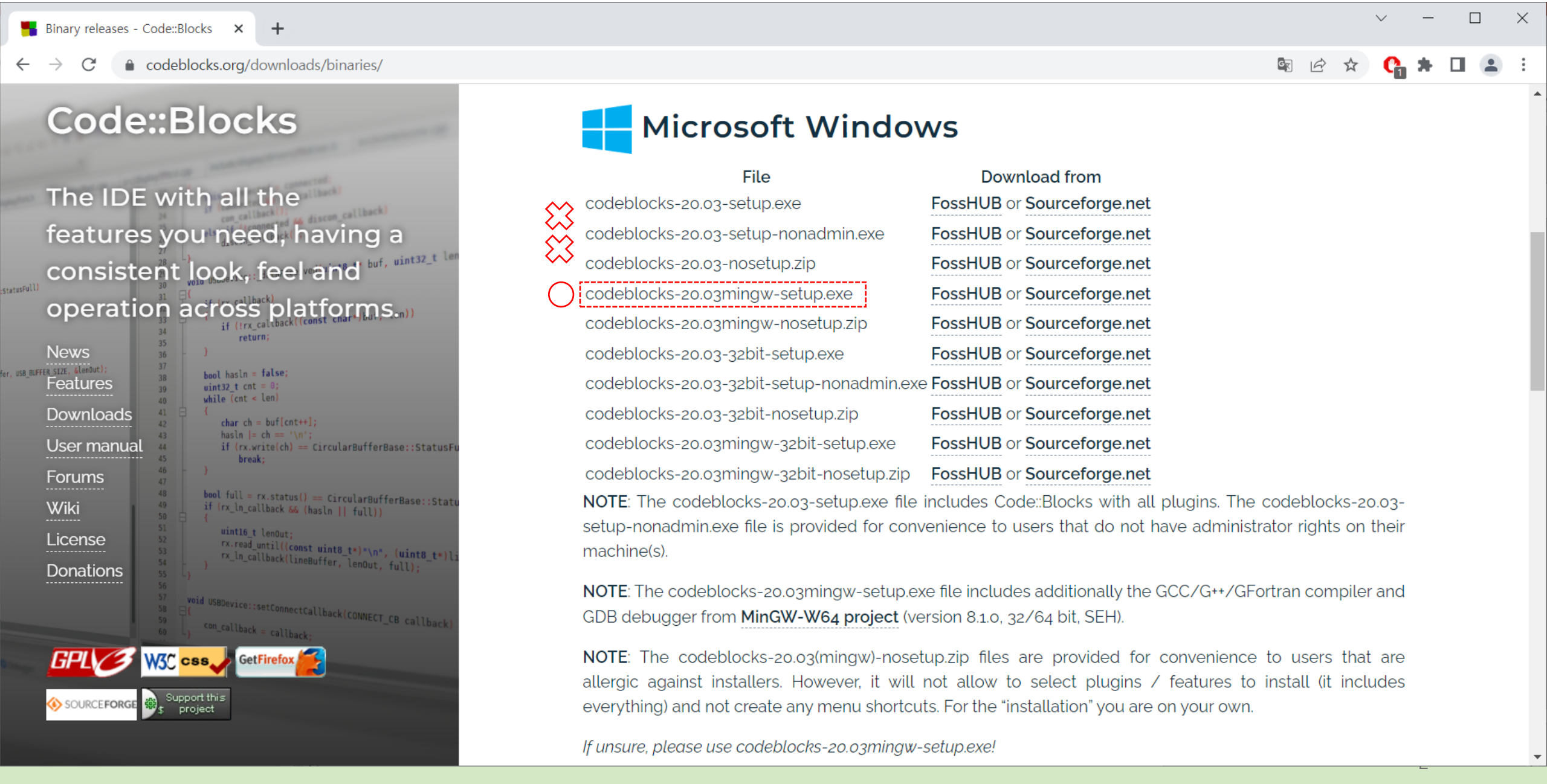
- 교육자료: https://datahub.pe.kr/
- 소스코드: https://algo.datahub.pe.kr/src/
- Smart OJ: https://oj.datahub.pe.kr/

정 보 과 학

< C언어 >

충북교육연구정보원 정보영재교육원 | SW·AI교실 | 정보아카데미 강사

흥덕고등학교 교사 박정진


Binary releases - Code::Blocks
codeblocks.org/downloads/binaries/
Code::Blocks
The IDE with all the features you need, having a consistent look, feel and operation across platforms.
News
Features
Downloads
User manual
Forums
Wiki
License
Donations
GPL v3
W3C CSS
GetFirefox
SOURCEFORGE
Support this project
Microsoft Windows
File
Download from
codeblocks-20.03-setup.exe FossHUB or Sourceforge.net
codeblocks-20.03-setup-nonadmin.exe FossHUB or Sourceforge.net
codeblocks-20.03-nosetup.zip FossHUB or Sourceforge.net
codeblocks-20.03mingw-setup.exe FossHUB or Sourceforge.net
codeblocks-20.03mingw-nosetup.zip FossHUB or Sourceforge.net
codeblocks-20.03-32bit-setup.exe FossHUB or Sourceforge.net
codeblocks-20.03-32bit-setup-nonadmin.exe FossHUB or Sourceforge.net
codeblocks-20.03-32bit-nosetup.zip FossHUB or Sourceforge.net
codeblocks-20.03mingw-32bit-setup.exe FossHUB or Sourceforge.net
codeblocks-20.03mingw-32bit-nosetup.zip FossHUB or Sourceforge.net
NOTE: The codeblocks-20.03-setup.exe file includes Code::Blocks with all plugins. The codeblocks-20.03-setup-nonadmin.exe file is provided for convenience to users that do not have administrator rights on their machine(s).
NOTE: The codeblocks-20.03mingw-setup.exe file includes additionally the GCC/G++/GFortran compiler and GDB debugger from MinGW-W64 project (version 8.1.0, 32/64 bit, SEH).
NOTE: The codeblocks-20.03(mingw)-nosetup.zip files are provided for convenience to users that are allergic against installers. However, it will not allow to select plugins / features to install (it includes everything) and not create any menu shortcuts. For the "installation" you are on your own.
If unsure, please use codeblocks-20.03mingw-setup.exe!

Start here - Code::Blocks 20.03

File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help

Management

Projects Files FSymbols

Workspace

Start here

Code::Blocks

The open source, cross-platform IDE

Release 20.03 rev 11983 (2020-03-12 18:24:30) gcc 8.1.0 Windows/unicode - 64 bit

Create a new project

Open an existing project

Tip of the Day

Visit the Code::Blocks forums

Report a bug or request a new feature

Recent projects

D:₩MyProjects₩Algorithm₩ShortestPathAll₩ShortestPathAll₩ShortestPathAll.cbp

D:₩MyProjects₩Test₩Test₩Test.cbp

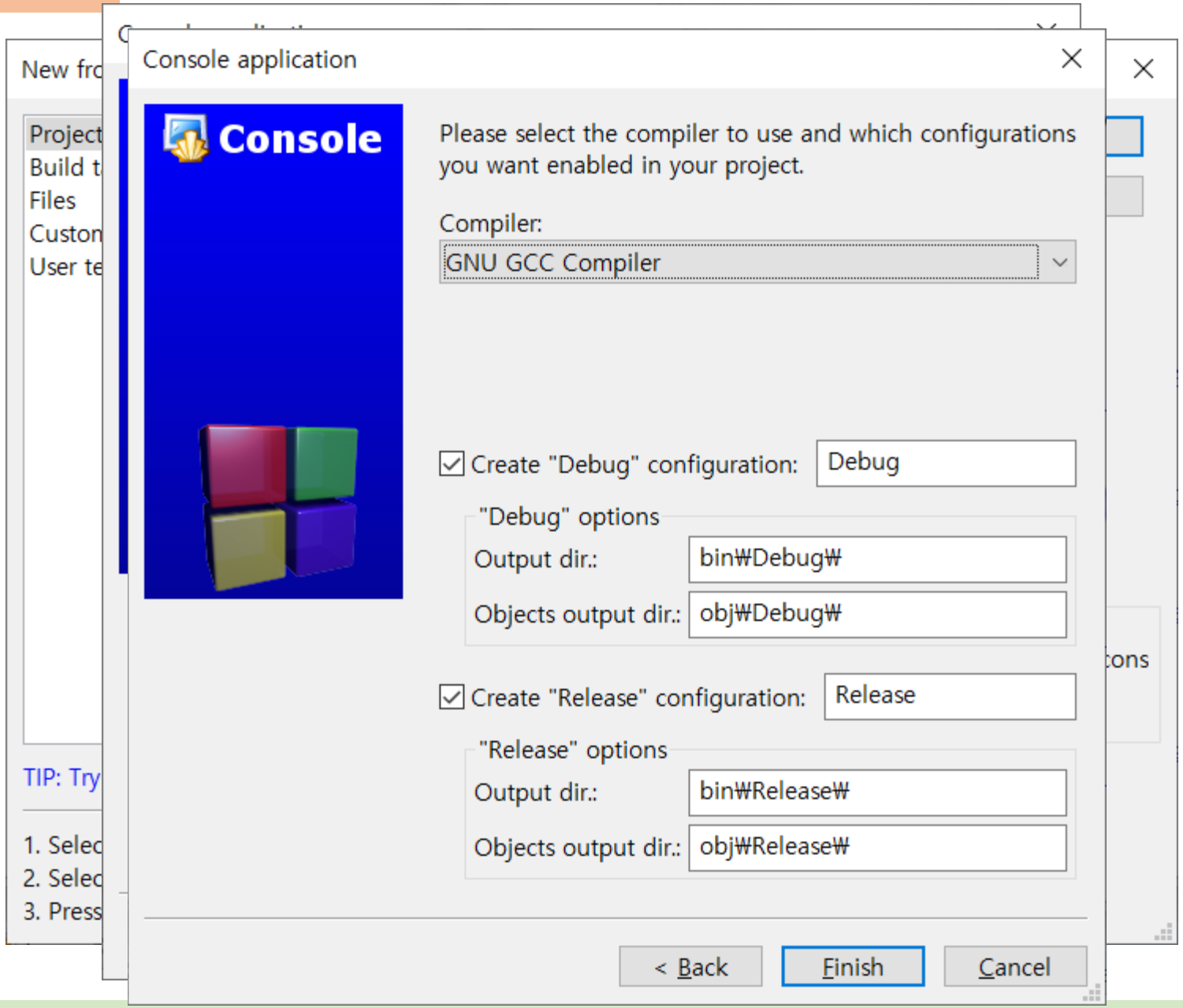
Recent files

Logs & others

Code::Blocks Search results Cccc Build log Build messages CppCheck/Vera++ CppCheck/Vera++ messages Cscope Debugger DoxyBlocks Fortran info

Start here

default


Console application
Console
Please select the compiler to use and which configurations you want enabled in your project.
Compiler:
GNU GCC Compiler
Create "Debug" configuration:
Debug
"Debug" options
Output dir.:
bin₩Debug₩
Objects output dir.:
obj₩Debug₩
Create "Release" configuration:
Release
"Release" options
Output dir.:
bin₩Release₩
Objects output dir.:
obj₩Release₩
< Back
Finish
Cancel

*main.cpp [HelloWorld] - Code::Blocks 20.03

File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help

Build & Run: F9

Management

Projects Files FSymbols

Workspace
HelloWorld
Sources
main.cpp

*main.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      cout << "Hello world!" << endl;
8      return 0;
9  }
10
```

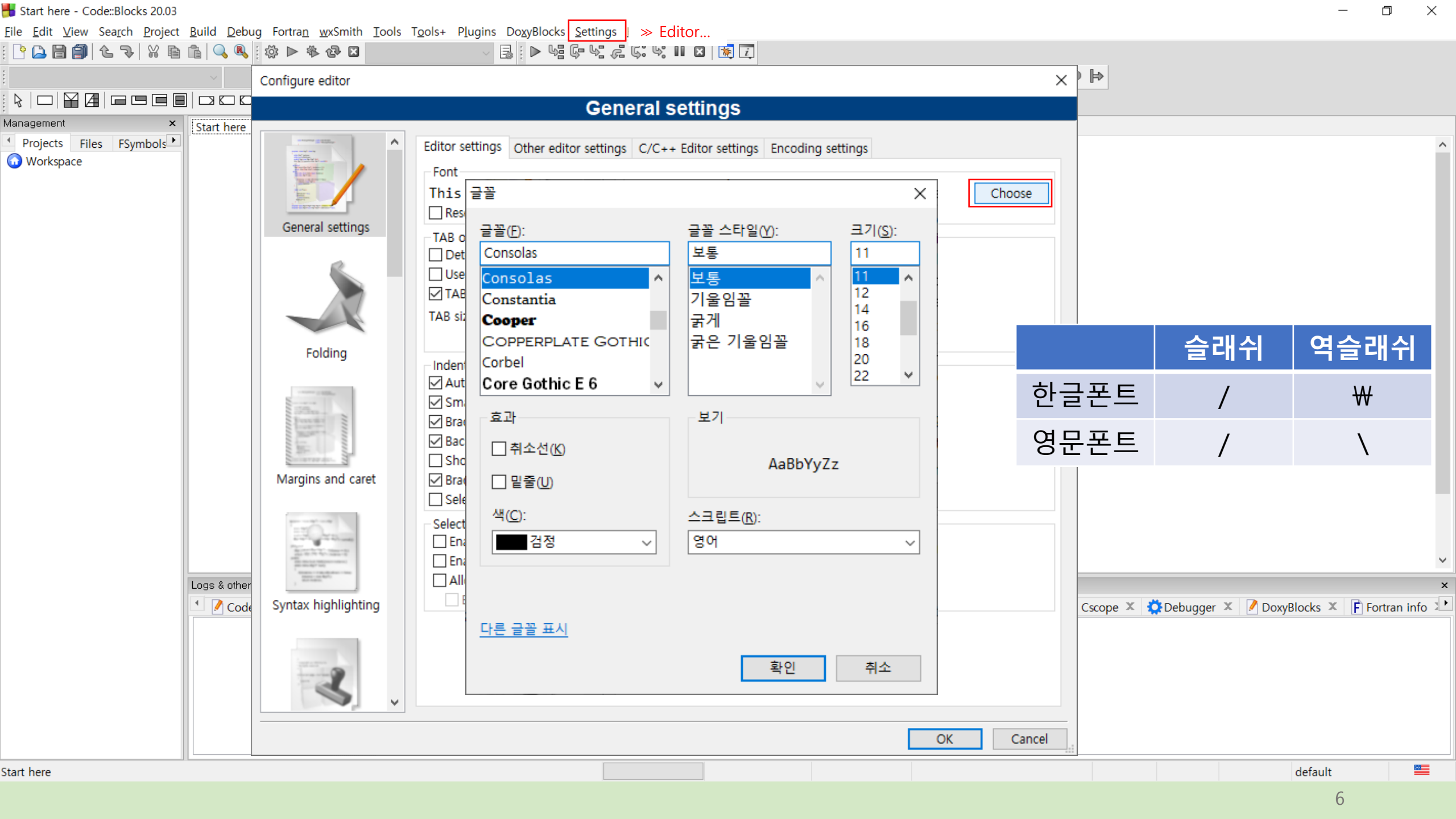
```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("Hello world! \n");
6      retrun 0;
7  }
```

	슬래쉬	역슬래쉬
한글폰트	/	₩
영문폰트	/	\

Logs & others

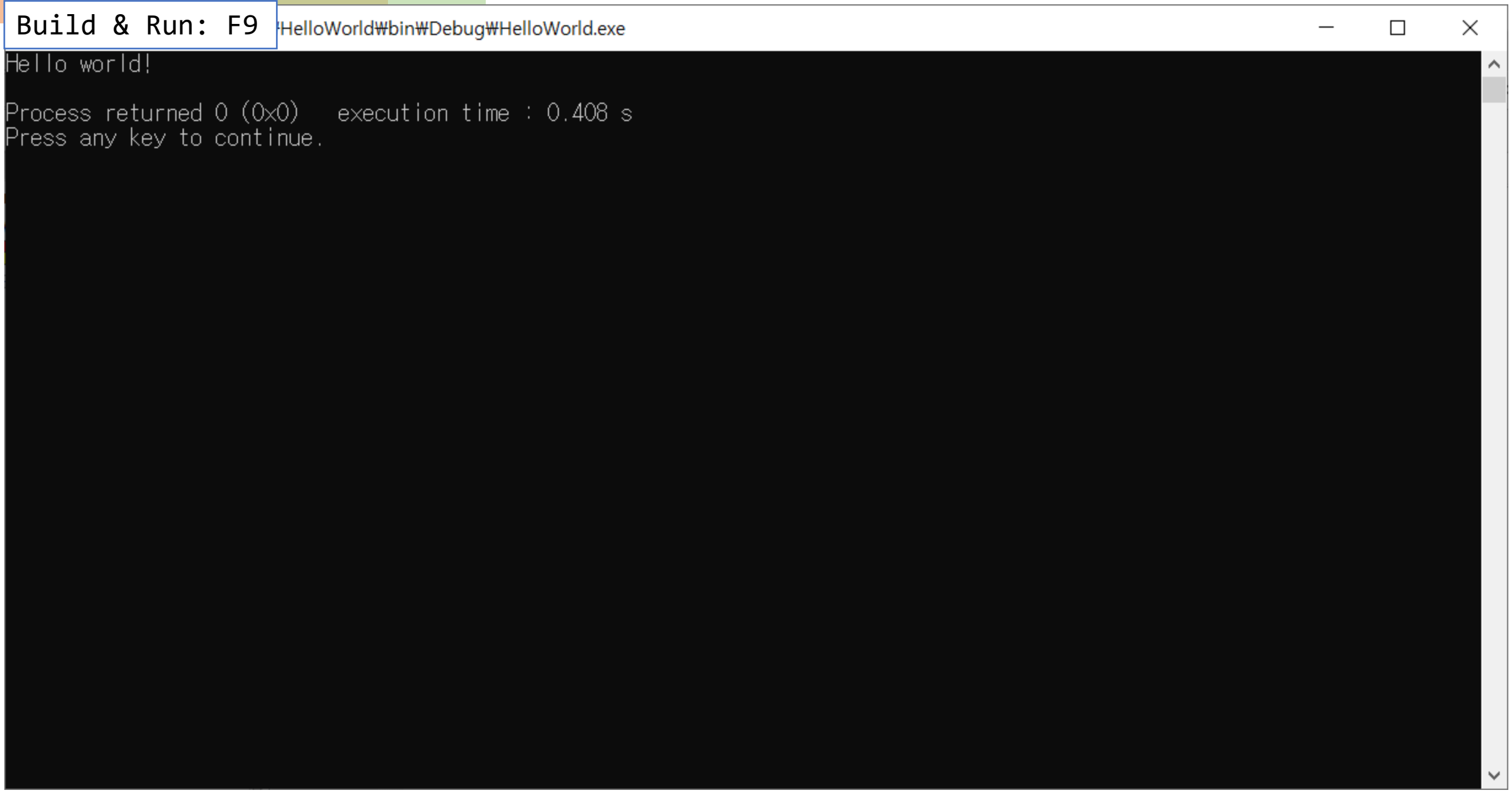
Code::Blocks | Search results | Cccc | Build log | Build messages | CppCheck/Vera++ | CppCheck/Vera++ messages | Cscope | Debugger | DoxyBlocks | Fortran info

D:₩MyProjects₩Algorithm₩HelloWorld₩main.cpp | C/C++ | Windows (CR+LF) | WINDOWS-949 | Line 10, Col 1, Pos 117 | Insert | Modified | Read/Write | default



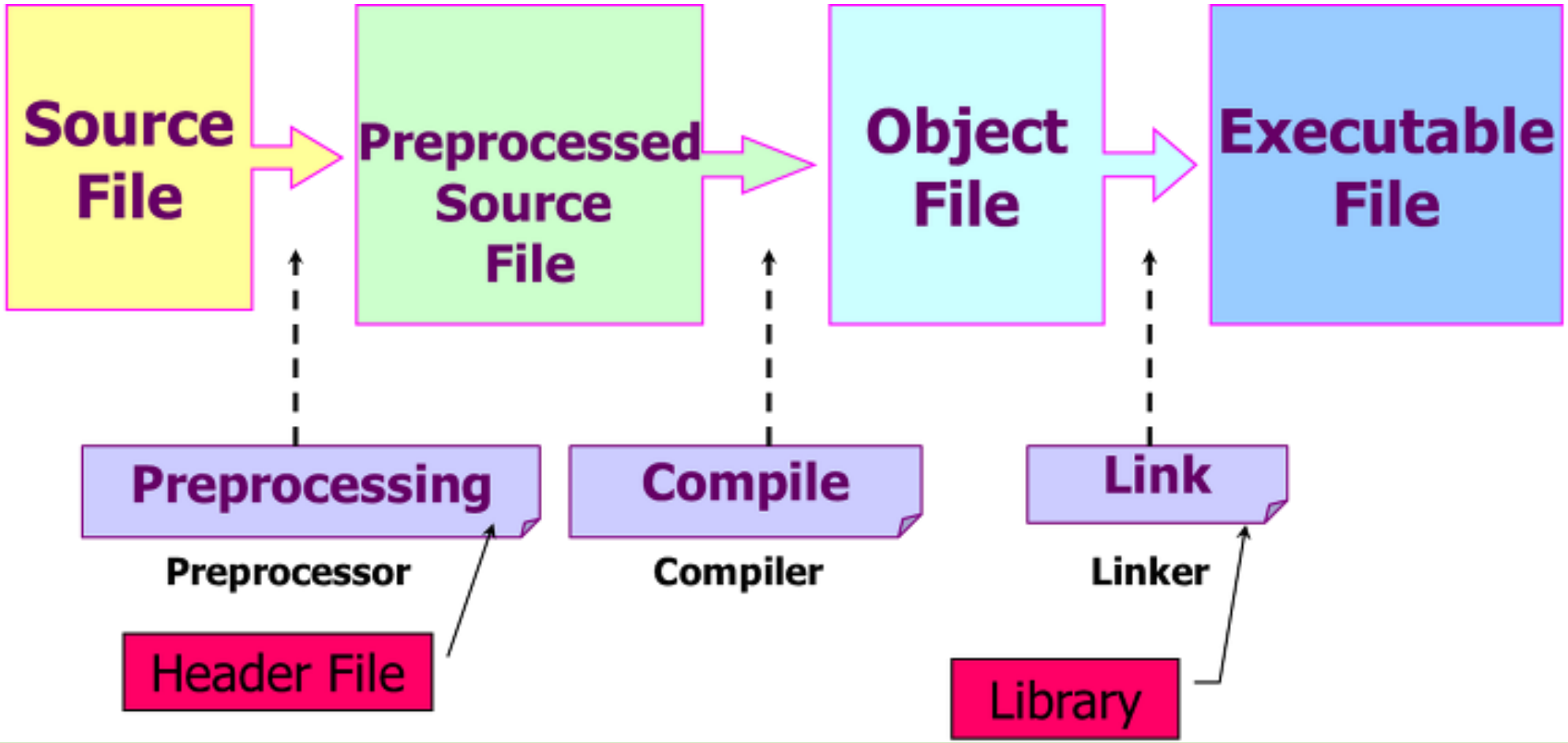
	슬래쉬	역슬래쉬
한글폰트	/	₩
영문폰트	/	\

Build & Run: F9


HelloWorld¥bin¥Debug¥HelloWorld.exe
Hello world!
Process returned 0 (0x0)   execution time : 0.408 s
Press any key to continue.

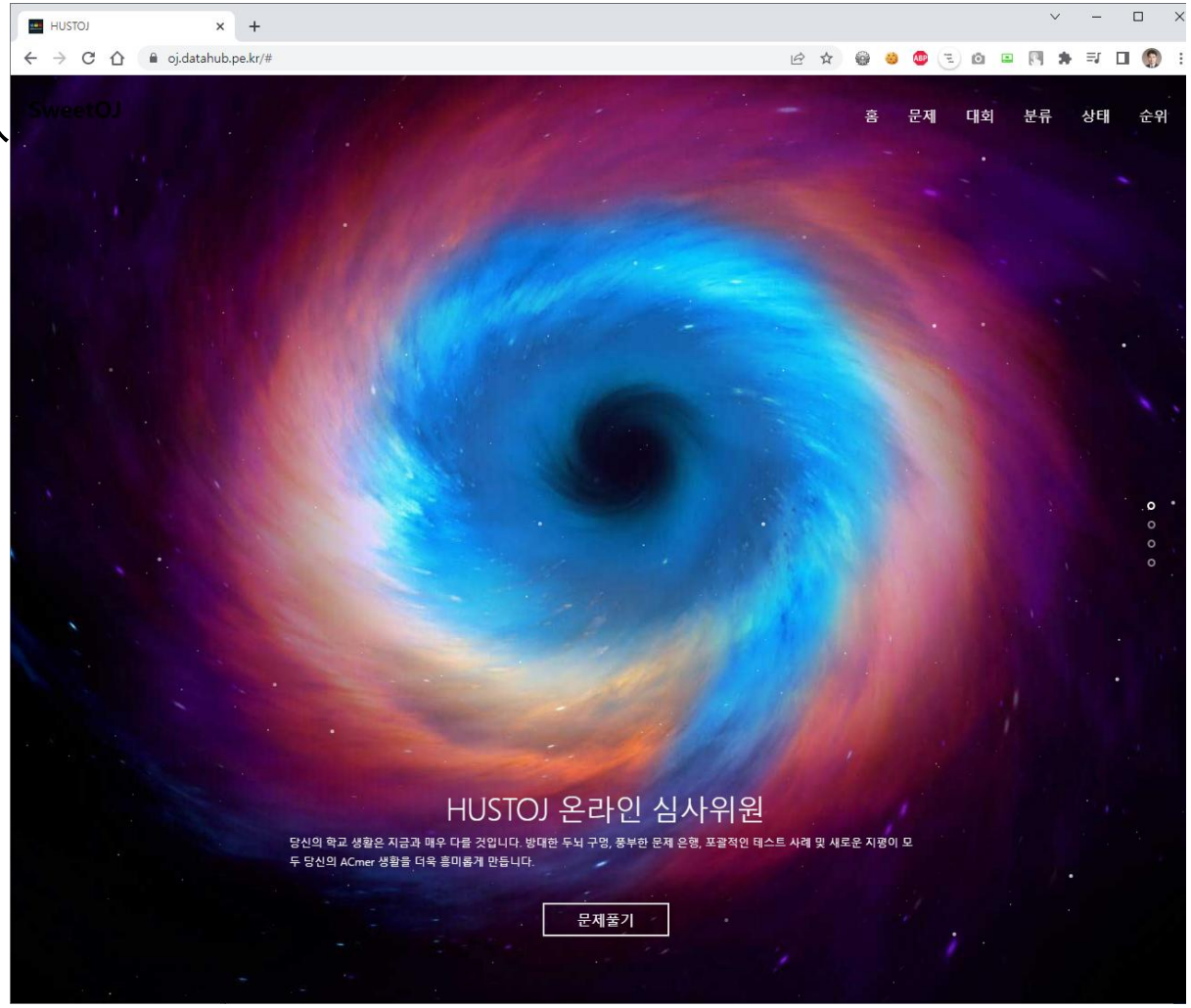
실행 파일 생성과정

- Compile & Linking



OJ 사용법

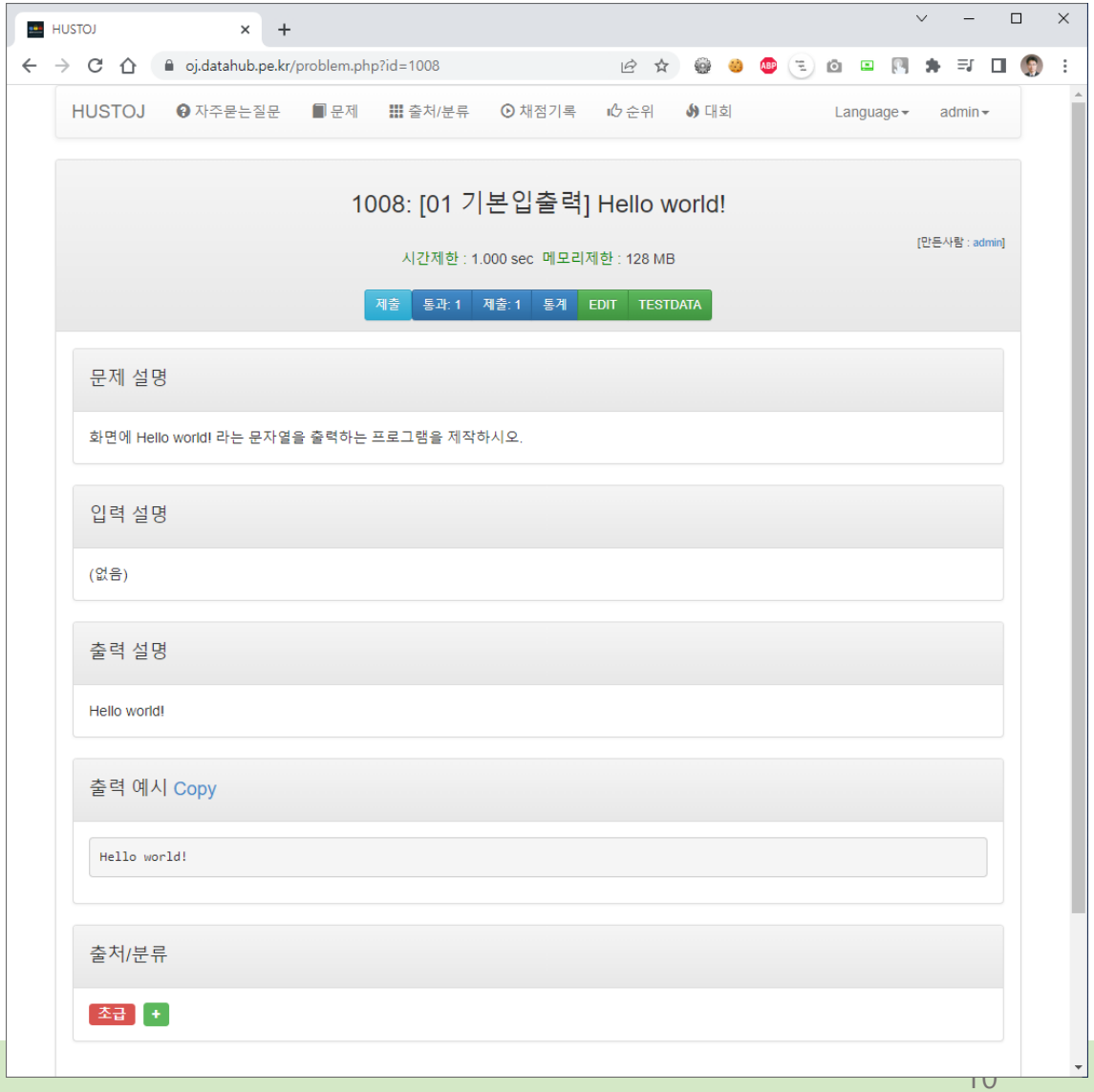
- OJ(온라인 저지)란?
  - 프로그래밍 대회에서 프로그램들을 ㅅ
    대회 연습용으로 사용되기도 한다.
  - 본 수업용 OJ
    - https://oj.datahub.pe.kr



OJ 사용법

- 파일제출
  - CodeBlock 등 IDE에서 프로그램 작성
  - 완성된 프로그램을 OJ에 업로드 후
  - 채점 결과 확인

  - 채점 결과 종류
    - 모두 맞음
    - 틀림 | 정확도: __%
    - 실행중 에러 | 정확도: __%
    - 컴파일 에러



C 언어 기초 문법

```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("Hello world! \n");
6      retrun 0;
7  }
```

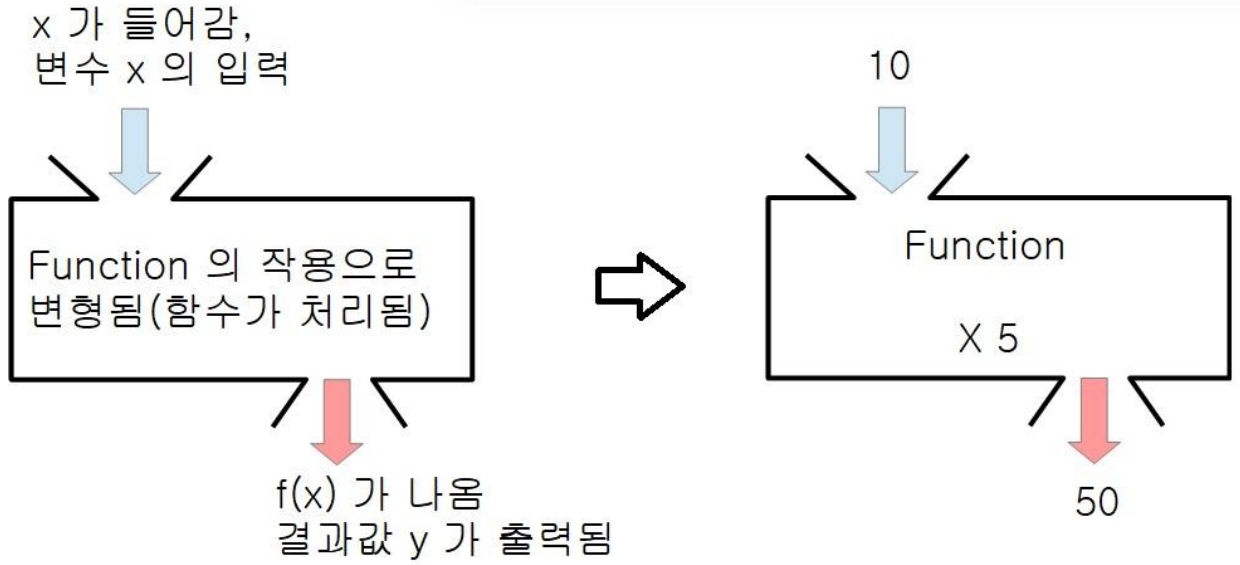
- 함수란?
  - 특정한 기능을 하는 코드들의 집합
  - C언어에서는 _____() 형태

    예시)

    최소값을 구하는 함수: min()

    절대값을 구하는 함수: abs()
  - main() 함수

    프로그램의 시작 지점



- C언어로 프로그램을 작성한다는
  - 함수를 만들고, 만든 함수들의 실행 순서를 결정하는 것

C 언어 기초 문법

```c
#include <stdio.h>

int main()
{
    printf("Hello world! \n");
    retrun 0;
}
```

- return 0;
  - 함수 결과값 0으로 함수 종료

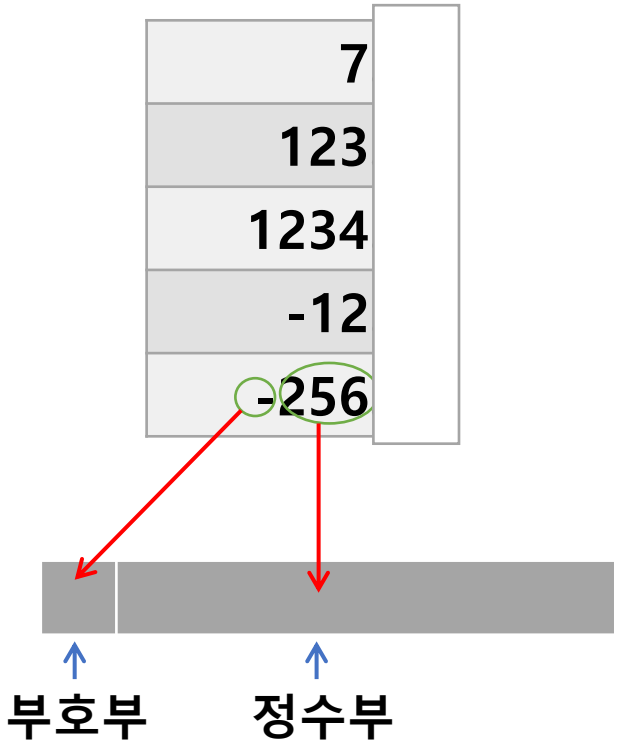
- printf("Hello world! \n");
  - printf() 함수: 표준 출력 장치에 출력하는 기능을 하는 함수
  - 인수: "Hello world! \n" 를 printf 라는 함수에 전달

- #include <stdio.h>
  - 표준 입출력 함수들에 대한 정보를 가지고 있는 stdio.h 라는 파일을 불러온다.

- 문장의 끝
  - 함수 내에 존재하는 문장의 끝에는 세미콜론 문자 ; 를 붙여준다.

수의 표현 방식

- **정수형**
  - 고정소수점 데이터형식
  - (fixed-point data format)



C언어의 자료형(data type)

- **정수형**
  - char
    (문자형) - 숫자도 저장하지만 주로 문자를 저장함
    character
  - short
    short int
  - **int**
    integer
  - long
    long integer
  - long long
    64bit long

수의 표현 방식

- **부동소수점**(浮動小數點) **데이터형식**
  - **2진수**에서는...

  ex) **10110.1001001001**

  = $1.01101001001001 \times 2^{4}$

부호부 **지수부** **가수부**

  - 장점 : 아주 큰 수, 아주 작은 수 표현 가능
  - 단점 : 계산하는데 시간이 오래 걸림

C언어의 자료형(data type)

- **실수형**
  - float
    - 단정도 정밀도 실수 (소수점 이하 6자리까지)
    - 4byte(32bit)

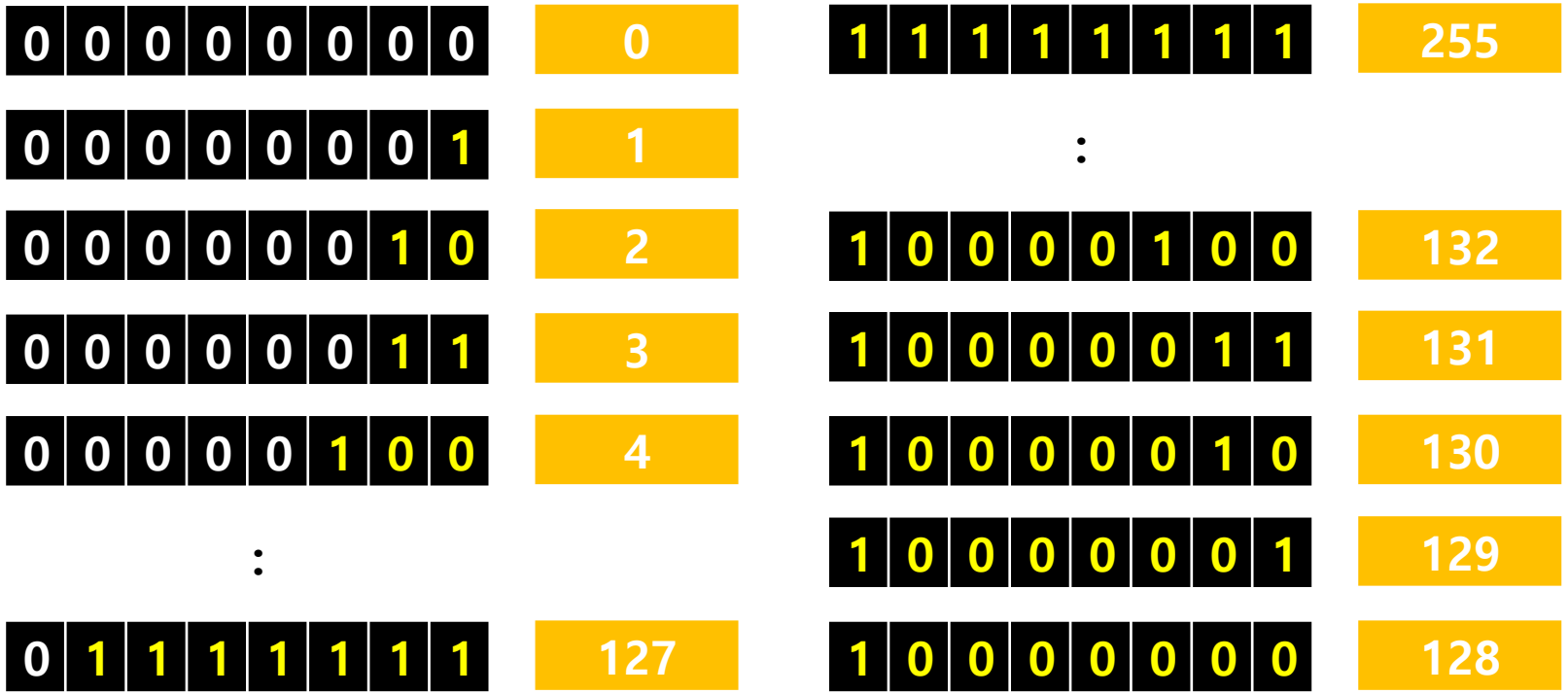
부호부 지수부 가수부

  - **double**
    - 배정도 정밀도 실수 (소수점 이하 15자리까지)
    - 8byte(64bit)

부호부 지수부 가수부

수의 표현 방식

- 정수(양수)의 표현 - unsigned



수의 표현 방식

- 정수 (양수, 음수)의 표현 - signed


