L' && visit[ny][nx] == 0) // 연결점 조건 if (visit[ny][nx] == 0) // 으로만 사용할 것이다 visit 배열 초기화를 아래와 같이 한다 (BFS() 호출전에 실행) if (a[i][j] == 'W') visit[i][j] = 1; else visit[i][j] = 0; </pre>
방문표시	별도의 visit배열을 사용한다 visit 배열을 그냥 0, 1로만 사용하면 낭비다. 여기에 시간도 저장하자

예5) 자외선을 피해 가기

BFS? : N의 범위가 너무 커서, DFS로 풀 수 없음 (Depth가 100*100은 풀이 불가)

입력	(1, 1)부터 사용, %1d (int a[MAX][MAX]배열 사용)
시작점	1개, (1, 1), b[1][1] = a[1][1]; 방문표시
도착점	1개, (N, N) 도착점을 만나도 바로 종료하지 않음, while(wp>rp)를 종료 후, b[N][N] 사용
연결점	4방향 중 b[ny][nx] > (b[out.y][out.x] + a[ny][nx]) 을 만족하는 정점
방문표시	a배열(입력배열)에 방문표시를 하면 안됨 → 서로 다른 경로가 같은 정점을 사용할 수 있기 때문임 b배열을 사용함 (임의의 정점까지의 자외선 누적의 최소값을 저장하는 배열) b[ny][nx] > (b[out.y][out.x] + a[ny][nx]) 일때 In_Queue(ny, nx)하고 b[ny][nx]를 갱신 한다 → 자연스럽게 더 나쁜 경로(<=)는 '가지치기'가 된다 → struct st{ } 에 int v; 를 사용하면 안 된다 이전 값이 나쁘고 새로운 값이 좋을 때, 이전 값은 새로운 값으로 갱신 되어야 하기 때문 → 배열에는 최소값이 저장되어야 하기 때문에 모든 셀에 초기값을 0x7FFF0000 으로 함

	→ 한 정점이 여러 번 큐에 들어갈 수 있으므로 Queue의 크기를 크게 잡아야 함 * 100
Special	<p>b배열은 "1,1로부터 모든 다른 좌표까지의 최소 자외선의 누적량"의 의미를 갖는다 그래서, b배열은 초기값으로 "발생가능 최대값"을 넣어두고, 더 작은 값이 발생될 때 갱신한다 현재 값보다 더 크거나 같은값이 발생되면 연결점으로 사용하지 않아 그 길을 더 이상 가지 못하도록 한다 (가지치기) 배열은 DFS에서 인수 S의 개념이다. 예) if (S >= sol) return; // 가지치기 DFS(L+1, S+a[L]); // 누적</p>

DFS(깊이 우선 탐색)의 문제 풀이 순서

1. DFS 함수의 프로토타입을 작성한다

1.1 리턴형 (리턴에 내가 찾는 값을 넣어 반환하지 않음! 찾는 값을 넣는 변수는 전역 변수 사용할 것)

int: 성공(1), 실패(0)를 알리는 경우 사용 (실행 중 원하는 것을 찾을 수 있을 때)

void: 모든 경우의 수를 해봐야 하는 경우 사용

1.2 인수리스트 작성

반드시 종료를 위한 인수가 있어야 함 (int L)

이전 level까지의 어떤 상태 정보를 사용해야 할 경우 인수로 넘겨줌

2. 리턴형이 int 인 경우 DFS 함수의 마지막 줄에 return 0; 를 삽입한다 (return 0; 는 실패를 의미한다)

3. Depth까지만 수행되도록 최소한의 종료조건을 기술한다

if (L>N) return;

4. 재귀함수 호출문 및 결과값을 찾기 위한 검색 조건을 작성한다

대부분, 여러 번 호출하게 되며, 최소한의 호출문을 직접 여러 번 작성하거나, for 문을 이용한다

5. 가지치기 조건문을 작성한다

가능성이 없는 경우의 수는 중간에 작성을 멈춘다.

DFS 문제 설계 예시