```
1111 1000 (-?)
0000 0111 2의보수
```

수의 표현 방식

- 큰 수의 표현

0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	1
0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0	2
0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 1	3
0 0 0 0 0 0 0 0	0 0 0 0 0 1 0 0	4
:		
0 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	32767 $2^{15}-1$
1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	-32768 -2^{15}

수의 표현 방식

- 더 큰 수의 표현

00000000	00000000	00000000	00000000	0
00000000	00000000	00000000	00000001	1
00000000	00000000	00000000	00000010	2
00000000	00000000	00000000	00000011	3
00000000	00000000	00000000	00000100	4
00000000	00000000	00000000	00000101	5
01111111	11111111	11111111	11111111	$2^{31}-1$
10000000	00000000	00000000	00000000	-2^{31}

수의 표현 방식

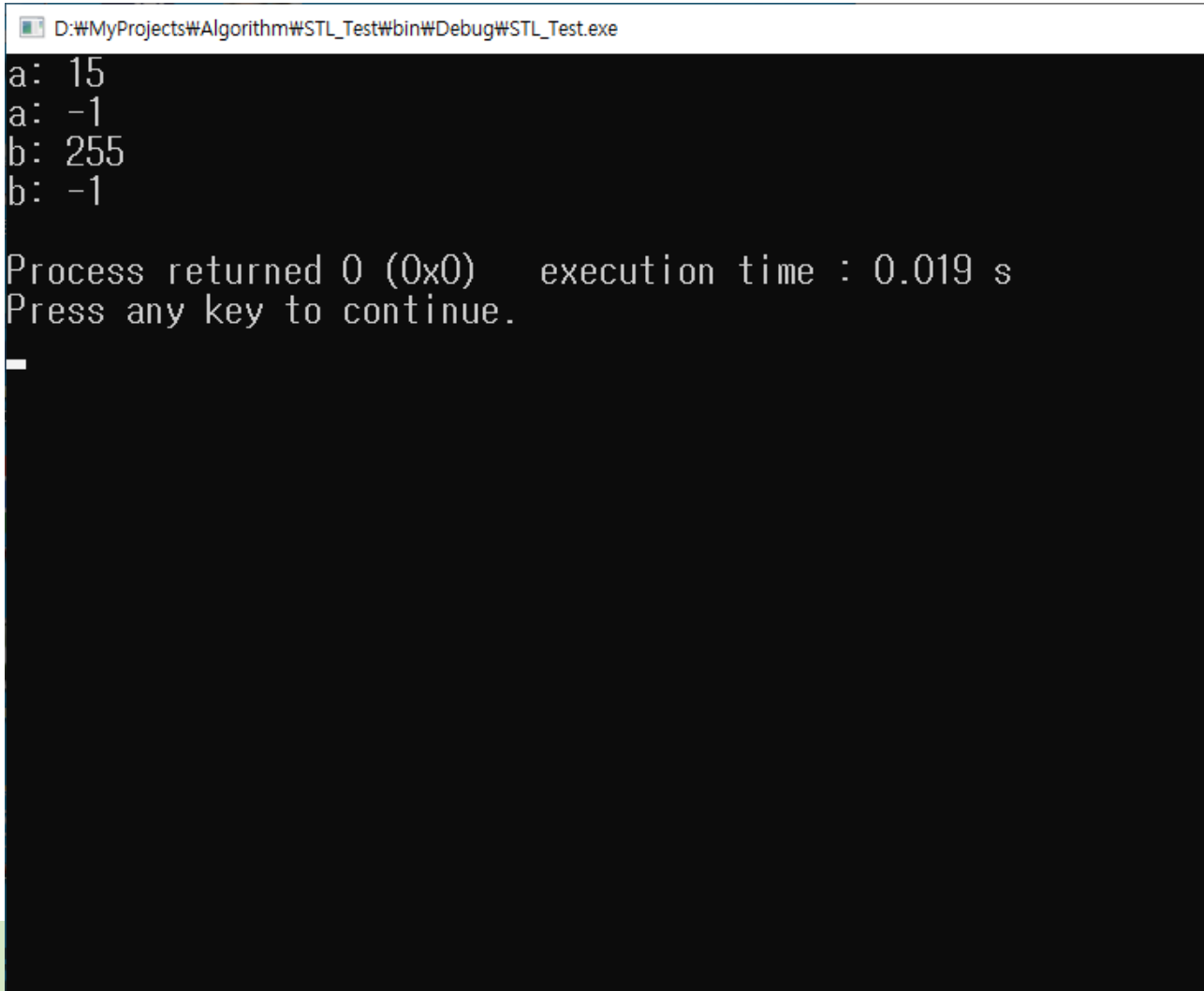
```
#include <iostream>
using namespace std;

int main() {
    signed char a = 0;
    for(int i=0; i<256; i++) {  // 256번 반복
        printf("%5d", a);
        a++;
    }
    puts("\n");

    unsigned char b = 0;
    for(int i=0; i<256; i++) {  // 256번 반복
        printf("%5d", b);
        b++;
    }
}
```

```
D:₩MyProjects₩Algorithm₩STL_Test₩bin₩Debug₩STL_Test.exe
    0    1    2    3    4    5    6    7    8    9   10   11   12
   24   25   26   27   28   29   30   31   32   33   34   35   36
   48   49   50   51   52   53   54   55   56   57   58   59   60
   72   73   74   75   76   77   78   79   80   81   82   83   84
   96   97   98   99  100  101  102  103  104  105  106  107  108
  120  121  122  123  124  125  126  127 -128 -127 -126 -125 -124 -
 -112 -111 -110 -109 -108 -107 -106 -105 -104 -103 -102 -101 -100
  -88  -87  -86  -85  -84  -83  -82  -81  -80  -79  -78  -77  -76
  -64  -63  -62  -61  -60  -59  -58  -57  -56  -55  -54  -53  -52
  -40  -39  -38  -37  -36  -35  -34  -33  -32  -31  -30  -29  -28
  -16  -15  -14  -13  -12  -11  -10   -9   -8   -7   -6   -5   -4

    0    1    2    3    4    5    6    7    8    9   10   11   12
   24   25   26   27   28   29   30   31   32   33   34   35   36
   48   49   50   51   52   53   54   55   56   57   58   59   60
   72   73   74   75   76   77   78   79   80   81   82   83   84
   96   97   98   99  100  101  102  103  104  105  106  107  108
  120  121  122  123  124  125  126  127  128  129  130  131  132
  144  145  146  147  148  149  150  151  152  153  154  155  156
  168  169  170  171  172  173  174  175  176  177  178  179  180
  192  193  194  195  196  197  198  199  200  201  202  203  204
  216  217  218  219  220  221  222  223  224  225  226  227  228
  240  241  242  243  244  245  246  247  248  249  250  251  252
Process returned 0 (0x0)   execution time : 0.189 s
Press any key to continue.
```

수의 표현 방식

- 비트열 직접 입력하기

```
#include <iostream>
using namespace std;

int main() {
    char a;
    a = 0b00001111;
    printf("a: %d\n", a);

    a = 0b11111111;
    printf("a: %d\n", a);

    short b;
    b = 0x00FF;
    printf("b: %d\n", b);

    b = 0xFFFF;
    printf("b: %d\n", b);
}
```



C언어의 자료형(data type)

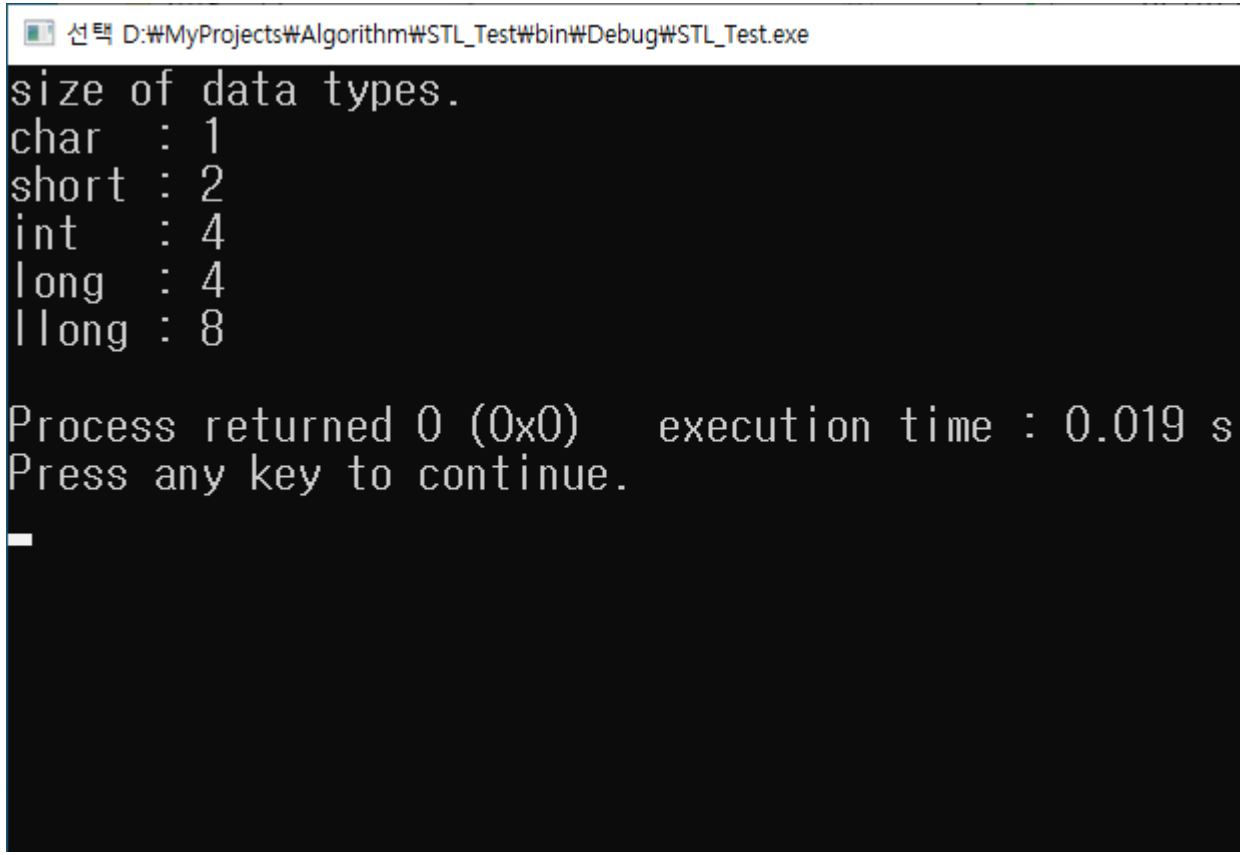
용도	타입	크기		signed(부호있음)	unsigned(부호없음)
정수형 (문자형)	**char**	1B	8bit	-2^{7} ~ $2^{7}-1$ (127)	0 ~ $2^{8}-1$ (256)
정수형	short	2B	16bit	-2^{15} ~ $2^{15}-1$ (≒3.2만)	0 ~ $2^{16}-1$ (≒6.5만)
	int	4B	32bit	-2^{31} ~ $2^{31}-1$ (≒21억)	0 ~ $2^{32}-1$ (≒42억)
	long	4B	32bit	-2^{31} ~ $2^{31}-1$ (≒21억)	0 ~ $2^{32}-1$ (≒42억)
	long long	8B	64bit	-2^{63} ~ $2^{63}-1$ (≒922경)	0 ~ $2^{64}-1$ (≒1844경)
실수형	float	4B	32bit	3.4×10^{-38} ~ 3.4×10^{38}	
	double	8B	64bit	1.7×10^{-308} ~ 1.7×10^{308}	

수의 표현 방식

- 데이터 타입 별 사이즈

```
#include <stdio.h>

int main() {
    puts("size of data types.");
    printf("char  : %d\n", sizeof(char));
    printf("short : %d\n", sizeof(short));
    printf("int   : %d\n", sizeof(int));
    printf("long  : %d\n", sizeof(long));
    printf("llong : %d\n", sizeof(long long));
    printf("float : %d\n", sizeof(float));
    printf("double: %d\n", sizeof(double));
}
```

- 결과



변수 (variable)

- 변수의 의미
  - '변하는 수' 라는 의미
  - 무언가를 기억해야 할 때 사용
  - 자료를 저장하는 공간에 이름은 붙인 것
  - 프로그래머가 이름(변수명)을 결정

- 변수의 사용
  - 선언을 한 뒤부터 사용 가능
  - = 연산자로 값을 할당
  - 선언과 동시에 할당 가능(초기화)
  - 초기화 하지 않으면 쓰레기 값

```c
#include <stdio.h>

int main() {
    // 변수 선언(초기화 없음)
    int age;
    // 변수에 값 할당
    age = 20;

    // 변수 선언(초기화 없음)
    int height;

    // 변수 선언과 동시에 할당
    char blood = 'A';
    // 변수 선언과 동시에 할당
    double pi = 3.14159;
}
```

주소	내용(값)	이름
1001	20	age
1002		
1003		
1004		
1005		height
1006		
1007		
1008		
1009	'A'	Blood
1010	3.14159	pi
1011		
1012		
1013		
1014		
1015		
1016		
1017		

서식문자

```c
#include <stdio.h>

int main() {
    int age;    // 변수 선언
    age = 20;   // 변수에 할당
    int height, weight;
    char blood = 'A';   // 선언과 할당
    double pi = 3.141592;

    printf("age: %d \n", age);
    printf("height: %d \n", height);
    printf("blood: %c \n", blood);
    printf("pi: %lf \n", pi);
    printf("pi: %.2lf \n", pi);
}
```

- 서식문자

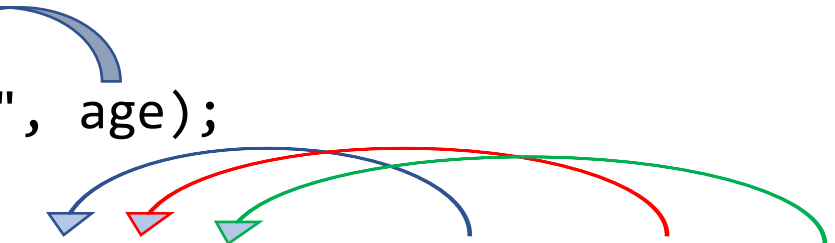
서식문자	출력 형태	사용 타입
%c	단일 문자	char
%d	부호 있는 10진 정수	char, short, int
%i	부호 있는 10진 정수, %d와 같음	
%f, %lf	부호 있는 10진 실수	float, double
%s	문자열	char[]
%o	부호 없는 8진 정수	char, short, int,
%u	부호 없는 10진 정수	
%x	부호 없는 16진 정수, 소문자 사용	
%X	부호 없는 16진 정수, 대문자 사용	
%e	e 표기법에 의한 실수	float, double
%lld, %llu	부호 있는, 부호 없는 long long 정수	long long
%g	값에 따라서 %f, %e 둘 중 하나를 선택	float, double
%G	값에 따라서 %f, %G 둘 중 하나를 선택	
%%	% 기호 출력	

printf 함수의 기본적인 이해

- 첫 번째 인수로 전달된 문자열의 서식에 맞게 출력

```c
#include <stdio.h>

int main() {
    int age = 20;
    int year = 2010, month = 10, day = 31;

    printf("my age: %d\n", age);

    printf("my birthday: %d/%d/%d\n", year, month, day);
          //      %d : decimal(10진)
    printf("Good\nmorning\neveryday\n");
          //       \n : new line
}
```

```
my age: 20
my birthday: 2010/10/31
Good
morning
everyday
```



%d 의 의미:
decimial(10진)
10진수로
출력하라는 의미

상수 (constant)

- 상수의 의미
  - 프로그램 실행도중 값을 변경할 수 없는 데이터

- 상수의 종류
  1. 리터럴 상수
  2. 매크로 상수
  3. 변수의 고정

1. 리터럴 상수

   프로그램내에서 지정된 값.

   1) 정수 상수

   ```
   int n = 10;
   ```

   2) 실수 상수

   ```
   double mili = 0.001;
   double mili = 1.0e-3;
   double ton = 1.0e+3;
   ```

   3) 문자 상수

   ```
   char grade = 'A';
   ```

상수 (constant)

2. 매크로 상수

- #define 문 사용

```
#include <stdio.h>
#define PI 3.141592
int main() {
   int r = 5;
   double s;
   s = PI*r*r;
   printf("%lf\n", s);
   r++;

   s = PI*r*r;
   printf("%lf\n", s);
   r++;
}
```

3. 변수의 고정

- const 키워드 사용

```
#include <stdio.h>
int main() {
   const double PI=3.141592;
   int r = 5;
   double s;
   s = PI*r*r;
   printf("%lf\n", s);
   r++;

   s = PI*r*r;
   printf("%lf\n", s);
   r++;
}
```

서식문자

- 서식 문자의 종류와 그 의미

서식문자	출력 형태
%c	단일 문자
%d	부호 있는 10진 정수
%i	부호 있는 10진 정수, %d와 같음
%f, %lf	부호 있는 10진 실수
%s	문자열
%o	부호 없는 8진 정수
%u	부호 없는 10진 정수
%x	부호 없는 16진 정수, 소문자 사용
%X	부호 없는 16진 정수, 대문자 사용
%e	e 표기법에 의한 실수
%lld, %llu	부호 있는, 부호 없는 long long 정수
%g	값에 따라서 %f, %e 둘 중 하나를 선택
%G	값에 따라서 %f, %G 둘 중 하나를 선택
%%	% 기호 출력

```c
#include <stdio.h>

int main() {
    char blood = 'A';
    int age = 20;
    double pi=3.1415926535;

    printf("blood: %c. \n", blood);
    printf("name : %s. \n", "Reinhard");
    printf("age  : %d. \n", age);
    printf("pi   : %lf. \n\n", pi);

    printf("blood: %10c. \n", blood);
    printf("name : %10s. \n", "Reinhard");
    printf("age  : %10d. \n", age);
    printf("pi   : %10.4lf. \n\n", pi);

    printf("blood: %-10c. \n", blood);
    printf("name : %-10s. \n", "Reinhard");
    printf("age  : %-10d. \n", age);
    printf("pi   : %-10.4lf.  \n\n", pi);
}
```

[기본서식]

```
blood: A.
name : Reinhard.
age  : 20.
pi   : 3.141593.
```

[오른쪽 정렬]

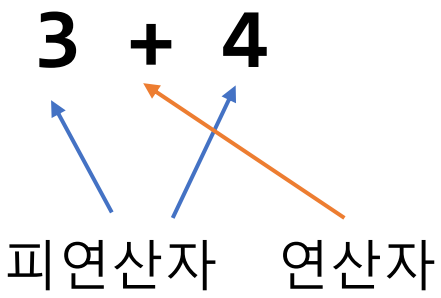
```
blood:          A.
name :   Reinhard.
age  :         20.
pi   :     3.1416.
```

[왼쪽 정렬]

```
blood: A          .
name : Reinhard   .
age  : 20         .
pi   : 3.1416     .
```

연산자(operator)

- 연산자와 피연산자



- 이항연산자
  - 피연산자가 두 개인 연산자
  - +, -, x, ÷
- 단항연산자
  - 피연산자가 한 개인 연산자
  - -, +, ~, !

- C언의 연산자 표기

의미	수학	C언어	비고
덧셈	+	+	
뺄셈	-	-	
곱셈	×	*	v
나눗셈	÷	/	v
나머지		%	
같다	=	==	v
다르다	≠	!=	v
크다	>, ≥	>, >=	
작다	<, ≤	<, <=	
그리고	And	&&	
또는	Or	||	

연산자(operator)

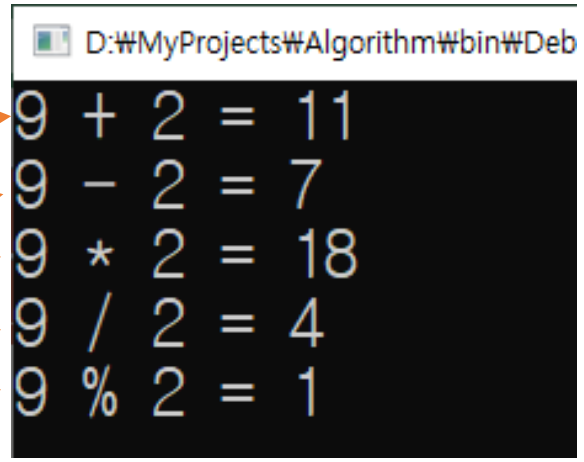
- 대입 연산자와 산술 연산자

연산자	설명	결합방향
=	연산자 오른쪽에 있는 값을 연산자 왼쪽에 있는 변수에 대입한다. 예) num = 20;	←
+	두 피연산자의 값을 더한다. 예) num = 4+3;	→
-	왼쪽 피연산자의 값에서 오른쪽 피연산자 값을 뺀다. 예) num = 4-3;	→
*	두 피연산자의 값을 곱한다. 예) num = 4*3;	→
/	왼쪽의 피연산자 값을 오른쪽 피연산자 값으로 나눈 몫을 구한다. 예) num = 7/3;	→
%	왼쪽의 피연산자 값을 오른쪽 피연산자 값으로 나눈 나머지를 구한다. 예) num = 7%3;	→

연산자(operator)

- 대입 연산자와 산술 연산자 실습

```
#include <stdio.h>

int main() {
    int n1 = 9, n2 = 2, res;
    res = n1 + n2;
    printf("%d + %d = %d \n", n1, n2, res);
    res = n1 - n2;
    printf("%d - %d = %d \n", n1, n2, res);

    printf("%d * %d = %d \n", n1, n2, n1*n2);
    printf("%d / %d = %d \n", n1, n2, n1/n2);
    printf("%d %% %d = %d\n", n1, n2, n1%n2);
}
```



자료의 형변환(type casting)

- 묵시적 형변환(=자동 형변환)
  - 자동으로 일어나는 형변환
  - 언제?
    - 대입문에서 좌변과 우변의 자료형이 다를 때
    - 연산식에서 피연산자간에 자료형이 다를 때
  - 변환 규칙
    - 대입문의 경우 우변의 자료형이 좌변의 자료형으로 변환
    - 연산식에서는 두 피연산자 중에서 표현 범위가 더 넓은 자료형으로 변환

```c
#include <stdio.h>

int main() {
    char  a = 127;  // 1111 1111
    short b;
    b = a;
    printf("%d\n", b);

    char  c;
    short d = 256;  // 1 0000 0000
    c = d;
    printf("%d\n", c);

    double e = 10/4.0 +0.5;
    printf("%lf\n", e);
}
```

자료의 형변환(type casting)

- 명시적 형변환
  - 프로그래머가 지정하는 형변환
  - (타입) 형식을 사용한다.
  - 예시

```
double c = 10/4;   // 2
double d = (double)10/4;   // 2.5
```

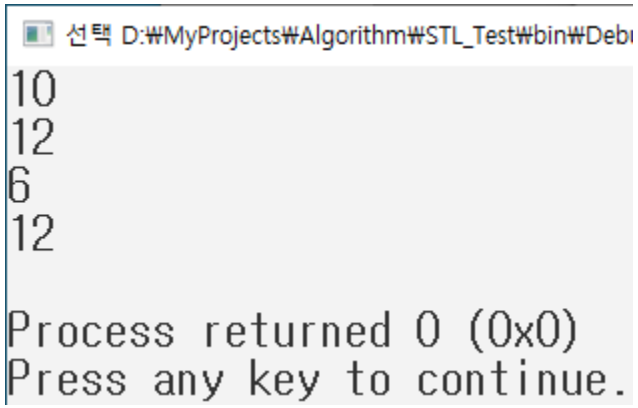
```
#include <stdio.h>

int main() {
    int i = 9 *1000*1000*1000;
    printf("%d\n", i);

    long long l = (long long)9 *1000*1000*1000;
    printf("%lld\n", l);

    double c = 10/4;
    printf("%lf\n", c);

    double d = (double)10/4;
    printf("%lf\n", d);
}
```

```
410065408
9000000000
2.000000
2.500000
```

연산자(operator)

- 증감 연산자

연산자	연산의 예	의미	결합성
++a	printf("%d", ++a)	선 증가, 후 연산	←
a++	printf("%d", a++)	선 연산, 후 증가	←
--b	printf("%d", --b)	선 감소, 후 연산	←
b--	printf("%d", b--)	선 연산, 후 감소	←




```
#include <stdio.h>

int main() {
    int a=10;
    printf("%d\n", a++);
    printf("%d\n", ++a);

    int x=2, y=3;
    printf("%d\n", (x++)*(y++));
    printf("%d\n", (x*y));
}
```

연산자(operator)

- 비트 연산자

연산자	의미	예	결합성
&	Bitwise AND	10 & 7	←
|	Bitwise OR	10 | 7	←
^	Bitwise XOR	10 ^ 7	←
~	Bitwise NOT	~ 7	←
<<	비트열 왼쪽 시프트	8 << 1	←
>>	비트열 오른쪽 시프트	8 >> 2	←

```
#include <iostream>
using namespace std;

int main() {
    printf("10 & 7 = %02X\n", 10 & 7);
    printf("10 | 7 = %02X\n", 10 | 7);
    printf("10 ^ 7 = %02X\n", 10 ^ 7);
    printf("8 << 1 = %02X\n", 8 << 1);
    printf("8 >> 2 = %02X\n", 8 >> 1);
}
```

```
  10: 0000 1010      10: 0000 1010      10: 0000 1010                        1111 1111
& 7: 0000 0111    | 7: 0000 0111    ^ 7: 0000 0111    ~ 7: 0000 0111    ^ 1000 0000
    : 0000 0010        : 0000 1111        : 0000 1101        : 1111 1000      0111 1111
```

Selected Decimal, Binary and Hexadecimal equivalents

연산자(operator)

- 복합 대입 연산자

연산자	의미	사용 예	같은표현
+=	값을 더하여 대입	a+=3	a=a+3
-=	값을 빼서 대입	a-=3	a=a-3
*=	값을 곱하여 대입	a*=3	a=a*3
/=	값으로 나누어 대입	a/=3	a=a/3
%=	값으로 나눈 나머지를 대입	a%=3	a=a%3

다음 중 의미가 다른 수식은?

ⓐ c=c+1;
ⓑ c++;
ⓒ ++c;
ⓓ c+=1;
ⓔ ++c++;

```
#include <iostream>
using namespace std;

int main() {
    int a=9;
    printf("%d\n", a+=3);
    printf("%d\n", a-=3);
    printf("%d\n", a*=3);
    printf("%d\n", a/=3);
    printf("%d\n", a%=3);
}
```

연산자(operator)

- 연산자의 우선순위와 결합방향

우선순위	연산자유형		연산자종류	결합방향
높음	식, 구조체, 공용체 연산자		( ) [ ] -> .	좌 → 우
	단항 연산자		! ~ ++ -- - (형명칭) * & sizeof	좌 ← 우
↑	이항 연산자	승, 제	* / %	좌 → 우
		가, 감	+ -	좌 → 우
		비트 이동	<< >>	좌 → 우
		대소 비교	< <= > >=	좌 → 우
		등가 판정	== !=	좌 → 우
↓		비트 AND	&	좌 → 우
		비트 XOR	^	좌 → 우
		비트 OR	|	좌 → 우
		논리 AND	&&	좌 → 우
		논리 OR	||	좌 → 우
	조건 연산자		? :	좌 ← 우
낮음	대입 연산자		+ =+ =+ *= /= %=	좌 ← 우
	나열 연산자		,	좌 → 우

- 꼭 기억해야할 연산자 우선순위
  ① ( )
  ② ++ -- !
  ③ * / %
  ④ + -
  ⑤ =

연산자(operator)

- 연산자 우선순위 실험

```
#include <stdio.h>

int main(void) {
    int a = 10+4*3-1;
    printf("%d \n", a);

    int b = 10+4*(3-1);
    printf("%d \n", b);

    int r=4+5*6/(2+1)+15-5*2;
    printf("%d \n", r);
}
```

```
21
18
19
```

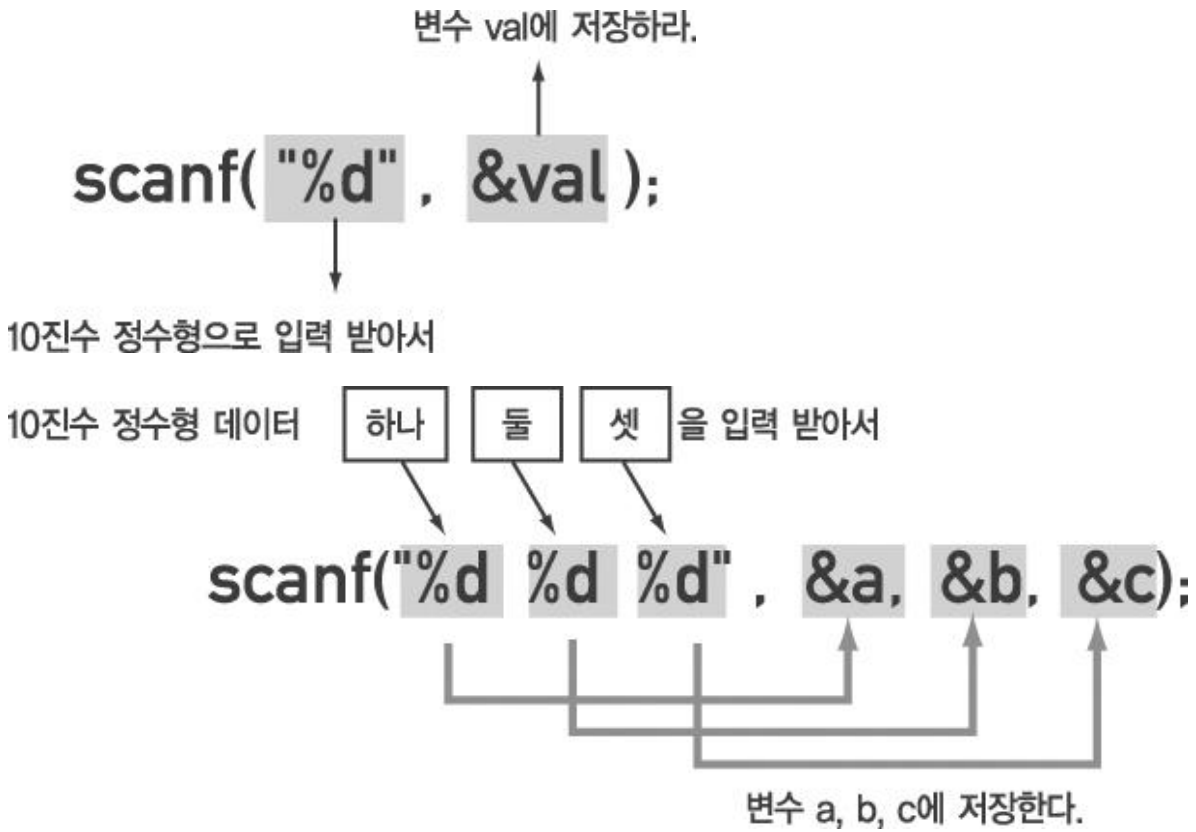
- 연산자 결합방향 실험

```
#include <stdio.h>

int main(void)
{
    int r = 10-1-2-3+4+5;
    printf("%d \n", r);

    int a=3, b=4, c=5, d=6;
    a = b = c = d;
    printf("%d %d %d %d", a,b,c,d);
}
```

```
13
6 6 6 6
```

scanf() 함수를 이용한 입력

- 키보드로부터 정수 입력을 위한 scanf 함수의 호출



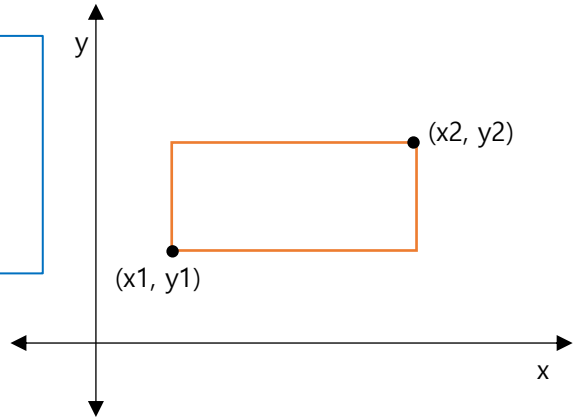
OJ에 제출

```c
#include <stdio.h>

int main() {
   int n1, n2;

   printf("input n1: ");
   scanf("%d", &n1);
   printf("input n2: ");
   scanf("%d", &n2);
   printf("%d + %d = %d\n",n1, n2, n1+n2);

   printf("input two nums:\n");
   scanf("%d %d", &n1, &n2);
   printf("%d * %d = %d\n", n1, n2, n1*n2);

   return 0;
}
```

연습문제 – 직사각형의 넓이

- 문제
  - 두 점의 x, y 좌표를 입력 받아서, 두 점이 이루는 직사각형의 넓이를 계산하여 출력하는 프로그램을 작성하시오. (x1, y1)의 좌표가 (x2, y2)보다 작다고 가정
  - 실행의 예

```
2 4
4 8
8
```



- 정답

OJ에 제출

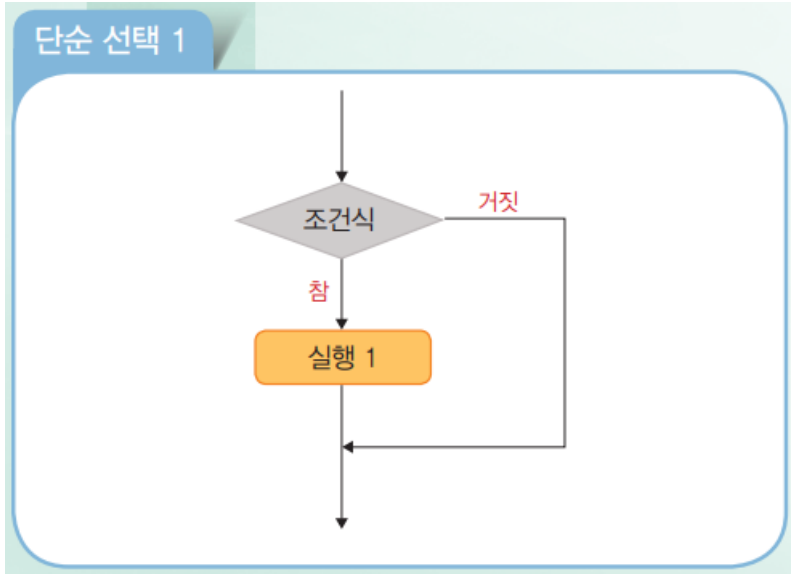
```
#include <stdio.h>

int main() {
    int x1, y1;
    int x2, y2;









    return 0;
}
```

제어문

- 조건문
  - if 문
    - 단순 if
    - if … else
    - if … else if … else
  - switch문
    - case
    - default
- 반복문
  - while 문
  - do…while 문
  - for문
- 기타
  - break 문
  - continue 문

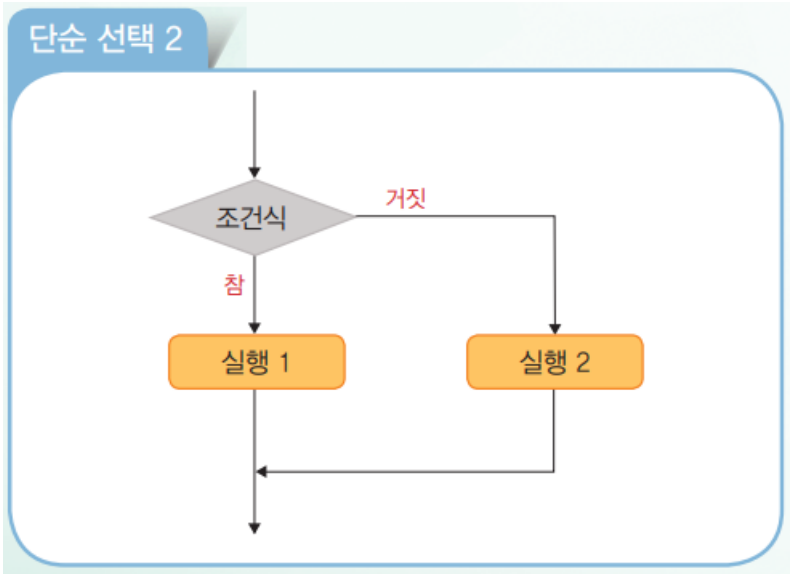
선택 실행 구조

- if 문



- 예시
  - 100점이면 congratulation 출력

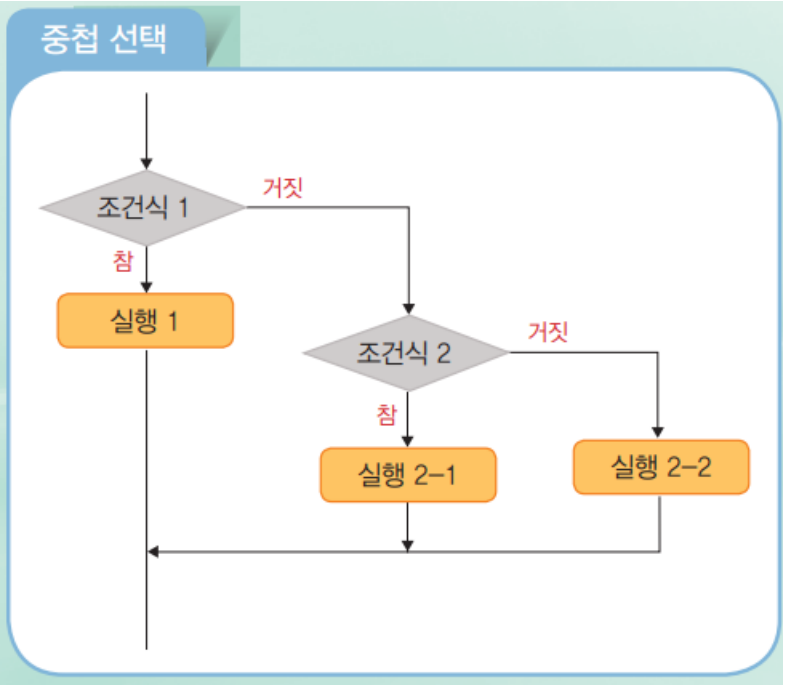
- if … else 문



- 예시
  - 60점 이상이면 "Pass", 아니면 "Fail"

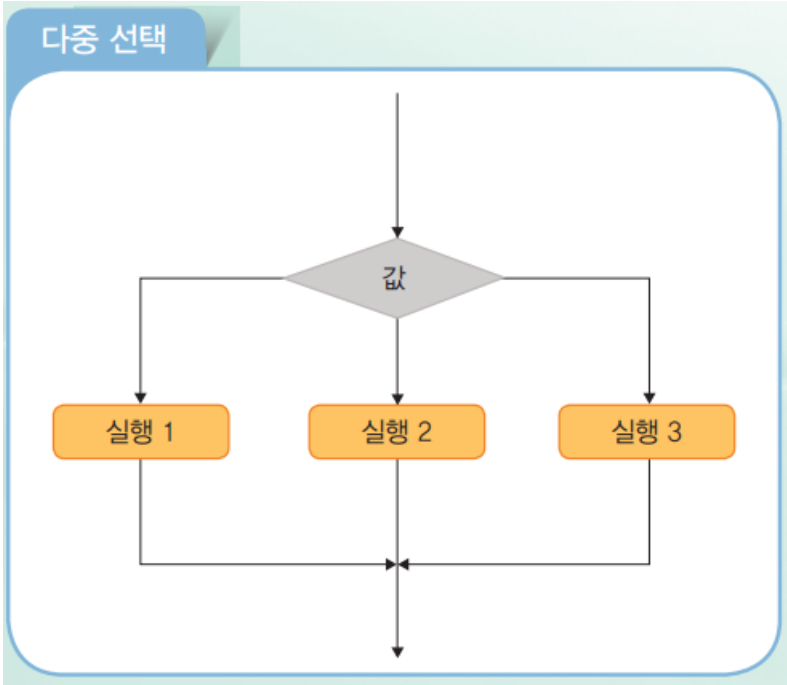
선택 실행 구조

- if … else if … else 문



- 예시
  - 90점 이상이면 A, 80점 이상이면 B,
  - 그 외에는 C

- switch 문



- 예시
  - ↑ : 전진, ← : 좌회전, → : 우회전

조건문 – if

- 단순 if 문
  - 특정 조건을 만족하는 경우 해야 할 일이 있을 때 사용
  - 처리해야 할 명령어가 2개 이상인 경우 반드시 중괄호로 묶어야 함

```
input score: 100
Perfect!
You good!
Test passed.
```

```
input score: 60
You good!
Test passed.
```

```c
#include <stdio.h>

int main() {
    int score;
    printf("input score: ");
    scanf("%d", &score);

    if(score == 100)
        printf("Perfect!\n");

    if(score >= 60) {
        printf("You good!\n");
        printf("Test passed.\n");
    }
}
```

연산자(operator)

- 비교 연산자(관계 연산자)
  - 두 피연산자의 관계(크다, 작다 혹은 같다)를 따지는 연산자
  - true(논리적 참, 1), false(논리적 거짓, 0) 반환

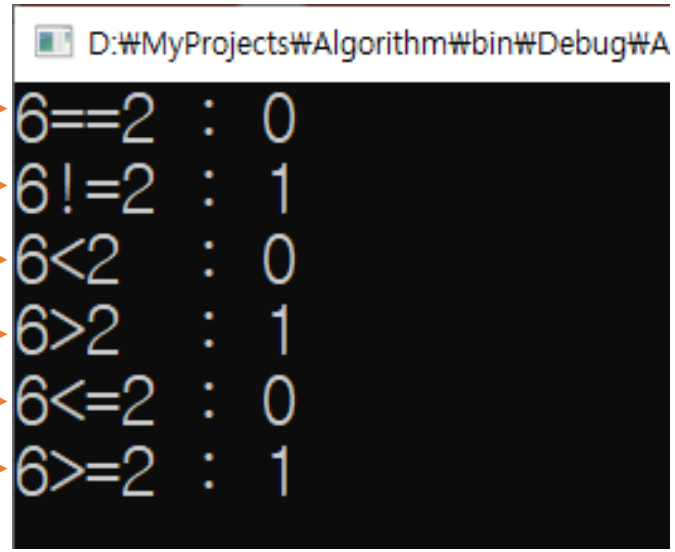
연산자	연산의 예	의미	결합성
==	a == b	a와 b가 같은가	→
!=	a != b	a와 b가 같지 않은가	→
<	a < b	a가 b보다 작은가	→
>	a > b	a가 b보다 큰가	→
<=	a <= b	a가 b보다 작거나 같은가	→
>=	a >= b	a가 b보다 크거나 같은가	→

연산자(operator)

- 비교 연산자(관계 연산자)

```
#include <stdio.h>

int main() {
    int a=6, b=2;

    printf("%d==%d : %d\n", a, b, a==b);
    printf("%d!=%d : %d\n", a, b, a!=b);
    printf("%d<%d  : %d\n", a, b, a<b);
    printf("%d>%d  : %d\n", a, b, a>b);
    printf("%d<=%d : %d\n", a, b, a<=b);
    printf("%d>=%d : %d\n", a, b, a>=b);
}
```

- 모든 연산자는 계산 결과를 반환한다.
- 비교연산자는 참(true) 이면 1을, 거짓(false)이면 0을 반환한다.



조건문 – if

- if … else 문
  - 특정 조건을 만족하는 경우 A를 하고 만족하지 않는 경우 B를 수행할 때 사용
  - 점수를 입력 받아 60점 이상이면 'You passed!' 를 아니면 'Failed.' 'Retry it.' 을 출력하는 프로그램을 작성하시오.

```
#include <stdio.h>

int main()
{
    int score;
    printf("input score: ");
    scanf("%d", &score);

    if(score >= 60)
        printf("You passed!\n");
    else {
        printf("Failed.\n");
        printf("Retry it.\n");
    }
    return 0;
}
```

조건문 – if

- if … else if … else 문

  - 점수를 입력 받아 90점 이상이면 'A', 80점 이상이면 'B', 70점 이상이면 'C', 그 밖의 경우 'F'를 출력하는 프로그램을 작성하시오.

OJ에 제출

```
#include <stdio.h>

int main() {
    int score;
    char grade;
    printf("input score: ");
    scanf("%d", &score);

    if(score >= 90)
        grade = 'A';
    else if(score >= 80)
        grade = 'B';
    else if(score >= 70)
        grade = 'C';
    else
        grade = 'F';

    printf("grade: %c\n", grade);
}
```

연습문제

- BMI 계산
  - 체질량지수는 자신의 몸무게(kg)를 키의 제곱(m)으로 나눈 값입니다.
  - 몸무게(kg단위)와 키(cm단위)를 입력받아 BMI를 계산하여 소수점 둘째 자리까지 출력하고,
  - BMI 수치에 따른 결과를 출력하시오.
    - 18.5 미만이면 '저체중'
    - 18.5 ~ 23미만이면 '정상'
    - 23.0 ~ 25 미만이면 '과체중'
    - 25.0 이상부터는 '비만'

OJ에 제출

```
#include <stdio.h>
int main() {
    double w;  // 몸무게
    double h;  // 키
    double bmi;

    scanf("%lf", &w);
    scanf("%lf", &h);









}
```

연산자(operator)

- 논리 연산자
  - and, or, not을 표현하는 연산자
  - true(1), false(0) 반환

연산	C연산자	연산의 예	의미	결합성
AND	&&	a && b	true면 true 리턴	→
OR	||	a || b	하나라도 true면 true 리턴	→
NOT	!	!a	true면 false를, false면 true 리턴	→

조건문 – if

- 필기/실기 모두 60점 이상이어야 합격

```c
#include <stdio.h>

int main() {
    int pilgi, silgi;
    printf("필기와 실기 점수를 입력: ");
    scanf("%d %d", &pilgi, &silgi);

    if(pilgi>=60 && silgi>=60)
        printf("You passed!\n");
    else {
        printf("Failed.\n");
        printf("Retry it.\n");
    }
    return 0;
}
```

- 필기/실기 둘 중 하나만 60점 이상이면 합격

```c
#include <stdio.h>

int main() {
    int pilgi, silgi;
    printf("필기와 실기 점수를 입력: ");
    scanf("%d %d", &pilgi, &silgi);

    if(pilgi>=60 || silgi>=60)
        printf("You passed!\n");
    else {
        printf("Failed.\n");
        printf("Retry it.\n");
    }
    return 0;
}
```

조건문 – if

- 점수가 60점 미만이 아니면 합격

```
#include <stdio.h>

int main() {
    int score;
    printf("input score: ");
    scanf("%d", &score);

    if( !(score<60) )
        printf("You passed!\n");
    else
        printf("Failed.\n");

    return 0;
}
```

- 필기가 60점미만이 아니고, 실기도 60점 미만이 아니면 합격

```
#include <stdio.h>

int main(){
    int pilgi, silgi;
    printf("필기와 실기 점수를 입력: ");
    scanf("%d %d", &pilgi, &silgi);

    if( !(pilgi<60) && !(silgi<60))
        printf("You passed!\n");
    else
        printf("Failed.\n");

    return 0;
}
```

조건문 – switch

- switch 문
  - 비교.선택 할 조건이 많은 경우 유용
  - switch(수식)
    - 수식은 정수형 변수, 정수형 수식만 가능
  - case 값
    - 값은 정수만 가능
  - default: 기본
  - break: switch 탈출

```c
#include <stdio.h>
int main() {
    int score;
    char grade;
    printf("input score: ");
    scanf("%d", &score);

    switch(score / 10) {
        case 10:
        case  9:
            grade = 'A';
            break;
        case  8:
            grade = 'B';
            break;
        case  7:
            grade = 'C';
            break;
        default:
            grade = 'F';
    }
    printf("grade: %c\n", grade);
}
```

score / 10
[정수] 나누기 [정수]는
결과도 정수

연습문제

- 두 개의 정수와 한 개의 사칙 연산자를 입력받아 사칙연산 결과를 처리하는 프로그램을 작성하시오.
- 입력되는 모든 숫자는 정수이고, 가운데 연산자는 + - * / % 이외는 없다.

```
예: 10 + 5
계산할 수식을 입력하세요
9 * 5
9 * 5 = 45
```

```
#include <stdio.h>

int main() {
    int n1, n2, res;
    char op;
    printf("예: 10 + 5\n");
    printf("계산할 수식을 입력하세요\n");
    scanf("%d %c %d", &n1, &op, &n2);









    printf("%d %c %d = %d\n", n1, op, n2, res);
}
```

반복문

- 반복문의 기능
  - 특정 영역을 특정 조건이 만족되는 동안에 반복 실행하기 위한 문장

- 세 가지 형태의 반복문이 제공됨
  1) while문에 의한 반복
     - 몇 번 반복해야 하는지 모를 때 사용, ex) 답 맞출 때까지 계속
  2) do ~ while문에 의한 반복
     - 일단 한번은 실행하고 그 결과에 따라 다시 반복할 수도 있을 때 사용, ex) 메뉴 입력
  3) for문에 의한 반복
     - 반복 횟수가 정해져 있는 경우 주로 사용, ex) 10번 출력

반복문 - while

- 형식

```
while(반복조건)
    반복할 문장;
```

```
while(반복조건) {
    반복할 문장1;
    반복할 문장2;
          :
}
```

- 반복조건이 참인 동안 반복할 문장을 실행

- 예시

```
int n = 1;
while(n < 5) {
    printf("%d \n", n);
    n++;  // n 1증가
}
```

n
5

```
1
2
3
4
```

반복문 - while

- 5회 반복 방법1

```
int c = 0;
while(c < 5) {
    printf("%d \n", c);
    c++;
}
```

```
0
1
2
3
4
```

- 5회 반복 방법2

```
int c = 1;
while(c <= 5) {
    printf("%d \n", c);
    c++;
}
```

```
1
2
3
4
5
```

반복문 - while

- 퀴즈

```c
#include <stdio.h>
int main() {
    int num = 10; // 10부터 시작

    puts("Rocket lunch countdown..");
    while(num > 0) {  // 0보다 크면
        printf("%2d \n", num);
        num--;  // 1씩 감소하면서…
    }
    printf("last num:%2d \n", num);
    return 0;
}
```

- 결과

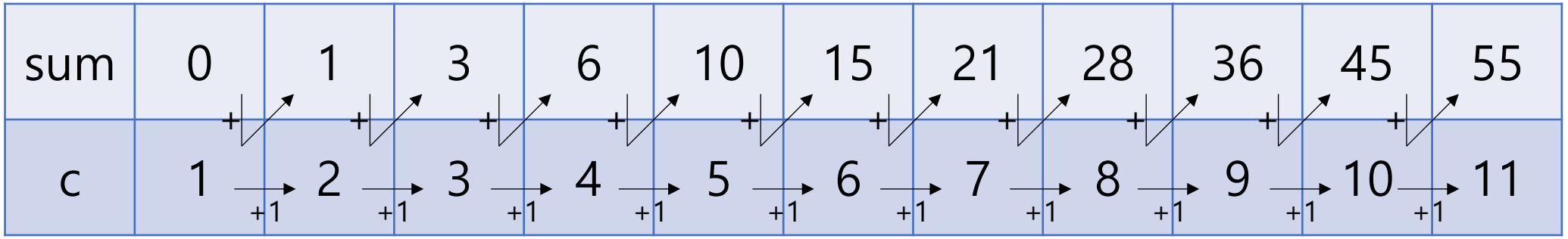
```
Rocket lunch countdown..
10
 9
 8
 7
 6
 5
 4
 3
 2
 1
```

- 카운트 다운 숫자는 얼마까지 출력될까?
- 종료 직전 num 값은? → 0

반복문 - while

- 1부터 10까지 누계 구하기

1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10



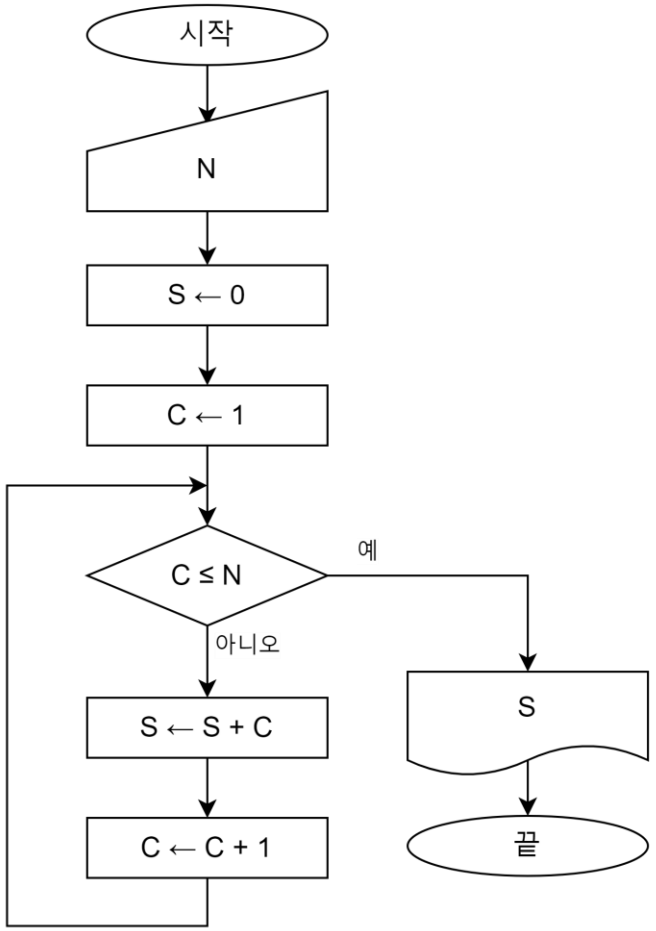
[초기값]
sum = 0
c = 1

```
while(c < 11)
    sum = sum + c
    c = c + 1
```

```
while(c <= 10)
    sum = sum + c
    c = c + 1
```

반복문 - while

- 1부터 10까지 누계 구하기



- 소스코드

```
#include <stdio.h>
int main() {
    int sum=0, c=1;
    while(c ? 10) {
        sum=sum+c;
        c++;
   }
   printf("%d \n", sum);
}
```

- 출력

```
55
```

STEP	sum	c
1	0	1
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

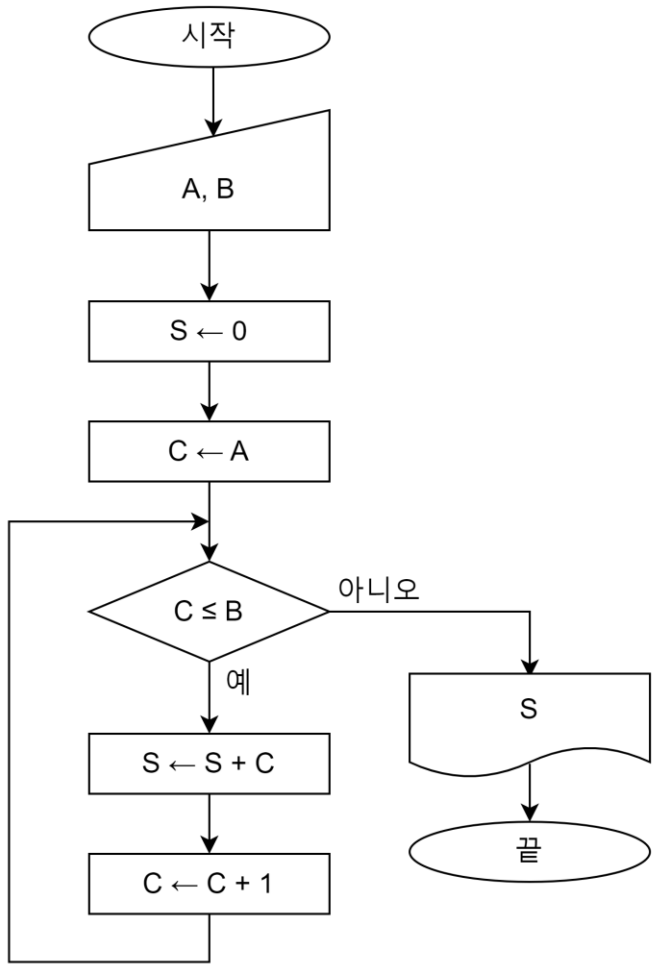
연습문제

- while문을 사용하여 두 정수 a와 b를 입력 받아 a부터 b까지 누계를 구하는 프로그램을 작성하시오.

  (a < b 라고 가정)

  - 예

```
1 10
55
```




```c
#include <stdio.h>

int main() {
    int a, b;

    return 0;
}
```

반복문 – do ... while

- 문법

```
do {
    반복할 문장1;
    반복할 문장2;
        :
}
while(반복조건);
```

- while문은 조건을 먼저 확인하고 반복 할지 결정하고,
- do…while문은 일단 한 번 해보고 반복 할지 결정한다.

- 비교

```
int main() {
    int n = 1;
    while(n < 1) {
        printf("%d \n", n);
        n++;
    }
}
```

```
int main() {
    int n = 1;
    do {
        printf("%d \n", n);
        n++;
    }
    while(n < 1);
}
```

소인수로 분해하기

- 문제

  양의 정수 한 개가 입력되었을 때, 그 수를 소인수로 분해하여 출력하는 프로그램을 작성하시오.

- 입력

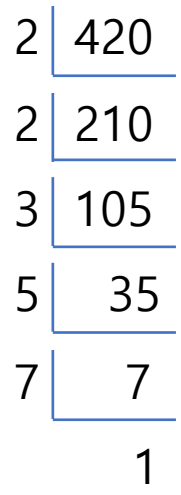
  양의 정수 한 개가 입력된다. (2이상)

- 출력

  그 수를 소인수로 분해하여 오름차순으로 출력한다.

- 입력과 출력의 예

입력 예	출력 예
420	2 2 3 5 7



- 프로그램

```
int main() {
    int n;
    scanf("%d", &n);

    int d=2;
    do {









    }
    while(d<=n);
}
```

중첩된 while

- 중첩된 while

```
while(반복조건1) {
   while(반복조건2) {


   }
}
```

  - 들여쓰기를 사용하여 반복 내용의 시작과 끝을 명확히 하자!

- 00:00 부터 ~ 05:59 까지 시간 출력

```
#include <stdio.h>

int main() {
    int hour=0, min=0;

    puts("clock time");
    while(hour < 6) {
        min = 0;
        while(min < 60) {
            printf("%02d:%02d ", hour, min);
            min++;
        }
        printf("\n");
        hour++;
    }
    return 0;
}
```

반복문 - for

- for문 형식

```
for(①초기식; ②조건식; ④증감식) {
    ③반복할 문장;
}
```

- 실행순서

1) ①초기식 ②조건식 ③반복할 문장 ④증감식
2) ②조건식 ③반복할 문장 ④증감식
3) :
4) ②조건식

- 예시

```
for(int i=0; i<3; i++) {
    printf("%d\n", i);
}
```

- 실행순서

①초기식	②조건식	③반복문장	④증감식	i
i=0	i<3	printf(0)	i++	1
	i<3	printf(1)	i++	2
	i<3	printf(2)	i++	3
	i<3			

반복문 - for

- 5회 반복 방법1

```
for(int i=0; i<5; i++) {
    printf("%d\n", i);
}
```

- 출력

```
0
1
2
3
4
```

- 5회 반복 방법2

```
for(int i=1; i<=5; i++) {
    printf("%d\n", i);
}
```

- 출력

```
1
2
3
4
5
```

반복문 - for

- 1부터 15사이 3의 배수 출력

```
#include <stdio.h>
int main() {
   for(int i=3; i<=15; i=i+3)
        printf("%d", i);
}
```

- 출력

```
3
6
9
12
15
```

- 10부터 1까지 카운트다운

```
#include <stdio.h>
int main() {
   puts("Rocket launch countdown…");
   for(int i=10; i>=1; i--)
        printf("%d ", i);
}
```

- 출력

```
Rocket launch countdown…
10 9 8 7 6 5 4 3 2 1
```

반복문 – for 문과 while 문 비교

- for 문

```c
// 1부터 5까지의 합계를 구하는 프로그램
#include <stdio.h>

int main() {
    int sum = 0;
    int n;

    for(n=1; n<=5; n++) {
        sum= sum + n;
        printf("sum of 1 to %d: %2d \n", n, sum);
    }
    return 0;
}
```

- while 문

```c
// 1부터 5까지의 합계를 구하는 프로그램
#include <stdio.h>

int main() {
    int sum = 0;
    int n;

    n=1;
    while(n<=5) {
        sum = sum + n;
        printf("sum of 1 to %d: %2d \n", n, sum);
        n++;
    }
    return 0;
}
```

3의 배수 게임

- 문제

  3의 배수 게임을 하던 정올이는 3의 배수 게임에서 작은 실수를 계속해서 벌칙을 받게 되었다.
  3의 배수 게임의 왕이 되기 위한 수련 프로그램을 작성해 보자.

  ** 3의 배수 게임이란?
  여러 사람이 순서를 정해 순서대로 수를 부르는 게임이다.
  만약 3의 배수를 불러야 하는 상황이라면, 그 수 대신 "박수"를 친다.

- 입력

  첫 줄에 하나의 정수 n이 입력된다.
  (n은 50미만의 자연수이다)

- 출력

  1부터 n까지 순서대로 공백을 두고 수를 출력하는데, 3의 배수(3, 6, 9 ...)인 경우 수 대신 영문 대문자 X 를 출력한다.

- 입력과 출력의 예

입력 예	출력 예
7	1 2 X 4 5 X 7

약수의 합 구하기

- 문제

  한 정수 n을 입력 받아서 n의 모든 약수의 합을 구하는 프로그램을 작성하시오.

  예를 들어 10의 약수는 1, 2, 5, 10이므로 이 값들의 합인 18이 10의 약수의 합이 된다.

- 입력

  첫번째 줄에 정수 n이 입력된다.

  (단, 1 <= n <= 100,000)

- 출력

  n의 약수의 합을 출력한다

- 입력과 출력의 예

입력 예	출력 예
5	6

입력 예	출력 예
10	18

- 고찰

  n의 약수들을 어떻게 알아낼 수 있을까?

약수의 합 구하기

- 문제

  한 정수 n을 입력 받아서 n의 모든 약수의 합을 구하는 프로그램을 작성하시오.

  예를 들어 10의 약수는 1, 2, 5, 10이므로 이 값들의 합인 18이 10의 약수의 합이 된다.

- 입력

  첫번째 줄에 정수 n이 입력된다.

  (단, 1 <= n <= 100,000)

- 출력

  n의 약수의 합을 출력한다

- 답안 예시

```
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);

    int ans = 0;









    printf("%d\n", ans);
    return 0;
}
```

공약수 찾기

- 문제
  - 입력된 두 자연수의 공약수를 모두 출력하는 프로그램을 작성하시오.
- 입력
  - 첫 번째 줄에 두 자연수 a와 b가 공백으로 분리되어 입력된다.
    (1 ≤ a, b ≤ 2,100,000,000)
- 출력
  - a와 b의 공약수를 작은 수부터 큰 수 순서로 공백으로 분리하여 출력한다.
- 예시

```
8 24
1 2 4 8
```

- 프로그램

```
#include <stdio.h>
int main() {
    int a, b;
    scanf("%d %d", &a, &b);




}
```

반복문 – 중첩된 for

1) 한 학급 1번 부터 30번까지 출력한다.

```
int main() {
    for(int n=1; n<=30; n++) {
        printf("%4d ", n);
    }
}
```

2) 열 개 학급에 대하여 출력한다.

```
int main() {
    for(int c=1; c<=10; c++) {
        printf("[%d반]\n", c);
        for(int n=1; n<=30; n++) {
            printf("%4d ", n);
        }
        printf("\n");
    }
}
```

3) 세 개 학년에 대하여 출력한다.

반복문 – 중첩된 for

- 중첩된 for 문을 이용하여 구구단 출력하기

```
#include <stdio.h>

int main() {
   for(int d=1; d<=5; d++) {    // 1단부터 5단까지
      printf(" %d 단\n", d);
      for(int x=1; x<=9; x++) {       // x1부터 x9까지
         printf("%d x %d = %2d \n", d, x, d*x);
      }
      printf("\n");
   }
   return 0;
}
```

```
 1 단
1 x 1 =  1
1 x 2 =  2
1 x 3 =  3
1 x 4 =  4
1 x 5 =  5
1 x 6 =  6
1 x 7 =  7
1 x 8 =  8
1 x 9 =  9

 2 단
2 x 1 =  2
2 x 2 =  4
2 x 3 =  6
2 x 4 =  8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18

 3 단
3 x 1 =  3
```

반복문 – 중첩된 for

- 연습문제

  삼각형의 밑변 길이 정수 a를 입력 받아 중첩된 반복문을 이용하여 아래 그림과 같은 직각 삼각형 모양을 출력하는 프로그램을 작성하시오.

*	1개
**	2개
***	3개
****	4개
*****	5개
******	:
*******	a개

- 정답

```
#include <stdio.h>

int main() {




}
```

```
int main(void) {
```

반복문 – 중첩된 for

- 연습문제

삼각형의 밑변 길이 정수 a를 입력 받아 중첩된 반복문을 이용하여 아래 그림과 같은 직각 삼각형 모양을 출력하는 프로그램을 작성하시오.

```
*
**
***
****
*****
******
*******
```

공백?+별n개=a개

∴ ? = a-n

- 정답

```
#include <stdio.h>

int main() {




}
```

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

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

제어문 – break

- break 문
  - switch, for, while, do ~ while문의 영역을 빠져 나오기 위해 사용
  - 가장 가까운 루프를 벗어난다.

- 사용 예
  - 1+2+3+…+n 의 합이 처음으로 100이상이 될 때, 그 때의 합과 n을 구하는 프로그램을 작성하시오.

```
#include <stdio.h>

int main() {
   int sum=0, n;

   for(n=1; true; n++) {
      sum = sum + n;
      if(sum >= 100)
         break;
   }
   printf("n: %d, sum: %d \n", n, sum);
   return 0;
}
```

소수 판별

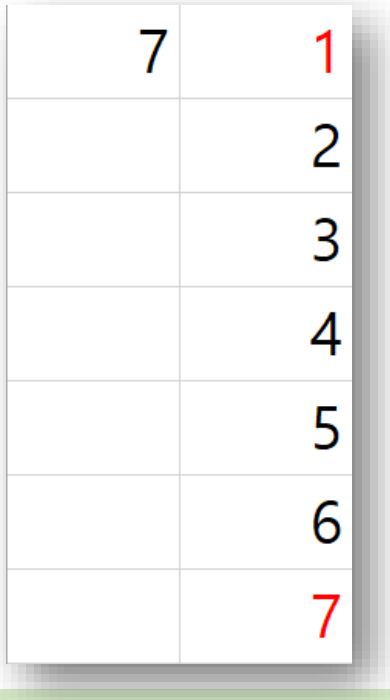
- 문제

  3 이상의 자연수(n)가 입력되었을 때, 소수 여부를 판별하는 프로그램을 작성해 보자.

  (3 ≤ n ≤ 1,000,000)

- 입력과 출력 예시

입력 예	출력 예
41	prime
111	composite



- 프로그램

```
#include <stdio.h>

int main() {
    int n;
    scanf("%d", n);

    for(d=2; d<n; d++)  {
        if(n%d==0) break;
    }
    if(d<n) printf("composite");
    else    printf("prime");
}
```

제어문 – continue

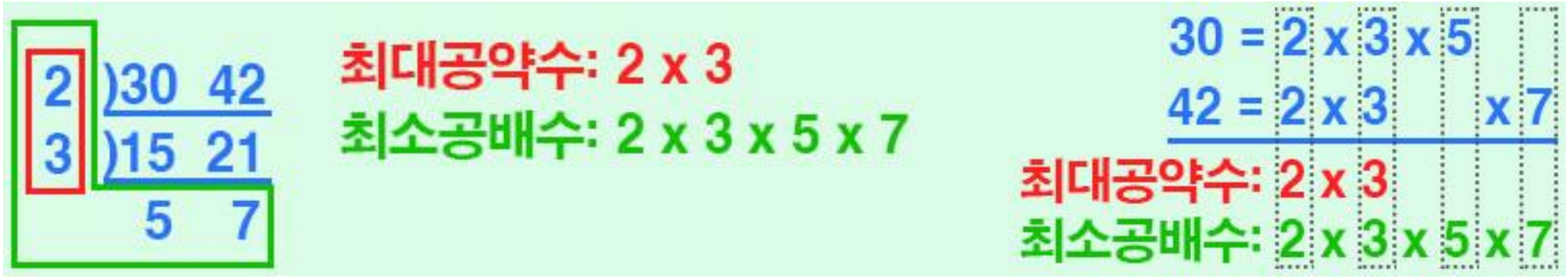
- continue 문
  - 반복문 내에서 사용되며, 남겨진 반복내용을 중단하고 다음 반복을 시작한다.

- 사용 예
  - 1부터 20까지의 정수 중에서 홀수만을 출력하시오. (for, continue 문을 사용할 것.)

```
#include <stdio.h>

int main() {
   int i;
   for(i=1; i<=20; i++)  {
      if(i%2==0)
         continue;
      printf("%3d ", i);
   }
   return 0;
}
```

최대공약수와 최소공배수

- 최대공약수
  - Greatest Common Divisor, GCD
  - 공약수: 여러 수의 공통된 약수
  - 최대공약수: 여러 수의 공약수 중 최대인 수
- 최소공배수
  - Lowest Common Multiple, LCM
  - 공배수: 여러 수의 공통된 배수
  - 최소공배수: 공배수 중 최소인 수



- G = gcd(30, 42) = 6
- L = lcm(30, 42) = 210

- A = 30, B = 42
- A × B = G × L
- 30 × 42 = 6 × 210

최대공약수 구하기

- 공약수 탐색 전략
  - a < b 조건 이용
  - 공약수는 1 부터 a 사이에 존재
  - a부터 1순서로 탐색

- 소스코드

```
int main() {
    int a, b;
    scanf("%d %d", &a, &b);

    printf("test at ");
    for(int i=a; i>=1; i--) {
        printf("%d ", i);
        if(a%i==0 && b%i==0) {
            printf("\nfound: %d\n", i);
            break;
        }
    }
}
```

- 실행결과

```
30 42
test at  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13
 12  11  10   9   8   7   6
found: 6
```

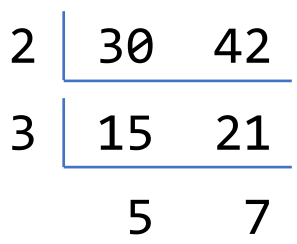
최대공약수 구하기

- 문제

  두 수 a, b를 입력 받아 최대공약수를 출력하는 프로그램을 작성하시오.

  예를 들어, 30과 42의 최대공약수는 6이다.

```
30 42
6
```

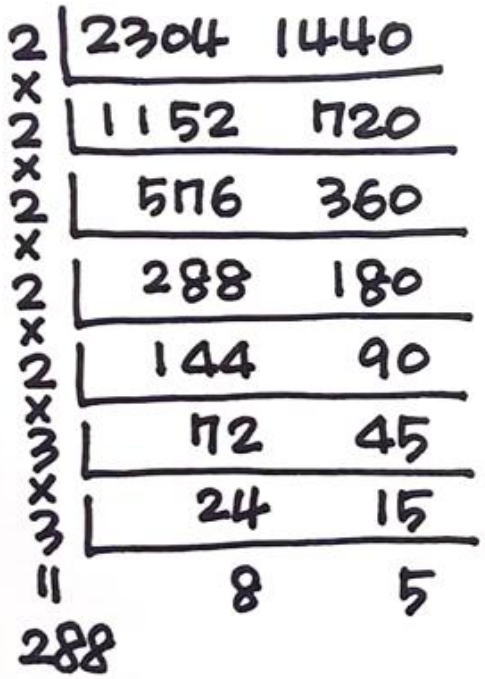


- 고찰
  - while 문이 적합한가?
  - do … while 문이 적합한가?

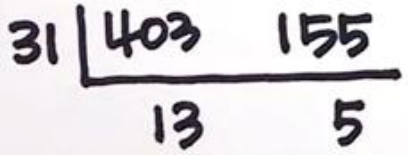
```c
#include <stdio.h>
int main() {
    int a, b;
    scanf("%d %d", &a, &b);

    int d=2, G=1;

    printf("%d\n", G);
}
```

최대공약수 구하기 <유클리드 호제법>

① 수가 크면 계산이 복잡



② 약수 찾기가 어려운 경우



a b 몫(s), 나머지(r)

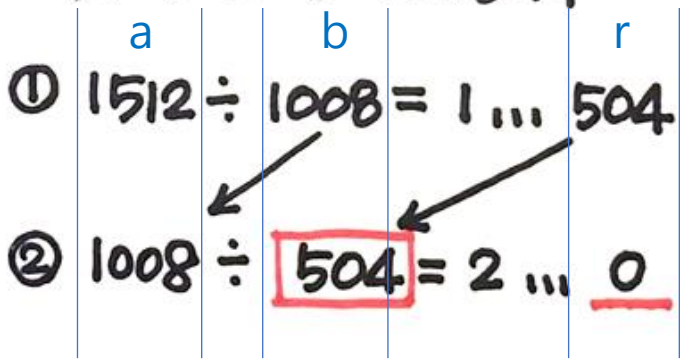
① 큰수를, 작은수로 나눈다

② 나누는 수를, 나머지로 계속 나눈다

나머지가 0이 나오면,

나누는 수가 최대공약수

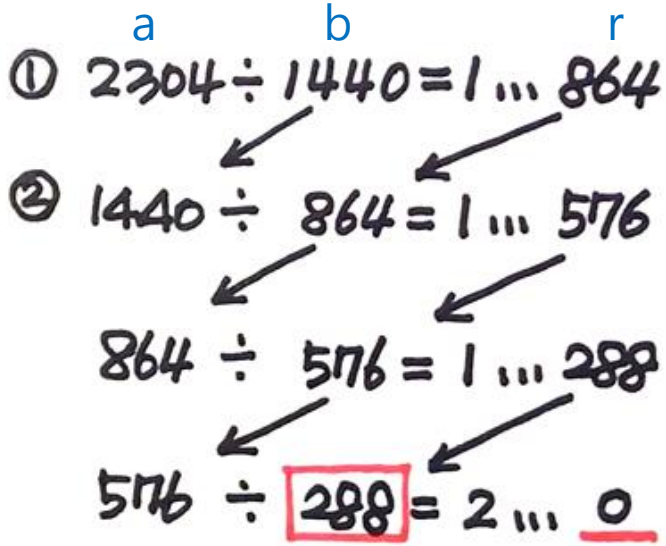
⑥ 1512, 1008 최대공약수

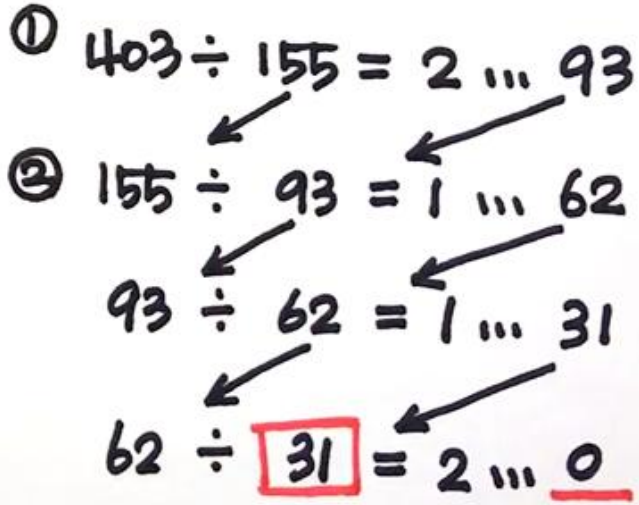


<유클리드 호제법>

a와 b의 최대공약수는

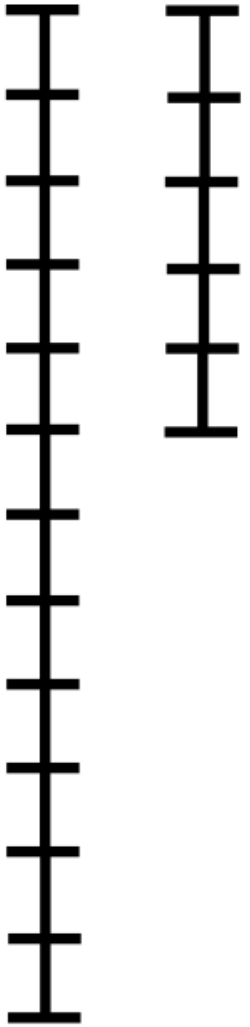
b과 r의 최대공약수와 같다.



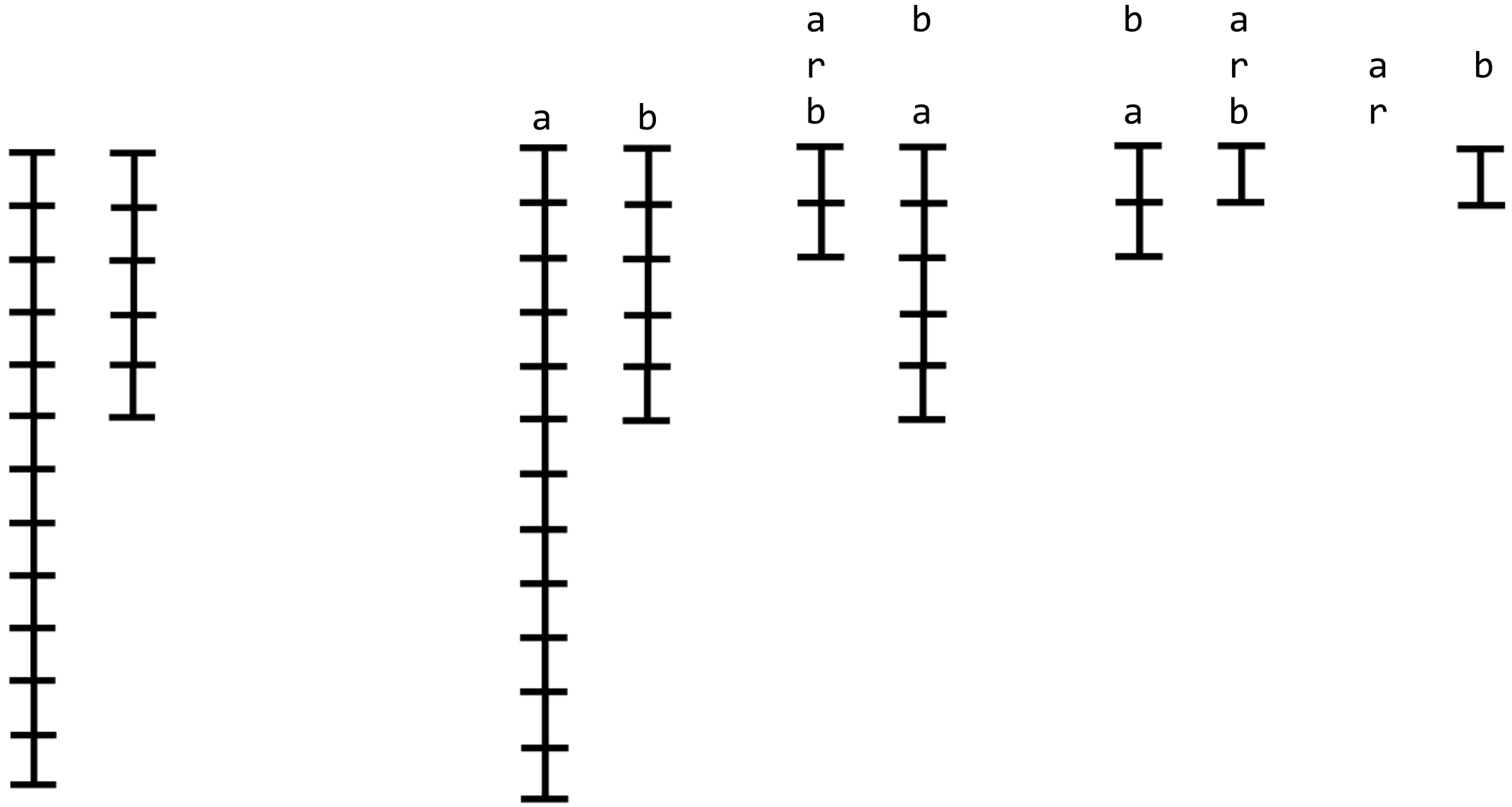


최대공약수 구하기 <유클리드 호제법>

- 왜?
  - A=1112, B= 695 일때,
  - A, B는 둘다 최대공약수(G)의 배수이다.
    - 1112 mod 695 = 417
    - 695 mod 417 = 278
    - 417 mod 278 = 139
    - 278 mod 139 = 0
  - 계산 중간결과인 417, 278, 139 도 모두 G의 배수이다.
- 그림으로 이해
  - 작은 녀석의 배수로 큰 녀석을 잘라내면 남는 녀석도 G의 배수가 된다.



최대공약수 구하기 <유클리드 호제법>



최대공약수 구하기

- 문제

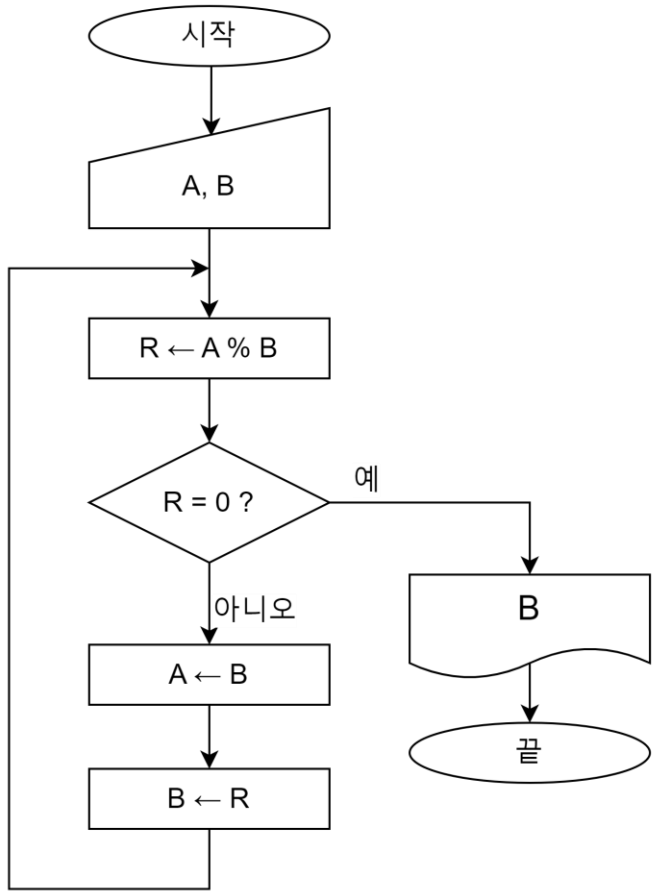
  두 수 a, b를 입력 받아 최대공약수를 출력하는 프로그램을 작성하시오.

  예를 들어, 12과 16의 최대공약수는 4이다.

```
12 16
4
```

- 고찰
  - while 문이 적합한가?
  - do … while 문이 적합한가?

- 유클리드 호제법 순서도



최대공약수 구하기 (유클리드 호제법이용)

- while 문으로 구현

```
#include <stdio.h>

int main() {
    int a, b, r;
    scanf("%d %d", &a, &b);
    while(1) {









    }
}
```

- do … while 문으로 구현

```
#include <stdio.h>

int main() {
    int a, b, r;
    scanf("%d %d", &a, &b);









}
```

최대공배수 구하기

- 문제

  두 수 a, b를 입력 받아 최대공배수를 출력하는 프로그램을 작성하시오.

  예를 들어, 6과 8의 최소공배수는 24이다.

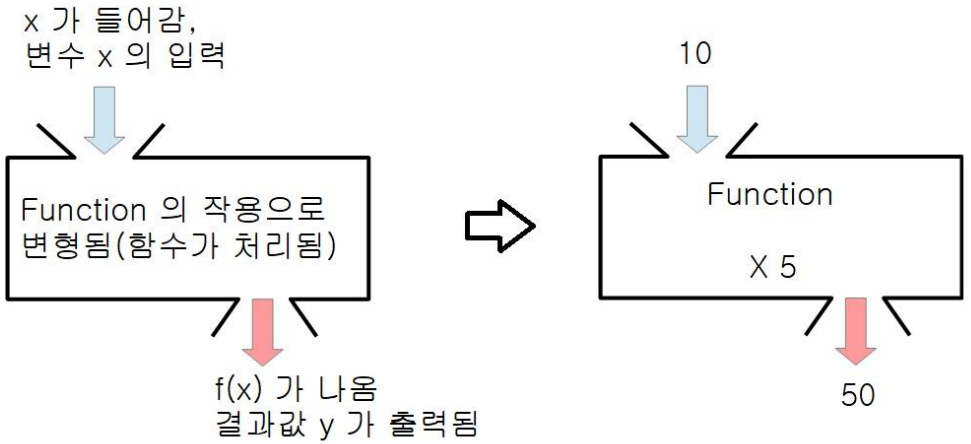
  ```
  6 8
  24
  ```

- 전략
  - 방금 전에 만든 최대공약수 프로그램을 약간 개조하자!

  - A × B = G × L

함수

- 함수(function) 란?
  - 특정한 처리.기능을 수행하는 코드를 하나로 묶어 둔 것.
  - 특정 인자를 받아 결과값을 반환하는 개체를 말하기 때문에 서브루틴(subroutine)이라고도 한다.



- 함수 사용의 효과
  - 코드들을 기능 단위로 묶을 수 있기 때문에 프로그램을 이해하고 만들기 쉽게 한다.
- 함수의 종류
  - 내장 함수
  - 사용자정의 함수
  - 매크로 함수
- 함수=프로시저=메소드

내장 함수

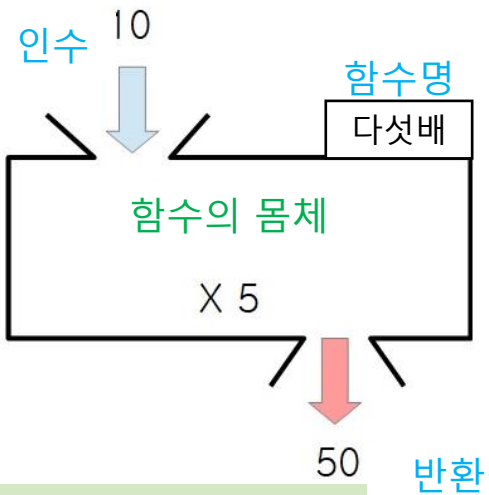
- 헤더파일의 종류

종류	기능	내장함수
stdio.h	표준 입출력 함수 등을 정의	**printf( )**, **scanf( )**, **gets( )**, getchar( ), **puts( )**, putchar( ), fgetc( ), fgets( ), fputc( ), fputs( ), fopen( ), fclose( ) 등
conio.h	직접 콘솔 입출력 함수 등을 정의	getch( ), getche( ), putch( ), cgets( ) 등
math.h	수학 함수와 매크로 정의	sin( ), cos( ), tan( ), exp( ), log( ), **sqrt( )**, **abs( )**, fabs( ), **pow( )**, fmod( ) 등
string.h	문자열 처리 함수 정의	**strlen,(), strcat( )**, **strcpy( )**, **strcmp( )**, strncat( ) 등
ctype.h	문자 검사 매크로 정의	isalpha( ), islower( ), isupper( ), tolower( ), toupper( ) 등

사용자 정의 함수

- 형식

```
[함수의 리턴형] 함수명([인수1,인수2…])
{
    문장1;
    문장2;
     …
     …
    문장n;
    [return] [리턴값]
}
```



- 예

```
[리턴형] 함수명 (인수)
int    main ( ) {
     함수의 몸체
}

int add(int a, int b) {
    int sum = a + b;
    return sum;
}

char upper(char ch) {
     return ch-32;
}
```

사용자 정의 함수

- 함수의 형태
  - 인수
    - 없는가?
    - 있는가? 있다면 한 개인가, 두 개인가?
  - 리턴값
    - 없는가?
    - 있는가? (한 개만 리턴 가능)
  - 다양한 형태의 함수 모양이 나올 수 있음

- 형태

인수	리턴	형태
X	X	void func() void func(void)
X	O	int func() double func(void)
O	X	void func(int ar) void func(char c)
O	O	int func(int ar) int func(int a, int b)

사용자 정의 함수

- Case1: 인수 X, 리턴값 X
  - 기능: "Copyright" 출력
  - 함수명: output
  - 인수: 없음
  - 리턴값: 없음

- 함수 구현

```
void output() {
    printf("---------------\n");
    printf(" function test\n");
    printf("---------------\n");
    return;
}
```

사용자 정의 함수

- Case2: 인수 O, 리턴 O
  - 기능: x+y 값을 구한다
  - 함수명: add
  - 인수: x, y 2개

    x:int, y:int
  - 리턴값: x+y

    리턴형: int

- 함수 구현

```
int add(int x, int y) {
    int sum = x + y;
    return sum
}
```

사용자 정의 함수

- 함수의 호출

```
#include <stdio.h>
int add(int x, int y) {
    int sum = x + y;
    return sum;
}

int main() {
    int a=3, b=4;
    int sum=0;

    sum = add(a, b);
    printf("%d \n", sum);
}
```

- 메모리에서 일어나는 일
  - 지역변수는 함수 호출 시 생성 되고, 함수가 종료되면 자동으로 파괴된다.

소속	변수	값
add	sum	7
	y	4
	x	3

copy

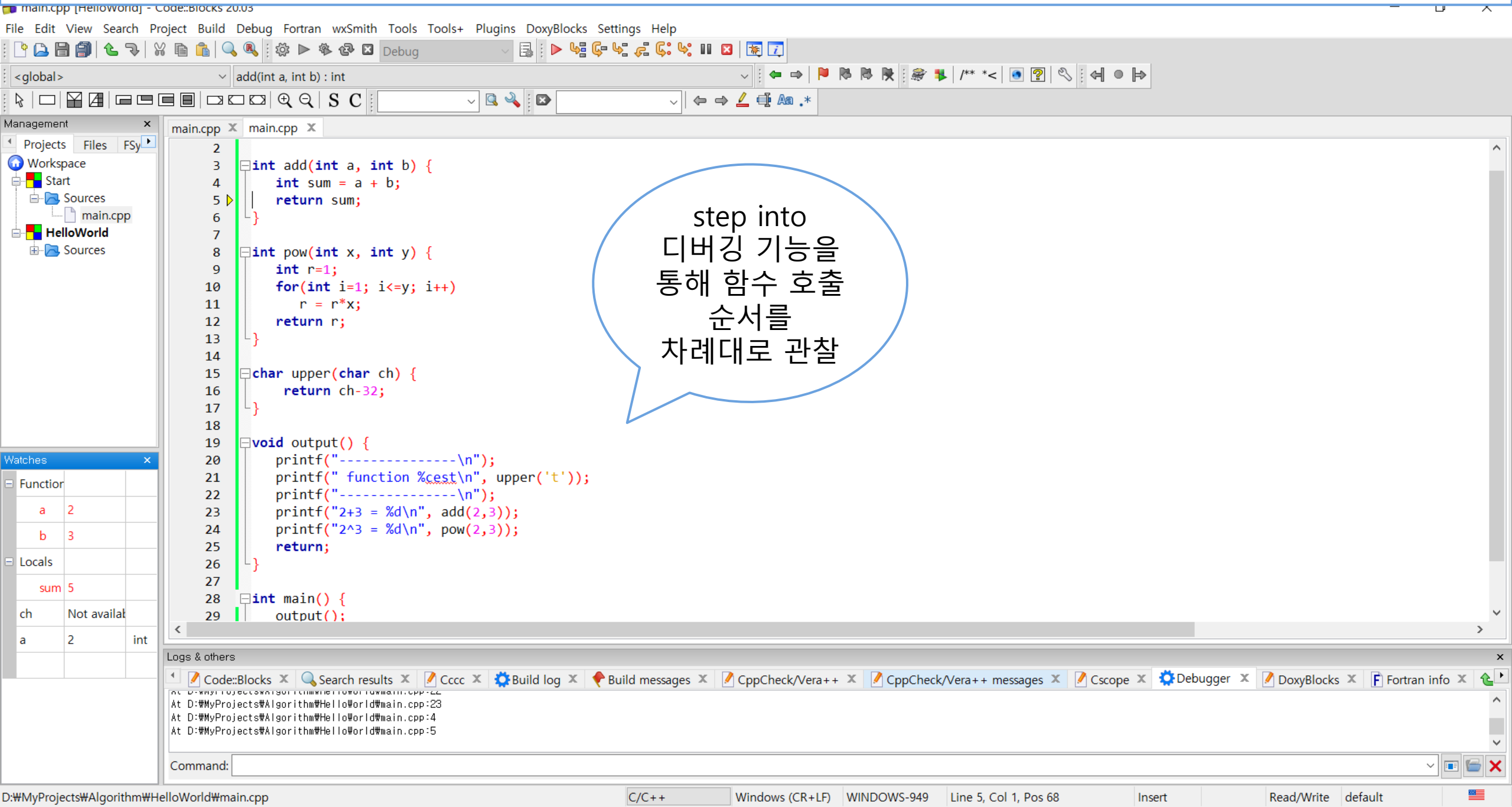
소속	변수	값
main	sum	7
	b	4
	a	3

사용자 정의 함수

```c
#include <stdio.h>

int add(int a, int b) {
   int sum = a + b;
   return sum;
}

int pow(int x, int y) {
   int r=1;
   for(int i=1; i<=y; i++)
      r = r*x;
   return r;
}

char upper(char ch) {
    return ch-32;
}
```

```c
void output() {
   printf("---------------\n");
   printf(" function %cest\n", upper('t')); //함수호출
   printf("---------------\n");
   printf("2+3 = %d\n", add(2,3)); //함수호출
   printf("2^3 = %d\n", pow(2,3)); //함수호출
   return;
}

int main() {
   output();  //함수호출
}
```

Debugging: Step into [shift]+F7



사용자 정의 함수

함수의 프로토타입을 미리 알려준다

```c
#include <stdio.h>

int add(int a, int b);
int pow(int x, int y);
char upper(char ch);

void output() {
   printf("---------------\n");
   printf(" function %cest\n", upper('t'));
   printf("---------------\n");
   printf("2+3 = %d\n", add(2,3));
   printf("2^3 = %d\n", pow(2,3));
   return;
}

int main() {
   output();
}
```

```c
int add(int a, int b) {
   int sum = a + b;
   return sum;
}

int pow(int x, int y) {
   int r=1;
   for(int i=1; i<=y; i++)
      r = r*x;
   return r;
}

char upper(char ch) {
    return ch-32;
}
```

Logs & others

Code::Blocks | Search results | Cccc | Build log

File	Line	Message
D:₩MyProjects...	5	error: 'upper' was not declared in this scope
D:₩MyProjects...	5	note: suggested alternative: '_popen'
D:₩MyProjects...	7	error: 'add' was not declared in this scope
D:₩MyProjects...	8	error: 'pow' was not declared in this scope
D:₩MyProjects...	8	note: suggested alternative: 'putw'

함수 만들기 연습

- 정수 k를 넘겨받아 별(*)을 k개 출력하는 함수 kstars(k)

- 왼쪽의 kstars(k)를 이용하여 * 로 삼각형 그리기

```
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);



}
```

```
5
*
**
***
****
*****
```

함수 만들기 연습

- 두 실수 a, b값을 받아 두 수의 차이(절대값)를 반환하는 함수

```
____ diff(____ a, ____ b) {


}
```

- 두 자연수 a와 b를 입력 받아 a^b를 계산하는 함수

```
____ power(____ a, ____ b) {


}
```

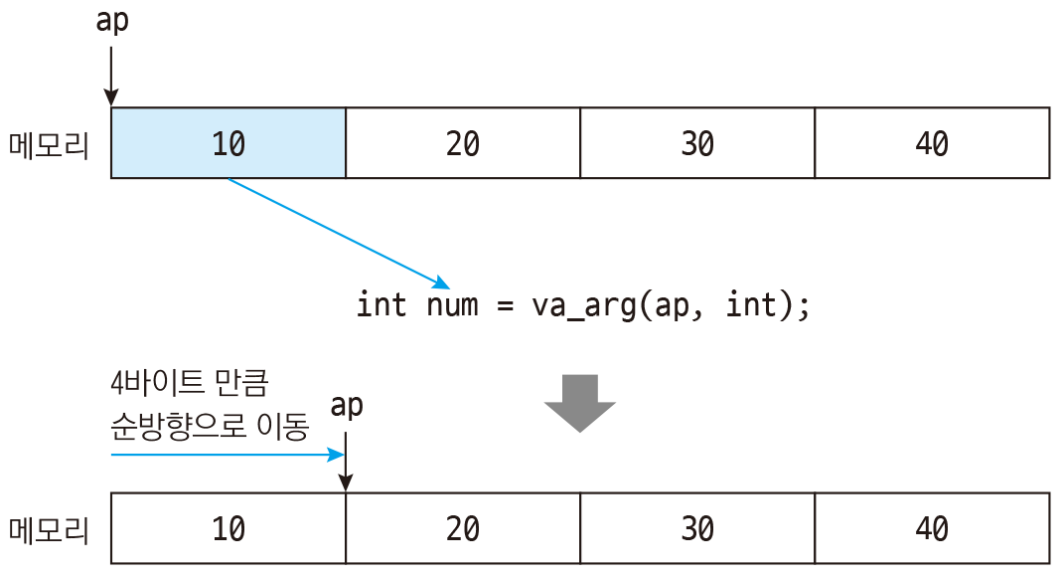
함수 만들기 연습

- 두 정수 a, b값을 받아 큰 수를 반환하는 max 함수

- 두 정수 a, b값을 받아 작은 수를 반환하는 min 함수

가변인자 함수

```c
#include <stdio.h>
#include <stdarg.h>    // va_list, va_start, va_arg, va_end가 정의된 헤더 파일

void printNumbers(int args, ...)  {  // 가변 인자의 개수를 받음, ...로 가변 인자 설정
    va_list ap;    // 가변 인자 목록 포인터

    va_start(ap, args);    // 가변 인자 목록 포인터 설정
    for (int i = 0; i < args; i++)  {   // 가변 인자 개수만큼 반복
        int num = va_arg(ap, int);    // int 크기만큼 가변 인자 목록 포인터에서 값을 가져옴
                                      // ap를 int 크기만큼 순방향으로 이동
        printf("%d ", num);           // 가변 인자 값 출력
    }
    va_end(ap);    // 가변 인자 목록 포인터를 NULL로 초기화
    printf("\n");    // 줄바꿈
}

int main() {
    printNumbers(1, 10);                // 인수 개수 1개
    printNumbers(2, 10, 20);            // 인수 개수 2개
    printNumbers(3, 10, 20, 30);        // 인수 개수 3개
    printNumbers(4, 10, 20, 30, 40);    // 인수 개수 4개
    return 0;
}
```



가변인자 함수

- 최소값을 알려주는 mins()

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

int mins(int args, ...) {
    va_list ap;
    va_start(ap, args);
    int M = va_arg(ap, int);
    for (int i=1; i < args; i++)  {
        int num = va_arg(ap, int);
        if(M > num) M = num;
    }
    va_end(ap);
    return M;
}

int main() {
    printf("%d\n", mins(2, 4, 3));
    printf("%d\n", mins(3, 8, 2, 6));
    printf("%d\n", mins(4, 9, 4, 6, 3));
    printf("%d\n", mins(5, 2, 4, 6, 1, 8));
}
```

- 최대값을 알려주는 maxs()

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

int mins(int args, ...) {
    va_list ap;
    va_start(ap, args);
    int M = va_arg(ap, int);
    for (int i=1; i < args; i++)  {
        int num = va_arg(ap, int);
        if(M > num) M = num;
    }
    va_end(ap);
    return M;
}

int main() {
    printf("%d\n", mins(2, 4, 3));
    printf("%d\n", mins(3, 8, 2, 6));
    printf("%d\n", mins(4, 9, 4, 6, 3));
    printf("%d\n", mins(5, 2, 4, 6, 1, 8));
}
```

매크로 함수

- 최대값, 최소값 함수

```
#include <stdio.h>

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

int main() {
    int x=2, y=3;
    int m=MIN(x, y);
    int M=MAX(x, y);
    printf("m: %d, M: %d\n", m, M);
}
```

- 매크로 함수의 장점
  - 매크로 함수는 단순 치환만을 해주므로, 인수의 타입을 신경 쓰지 않습니다.
  - 함수 호출에 의한 성능 저하가 일어나지 않으므로, 프로그램의 실행속도가 향상됩니다.
- 단점
  - 원하는 결과를 얻는 정확한 매크로 함수의 구현은 어려우며, 따라서 디버깅 또한 매우 어렵습니다.
  - 매크로 함수의 크기가 증가하면 증가할수록 사용되는 괄호 또한 매우 많아져서 가독성이 떨어집니다.

지역변수 / 전역변수

- 지역변수
  - 함수 안에서 선언된 변수
  - 해당 함수 안에서만 사용가능
  - 초기값이 쓰레기 값이다
  - 함수가 호출되면 생성되고 함수가 종료되면 사라진다
  - 동일한 이름의 전역/지역변수가 존재하면 지역변수가 우선한다
  - 스택에 저장
- 전역변수
  - 함수 외부에서 선언된 변수
  - 어느 함수에서든 사용가능
  - 초기값이 `0` 이다
  - 프로그램이 실행 중이면 항상 존재한다
  - 프로그램을 이해하기 어렵게 만드므로 꼭 필요한 경우에만 사용하자
  - 전역공간에 저장

지역변수 / 전역변수

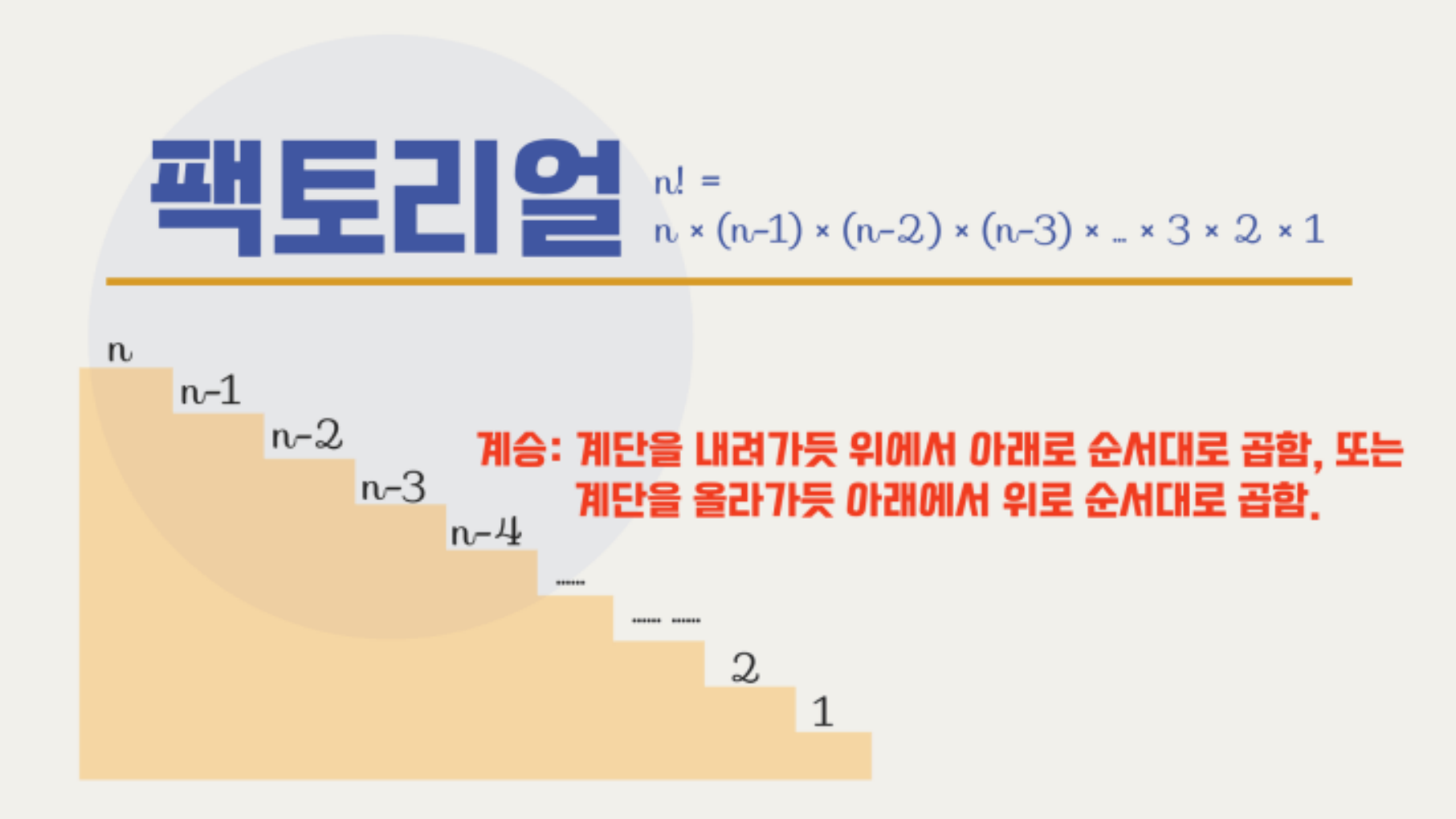
- 지역변수 예시

```c
#include <stdio.h>
// add의 sum과 main의 sum은 동명이인

int add(int a, int b) {
    int sum = a + b;
    return sum;
}

int main(){
    int sum=0;
    add(1, 2);
    printf("%d\n", sum);
}
```

- 전역변수 예시

```c
#include <stdio.h>
// add의 sum과 main의 sum은 동일변수

int sum;
void add(int a, int b) {
    sum = a + b;
}

int main(){
    add(1, 2);
    printf("%d\n", sum);
}
```

팩토리얼 계산


팩토리얼
n! =
n × (n-1) × (n-2) × (n-3) × … × 3 × 2 × 1
n
n-1
n-2
n-3
n-4
……
…… ……
2
1
계승: 계단을 내려가듯 위에서 아래로 순서대로 곱함, 또는
계단을 올라가듯 아래에서 위로 순서대로 곱함.

팩토리얼 계산

- 비 재귀적 해결
  - ex) 팩토리얼 계산

```
n! = n x (n-1) x (n-2) x … x 1

5! = 5 x 4 x 3 x 2 x 1 = 120

3! = 3 x 2 x 1 = 6
```

- 답안

```
#include <stdio.h>
int factorial(int n) {



                    ?



    return r;
}
int main() {
    printf("6! = %d\n", factorial(6));
}
```

재귀함수

- 재귀함수
  - 실행 도중 자기 자신을 호출(재귀 호출)하는 함수
  - ex) 팩토리얼 계산

```
5! = 5 x 4 x 3 x 2 x 1
5! = 5 x 4!
        4! =4x3!
              3!=3x2!
                    2!=2x1!
                          1!=1
```

$$f(n) \begin{cases} n \times f(n-1) & \dots\ n \geq 2 \\ 1 & \dots\ n = 1 \end{cases}$$

- 함수 예시
  - 탈출조건이 없으면 무한루프가 되므로 유의

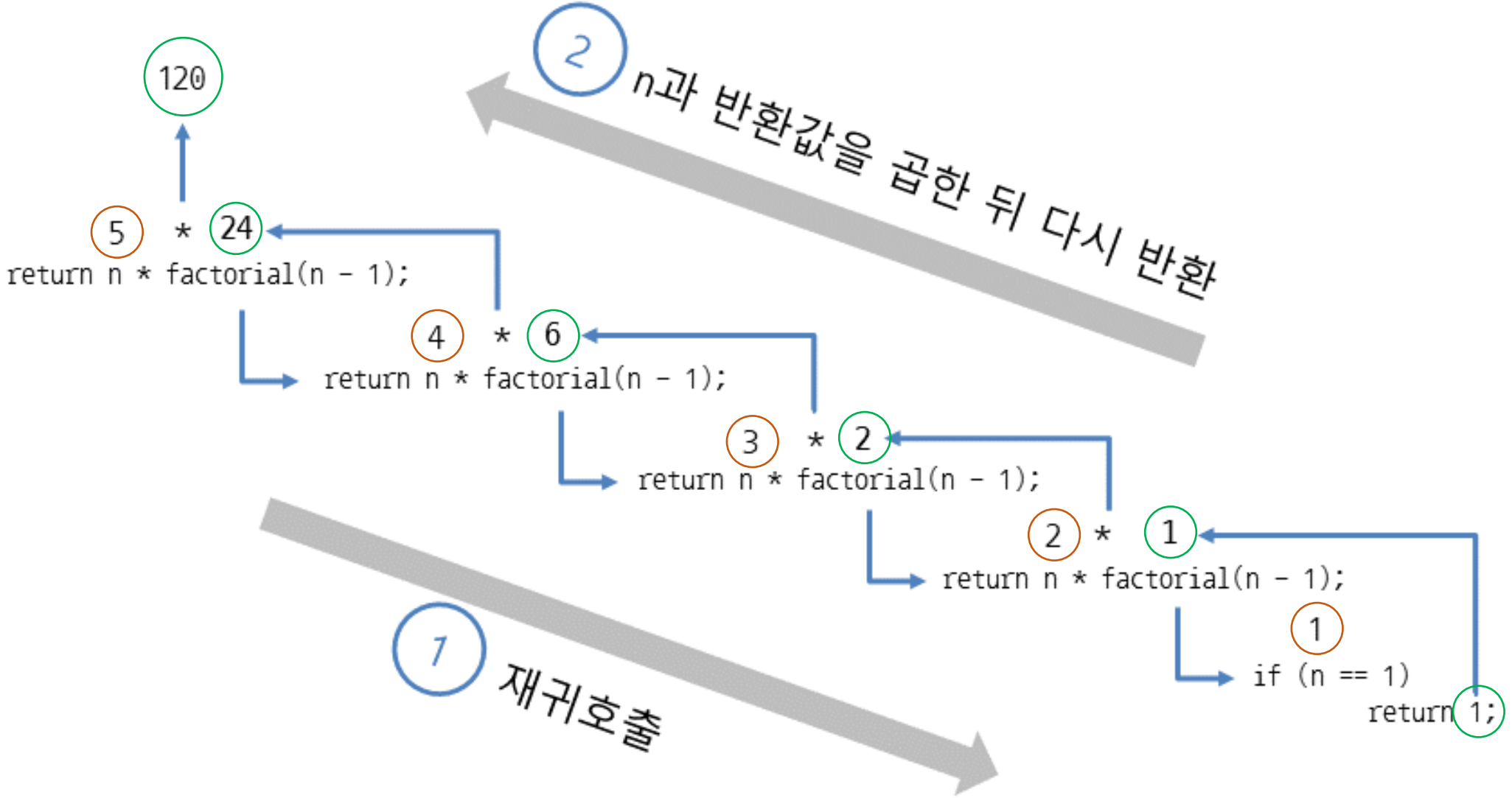
```
int factorial(int n) {
   if(n >= 2)
      return n*factorial(n-1);
   else
      return 1;
}

// 3항 조건 연산자를 활용하여
// 아래와 같이 표현해도 동일한 효과
int factorial(int n) {
   return (n>=2)? n*factorial(n-1) : 1;
}
```

재귀함수

factorial(5)

계산과정

묘사

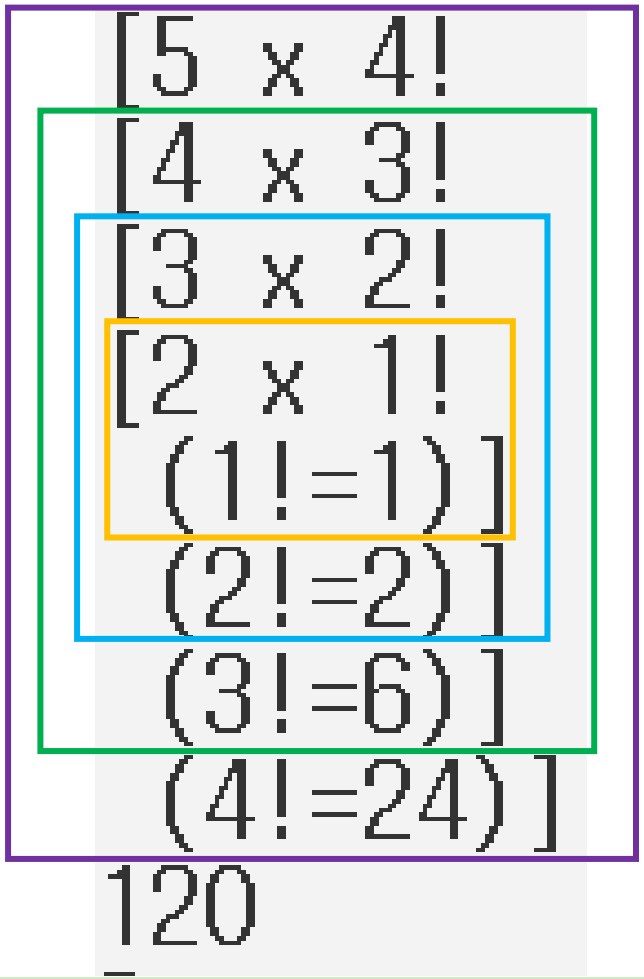


재귀함수

- 재귀함수 호출 관찰

```
#include <stdio.h>

int fact(int n) {
    if(n >= 2) {
        printf("[%d x %d!\n", n, n-1);
        int f = fact(n-1);
        printf(" (%d!=%d)]\n", n-1, f);
        return n*f;
    }
    else
        return 1;
}

int main() {
    printf("%d", fact(5));
}
```

- 함수 예시



연습문제

- 피보나치 수 찾기
  - 첫째 항 및 둘째 항이 1이며, 그 뒤의 모든 항은 바로 앞 두 항의 합인 수열
  - 처음 여섯 항은 각각 1, 1, 2, 3, 5, 8 이다.
  - 숫자 K를 입력 받아 K번째에 해당하는 피보나치 수를 출력하는 알고리즘을 작성하시오.

- 함수 구현

```
int fibo(int n) {



}
```

재귀 함수 – 연습문제1

- 피보나치 수 찾기
  - 첫째 항 및 둘째 항이 1이며, 그 뒤의 모든 항은 바로 앞 두 항의 합인 수열
  - 처음 여섯 항은 각각 1, 1, 2, 3, 5, 8 이다.
  - 숫자 K를 입력 받아 K번째에 해당하는 피보나치 수를 출력하는 알고리즘을 작성하시오.

- 함수 구현

```
int fibo(int n) {



}
```

재귀 함수 – 연습문제2

- 계단을 오르는 방법
  - 계단을 한 번에 한 칸 또는 두 칸 만 오를 수 있다고 할 때 n 칸으로 되어 있는 계단 전체를 오르는 방법은 몇 가지가 있는가?

  - 힌트1
    - 1칸 계단: 1가지 방법
    - 2칸 계단: 2가지 방법
    - 3칸 계단은?

  - 힌트2: n칸 계단에 오르는 방법
    - n-2칸 까지 올라온 다음 두 칸 오른다 +
    - n-1칸 까지 올라온 다음 한 칸 오른다

- 함수로 표현

예를 들어,

f(n) : n개의 계단일 때 오르는 방법의 수

```
f(1) = 1
f(2) = 2
f(3) = f(2) + f(1)
f(4) = f(3) + f(2)
f(5) = f(4) + f(3)
        :
f(n) = f(n-1) + f(n-2)
```

연습문제 풀이: 계단오르기

- 고찰

계단	오르는 방법		방법 개수	
(1)	①	①	1	1
(2)	①+① ②	(1)+① ②	1 1	2
(3)	①+② ①+①+①, ②+①	(1)+② (2)+①	1 2	3
(4)	①+①+②, ②+② ①+②+①, ①+①+①+①, ②+①+①	(2)+② (3)+①	2 3	5
(5)	①+②+②, ①+①+①+②, ②+①+② ①+①+②+①, ②+②+①, ①+②+①+①, …	(3)+② (4)+①	3 5	8
(6)	생략 생략	(4)+② (5)+①	5 8	13

- 점화식 표현

```
f(1) = 1
f(2) = 2
f(3) = 3
f(n) = f(n-2)+f(n-1)
```

연습문제 풀이: 계단오르기

```
#include <stdio.h>

int count(int stairs) {




}

int main() {
   int stairs;
   scanf("%d", &stairs);
   printf("%d\n", count(stairs));
}
```

- 함수로 표현

예를 들어,

f(n) : n개의 계단일 때 오르는 방법의 수

f(1) = 1

f(2) = 2

f(3) = f(2) + f(1)

f(4) = f(3) + f(2)

f(5) = f(4) + f(3)

:

f(n) = f(n-1) + f(n-2)

배열

- 배열
  - 같은 형식의 여러 데이터를 하나의 변수에 긴 띠 모양으로 저장하여 사용하는 자료의 집합체
  - 줄줄이 연결된 타입이 동일한 변수들의 집합

- 선언
  - 형식

    데이터형 배열명[원소의 수]

  - 선언 예

    int kor[5];

    kor변수 5개 만듦

    kor[0] ~ kor[4]

  - 배열의 구조
    - 0번 인덱스부터 시작됨에 유의

kor[0]	kor[1]	kor[2]	kor[3]	kor[4]

배열

- 대입

```
kor[0] = 60;
kor[1] = 60;
kor[2] = 60;
kor[3] = 60;
kor[4] = 60;
```

- 반복문 이용한 대입

```
for(i=0; i<5; i++)
    kor[i] = 60;
```

- 초기화

```
int a[5] = {3,2,7,6,9};
int b[] = {3,6,5,4};
int c[5] = {5,8,3};
int d[5] = {4,};
static int e[5];
```

배열

- 배열의 순회1

```
#include <stdio.h>

int main() {
  int a[10]={1,3,7,6,4,8,9,12,2,10};

  // 0번부터 시작하여 n-1에서 끝남에 유의
  for(int i=0; i<10; i++) {
    printf("%4d", a[i]);
  }
  return 0;
}
```

- 배열의 순회2

```
#include <stdio.h>

int main() {
  int a[10]={1,3,7,6,4,8,9,12,2,10};
  int i;
  // 0번부터 시작하여 n-1에서 끝남에 유의
  i=0;
  while(i<10) {
    printf("%4d", a[i]);
    i++;
  }
  return 0;
}
```

배열

- 배열의 합

```
#include <stdio.h>

int main() {
  int i, sum=0;
  int a[10]={1,3,7,6,4,8,9,12,2,10};

  for(i=0; i<10; i++) {
    sum = sum + a[i];
  }
  printf("sum = %d\n", sum);
}
```

- 피보나치수

```
#include <stdio.h>

int main() {
  int i, fibo[10]={1,1};

  for(i=2; i<10; i++) {
    fibo[i]=fibo[i-1]+fibo[i-2];
  }

  for(i=0; i<10; i++) {
    printf("%4d\n", fibo[i]);
  }
}
```

입력된 자연수 개수 출력하기

- 문제

  1~6 범위의 n개의 자연수가 입력되었을 때, 각 수가 입력된 개수를 출력하는 프로그램을 작성해 보자.

- 입력
  - 첫 줄에 자연수의 갯수 n이 입력된다.

    (1 <= n <= 1,000,000)
  - 두 번째 줄에 n 개의 자연수가 공백을 두고 입력된다.

- 출력

  1~6까지 각 자연수가 입력된 개수를 공백으로 분리하여 출력한다.

- 입력과 출력의 예

입력 예	출력 예
10 4 3 2 5 3 1 4 6 2 3	1 2 3 2 1

입력된 자연수 개수 출력하기(풀이)

- 소스코드

```c
#include <stdio.h>

int main() {
    // 몇 번 입력되었는지 저장하기 위한 배열
    int cnts[7] = {0,};
    int n, s;
    scanf("%d", &n);

    for(int i=0; i<n; i++) {  // n회 반복
        scanf("%d", &s);  // 입력된 숫자 s
        cnts[s]++;  // 입력된 숫자 카운트
    }
    // 1부터 6까지 입력횟수 결과출력
    for(int i=1; i<=6; i++)
        printf("%d ", cnts[i]);
}
```

- cnts 배열

idx	0	1	2	3	4	5	6
val	0	0	0	0	0	0	0

- 4 입력

idx	0	1	2	3	4	5	6
val	0	0	0	0	1	0	0

- 3 입력

idx	0	1	2	3	4	5	6
val	0	0	0	1	1	0	0

- 핵심코드
  - `cnts[s]++;  //s번 idx의 값을 증가`

숫자 목록에서 수 찾기

- 문제

  n개로 이루어진 정수 목록에서 원하는 수의 위치를 찾으시오.
  단, 입력되는 정수 목록에 같은 수는 없다.

- 입력

  첫 줄에 한 정수 n이 입력된다.
  (2 <= n <= 100,000)
  둘째 줄에 n개의 정수가 공백으로 구분되어 입력된다.
  (입력되는 모든 정수는 21억 보다 작다)
  셋째 줄에는 찾고자 하는 수가 입력된다.

- 출력

  찾고자 하는 원소의 위치를 출력한다.
  없으면 -1을 출력한다.

- 입력과 출력의 예

입력 예	출력 예
8 1 2 3 5 7 9 11 15 11	7

숫자 목록에서 수 찾기(풀이)

```
#include <stdio.h>
int main() {
    int n;  // 입력되는 자료 개수
    scanf("%d", &n);

    int nums[n+1];      // 자료가 저장되는 공간
    for(int i=1; i<=n; i++)    // n회 반복
        scanf("%d", &nums[i]);

    int s;  // 찾을 수
    scanf("%d", &s);

    for( … ) {  // n회 반복
        if( … ) {    //nums배열에서 s를 찾으면,
            printf("%d\n", i);  // 그 위치를 출력
            return 0;
        }
    }
    printf("%d\n", -1);
}
```

- nums 배열

idx	0	1	2	3	4	5	6	7	8
val	x								

- 데이터 입력 후

idx	0	1	2	3	4	5	6	7	8
val	x	1	2	3	5	7	9	11	15

최댓값 찾기

- 문제

  9개의 서로 다른 자연수가 주어질 때, 이들 중 최댓값을 찾고 그 값이 몇 번째 수 인지를 구하는 프로그램을 작성하시오. 예를 들어, 서로 다른 9개의 자연수가 각각 3, 29, 38, 12, 57, 74, 40, 85, 61 라면, 이 중 최댓값은 85이고, 이 값은 8번째 수이다

- 입력

  첫째 줄부터 아홉째 줄까지 한 줄에 하나의 자연수가 주어진다. 주어지는 자연수 는 100 보다 작다.

- 출력

  첫째 줄에 최댓값을 출력하고, 둘째 줄에 최댓값이 몇 번째 수인지를 출력한다.

- 입력과 출력의 예

입력 예	출력 예
3 29 38 12 57 74 40 85 61	85 8

- 출처

  한국정보올림피아드(2007 지역본선 초등부)

최댓값 찾기(풀이)

```c
#include <stdio.h>

int main() {
    int nums[10];
    for(int i=1; i<=9; i++)
        scanf("%d", &nums[i]);

    // 첫 번째 원소를 최댓값이라고 가정하고 시작
    int max = nums[1];
    int idx = 1;

    for( … ) {  // 배열 내 나머지 원소들에 대하여
        if( … ) {  // 더 큰 수를 발견하면,
            max = nums[i];
            idx = i;
        }
    }
    printf("%d\n", max);  // 최댓값 출력
    printf("%d\n", idx);  // 몇 번째 수인지 출력
}
```

- nums 배열

idx	0	1	2	3	4	5	6	7	8	9
val	x									

- 데이터 입력 후

idx	0	1	2	3	4	5	6	7	8	9
val	x	3	29	38	12	57	74	40	85	61

- 최대값 탐색

max	max	max	max	max	max	max	max	max
3	29	38	38	57	74	74	85	85

2진수로 변환하기

- 문제

  10진수 n이 입력되었을 때, 2진수로 변환해 출력하는 프로그램을 작성해 보자.

- 입력

  첫 줄에 10진수 n이 입력된다.

  (0 ≤ n ≤ 100,000,000)

- 출력

  10진수를 2진수로 변환한 결과를 출력한다.

- 입력과 출력의 예

입력 예	출력 예
11	1011

- 출처

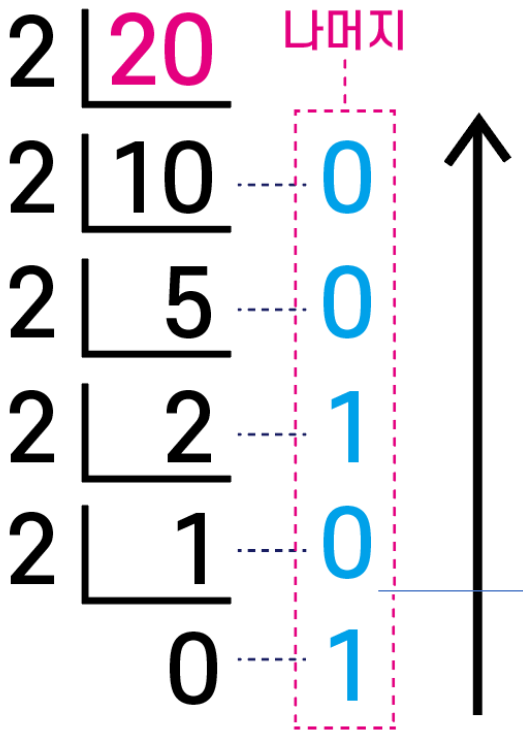
  정보과학 교과서 p.83

2진수로 변환하기(풀이)

- 2진수 변환

p

i	0	1	2	3	4	5	6	7	8
d[i]	0	0	1	0	1				



2로 나눈 나머지 구하기

- 소스코드

```
#include <stdio.h>

int main() {
    int d[32];  // 변환결과를 저장할 공간
    int n;
    scanf("%d", &n);

    int p=0; //0번 인덱스부터~
    do {
        ________;  //2로 나눈 나머지 저장
        _______;      //p를 다음으로 이동
        _______;      //2로 나눈 몫 계산
    }
    while(n>0); //아직 0이 되지 않았다면 계속

    // d배열의 p-1번 인덱스부터 역순으로 출력
    for(int i=p-1; i>=0; i--)
        printf("%d", d[i]);
}
```

배열

- 에라토스테네스의 체

isPrime[]

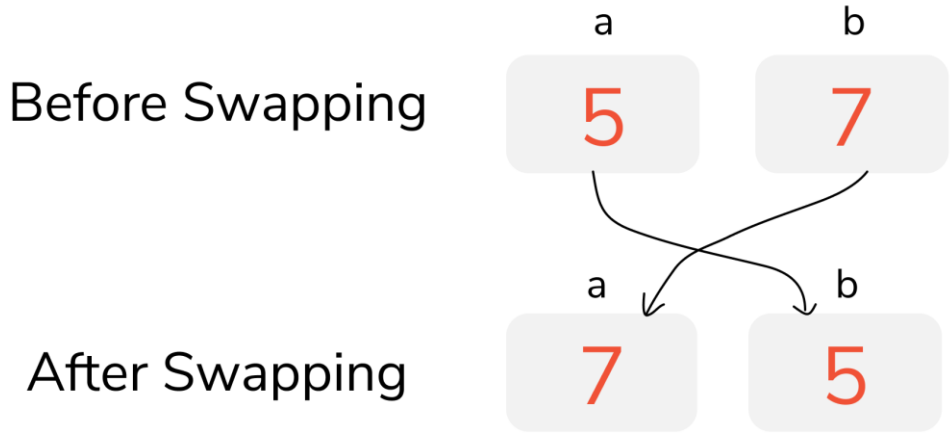
idx	0	1	2	3	4	5	6	…
val	1	1	0	0	0	0	0	…

- 소스코드

```
#include <stdio.h>
#define MAX  101
int isPrime[MAX] = {1, 1, 0, };
// 1: 소수아님, 0: 소수
int main() {
  int cnt=0;
  for(int i=2; i<MAX; i++) {
    if(isPrime[i]==0) { // 현재수만 소수이고
      printf("%5d", i);
      cnt++;
      for(int j=i; j<MAX; j+=i) // 배수들은
         isPrime[j]=1;  // 소수아님으로 셋팅
    }
  }
  printf("1~100 %d개의 소수를 찾아냄\n", cnt);
}
```

SWAP

- 두 변수 내용물을 서로 교환하는 연산



- 잘못된 구현

```
#include <stdio.h>

int main() {
   int a=5, b=7;
   printf("%d %d\n", a, b);

   a=b;
   b=a;

   printf("%d %d\n", a, b);
   return 0;
}
```

```
5 7
7 7
```

SWAP

- 두 변수 내용물을 서로 교환하는 연산
- 임시 변수가 필요함.

- 올바른 구현

```
#include <stdio.h>

int main() {
   int a=5, b=7;
   printf("%d %d\n", a, b);

   int t=a;
   a=b;
   b=t;

   printf("%d %d\n", a, b);
   return 0;
}
```

```
5 7
7 5
```

SWAP

- 고급 구현

```
#include <stdio.h>

// C++의 Generic과 참조자를 사용
template <class T>
inline void SWAP(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

제네릭과 참조자는 본 수업의 범위를 벗어나는 내용이므로 자세한 설명은 생략한다.

```
int main() {
    int a=5, b=7;
    printf("%d %d\n", a, b);

    SWAP(a, b);

    printf("%d %d\n", a, b);
    return 0;
}
```

```
5 7
7 5
```

정렬 알고리즘

- 선택 정렬
  - 각 회전마다 최소 값을 찾아 1번째부터 차례대로 배열
  - 1회전이 종료될 때 마다 가장 앞쪽부터 차례대로 숫자가 결정됨

  - 예
    - 4 3 6 2 5 를 선택정렬

- 1회전 (첫 번째 숫자를 결정하기 위함)

4 3 6 2 5 (교환)

3 4 6 2 5

3 4 6 2 5 (교환)

2 4 6 3 5

- 2회전 (두 번째 숫자를 결정하기 위함)

2 4 6 3 5 (최소값을 제외하고 2번째부터 시작)

2 4 6 3 5 (교환)

2 3 6 4 5

- 3회전 (세 번째 숫자를 결정하기 위함)

2 3 6 4 5 (교환) (2, 3을 제외하고 3번째부터 시작)

2 3 4 6 5

- 4회전 (네 번째 숫자를 결정하기 위함), 5개를 정렬 하려면 4회전 필요

2 3 4 6 5 (교환)

2 3 4 5 6 (정렬 완료)

정렬 알고리즘

- 구현

```
void selection_sort(int a[], int len)
{
    int i, j, t;

    for(i=0; i<len-1; i++) {
        for(j=i+1; j<len; j++)  {
            if(a[i] > a[j]) {
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
        }
    }
}
```

- 1회전

i	j			
4	3	6	2	5

i		j		
3	4	6	2	5

i			j	
3	4	6	2	5

i				j
2	4	6	5	5

정렬 알고리즘

- 구현

```
void selection_sort(int a[], int len)
{
    int i, j, t;

    for(i=0; i<len-1; i++) {
        for(j=i+1; j<len; j++)  {
            if(a[i] > a[j]) {
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
        }
    }
}
```

- 2회전

	i	j		
2	4	6	3	5

	i		j	
2	4	6	3	5

	i			j
2	3	6	4	5

정렬 알고리즘

- 구현

```
void selection_sort(int a[], int len)
{
    int i, j, t;

    for(i=0; i<len-1; i++) {
        for(j=i+1; j<len; j++)  {
            if(a[i] > a[j]) {
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
        }
    }
}
```

- 3회전

		i	j	
2	3	6	4	5

		i		j
2	3	4	6	5

- 4회전

			i	j
2	3	4	6	5

			i	j
2	3	4	5	6

정렬 알고리즘

- 선택 정렬 테스트

```c
#include <stdio.h>

// 길이가 len인 배열 a를 오름차순 정렬
void selection_sort(int a[], int len) {
    int i, j, t;
    for(i=0; i<len-1; i++) {
        for(j=i+1; j<len; j++) {
            if(a[i] > a[j]) {
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
        }
    }
}
```

```c
// 길이가 len인 배열 a의 모든 원소를 출력
void show_array(int a[], int len) {
    for(int i=0; i<len; i++)
        printf("%5d", a[i]);
    printf("\n\n");
}

int main() {
    int a[10] = {7, 5, 8, 1, 4, 9, 2, 10, 6, 3};
    show_array(a, 10);

    selection_sort(a, 10);
    show_array(a, 10);
    return 0;
}
```

Quiz) 만약 내림차순 정렬로 바꾸려면 어느 곳을 수정하며 될까?

STL sort() 함수 사용하기

- sort() 함수의 사용
  - C++의 STL 사용
  - #include <algorithm> 필요
  - sort(배열시작, 배열끝, [비교함수])
  - 기본 비교함수는 내림차순
- 비교함수(규칙)

```
// 오름차순용
bool asc_order(int a, int b) {
    return a < b;    // 뒤 쪽이 더 크게
}

// 내림차순용
bool desc_order(int a, int b) {
    return a > b;    // 앞 쪽이 더 크게
}
```

```
#include <stdio.h>
#include <algorithm>
using namespace std;

void show_array(int a[], int len) {
    for(int i=0; i<len; i++)
        printf("%5d", a[i]);
    printf("\n\n");
}

int main() {
    int a[10] = {7, 5, 8, 1, 4, 9, 2, 10, 6, 3};
    show_array(a, 10);

    sort(a, a+10);                  // 오름차순 정렬
    show_array(a, 10);

    sort(a, a+10, desc_order);  // 내림차순 정렬
    show_array(a, 10);
    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <algorithm>
using namespace std;

#define LEN 100

void selection_sort(int a[], int len) {
    int i, j, t;

    for(i=0; i<len-1; i++) {
        for(j=i+1; j<len; j++) {
            if(a[i] > a[j]) {
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
        }
    }
}

void show_array(int a[], int len) {
    for(int i=0; i<len; i++)
        printf("%5d", a[i]);
    printf("\n\n");
}
```

```
void rand_array(int a[], int len, int max) {
    srand(time(NULL));

    for(int i=0; i<len; i++)
        a[i] = rand() % max;
}

bool desc_order(int a, int b) {
    return a > b;
}

int main() {
    int a[LEN];
    rand_array(a, LEN, LEN*10);
    show_array(a, LEN);  // before sorting

    selection_sort(a, LEN);  // selection ASC sort
    show_array(a, LEN);      // after sorting

    sort(a, a+LEN, desc_order);  // DESC sort
    show_array(a, LEN);          // after sorting

    return 0;
}
```

정렬하여 k번째 수 찾기

- 문제

  n개의 정수를 배열에 입력 받아 정렬한 뒤, k번째로 큰 숫자를 찾는 프로그램을 작성하시오. 만약 네 개의 정수 1, 2, 3, 4가 입력되었다면, 3번째로 큰 수는 2이다.

- 입력

  첫 번째 줄에 입력 받을 자료의 개수 n이 입력된다. 두 번째 줄부터 정수 n개가 한 줄에 하나씩 차례대로 입력된다. 마지막 줄에는 k가 입력된다.

- 출력

  입력된 자료들 가운데 k번째로 큰 숫자를 출력한다.

- 입력과 출력의 예

입력 예	출력 예
4 1 2 3 4 3	2

- 고찰

  이 문제를 풀려면 오름차순 정렬을 사용해야 하는가? 내림차순 정렬을 사용해야 하는가?

다차원 배열

- 2차원 배열의 선언
  - 1차원 배열을 여러 개 겹쳐 놓은 것
  - 행과 열의 평면 구조를 가진 배열
  - 선언

    데이터형 배열명[행 수][열 수]

  - 선언 예

    int a[4][5];

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]
a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]

- 2차원 배열의 초기화

int a[2][3] = {{3, 2, 7}, {6, 9, 8}};

3	2	7
6	9	8

int b[2][3] = {5,8,3,7};

5	8	3
7	0	0

int c[2][3] = {4, };

4	0	0
0	0	0

int d[2][3] = { {3, }, {7,6,9} };

3	0	0
7	6	9

int e[][3] = { {3,2,6}, {7,6,9} };

다차원 배열

- 2차원 배열의 순회(행 우선)

```
#include <stdio.h>
#define ROW 3
#define COL 4

int main() {
    int a[ROW][COL] = {
        {1,2,3,4},{5,6,7,8},{9,10,11,12} };

    for(int r=0; r<ROW; r++) {      //행 순회
        for(int c=0; c<COL; c++) { //열 순회
            printf("%4d", a[r][c]);
        }
        printf("\n");
    }
    return 0;
}
```

1	2	3	4
5	6	7	8
9	10	11	12

```
   1   2   3   4
   5   6   7   8
   9  10  11  12
```

- 2차원 배열의 순회(열 우선)

```
#include <stdio.h>
#define ROW 3
#define COL 4

int main() {
    int a[ROW][COL] = {
        {1,2,3,4},{5,6,7,8},{9,10,11,12} };

    for(int c=0; c<COL; c++) {      //열 순회
        for(int r=0; r<ROW; r++) { //행 순회
            printf("%4d", a[r][c]);
        }
        printf("\n");
    }
    return 0;
}
```

1	2	3	4
5	6	7	8
9	10	11	12

```
   1   5   9
   2   6  10
   3   7  11
   4   8  12
```

격자판의 최댓값

- 문제

  <그림1>과 9x9 격자판에 쓰여진 81개의 자연수가 주어질 때, 이들 중 최댓값을 찾고 그 최댓값이 몇 행 몇 열에 위치한 수인지 구하는 프로그램을 작성하시오.

	1열	2열	3열	4열	5열	6열	7열	8열	9열
1행	3	23	85	34	17	74	25	52	65
2행	10	7	39	42	88	52	14	72	63
3행	87	42	18	78	53	45	18	84	53
4행	34	28	64	85	12	16	75	36	55
5행	21	77	45	35	28	75	90	76	1
6행	25	87	65	15	28	11	37	28	74
7행	65	27	75	41	7	89	78	64	39
8행	47	47	70	45	23	65	3	41	44
9행	87	13	82	38	31	12	29	29	80

<그림 1>

예를 들어, 왼쪽과 같이 81개의 수가 주어질 경우에는 이들 중 최댓값은 90이고, 이 값은 5행 7열에 위치한다.

출처: 한국정보올림피아드(2007 지역예선 중고등부)

- 입력

  첫째 줄부터 아홉째 줄까지 한 줄에 아홉 개씩 자연수가 주어진다. 주어지는 자연수는 100보다 작다.

- 출력

  첫째 줄에 최대값을 출력하고, 둘째 줄에 최댓값이 위치한 행 번호와 열번호를 빈칸을 사이에 두고 차례로 출력한다. 최댓값이 두 개 이상인 경우 행 숫자가 가장 작은 위치를 출력한다.

입력 예	출력 예
3 23 85 34 17 74 25 52 65 10 7 39 42 88 52 14 72 63 87 42 18 78 53 45 18 84 53 34 28 64 85 12 16 75 36 55 21 77 45 35 28 75 90 76 1 25 87 65 15 28 11 37 28 74 65 27 75 41 7 89 78 64 39 47 47 70 45 23 65 3 41 44 87 13 82 38 31 12 29 29 80	90 5 7

격자판의 최댓값

- 문제

```
#include <stdio.h>

#define ROW 9
#define COL 9

int a[ROW][COL];

void input() {
    for(int r=0; r<ROW; r++)
        for(int c=0; c<COL; c++) {
            scanf("%d", &a[r][c]);
}
```

```
int main() {
    input();

    int mr, mc, max=-1;









    printf("%d\n", max);
    printf("%d %d\n", mr+1, mc+1);
    return 0;
}
```

구조체

- 구조체란?
  - C언어의 기본 타입을 가지고 새롭게 정의하는 사용자 정의 타입
  - 기본 타입만으로는 나타낼 수 없는 복잡한 데이터를 표현 가능
  - 배열이 같은 타입의 변수 집합이라고 한다면, 구조체는 다양한 타입의 변수 집합을 하나의 타입으로 나타낸 것
  - 구조체를 구성하는 변수를 구조체의 멤버(member)라고 부른다

- 구조체의 정의와 선언 예

```
#include <stdio.h>

typedef struct {
    int x, y;  // 원의 중심점
    double r;  // 원의 반지름
} circle;

int main() {
    circle c1 = {1, 2, 5};
    printf(" (%d, %d) %lf", c1.x, c1.y, c1.r);
}
```

알고리즘의 효율성

- 이해의 복잡도
  - difficulty
  - 알고리즘 이해와 구현에 필요한 시간과 노력의 양

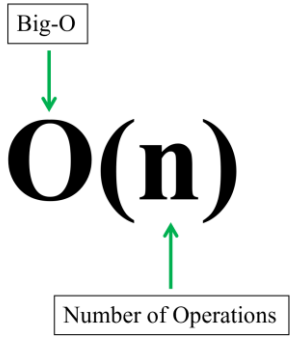
- 시간 복잡도
  - time complexity
  - 문제를 해결하는데 걸리는 시간과의 함수 관계
  - 반복문, 중첩된 반복문의 구조와 개수에 의해 결정

- 공간 복잡도
  - space complexity
  - 문제를 해결하기 위해 필요한 메모리(저장) 공간의 양
  - 변수 및 배열의 개수와 크기에 의해 결정
  - 함수가 호출될 때마다 사용되는 스택 공간이 늘어남

알고리즘의 시간 복잡도

- 빅오(O)표기법
  - 빅오 표기법은 알고리즘의 성능 평가 방법 중 가장 많이 사용하는 방법 중 하나
  - 가장 많이 사용하는 이유는 최악의 성능을 표시하기 때문
  - 최악의 성능 지표는, 적어도 이 정도의 성능은 보장한다는 의미
  - 실행 횟수를 점근적 표기법으로 표시

- 표기 형식



최소 n번은 연산해야 답이 나온다.

- 실행 횟수 계산
  - 프로그램은 첫번째 줄부터 마지막 줄까지 차례로 실행된다고 가정.
  - 헤더 파일은 알고리즘의 성능에 영향을 주지 않는다.
  - 함수 진입, 함수 반환은 알고리즘 성능에 영향을 주지 않는다.

알고리즘의 시간 복잡도

- 프로그램 예시

```
#define N 100                    // 영향을 주지 않는다.
#include <stdio.h>               // 영향을 주지 않는다.

void main(int)                   // 영향을 주지 않는다.
{
    int sum = 0;                 // 실행 횟수: 1회
    int i;                       // 실행 횟수: 1회

    for(i=1; i<=N; i++) {        // 실행 횟수: N+1회
        sum = sum + i;           // 실행 횟수: N회
    }

    printf("sum:%d\n",sum);      // 실행 횟수: 1회
    // 총 횟수: 1 + 1 + N+1 + N + 1 = 2N + 4회
}
```

- 실행 횟수 계산
  - 상수항은 무시
    - $O(101) \rightarrow O(1)$
    - $O(2N+1) \rightarrow O(N)$
  - 지배적이지 않은 항은 무시
    - $O(N^2+N) \rightarrow O(N^2)$
    - $O(N+logN) \rightarrow O(N)$
    - $O(100 \times 2^N + 500N^2) \rightarrow O(2^N)$

- 예시 프로그램의 Big-O: $O(N)$

빅오 표기의 종류

- $O(1)$
  - 상수시간(constant time)
  - 데이터 양과 상관없이 문제 해결에 항상 정해진 시간이 걸림
  - 평가: 최상의 알고리즘
  - 알고리즘 예
    - 정수의 홀짝 판별
    - 가우스의 1~N 누계 구하기
      - (시작수+마지막수) x n / 2

- $O(logN)$
  - 로그시간(logarithmic)
  - 데이터 양이 증가 함에 따라 실행 시간이 로그 함수 그래프로 나타남
  - 데이터가 많이 늘어나도 실행시간은 약간만 증가하는 특징
  - 평가: 매우 좋은 알고리즘
  - 알고리즘 예
    - 이진탐색
      - 10개 일때, $\log_2 10 = 3.x$
      - 100개 일때, $\log_2 100 = 6.x$
      - 1000개 일때, $\log_2 1000 = 9.x$

빅오 표기의 종류

- $O(N)$
  - 선형시간(linear time)
  - 데이터 양의 증가에 따라 실행 시간이 일차 함수 그래프로 나타남
  - 테이터 증가량과 정비례하여 실행 시간이 증가하는 특징
  - 평가: 좋은 알고리즘
  - 알고리즘 예
    - 정렬되지 않은 배열에서 최댓값 찾기

- $O(N\ logN)$
  - 선형 로그 시간(linearithmic time)
  - 데이터 양의 증가에 따라 실행 시간이 일차 함수 + 로그함수 형태의 그래프로 나타남
  - 데이터의 증가량보다 실행시간이 더 많이 증가하는 특징
  - 평가: 준수한 알고리즘
  - 알고리즘 예
    - 힙 정렬
    - 자이델(Seidel)의 다각형 삼각

빅오 표기의 종류

- $O(N^2)$
  - 이차식 시간(quadratic time)
  - 데이터 양의 증가에 따라 실행 시간이 이차 함수(N^2) 그래프로 나타남
  - 테이터 증가량에 제곱으로 비례하여 실행 시간이 증가하는 특징
  - 평가: 그저 그런 알고리즘
  - 알고리즘 예
    - 선택정렬
    - 버블정렬
- $O(N^3)$
  - 삼차식 시간(cubic time)
  - 데이터 양의 증가에 따라 실행 시간이 삼차 함수(N^3) 그래프로 나타남
  - 테이터 증가량과 정비례하여 실행 시간이 증가하는 특징
  - 평가: 나쁜 알고리즘
  - 알고리즘 예
    - 행렬 2개의 무식한 곱셈

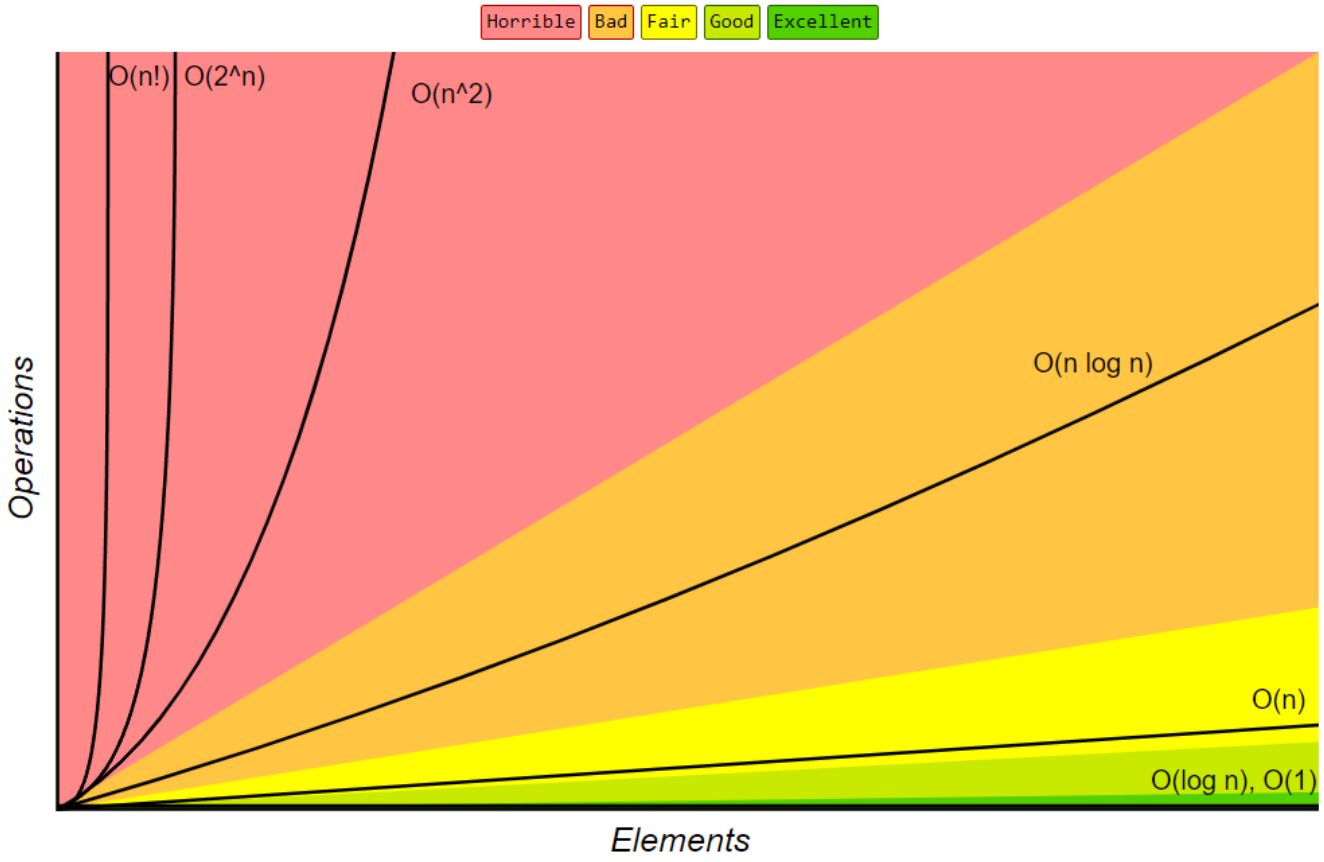
빅오 표기의 종류

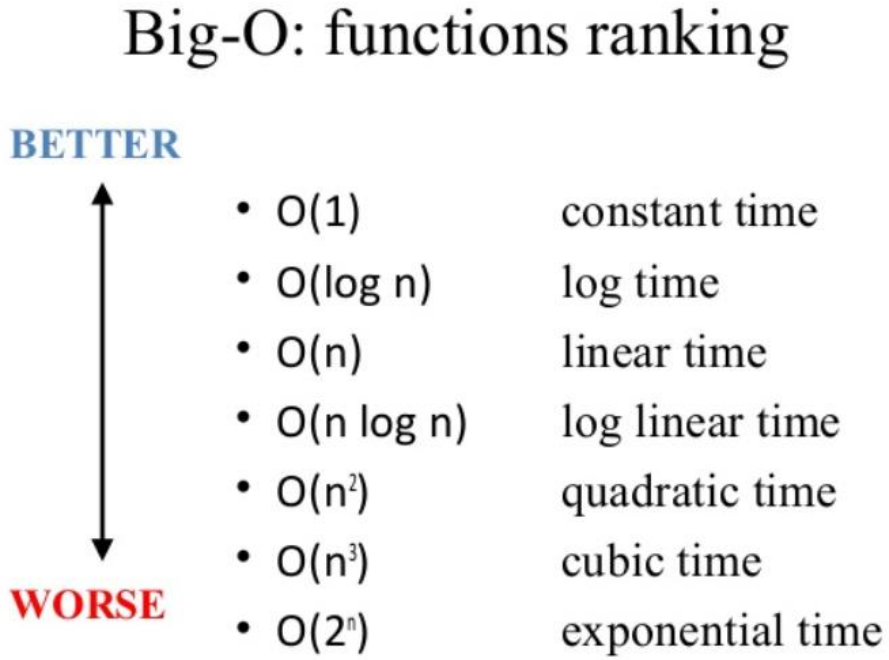
- $O(2^N)$
  - 지수 시간(exponential time)
  - 데이터 양의 증가에 따라 실행 시간이 지수 함수(2^N) 그래프로 나타남
  - 테이터 증가량에 따라 실행 시간이 지수 형태로 증가하는 특징
  - 평가: 끔찍한 알고리즘
  - 알고리즘 예
    - 2^N을 재귀 호출로 계산

- $O(N!)$
  - 계승 시간(factorial time)
  - 데이터 양의 증가에 따라 실행 시간이 팩토리얼 함수($N!$) 그래프로 나타남
  - 평가: 최악의 알고리즘
  - 알고리즘 예
    - 브루트포스 탐색을 통한 외판원 문제 해결방법

빅오 표기의 종류

- 알고리즘 성능 비교
  - $O(1) > O(\log N) > O(N) > O(N \log N) > O(N^2) > \ \ldots \ > O(2^N) > O(N!)$





시간제한 피하기

- 주어진 입력 N의 크기에 따른 허용 시간 복잡도

N의 크기	시간복잡도
N ≤ 11	O(N!)
N ≤ 25	$O(2^N)$
N ≤ 100	$O(N^4)$
N ≤ 500	$O(N^3)$
N ≤ 3,000	$O(N^2 \log N)$
N ≤ 5,000	$O(N^2)$
N ≤ 1,000,000	O(NlogN)
N ≤ 10,000,000	O(N)
N > 10,000,000	O(logN), O(1)

- 활용 방법
  - 컴퓨터는 대략 1초에 1억회의 연산 수행한다고 가정하고 왼쪽 표를 얻어냄
  - 시간 제한은 대부분 1~5초
  - 입력 데이터가 5000개 이하로 주어진다면 $O(N^2)$ 또는 그보다 빠른 알고리즘을 설계하여 문제를 풀어야 함.
  - 입력 데이터가 25개 이하로 주어진다면 $O(2^N)$ 알고리즘만 되어도 통과 가능할 것임.

알고리즘의 공간 복잡도

- 공간 복잡도(Space Complexity)란?
  - 프로그램을 실행시킨 후 완료하는 데 필요로 하는 자원 공간의 양
  - $S(P) = c + S_p(N)$
    - 총 공간 요구 = 고정 공간 요구 + 가변 공간 요구
    - 고정 공간: 입출력 횟수나 크기와 관계없는 공간 요구
    - 가변 공간: 문제 해결을 위해 필요한 공간 + 재귀 호출에 요구 되는 공간

- 빅오 표기법
  - 알고리즘의 공간복잡도 역시 빅오 표기법으로 표현 가능하며 계산법 역시 동일
  - 단, 재귀 호출에 사용되는 스택 공간도 고려해야 함.
  - 어떤 알고리즘이 N개의 입력 데이터에 대하여 N x N 크기의 2차원 배열과 N 크기의 1차원 배열이 필요하다면,
  - 이때 이 알고리즘의 공간복잡도는 $O(N^2)$이다.

선형 탐색

feat. 탐색공간의 수학적 배제

탐색공간의 배제

- 필요성
  - 전체 탐색으로 대부분의 경우 해를 구할 수 있음
  - 하지만 실행 시간이 너무 길어 제한 시간 내에 문제를 해결하기 힘든 경우가 많음
  - 전체탐색에서 불필요한 탐색 공간을 탐색하지 않음으로써 알고리즘의 효율 향상 가능
  - 모든 공간을 탐색할 것이 아니라 일정한 조건을 두어 탐색에서 제외

- 수학적 배제
  - 수학적으로 탐색할 필요가 없음이 증명된 공간을 탐색에서 제외

- 경험적 배제(가지치기)
  - 일정 조건을 만족하는 경우 탐색에서 배제하는데 이 조건은
  - 이전에 탐색한 정보를 이용하며,
  - 배제 조건은 계속 갱신될 수 있음

약수의 합

- 문제

  한 정수 n을 입력 받는다.

  1부터 n의 자연수들 중 n 약수의 합을 구하는 프로그램을 작성하시오.

  예를 들어 n이 10이라면,

  10의 약수는 1, 2, 5, 10이므로 구하고자 하는 값은 1 + 2 + 5 + 10을 더한 18이 된다.

- 입력

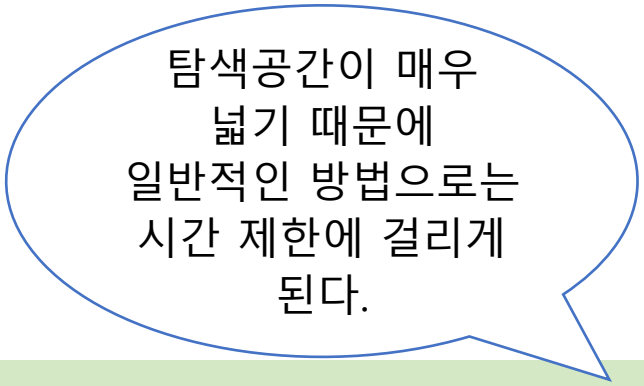
  첫 번째 줄에 정수 n이 입력된다.

  (단, 1 <= n <= 10,000,000,000(100억))

- 출력

  n의 약수의 합을 출력한다.

입력 예	출력 예
10	18



약수의 합

- 단순 풀이

```
#include <stdio.h>
long long n;
long long solve() {
  long long ans=0;

  for(long long i=1; i<=n; i++) {
     //n이 i로 나누어 떨어지면 i는 n의 약수이다
     if(n%i==0)
        ans+=i;
  }
  return ans;
}

int main() {
    scanf("%lld", &n);
    printf("%lld\n", solve());
    return 0;
}
```

- 평가
  - 이 소스코드는 1부터 n까지의 모든 원소들을 탐색하여, 탐색 대상인 수 i가 n의 약수라면 취하는 방식으로 진행된다.
  - 따라서 계산량은 O(n)이다.
  - 이번 문제는 n의 최댓값이 100억이므로 이 방법으로는 너무 많은 시간이 걸린다.
  - 따라서 탐색영역을 배제해야 할 필요가 있다.

약수의 합

- 고찰

1) 배제를 위한 수학적 아이디어 1

모든 자연수 n에 대하여 1 과 n 은 항상 n 의 약수이다.

ex) 10의 약수
1, 2, 5, 10

ex) 16의 약수
1, 2, 4, 8, 16

```
for(int i=2; i<n; i++) {
    if(n % i==0)
        ans+=i;
}
```

- 고찰

2) 배제를 위한 수학적 아이디어 2

모든 자연수 n에 대하여,
2이상 n미만의 자연수들 중 가장 큰 n의 약수는 n/2를 넘지 않는다.

ex) 10의 약수
1, 2, 5, 10

ex) 16의 약수
1, 2, 4, 8, 16

```
for(int i=2; i<=n/2; i++) {
    if(n % i==0)
        ans+=i;
}
```

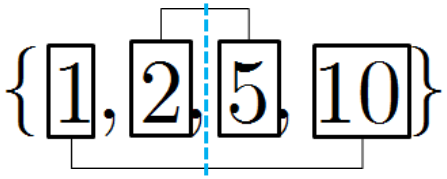
약수의 합

- 고찰

3) 배제를 위한 수학적 아이디어 3

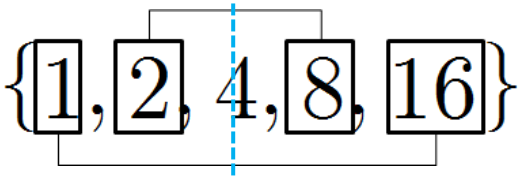
임의의 자연수 n의 약수들 중 두 약수의 곱은, n이 되는 약수a와 약수b가 반드시 존재한다. 단, n이 완전제곱수 일 경우에는 약수 a와 약수 b가 같을 수 있다.

반쪽만 찾으면 됨



반쪽만 찾으면 됨

10의 약수간 관계



16의 약수간 관계

약수의 개수를 c개라고 하고, d를 n의 약수 중 i번째 약수라 하면,

$$n = d_k \times d_{c-k+1}$$

완전 제곱수일 경우 우변 두 항이 동일,
최악의 경우 n 부터 $\sqrt{n}$ 까지만 검색하면 나머지 약수를 모두 찾아낼 수 있음

```
#include <math.h>
  :
    for(int i=1; i<=sqrt(n); i++) {
        if(n % i==0)
            ans+=i;
    }
```

```
    for(int i=1; i*i<=n; i++) {
        if(n % i==0)
            ans+=i;
    }
```

약수의 합

- 고찰

3) 배제를 위한 수학적 아이디어 3 구현

ex) 100의 약수 모두 구하기

① $1 \sim \sqrt{100} = 10$ 까지 조사

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

② 약수 집합a { 1, 2, 5, 10}으로부터

약수 집합b {100, 50, 20, 10}유도

③ 약수 합집합 {1,2,5,10,20,50,100}

- 수학적 배제 적용

```
#include <stdio.h>
long long int n;
long long int solve() {
    long long int i, ans = 0;









    return ans;
}
int main() {
    scanf("%lld", &n);
    printf("%lld\n", solve());
    return 0;
}
```

N번째 소수 찾기

- 문제

  한 정수 n을 입력 받는다.

  n번째로 큰 소수를 구하여 출력한다.

  예를 들어 n이 5라면,

  자연수들 중 소수는 2, 3, 5, 7, 11, 13, ...

  이므로 구하고자 하는 5번째 소수는 11이 된다.

- 입력

  첫 번째 줄에 정수 n이 입력된다.

  (단, 1 <= n <= 100,000)

- 출력

  n 번째 소수를 출력한다.

입력 예	출력 예
5	11

입력 예	출력 예
77	389

N번째 소수 찾기

- 단순 풀이

```
#include <stdio.h>

bool is_prime(int k) {
    int cnt = 0;
    for(int i=1; i<=k; i++)
        if(k%i==0)  // k의 약수의 갯수 cnt를 구한다
            cnt++;

    if(cnt==2)
        return true;
    else
        return false;
}
```

임의의 자연수 k가 소수라면 k의 약수는 1과 k만 존재한다. (약수가 2개 뿐임)

```
int main() {
    int nth; // 몇 번째
    scanf("%d", &nth);

    int prime_cnt=0;
    int n=2;
    while(true) {
        if(is_prime(n)) //소수이면 cnt증가
            prime_cnt++;
        // 찾고자 하는 번째 이면
        if(prime_cnt == nth)
            break;
        n++;
    }
    printf("%d", n);
}
```

N번째 소수 찾기

- 고찰

1) 배제를 위한 수학적 아이디어 1

약수를 두 개 이상 발견하면 바로 탈출

```
bool is_prime(int k) {
    int cnt = 0;
    for(int i=1; i<=k; i++) {
        if(k%i==0) cnt++;
        if(cnt > 2)
            break;
    }
    return (cnt==2);
}
```

2) 배제를 위한 수학적 아이디어 2

임의의 자연수 k가 소수라면 구간 [2, k-1] 에서 약수는 존재하지 않는다.

```
bool is_prime(int k) {
    for(int i=2; i<k; i++) {
        //2~ k-1사이 숫자로 나누어 떨이지면,
        // 즉, 약수가 존재하면
        if(k%i==0)
            return false;
    }
    return true;
}
```

N번째 소수 찾기

- 고찰

3) 배제를 위한 수학적 아이디어 3

k의 약수를 구하기 위해서는 ____ 까지만 검사하면 된다.

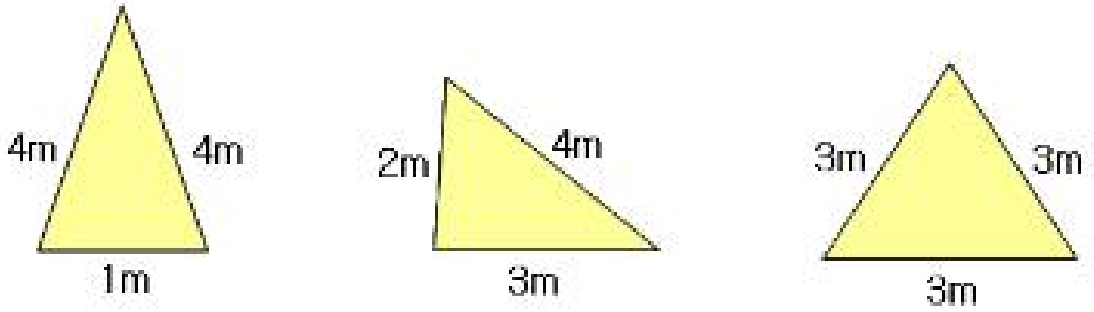
```
bool is_prime(int k) {
    for(int i=2;   ?   ; i++) {
        if(k%i==0)
            return false;
    }
    return true;
}
```

```
#include <stdio.h>

bool is_prime(int k) {
    for(int i=2; i*i<=k; i++) {
        if(k%i==0)
            return false;
    }
    return true;
}

int main() {
    int nth;
    scanf("%d", &nth);

    int prime_cnt=0;
    int n=2;
    while(true) {
        if(is_prime(n))
            prime_cnt++;
        if(prime_cnt == nth)
            break;
        n++;
    }
    printf("%d\n\n", n);
}
```

삼각화단 만들기

주어진 화단 둘레의 길이를 이용하여 삼각형 모양의 화단을 만들려고 한다. 이 때 만들어진 삼각형 화단 둘레의 길이는 반드시 주어진 화단 둘레의 길이와 같아야 한다. 또한, 화단 둘레의 길이와 각 변의 길이는 자연수이다. 예를 들어, 만들고자 하는 화단 둘레의 길이가 9m라고 하면,

- 한 변의 길이가 1m, 두 변의 길이가 4m인 화단
- 한 변의 길이가 2m, 다른 변의 길이가 3m, 나머지 변의 길이가 4m인 화단
- 세 변의 길이가 모두 3m인 3가지 경우의 화단을 만들 수 있다.



화단 둘레의 길이를 입력 받아서 만들 수 있는 서로 다른 화단의 수를 구하는 프로그램을 작성하시오.

- **입력**

  화단의 길이 n이 주어진다. (1<=n<=50,000)

- **출력**

  입력받은 n으로 만들 수 있는 서로 다른 화단의 수를 출력한다.

입력 예	출력 예
9	3

- 주의

  2,3,4 화단과 3,2,4 화단 2,4,3 화단은 모두 같은 모양의 화단임.

- 출처: 한국정보올림피아드(2002 전국본선 초등부)

삼각화단 만들기

- 단순 풀이 (오답)

```
#include <stdio.h>

int main(void) {
    int n;
    int cnt=0;
    scanf("%d", &n);
    for(int a=1; a<=n; a++)
        for(int b=1; b<=n; b++)
            for(int c=1; c<=n; c++) {
                if(a+b+c==n) {
                    printf("[%d %d %d]\t", a, b, c);
                    cnt++;
                    // 5개 출력할 때마다 줄 내림
                    if(cnt%5 == 0) puts("");
                }
            }
    printf("\nfound %d\n", cnt);
}
```

- 평가
  - $O(n^3)$ 의 시간 소모
  - 중복된 삼각형이 포함됨
  - 삼각형 만들기가 불가능한 길이도 포함

```
9
[1 1 7]  [1 2 6]  [1 3 5]  [1 4 4]
[1 5 3]  [1 6 2]  [1 7 1]  [2 1 6]
[2 2 5]  [2 3 4]  [2 4 3]  [2 5 2]
[2 6 1]  [3 1 5]  [3 2 4]  [3 3 3]
[3 4 2]  [3 5 1]  [4 1 4]  [4 2 3]
[4 3 2]  [4 4 1]  [5 1 3]  [5 2 2]
[5 3 1]  [6 1 2]  [6 2 1]  [7 1 1]

found 28
```

삼각화단 만들기

- 풀이
  - 동일한 길이 쌍 제거 조건

$$a \leq b \leq c$$

  - 삼각형의 조건
    - $a + b > c$
    - $a + b + c = n$

- 정답 알고리즘

```
#include <stdio.h>
int n;
int solve() {
    int cnt = 0;
    scanf("%d", &n);

    for(int a=1; a<=n; a++)
        for(int b=a; b<=n; b++)
            for(int c=b; c<=n; c++ )
                if(a+b+c==n && a+b>c)
                    cnt++;
    return cnt;
}

int main() {
    printf("%d\n", solve());
}
```

삼각화단 만들기

- 고찰

1) 배제를 위한 수학적 아이디어 1

둘레 길이가 n인 삼각형의 a, b 길이가 정해지면 c변은 n-(a+b) 계산으로 구할 수 있다.

```
for(int a=1; a<=n; a++)
   for(int b=a; b<=n; b++) {
      int c=n-(a+b);     // a+b+c=n 조건만족
      if(b<=c && a+b>c) // a≤b는 이미 만족
          cnt++;
}
```

- 공간복잡도가 $O(n^3)$ 에서 $O(n^2)$ 으로 줄어든다.

2) 배제를 위한 수학적 아이디어 2

둘레가 n인 삼각형의 각 변의 길이를 오름차순으로 정렬한 결과를 a, b, c 라고 할 때, 다음 조건을 만족한다.

$$[\frac{n}{3}] \le c < [\frac{n}{2}],\ 1 \le a \le [\frac{n}{3}]$$

```
for(int c=n/3; c<=n/2; c++)
    for(int a=1; a<=n/3; a++) {
        int b=n-(a+c);
        if(a+b>c && (a<=b && b<=c))
            cnt++;
    }
```

삼각화단 만들기

- 실행시간 비교

1) 배제를 위한 수학적 아이디어 1

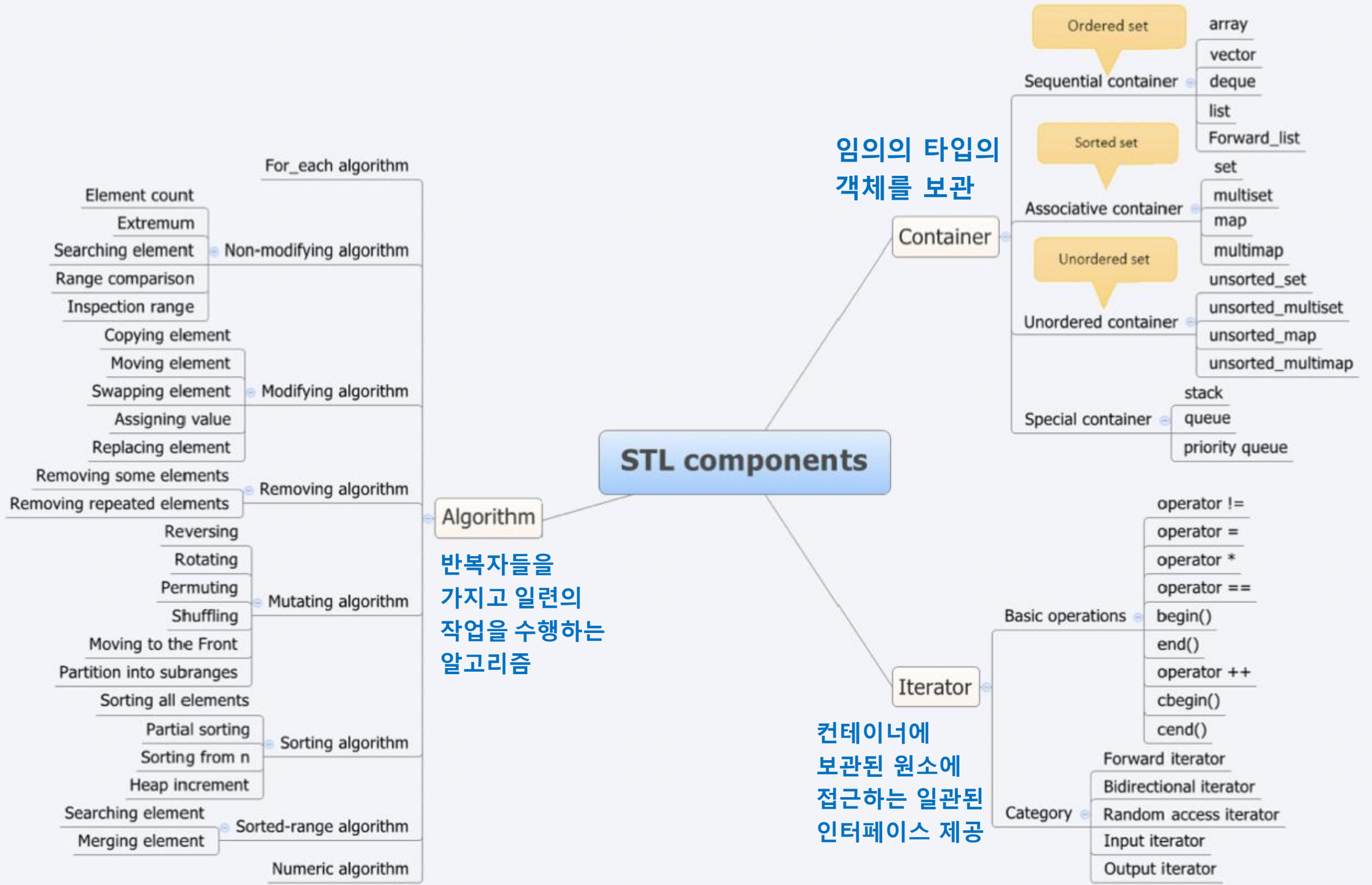
```
time_space_table:
/1030/sample.in:AC mem=0k time=3ms
/1030/test01.in:AC mem=0k time=2ms
/1030/test02.in:AC mem=0k time=2ms
/1030/test03.in:AC mem=0k time=3ms
/1030/test04.in:AC mem=0k time=3ms
/1030/test05.in:AC mem=0k time=11ms
/1030/test06.in:AC mem=0k time=38ms
/1030/test07.in:AC mem=0k time=149ms
/1030/test08.in:AC mem=0k time=355ms
/1030/test09.in:AC mem=0k time=586ms
/1030/test10.in:AC mem=0k time=925ms
```

2) 배제를 위한 수학적 아이디어 2

```
time_space_table:
/1030/sample.in:AC mem=0k time=2ms
/1030/test01.in:AC mem=0k time=2ms
/1030/test02.in:AC mem=0k time=2ms
/1030/test03.in:AC mem=0k time=3ms
/1030/test04.in:AC mem=0k time=3ms
/1030/test05.in:AC mem=0k time=4ms
/1030/test06.in:AC mem=0k time=7ms
/1030/test07.in:AC mem=0k time=19ms
/1030/test08.in:AC mem=0k time=40ms
/1030/test09.in:AC mem=0k time=68ms
/1030/test10.in:AC mem=0k time=107ms
```

STL

(Standard Template Library)


STL components
Container
임의의 타입의
객체를 보관
Ordered set
Sequential container
array
vector
deque
list
Forward_list
Sorted set
Associative container
set
multiset
map
multimap
Unordered set
Unordered container
unsorted_set
unsorted_multiset
unsorted_map
unsorted_multimap
Special container
stack
queue
priority queue
Algorithm
반복자들을
가지고 일련의
작업을 수행하는
알고리즘
For_each algorithm
Non-modifying algorithm
Element count
Extremum
Searching element
Range comparison
Inspection range
Modifying algorithm
Copying element
Moving element
Swapping element
Assigning value
Replacing element
Removing algorithm
Removing some elements
Removing repeated elements
Mutating algorithm
Reversing
Rotating
Permuting
Shuffling
Moving to the Front
Partition into subranges
Sorting algorithm
Sorting all elements
Partial sorting
Sorting from n
Heap increment
Sorted-range algorithm
Searching element
Merging element
Numeric algorithm
Iterator
컨테이너에
보관된 원소에
접근하는 일관된
인터페이스 제공
Basic operations
operator !=
operator =
operator *
operator ==
begin()
end()
operator ++
cbegin()
cend()
Category
Forward iterator
Bidirectional iterator
Random access iterator
Input iterator
Output iterator

1) 컨테이너

- 같은 타입의 여러 객체를 저장하는 객체
  - 클래스 템플릿으로 작성됨 (즉, 무엇이든 넣을 수 있다)
- 컨테이너의 종류

종류	설명	컨테이너
순차 컨테이너	특별한 규칙이 없는 일반적인 컨테이너 순서가 있는 선형구조.	array, vector, dequeue, list, forward_list
연관 컨테이너	특정 규칙에 의해서 정렬, 저장, 관리 순서가 없는 비선형구조.	map, miltimap, set, multiset
비정렬 컨테이너	내용물이 정렬되지 않은 상태로 보관되는 순서 없는 비선형 구조.	unordered_map, unordered_multimap, unordered_set, unordered_multiset
컨테이너 어뎁터	간결함과 명료성을 위해 인터페이스를 제한한 시퀀스나 연관 컨테이너의 변형. 반복자를 지원하지 않음.	queue, priority_queue, stack

1) 컨테이너 (container)

- 컨테이너별 성능비교

Container	Insertion	Access	Erase	Find	Persistent Iterator
vector / string	Back: O(1) or O(n) Other: O(n)	O(1)	Back: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	No
deque	Back/Front: O(1) Other: O(n)	O(1)	Back/Front: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	Pointers only
list / forward_list	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	O(n)	Yes
set / map	O(log n)	-	O(log n)	O(log n)	Yes
unordered_set / unordered_map	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	Pointers only
priority_queue	O(log n)	O(1)	O(log n)	-	-

2) 반복자 (interator)

- 컨테이너에 저장된 요소를 반복적으로 순회하여 요소를 가리키는 객체
- **컨테이너의 구조나 요소의 타입과 상관없이 동일한 방식으로 데이터를 순회할 수 있도록 한다.**
- 반복자의 종류

종류	설명
입력 반복자	현재 위치의 객체의 값을 읽어 오는 반복자
출력 반복자	현재 위치의 객체의 값을 변경할 수 있는 반복자
순방향 반복자	순방향으로 이동(++) 가능하면 재할당이 가능하다.
양방향 반복자	순방향 반복자 기능에 역방향으로 이동(--)이 가능한 반복자 이다.
임의 접근 반복자	양방향 반복자 기능과 []을 사용하여 임의의 요소에 접근 가능한 반복자이다.

3) 알고리즘

- 컨테이너를 알고리즘을 통해 동작시키는데 필요한 많은 함수를 제공
- STL 알고리즘과 함께 사용되며 반복자를 통해 컨테이너에 적용시킨다
- 알고리즘의 종류

종류	설명	대표 함수

읽기 알고리즘	컨테이너를 변경하지 않으며, 컨테이너의 지정된 범위에서 특정 데이터를 읽기만 하는 알고리즘	#include <algorihth>
		find(), for_each()

변경 알고리즘	컨테이너를 변경하지 않으며, 컨테이너의 지정된 범위에서 요소의 값만을 변경할 수 있는 알고리즘	#include <algorihth>
		copy(), swap(), transform()

정렬 알고리즘	컨테이너의 지정된 범위의 요소들이 정렬되도록 컨테이너를 변경하는 알고리즘	#include <algorihth> sort(), stable_sort(), binary_search()

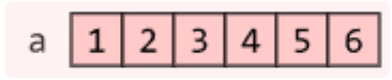
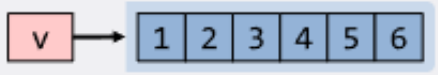
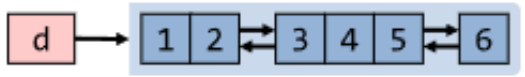
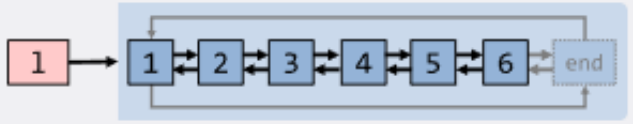
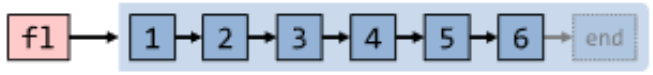
수치 알고리즘	STL에 직접 속하지 않고 C++라이브러리로 분류되는 알고리즘으로 수치적 해석을 위해 사용	#include <numeric>
		accumulate()

STL Container

Standard Template Library

1. Array
2. Vector
3. Deque (Double Ended Queue)
4. Queue
5. Heap (Priority Queue)

Standard Sequence Containers Overview

`array<T,size>`	 fixed-size contiguous array
`vector<T>`	 dynamic contiguous array; amortized *O*(1) growth strategy; C++'s "default" container
`deque<T>`	 double-ended queue; fast insert/erase at both ends
`list<T>`	 doubly-linked list; *O*(1) insert, erase & splicing; in practice often slower than vector
`forward_list<T>`	 singly-linked list; *O*(1) insert, erase & splicing; needs less memory than `list`; in practice often slower than vector

array 컨테이너

`std::array` 는 고정된 크기의 배열을 담고 있는 컨테이너 이다.

이 컨테이너는 마치 C 언어에서의 배열인 `T[N]` 과 비슷하게 작동하는데, 예를 들어서 C 배열 처럼 `{}` 를 통해 초기화 할 수 있습니다. (예컨대 `std::array<int, 3> a = {1,2,3}`). 다만 한 가지 차이점은 C 배열과는 다르게 배열의 이름이 `T*` 로 자동 형변환 되지 않습니다.

`std::array` 를 통해서 기존의 C 배열과 같은 형태를 유지하면서 (오버헤드가 없습니다), C++ 에서 추가된 반복자라던지, 대입 연산자 등을 사용할 수 있습니다. (즉 `<algorithm>` 에 정의된 함수들을 `std::array` 에도 사용할 수 있다는 의미 입니다.)

참고로 크기가 0 인 `std::array` 의 경우 (즉 `N` 이 0 일 때), `array.begin() == array.end()` 이며, `front()` 나 `back()` 을 호출할 시 그 결과는 정의되지 않습니다 (Undefined behavior).

쉽게 생각해서 `std::array` 는 `N` 개의 같은 타입의 원소들을 담고 있는 `tuple` 이라 보시면 됩니다.

반복자 무효화

`std::array` 의 반복자는 배열 객체가 살아있는 동안 **절대로 무효화 되지 않습니다**. 다만 `swap` 후에, 기존의 반복자가 같은 위치를 계속 가리키고 있으므로 다른 값을 가리킬 수 도 있습니다.

array 컨테이너

▪ 대입연산자 (oprator=)

배열의 각각의 원소들에 대해 대입 연산자를 호출합니다. 쉽게 말해

C/C++ | 확대 | 축소

```
std::array<int, 3> a = {1, 2, 3};
std::array<int, 3> b;

b = a;  // b 에 {1,2,3} 이 들어간다
```

가 됩니다. 반면에 C 배열의 경우

C/C++ | 확대 | 축소

```
int arr[3] = {1, 2, 3};
int b[3];

b = arr;  // <-- 불가능!!
```

▪ at, operator[]

```
#include <array>
#include <iostream>
using namespace std;

int main() {
  array<int, 6> data = {1, 2, 4, 5, 5, 6};
  // Set element 1
  data[1] = 88;  // 경계 검사 안함
  // Read element 2
  cout << "인덱스 2 에 위치한 원소 : " << data.at(2) << '\n';
  cout << "data 배열의 크기 = " << data.size() << '\n';

  try {
    data.at(6) = 666;  // 0-base 6th elements
  } catch (out_of_range const& ex) {
    cout << "예외 발생 : " << ex.what() << '\n';
  }
  // Print final values
  cout << "data:";
  for(int elem : data)
    cout << " " << elem;
  cout << '\n';
}
```

vector container

- vector의 특징
  - 크기를 바꿀 수 있는 순차 컨테이너
  - 가변길이 배열이라고 생각하면 쉬움
  - 특정 위치의 원소에 빠르게 접근 가능
  - 벡터에 원소가 삽입되고 삭제됨에 따라 자동으로 크기 조절됨
  - #include <vector> 필요
  - 주요 사용 사례
    - 주로 임의 접근이 빈번하고, 끝에서의 삽입과 삭제가 주로 발생하는 경우에 사용

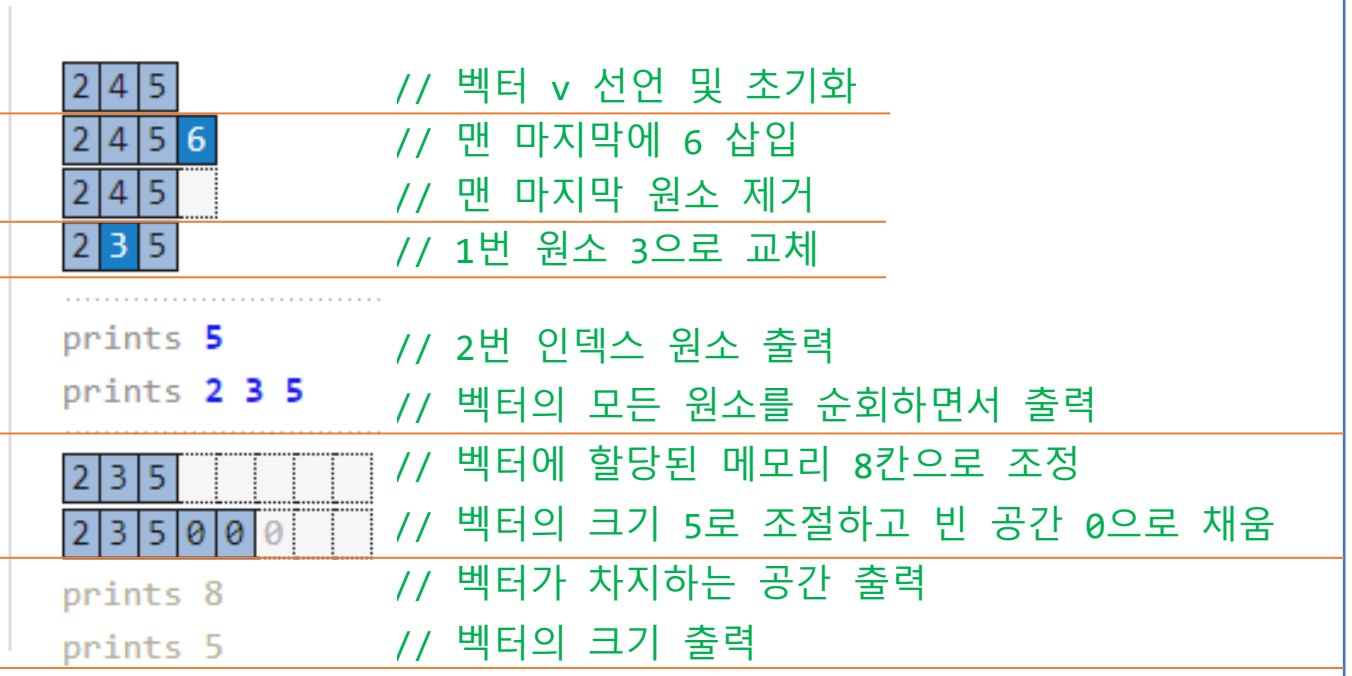
- vector의 선언

```
vector<자료형> 변수명;
vector<자료형> 변수명 = { 초기값 };

vector<int> v1;     // int를 담는 벡터
vector<double> v2; //double을 담는 벡터
```

vector container

- vector의 사용 예시

```
#include <stdio.h>
#include <vector>
using namespace std;

int main() {
    vector<int> v = {2, 4, 5};
    v.push_back(6);
    v.pop_back();
    v[1] = 3;

    printf("%d\n", v[2]);
    for(int x: v) printf("%d ", x);
    v.reserve(8);
    v.resize(5, 0);
    printf("\n%d\n", v.capacity());
    printf("%d\n", v.size());
}
```



vector container

- vector의 메소드
  - v.back() : v의 마지막 원소를 참조한다.
  - v.front() : v의 첫 번째 원소를 참조한다.
  - v.begin() : v의 시작을 가리키는 반복자를 반환한다.
  - v.end() : v의 끝을 가리키는 반복자를 반환한다.
  - v.push_back(x) : v의 끝에 x를 추가한다
  - v.pop_back() : 마지막 원소를 제거한다.
  - v.size() : v벡터의 원소의 개수 리턴, 값은 unsigned int가 나옴 (모든 컨테이너가 가진 함수)
  - v.resize(n) : v의 크기를 n으로 변경하고 확장되는 공간을 기본값으로 초기화
  - v.resize(n, x) : v의 크기를 n으로 변경하고 확장되는 공간의 값을 x로 초기화
  - v.capacity() : 실제 할당된 메모리 공간의 크기(vector만이 가지고 있는 함수)
  - q = v.insert(p, x) : p가 가리키는 위치에 x값을 삽입한다. q는 삽입한 원소를 가리키는 반복자
  - v.insert(p, n, x) : p가 가리키는 위치에 n개의 x값을 삽입한다.
  - v.insert(p, b, e) : p가 가리키는 위치에 반복자 구간[b, e) 원소를 삽입한다.
  - v.push_back(x) : v의 끝에 x를 추가한다
  - v.pop_back() : 마지막 원소를 제거한다.
  - v.earase(iterator) : 반복자가 가리키는 걸 지움
  - v.clear() : v의 모든 원소를 제거한다.

vector container

- 1차원 벡터의 순회

v: 6 2 9 7

```
#include <stdio.h>
#include <vector>
using namespace std;

int main() {
    vector <int> v = { 6,2,9,7 };
    //방법1
    for(int i=0; i<v.size(); i++) {
       printf("%d ", v[i]);
    }
    putchar('\n');
    //방법2
    for(int i : v) {
       printf("%d ", i);
    }
}
```

```
6 2 9 7
6 2 9 7
```

- 2차원 백터의 순회

v[0]: 3 1
v[1]: 2 1 5
v[2]: 6

```
#include <stdio.h>
#include <vector>
using namespace std;

int main() {
    // 벡터안에 벡터를 보관하는 형태
    vector<vector<int>> v =
       { {3, 1}, {2, 1, 5}, {6} };
      //  v[0]    v[1]      v[2]
    for(int i=0; i<v.size(); i++) {
        for(int j : v[i])
            printf("%d ", j);
        putchar('\n');
    }
}
```

```
3 1
2 1 5
6
```

std::vector<*ValueType*>

C++'s "default" dynamic array

`#include <vector>`

h/cpp hackingcpp.com

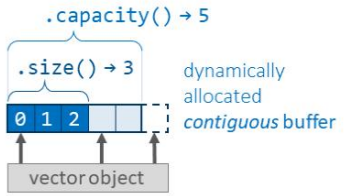
Construct A New Vector Object

```
vector<int> v1 {2,9,1,8,5,4}
vector<int> v2 (begin(v1)+3, end(v1))
vector<int> v3 (5,3)
vector<int> deep_copy_of_v1 (v1)
```

C++17 value type deducable from argument type

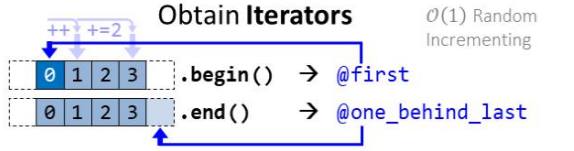
```
vector w {7,4,2};   // vector<int>
```

Typical Memory Layout

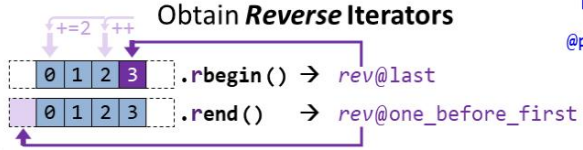


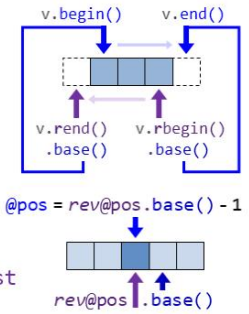
Obtain Iterators

$\mathcal{O}(1)$ Random Incrementing



Obtain *Reverse* Iterators



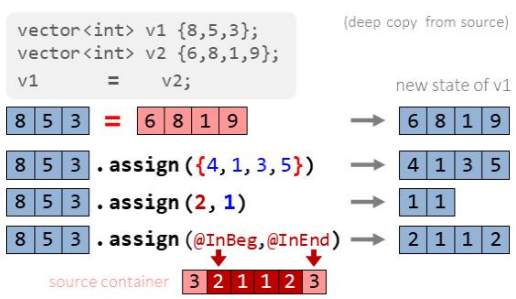


Assign New Content To An Existing Vector

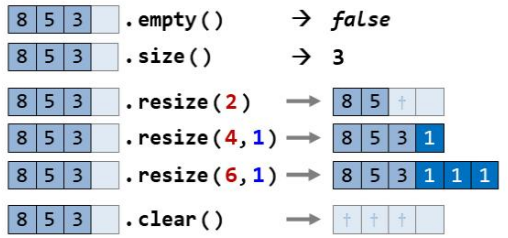
```
vector<int> v1 {8,5,3};
vector<int> v2 {6,8,1,9};
v1      =     v2;
```

(deep copy from source)

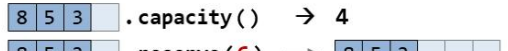
new state of v1



Query/Change Size (= Number of Elements)



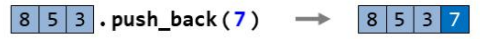
Query/Grow Capactiy (= Memory Buffer Size)



Avoid expensive memory allocations: `.reserve` capacity before appending / inserting if you know the (approximate) number of elements to be stored in advance!

Append Elements

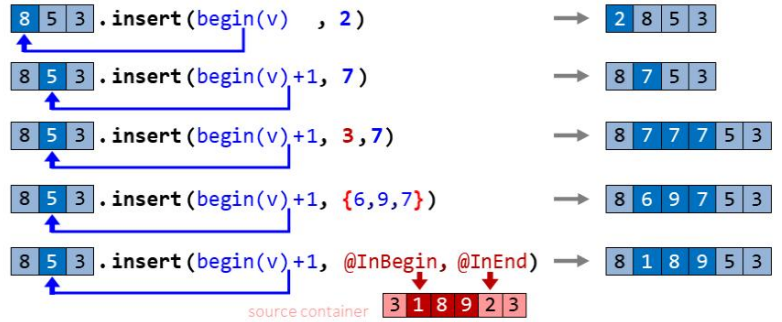
$\mathcal{O}(1)$ Amortized Complexity



Insert Elements at Arbitrary Positions

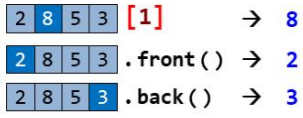
$\mathcal{O}(n)$ Worst Case

```
vector<int> v {8,5,3};
```

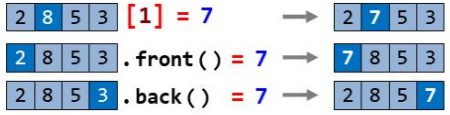


Get Element Values

$\mathcal{O}(1)$ Random Access



Change Element Values

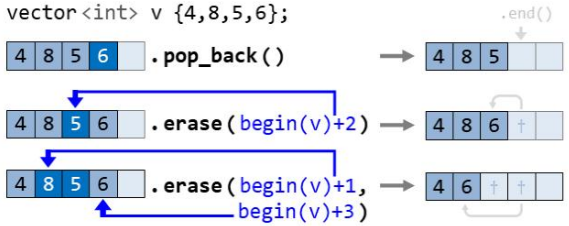


Out of Bounds Access

Erase Elements

$\mathcal{O}(n)$ Worst Case

```
vector<int> v {4,8,5,6};
```



Shrink The Capacity

(might be inefficient)

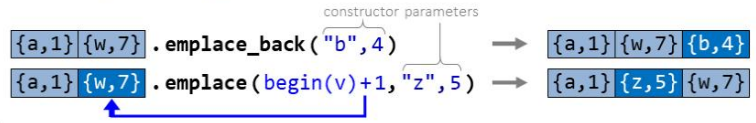
Erasing, resizing or clearing will not shrink the capacity!

```
vector<int> v (1024,0);   // capacity is at least 1024
v.resize(40);             // capacity unchanged!
v.shrink_to_fit();        // may shrink (not guaranteed)
v.swap(vector<int>(v));   // shrinks but has copy overhead
```

Insert & Construct Elements in Place

$\mathcal{O}(n)$ Worst Case

```
vector<pair<string,int>> v {{"a",1},{"w",7}};
```

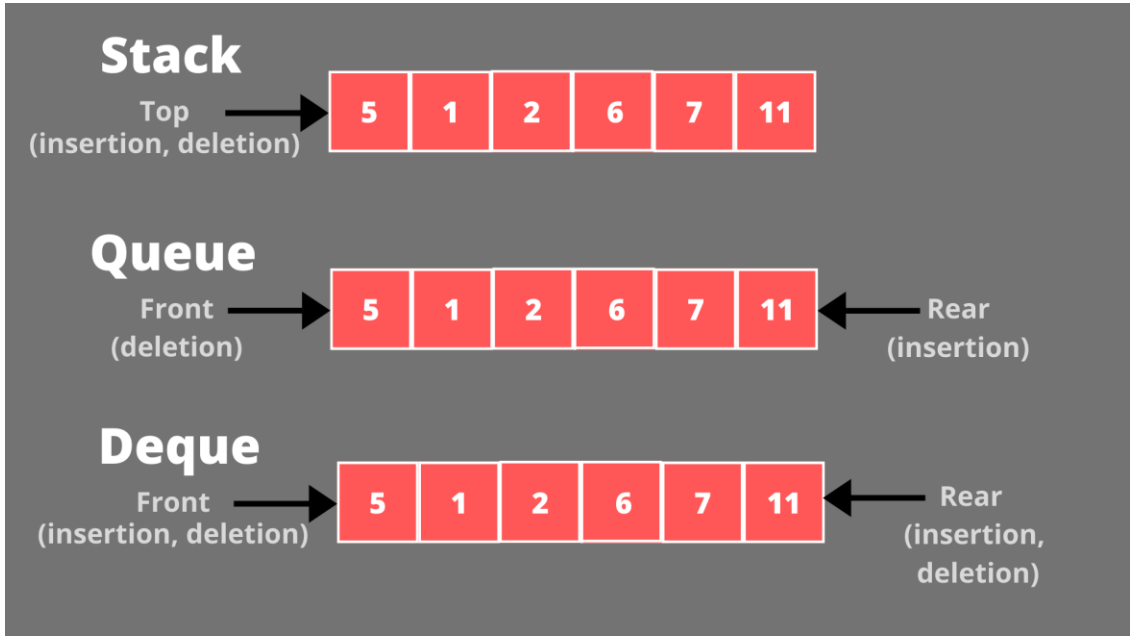


v1.1

deque container

- deque (double ended queue)
  - #include <deque> 필요
  - 양쪽 끝에서 삭제와 입력 모두 수행 가능
  - 메모리의 할당 정책과정에서 vector의 단점을 개선
  - 데크 끝과 시작 부분에 효율적으로 원소를 추가하거나 삭제 가능
- 특징
  - 벡터와는 달리 모든 원소가 메모리 상에 연속적으로 존재하지 않을 수 있음

- 컨테이너 비교



deque container

- 할당관련 메소드

  (참고로 벡터와는 다르게 capacity 와 reserve 없음)

  - size : 데크의 size 를 리턴한다 (현재 원소의 개수)
  - max_size : 데크 최대 크기를 리턴
  - resize : 데크가 size 개의 원소를 포함하도록 변경
  - empty : 데크가 비었는지 체크

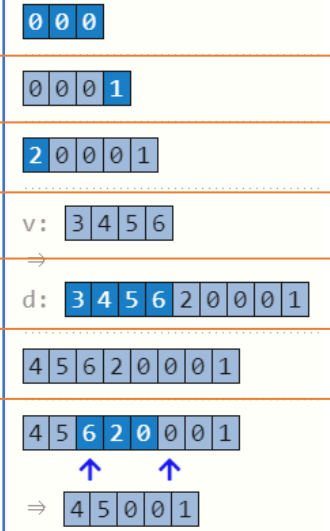
- 수정자(Modifier) 메소드
  - assign : 데크에 원소를 집어넣는다.
  - push_back : 데크 끝에 원소를 집어 넣는다.
  - push_front : 데크 맨 앞에 원소를 집어 넣는다.
  - pop_back : 마지막 원소를 제거한다.
  - pop_front : 첫번째 원소를 제거한다.
  - insert : 데크 중간에 원소를 추가
  - erase : 원소를 제거한다.
  - swap : 다른 데크와 원소와 교환한다.
  - clear : 원소를 모두 제거한다.

deque container

- deque의 사용 예시

```
#include <iostream>
#include <vector>
#include <deque>
using namespace std;

// dq의 모든 내용물 출력하기
template <typename T>
void print_deque(deque<T>& dq) {
  typename deque<T>::iterator itr;
  cout << "[ ";
  for (itr = dq.begin(); itr != dq.end(); itr++) {
    cout << *itr << " ";
  }
  cout << "]" << endl;
}
```

```
int main() {
    deque<int> d {0, 0, 0};
    d.push_back(1);
    d.push_front(2);
    vector<int> v {3, 4, 5, 6};
    d.insert(d.begin(), v.begin(), v.end());
    d.pop_front();

    d.erase(d.begin()+2, d.begin()+5);
    print_deque(d);
    for(int x: d) cout << x << " ";
}
```



std::deque<*ValueType*>

"double-ended queue" #include <deque> h/cpp hackingcpp.com

Construct A New Deque Object

```
deque<int> d1 {2,9,1,8,5,4}
deque<int> d2 (begin(d1)+3, end(d1))
deque<int> d3 (5,3)
deque<int> deep_copy_of_d1 (d1)
```

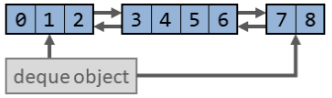
C++17 value type deducable from argument type

```
deque d4 {7,4,2};   // deque<int>
```

Typical Memory Layout

Note that the ISO standard only specifies the properties of deque (e.g., constant-time insert at both ends) but not how that should be implementd.

dynamically allocated contiguous chunks



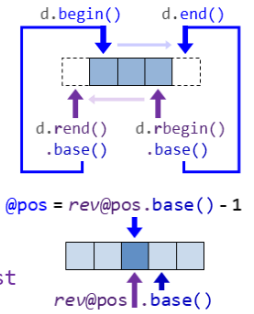
Obtain Iterators

$\mathcal{O}(1)$ Random Incrementing

```
.begin()  → @first
.end()    → @one_behind_last
```

Obtain *Reverse* Iterators

```
.rbegin() → rev@last
.rend()   → rev@one_before_first
```



Assign New Content To An Existing Deque

```
deque<int> d1 {8,5,3};
deque<int> d2 {6,8,1,9};
d1      =     d2;
```

(deep copy from source)

new state of d1

```
8 5 3 = 6 8 1 9   → 6 8 1 9
8 5 3 .assign({4,1,3,5})   → 4 1 3 5
8 5 3 .assign(2,1)         → 1 1
8 5 3 .assign(@InBeg,@InEnd) → 2 1 1 2
```

source container 3 2 1 1 2 3

Query Size (= Number of Elements) $\mathcal{O}(1)$

```
8 5 3 .empty() → false
8 5 3 .size()  → 3
```

Change Size $\mathcal{O}(|n - \text{newSize}|)$

```
8 5 3 .resize(2)   → 8 5
8 5 3 .resize(4,1) → 8 5 3 1
8 5 3 .resize(6,1) → 8 5 3 1 1 1
8 5 3 .clear()     →
```

Append Elements $\mathcal{O}(1)$

```
8 5 3 .push_back(7)  → 8 5 3 7
```

Prepend Elements $\mathcal{O}(1)$

```
8 5 3 .push_front(7) → 7 8 5 3
```

Insert Elements at Arbitrary Positions $\mathcal{O}(n)$ Worst Case

```
deque<int> d {8,5,3};
8 5 3 .insert(begin(d)+1, 7)          → 8 7 5 3
8 5 3 .insert(begin(d)+1, 3,7)        → 8 7 7 7 5 3
8 5 3 .insert(begin(d)+1, {6,9,7})    → 8 6 9 7 5 3
8 5 3 .insert(begin(d)+1, @InBegin,@InEnd) → 8 1 8 9 5 3
```

source container 3 1 8 9 2 3

Get Element Values $\mathcal{O}(1)$ Random Access

```
2 8 5 3 [1]    → 8
2 8 5 3 .front() → 2
2 8 5 3 .back()  → 3
```

Change Element Values

```
2 8 5 3 [1] = 7   → 2 7 5 3
2 8 5 3 .front() = 7 → 7 8 5 3
2 8 5 3 .back()  = 7 → 2 8 5 7
```

Out of Bounds Access

```
2 8 5 3 [6]      → Undefined Behavior
2 8 5 3 .at(6)   → Throws Exception std::out_of_range
```

Invalid Index!

Erase Elements At The Ends $\mathcal{O}(1)$

```
4 8 5 6 .pop_back()  → 4 8 5
4 8 5 6 .pop_front() → 8 5 6
```

Erase Elements At Arbitrary Positions

```
deque<int> d {4,8,5,6};
```

$\mathcal{O}(n)$ Worst Case Complexity

```
4 8 5 6 .erase(begin(d)+2)          → 4 8 6
4 8 5 6 .erase(begin(d)+1, begin(d)+3) → 4 6
```

Insert & Construct Elements in Place $\mathcal{O}(n)$ Worst Case

```
deque<pair<string,int>> d {{"a",1},{"w",7}};
```

constructor parameters

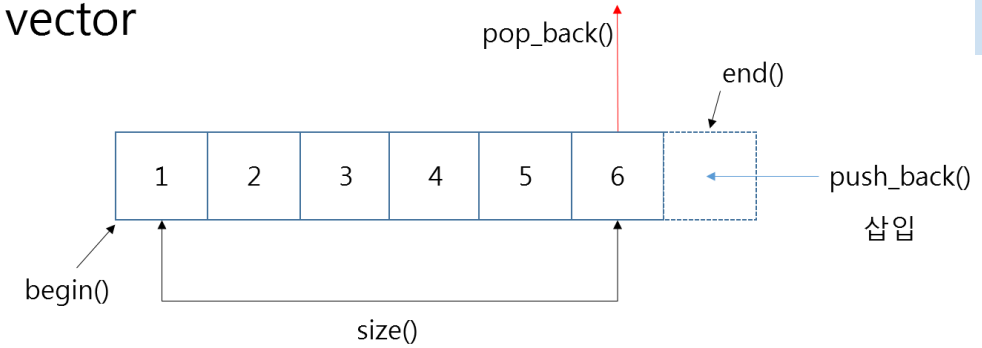
```
{a,1}{w,7} .emplace_back ("b",4)   → {a,1}{w,7}{b,4}
{a,1}{w,7} .emplace_front("c",6)   → {c,6}{a,1}{w,7}
{a,1}{w,7} .emplace(begin(d)+1,"z",5) → {a,1}{z,5}{w,7}
```

STL iterator

Standard Template Library

1. vector에 적용 예시

STL 반복자 (iterator)



- 원소의 삽입과 삭제 (반복자의 활용)

```
#include <iostream>
#include <vector>
using namespace std;

// 컨테이너의 종류와 내용물을 타입에 상관없이 작동
template <typename Iter>
void print_elements(Iter begin, Iter end) {
  cout << "[ ";
  while (begin != end) {
    cout << *begin << " ";
    begin++;
  }
  cout << "]" << endl;
}

int main() {
  // index           0,1,2,3,4,5,6,7,8,9
  vector <int> v = { 1,2,3,4,5,6 };
  cout << "처음 벡터 상태" << endl;
  print_elements(v.begin(), v.end());

  v.insert(v.begin()+6, 1);
  v.insert(v.begin()+6, 2);
  v.insert(v.begin()+6, 3);
  cout << "insert() 이후 " << endl;
  print_elements(v.begin(), v.end());

  cout << "값이 2 인 원소를 삭제" << endl;
  for(vector<int>::size_type i=0; i!=v.size(); i++) {
    if(v[i] == 2) {
      v.erase(v.begin() + i);
      //i번 인덱스 원소가 삭제됐고 뒷 내용물이 앞으로 당겨짐.
      i--; //그래서 i를 감소시켜야 함.
    }
  }
  print_elements(v.begin(), v.end());
}
```

```
처음 벡터 상태
[ 1 2 3 4 5 6 ]
insert() 이후
[ 1 2 3 4 5 6 3 2 1 ]
값이 2 인 원소를 삭제
[ 1 3 4 5 6 3 1 ]
```

STL algorithm

Standard Template Library

1. sort
2. copy
3. remove_if
4. tramsform
5. find
6. for_each

STL 알고리즘 **sort** (primitive type)

- 기본 데이터 타입 sort 예시

```
#include <iostream>
#include <array>
#include <vector>
#include <deque>
#include <algorithm>
using namespace std;

template <typename Iter>
void print_elements(Iter begin, Iter end) {
  cout << "[ ";
  while (begin != end) {
    cout << *begin << " ";
    begin++;
  }
  cout << " ]" << endl;
}

int main() {
    int ary[] = { 5,3,1,2,4,3,7 };
    cout << "initial array sequence\n";
    print_elements(begin(ary), end(ary));
    vector<int> vec(begin(ary), end(
    cout << "initial vector sequence
    print_elements(vec.begin(), vec.

    cout << "vector sort by iterator
    sort(vec.begin(), vec.end());
    print_elements(vec.begin(), vec.

    cout << "vector sort by reverse
    sort(vec.rbegin(), vec.rend());
    print_elements(vec.begin(), vec.end());

    deque <int> deq(begin(ary), end(ary));
    cout << "initial deque sequence\n";
    print_elements(deq.begin(), deq.end());

    cout << "deque sort by less<int>()\n";
    sort(deq.begin(), deq.end(), less<int>());
    print_elements(deq.begin(), deq.end());

    cout << "deque sort by greater<int>()\n";
    sort(deq.begin(), deq.end(), greater<int>());
    print_elements(deq.begin(), deq.end());
}
```

```
initial array sequence
[ 5 3 1 2 4 3 7 ]
initial vector sequence
[ 5 3 1 2 4 3 7 ]
vector sort by iterator
[ 1 2 3 3 4 5 7 ]
vector sort by reverse iterat
[ 7 5 4 3 3 2 1 ]
initial deque sequence
[ 5 3 1 2 4 3 7 ]
deque sort by less<int>()
[ 1 2 3 3 4 5 7 ]
deque sort by greater<int>()
[ 7 5 4 3 3 2 1 ]
```

STL 알고리즘 **sort** (custom type)

- 사용자 정의 타입 sort 예시

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct noco {
    int node;
    int cost;
};

template <typename Iter>
void print_elements(Iter begin, Iter end) {
  while (begin != end) {
    cout << *begin << "\n";
    begin++;
  }
}
```

```
ostream& operator<<(ostream& os, const noco& nc) {
    os << '(' << nc.node << ", " << nc.cost << ')';
    return os;
}

int main() {
    vector<noco> nc;
    nc.push_back({1, 90});
    nc.push_back({2, 80});
    nc.push_back({3, 70});
    nc.push_back({4, 60});
    nc.push_back({3, 75});
    nc.push_back({3, 85});
    nc.push_back({4, 85});
    nc.push_back({4, 65});

    // sort() 함수를 사용하려하면,
    // < 연산자가 정의되지 않았다는 에러가 발생한다.
    //sort(nc.begin(), nc.end());
    print_elements(nc.begin(), nc.end());
}
```

```
(1, 90)
(2, 80)
(3, 70)
(4, 60)
(3, 75)
(3, 85)
(4, 85)
(4, 65)
```

STL 알고리즘 **sort** (custom type) cont.

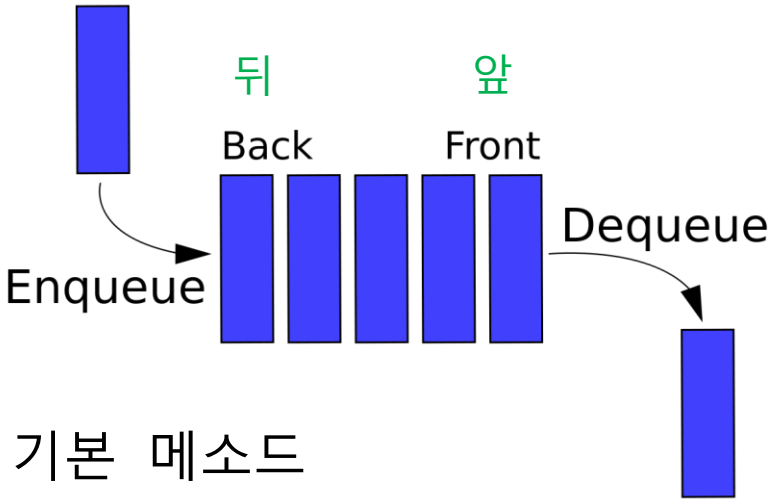
- 사용자 정의 타입 sort 예시

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct noco {
    int node;
    int cost;

    //noco(int node, int cost) : node(node), cost(cost) {}
    //sort() 함수가 사용할 < 연산자를 정의한다.
    bool operator<(const noco& nc) {
        if(cost != nc.cost)  // cost가 다르면,
            return cost > nc.cost;  // cost 내림차순
        else  // cost가 같으면,
            return node > nc.node;  // node 내림차순
    }
};
 :
```

```
  :
  :

int main() {
    vector<noco> nc;
    nc.push_back({1, 90});
    nc.push_back({2, 80});
    nc.push_back({3, 70});
    nc.push_back({4, 60});
    nc.push_back({3, 75});
    nc.push_back({3, 85});
    nc.push_back({4, 85});
    nc.push_back({4, 65});

    sort(nc.begin(), nc.end());
    print_elements(nc.begin(), nc.end());
}
```

queue container

- Queue
  - 대표적인 FIFO(First In First Out) 구조



  - 기본 메소드
    - push, pop, empty, front, back

- 선언과 사용 예

```
#include <stdio.h>
#include <queue>
using namespace std;

int main(void) {
    queue<int> q;
    q.push(1);      q.push(2);
    q.push(10);     q.push(20);

    while(!q.empty()) {
        printf("%d\n", q.front());
        q.pop();  // 윈소 제거
    }
}
```

```
1
2
10
20
```

queue container

- Queue의 내용물을 확인하려면?

```
#include <stdio.h>
#include <queue>
using namespace std;

// 아래와 같이 복사본을 인수로 받아서 확인한다.
// pop()을 호출하는 순간 큐의 내용물이 바뀌기 때문
void output_Q(queue<int> Q) {
  printf("Q:[");
  while(!Q.empty()) {
    printf("%2d ", Q.front());
    Q.pop();
  }
  printf("], ");
}
```

Q	1	3	5	7	9

```
int main(void) {
    queue<int> q;

    // Q에 1부터 10사이 홀수를 채운다.
    for(int i=1; i<10; i+=2) {
        q.push(i);
    }

    output_Q(q);
}
```

q	1	3	5	7	9

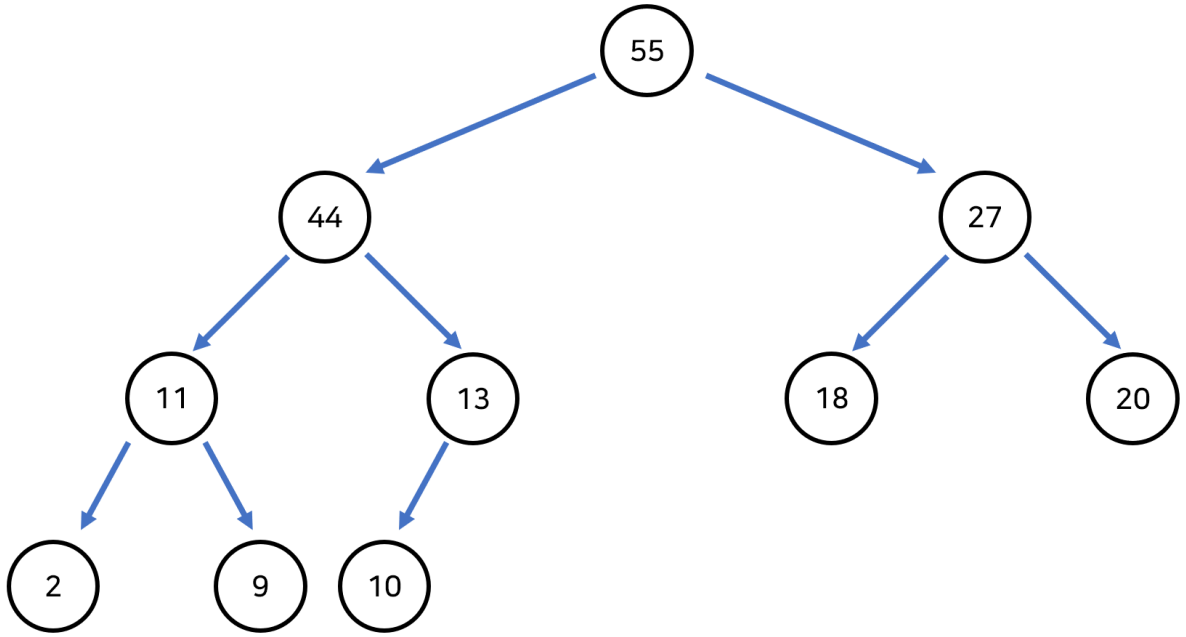
힙(heap)

- 정의
  - 영단어 힙(heap)은 ' 무엇인가를 차곡차곡 쌓아올린 더미 ' 라는 뜻
  - 힙은 완전 이진 트리로 구현된 자료구조
  - 가장 크거나 작은 값을 찾아내는 연산을 빠르게 하기위해 고안된 자료구조

- 힙의 종류
  - 최대힙(Max Heap) : 부모 노드의 값이 무조건 자식 노드의 값보다 큰 힙
  - 최소힙(Min Heap) :부모 노드의 값이 무조건 자식 노드의 값보다 작은 힙

힙(heap)

- 최대 힙의 예

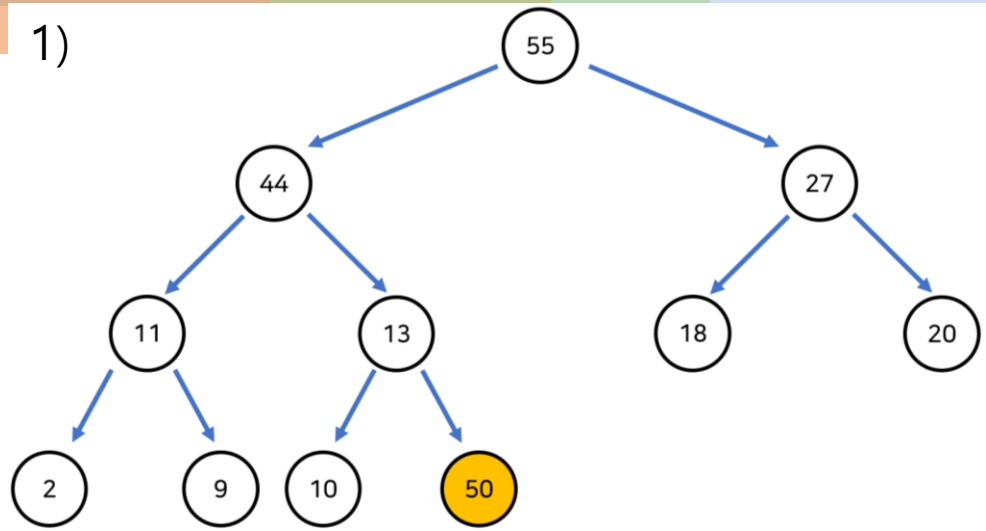


- Root의 값이 가장 크다.
- 또한 모든 서브트리에 대해서도 같다.

- 자료의 추가

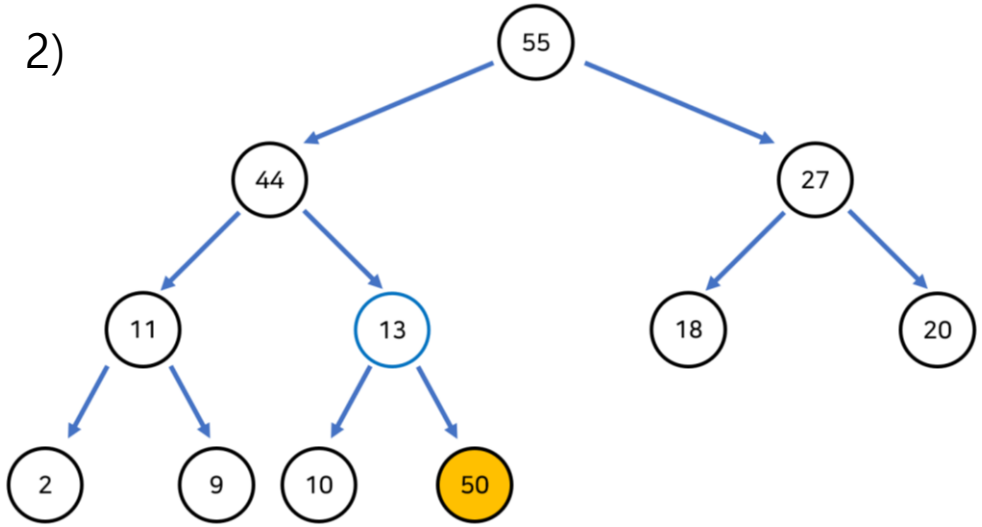
1) 새로운 노드를 트리의 맨 뒤에 추가한다. (완전 이진 트리의 형태를 깨면 안됨)
2) 추가된 노드와 부모 노드를 비교하여 자식 노드가 크다면 서로의 위치를 바꾼다.
3) 2번 작업을 부모 노드가 더 클때까지 반복한다.

1)



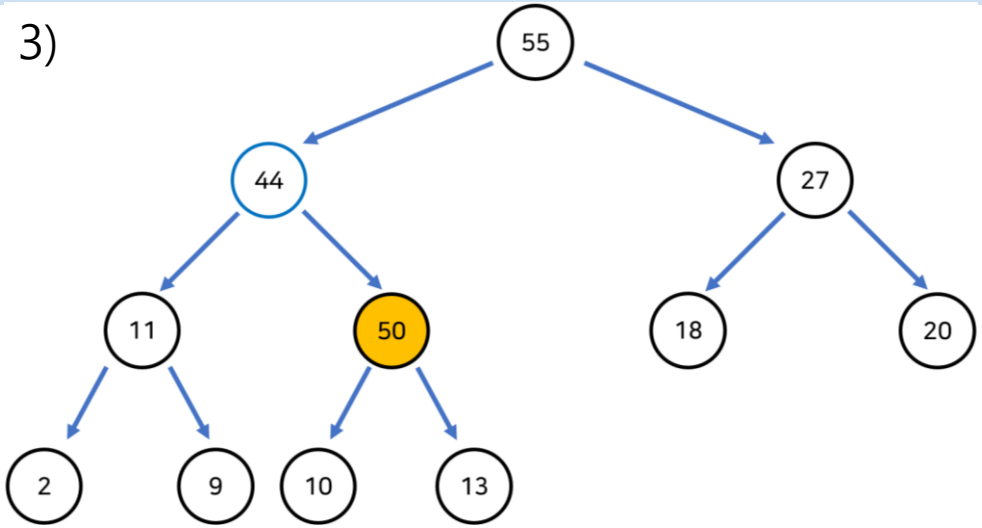
MaxHeap에 50의 값을 가진 노드를 추가하려는 상황이다. 그러면 이 노드는 트리의 맨 뒤인 13의 rightChild로 들어가게 된다.

2)



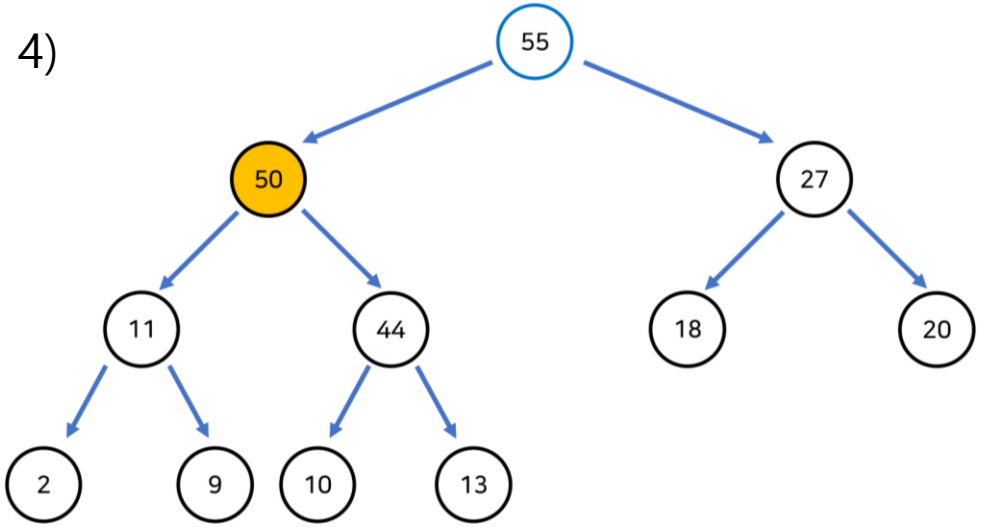
50의 부모노드인 13과 비교한다.
50이 크므로 둘의 자리를 바꿔준다.

3)



자리를 바꾸고 또 바꾼 자리의 부모노드인 44와 비교한다. 또 50이 크므로 자리를 바꿔준다.

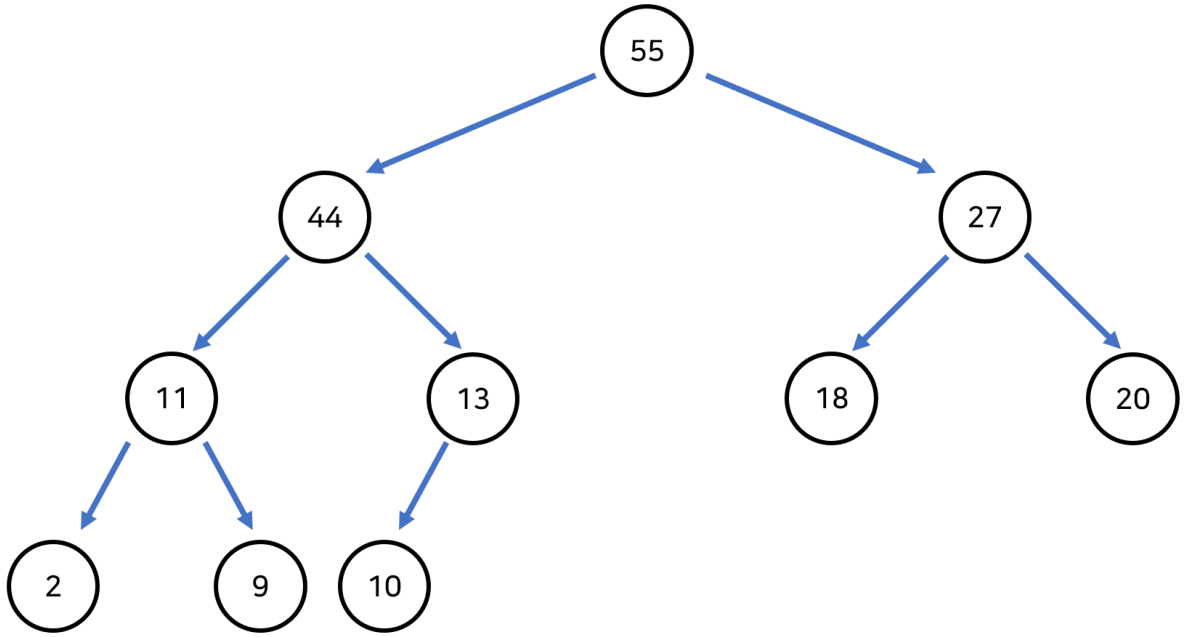
4)



부모 노드인 55와 비교하지만 55가 크므로 자리를 고정하고 자료의 추가가 완료된다.

힙(heap)

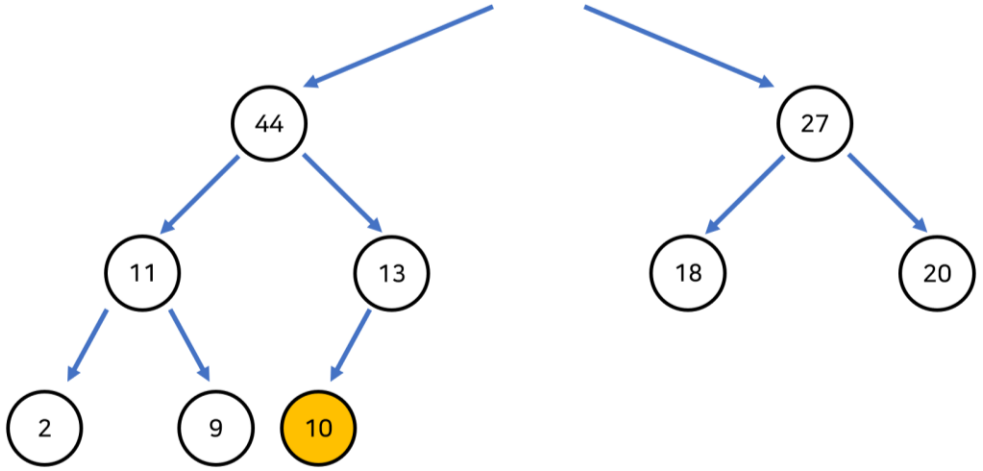
- 자료의 삭제



- 자료가 삭제되는 경우는 맨 위에 있는 Root노드가 빠지는 경우밖에 없다.
- 그렇게되면 다시 힙의 형태를 갖추어야...

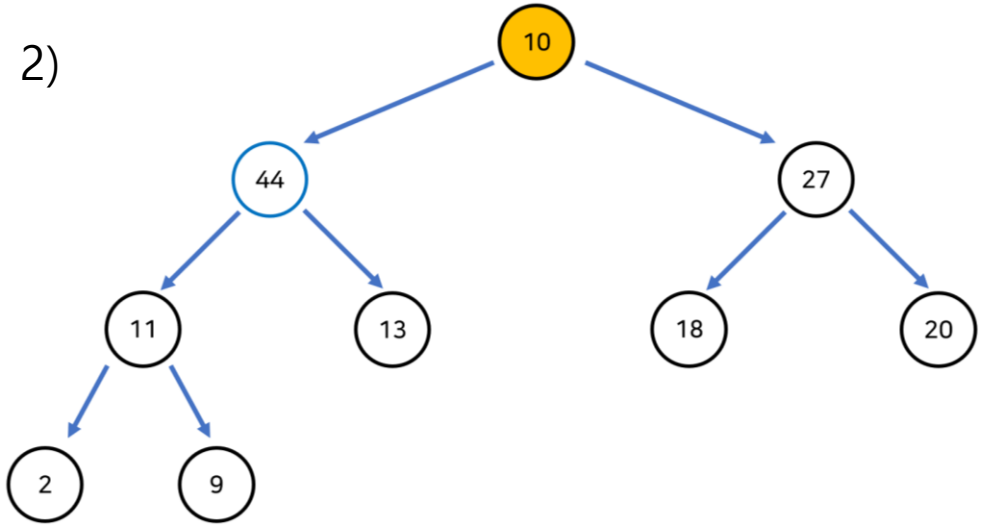
- 삭제 알고리즘
  1) 맨 뒤에있는 노드를 Root자리로 옮긴다.
  2) 자식 노드중 값이 더 큰 노드와 비교하여 자식 노드가 값이 더 크다면 위치를 바꾼다.
  3) 2번의 작업을 자식노드보다 자신이 클때까지 반복한다.

1)



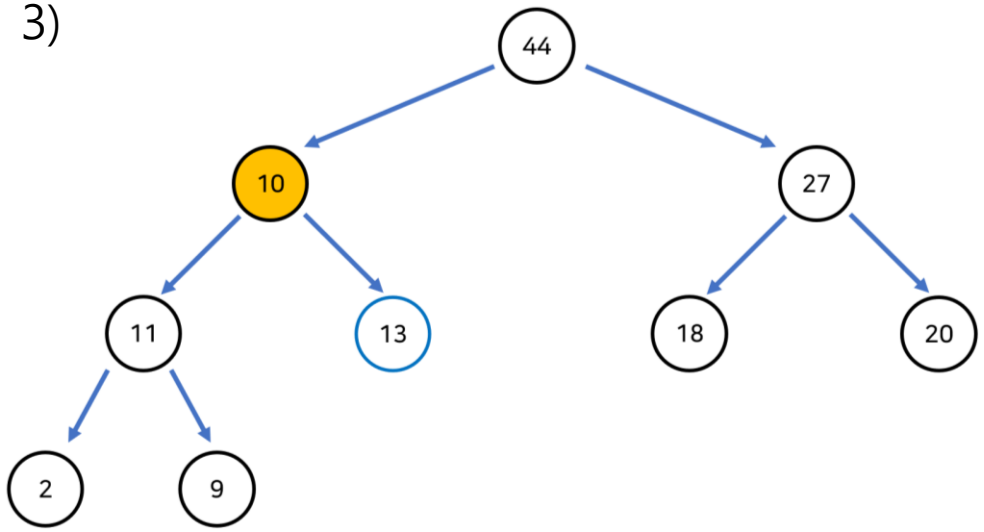
Root노드가 빠진 상황이다. 이렇게 되면 맨뒤에 있던 노드인 10을 Root로 올린다.

2)



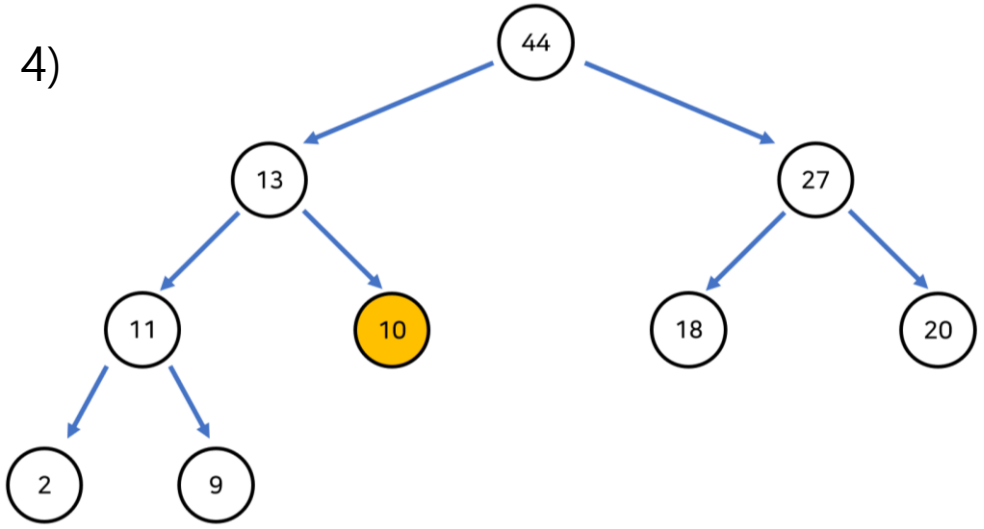
Root로 올리고 자식들과 비교를 시작하게 되는데, 자식 둘 중에 44가 더 크므로 44와 비교를 한다. 그런데 44가 10보다 크므로 44와 자리를 바꾼다.

3)



자리를 바꾸고 다시 같은 작업을 반복한다. 11과 13중 13이 크므로 13과 비교한다. 13이 더크므로 자리를 바꾼다.

4)



그 결과 이렇게 힙의 구조를 다시 유지할수 있게 되었다.

priority_queue container

- 메소드 정리
  - `bool empty();`
    - 비어있으면 `true` 반환
    - 비어있다는 것은 `size`가 `0` 이기도함.
  - `size_type size();`
    - 원소의 개수를 반환합니다.
  - `value_type& top();`
    - 맨 위에있는 원소를 참조 및 반환 합니다(삭제하는거 아님에 유의)
  - `void push(value_type& val);`
    - 인자를 삽입합니다. 내부적으로는 `push_back` 함수를 이용하여 삽입이 됩니다.
  - `void pop();`
    - 맨위에있는 인자를 삭제합니다.
    - 내부적으로는 `pop_heap` 알고리즘과 `pop_back` 함수가 이용되어 우선순위 큐 형태를 유지합니다.

priority_queue container

- 우선순위 큐 테스트

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
    priority_queue <int> pq;
    pq.push(45);
    pq.push(90);
    pq.push(30);
    pq.push(10);
    pq.push(49);
    pq.push(22);
    pq.push(14);

    while(!pq.empty()) {
        int t = pq.top(); pq.pop();
        cout << t << ' ‘;
   }
}
```

90 49 45 30 22 14 10

max heap 으로 작동함.

- min heap(최소값 우선)으로 하려면?

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
    //priority_queue<자료형, 구현체, 비교연산자>
    priority_queue<int, vector<int>, less<int>> pq;
    pq.push(45);
    pq.push(90);
    pq.push(30);
    pq.push(10);
    pq.push(49);
    pq.push(22);
    pq.push(14);

    while(!pq.empty()) {
        int t = pq.top(); pq.pop();
        cout << t << ' ';
   }
}
```

10 14 22 30 45 49 90

min heap 으로 작동함.

priority_queue container

우선순위 큐를 배웠으니 중등부 5번 창고를 풀러 가보자!

- operator< 오버라이딩 방법

```
#include <iostream>
#include <queue>

using namespace std;

struct noco {
    int node;
    int cost;

    //C++에서 우선순위 큐는 기본 최대힙(오름차순)으로 구현
    //되어있으므로 반대 논리값을 리턴해야 함.
    bool operator<(noco nc) const {
        if(cost != nc.cost)
            return cost > nc.cost; // 오름차순
        else
            return node > nc.node;
    }
};
```

```
[4] (55)
[4] (60)
[3] (70)
[3] (75)
[2] (80)
[3] (85)
[4] (85)
[1] (90)
```

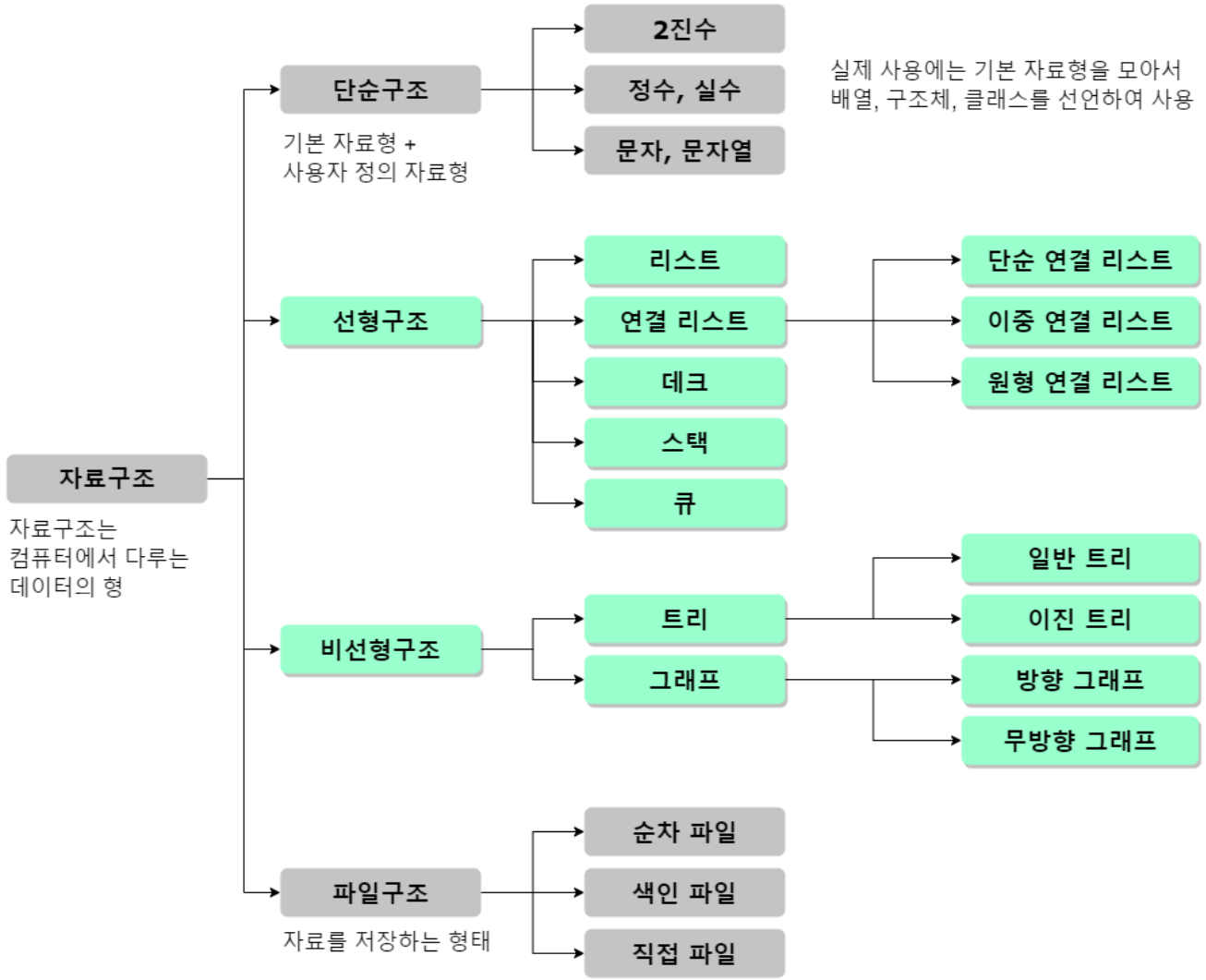
```
int main() {
    priority_queue <noco> pq;

    pq.push({1, 90});
    pq.push({2, 80});
    pq.push({3, 70});
    pq.push({4, 60});
    pq.push({3, 75});
    pq.push({3, 85});
    pq.push({4, 85});
    pq.push({4, 55});

    while(!pq.empty()) {
        noco t = pq.top();
        printf("[%d] (%d)\n", t.node, t.cost);
        pq.pop();
    }
    return 0;
}
```

자료구조

선형구조와 비선형구조

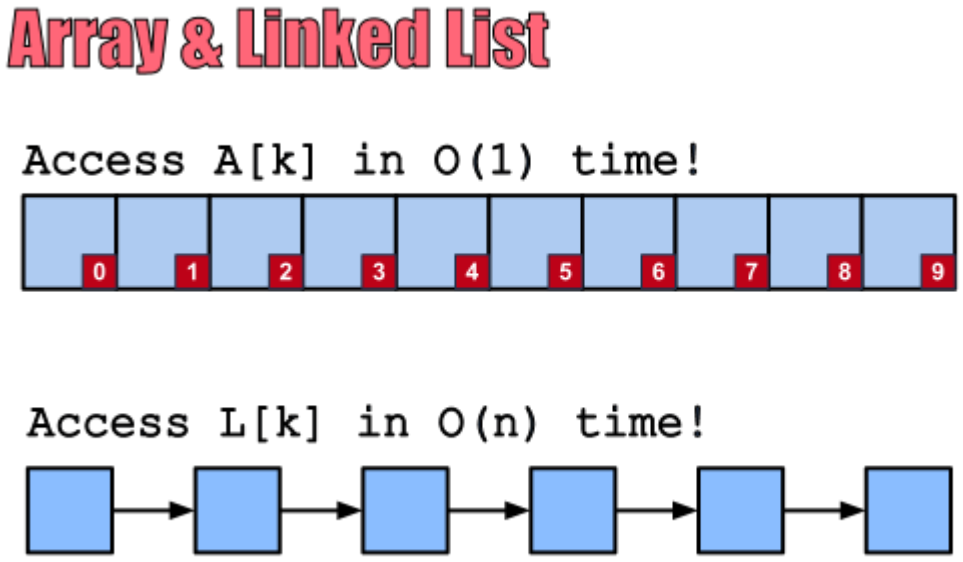
자료구조

- 자료구조(data structure)
  - 전산학에서 자료를 효율적으로 이용할 수 있도록 컴퓨터에 저장하는 방법이다.
  - 신중히 선택한 자료구조는 보다 효율적인 알고리즘을 사용할 수 있게 한다.

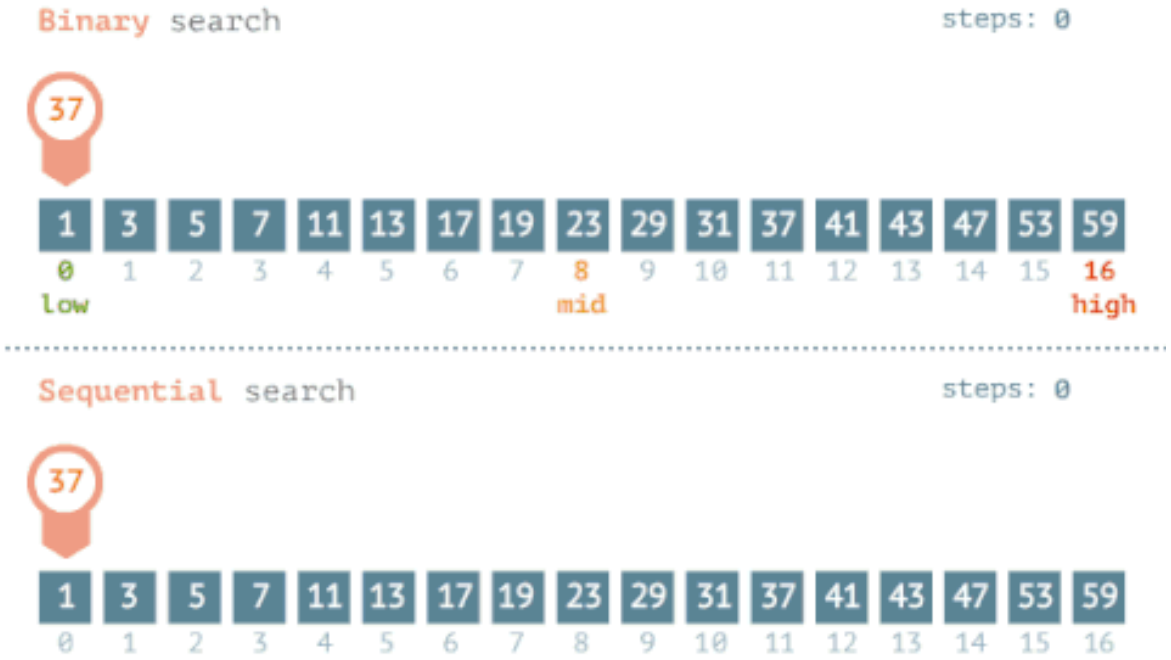


선형구조

- 선형구조란?
  - 자료를 구성하는 데이터를 순차적으로 나열시킨 형태를 의미

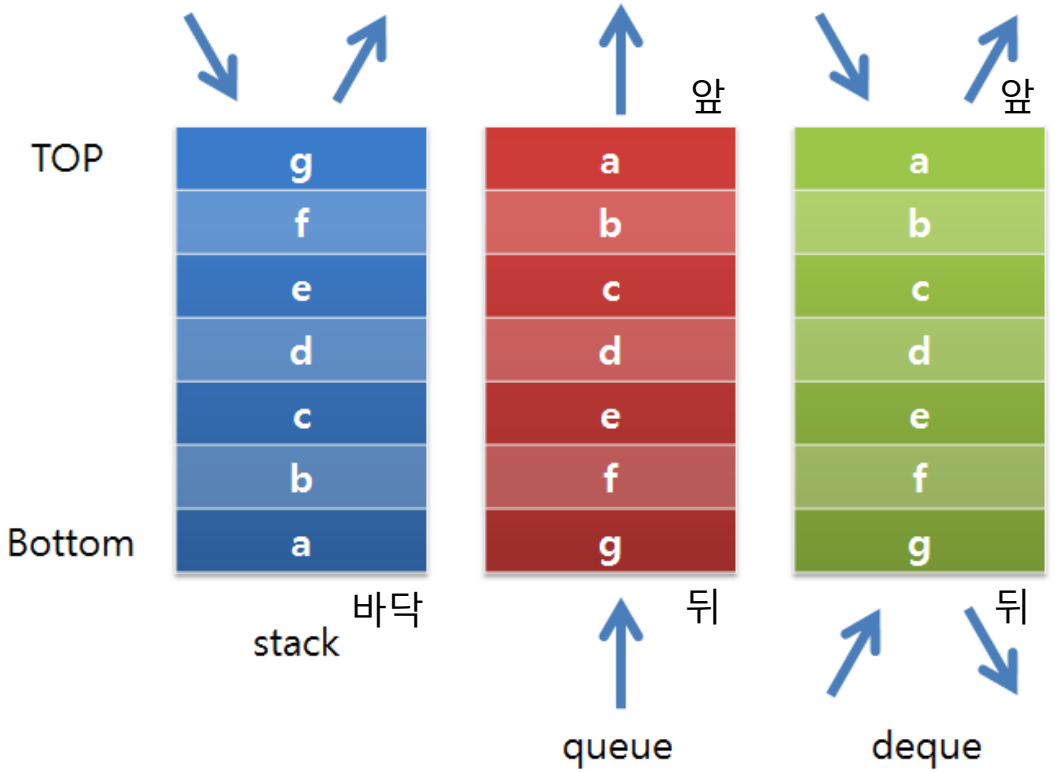


- 탐색법
  - 순차탐색
  - 이분탐색



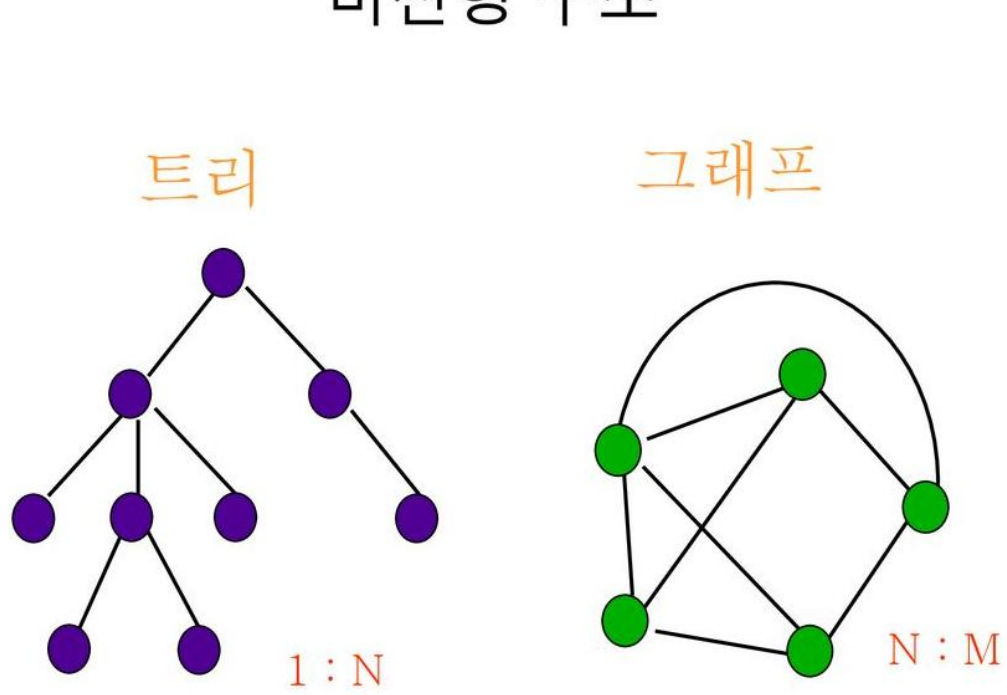
선형구조

- 배열
  - 고정 배열, 동적 배열(vector)
- 리스트
  - 연결리스트, 이중연결, 원형연결 리스트
- 스택
  - 후입선출(Last In First Out)
- 큐
  - 선입선출(First In First Out)
- 데크
  - Double Ended Queue



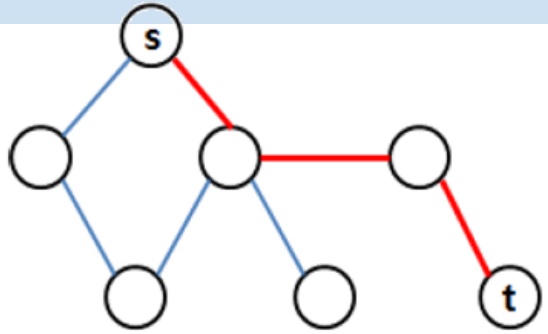
비선형구조

- 비선형구조란 i번째 원소를 탐색한 다음 그 원소와 연결된 다른 원소를 탐색하려고 할 때, 여러 개의 원소가 존재하는 탐색구조
- 트리나 그래프로 구성된 경우
- 선형과 달리 자료가 순차적이지 않으므로 단순히 반복문을 이용하여 탐색하기 어려움
- 비선형구조는 스택이나 큐와 같은 자료구조를 활용하여 탐색
- 비선형구조 탐색법
  - 깊이우선탐색(DFS, depth first search)
  - 너비우선탐색(BFS, breadth first searh)

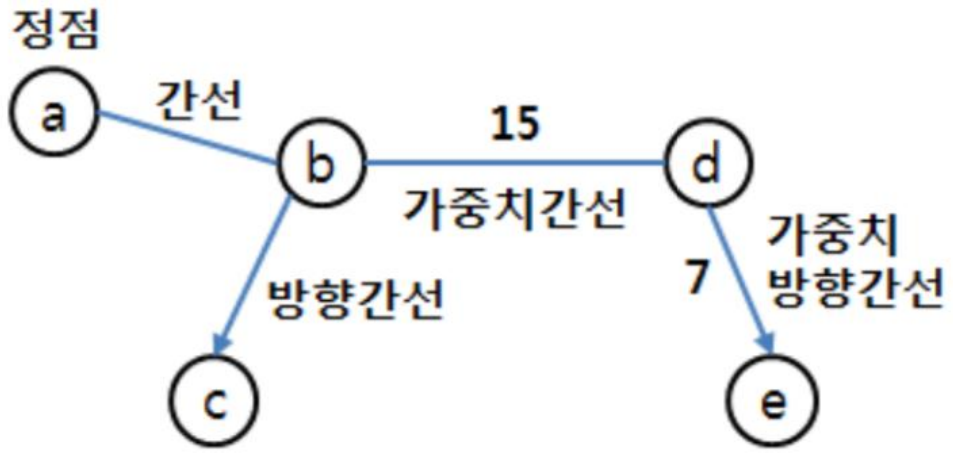


- 그래프 중에 회로가 없는 그래프를 트리라고 한다.

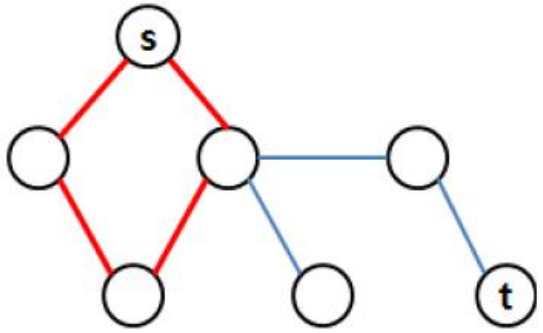
비선형구조 - 그래프



- 정점(vertex)
  - 노드(node)라 부르기도 한다
- 간선(edge)
  - 일반간선, 가중치 간선
  - 방향간선, 양방향간선, 무방향간선

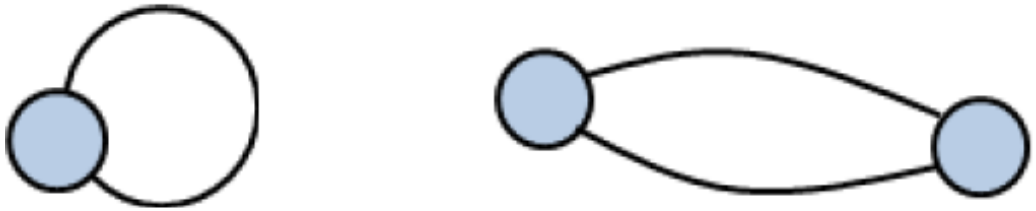


- 경로(path)
  - 임의의 정점 s에서 임의의 정점 t로 이동할 때, s에서 t로 이동하는데 사용한 정점들을 연결하고 있는 간선들의 순서로된 집합
- 회로(cycle)
  - 그래프에서 임의의 정점 s에서 같은 정점 s로의 경로들



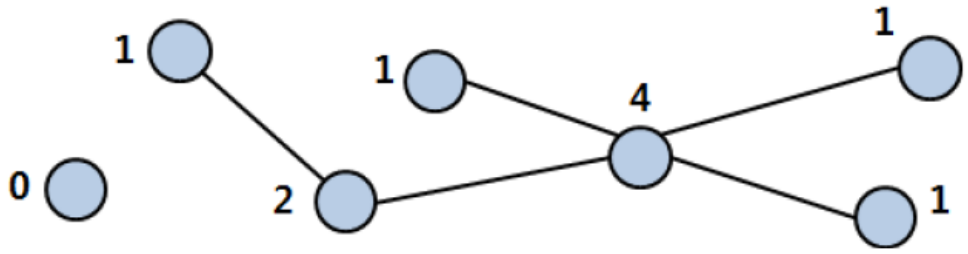
비선형구조 - 그래프

- 자기간선(loop)
  - 임의의 정점에서 자기 자신으로 연결하고 있는 간선
- 다중간선(multi edge)
  - 임의의 정점에서 다른 점점으로 연결된 간선의 수가 2개 이상일 경우



왼쪽은 자기간선 오른쪽은 다중간선을 나타낸다.

- 그래프의 차수
  - 그래프의 임의의 한 정점에서 다른 정점으로 연결된 간선의 수



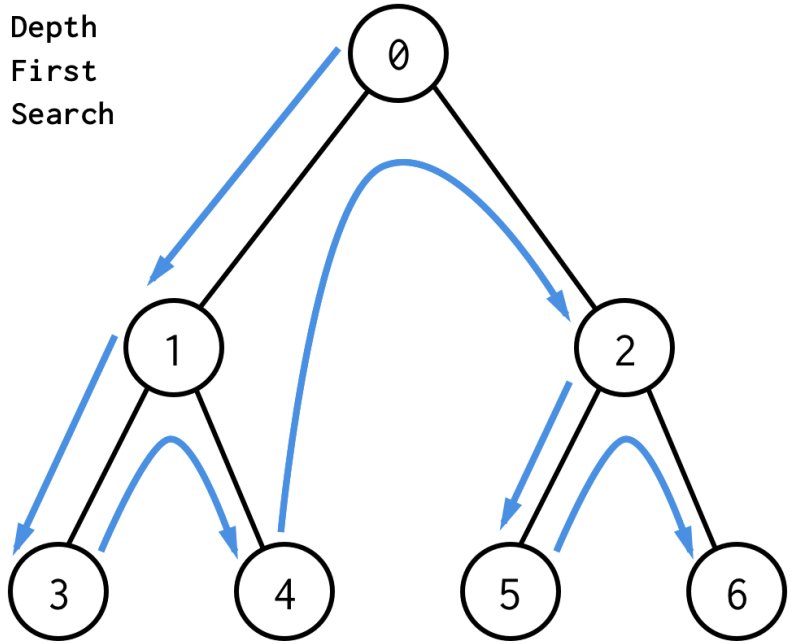
각 정점에서의 차수

비선형 탐색

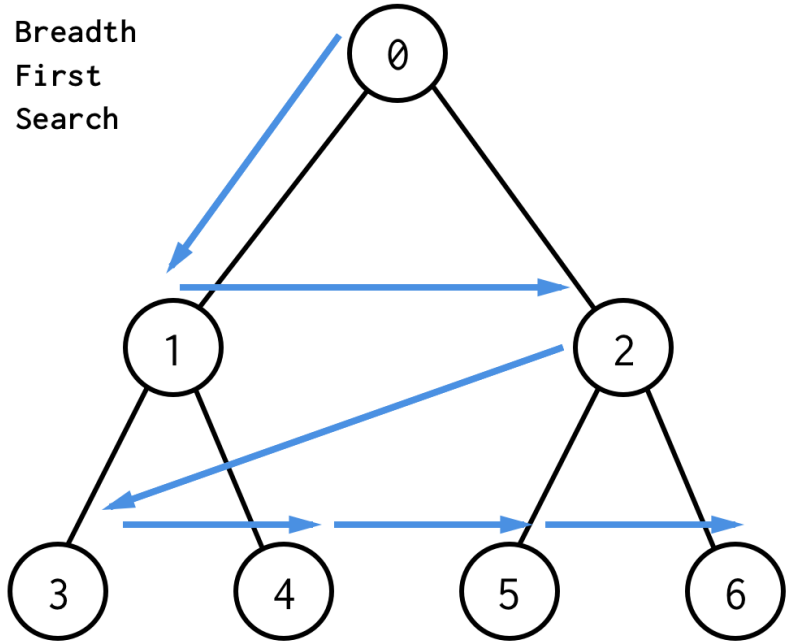
비선형구조의 전체탐색(DFS vs BFS)

탐색

- 깊이우선탐색(DFS)



- 너비우선탐색(BFS)



DFS(깊이우선탐색)	BFS(너비우선탐색)
현재 정점에서 갈 수 있는 점들까지 들어가면서 탐색	현재 정점에 연결된 가까운 점들부터 탐색
스택 또는 재귀함수로 구현	큐를 이용해서 구현

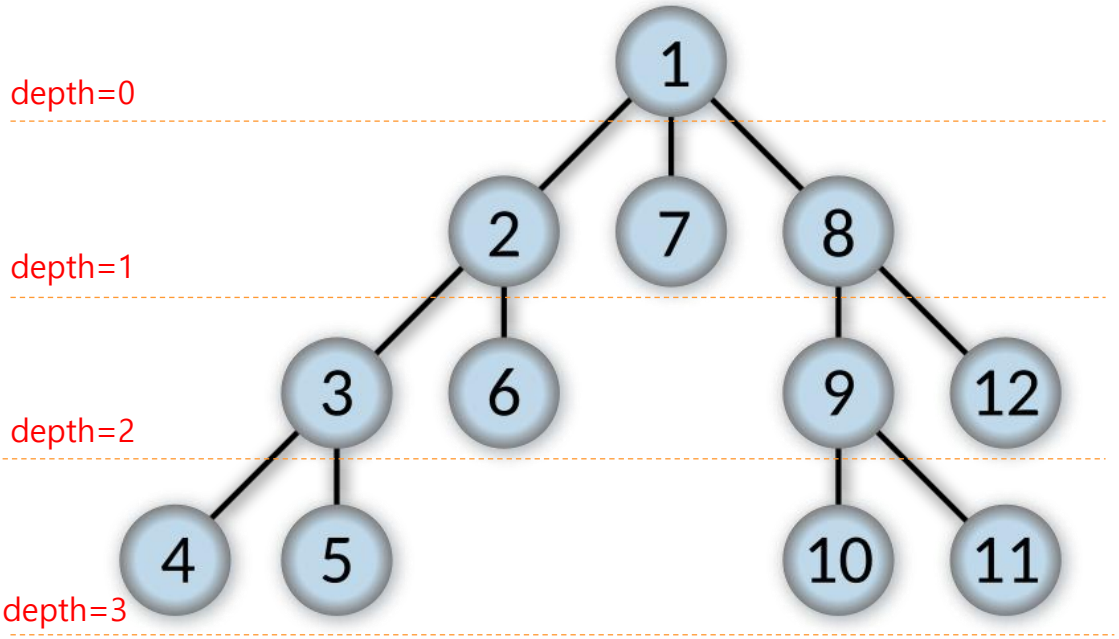
탐색

- 깊이우선탐색(DFS)
  - 최대한 깊이 내려간 뒤, 더이상 깊이 갈 곳이 없을 경우 옆으로 이동
  - 루트 노드(혹은 다른 임의의 노드)에서 시작해서 다음 분기(branch)로 넘어가기 전에 해당 분기를 완벽하게 탐색하는 방식
- 평가
  1. 깊이 우선 탐색(DFS)이 너비 우선 탐색(BFS)보다 좀 더 간단함
  2. 검색 속도 자체는 너비 우선 탐색(BFS)에 비해서 느림

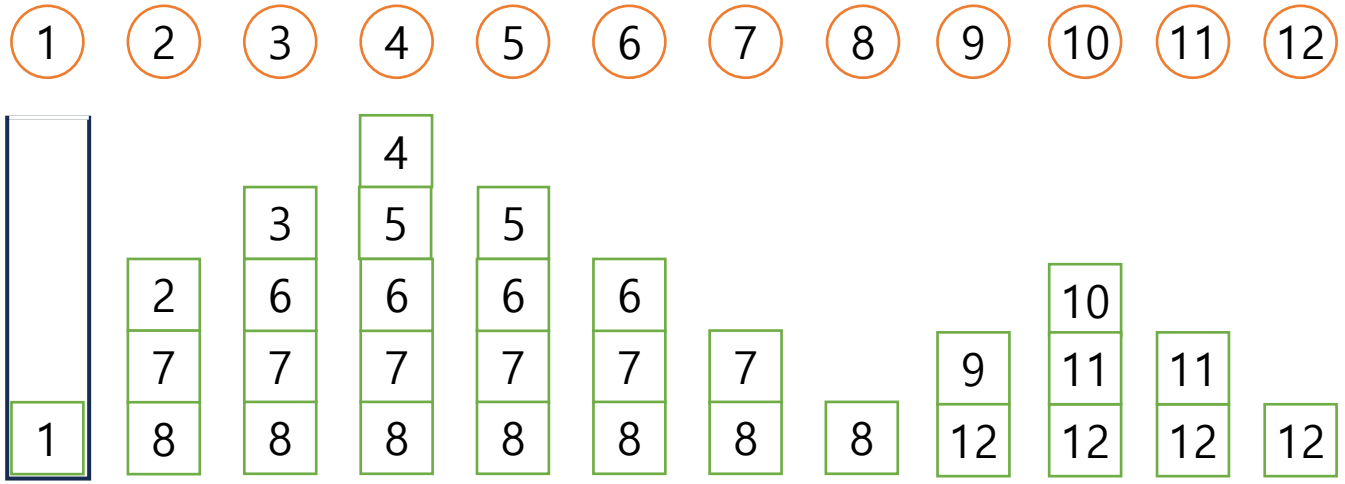
- 너비우선탐색(BFS)
  - 최대한 넓게 이동한 다음, 더 이상 갈 수 없을 때 아래로 이동
  - 루트 노드(혹은 다른 임의의 노드)에서 시작해서 인접한 노드를 먼저 탐색하는 방법
  - 시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회 방법

트리구조의 깊이우선탐색(DFS)

- 트리 구조



- 스택을 이용한 DFS 순회



- dfs(1)
- dfs(2),dfs(7),dfs(8)
- dfs(3),dfs(6),dfs(7),dfs(8)
- dfs(4),dfs(5),dfs(6),dfs(7),dfs(8)
- dfs(5),dfs(6),dfs(7),dfs(8)

계속 앞에 끼어든다

DFS 활용 순열 조합 1

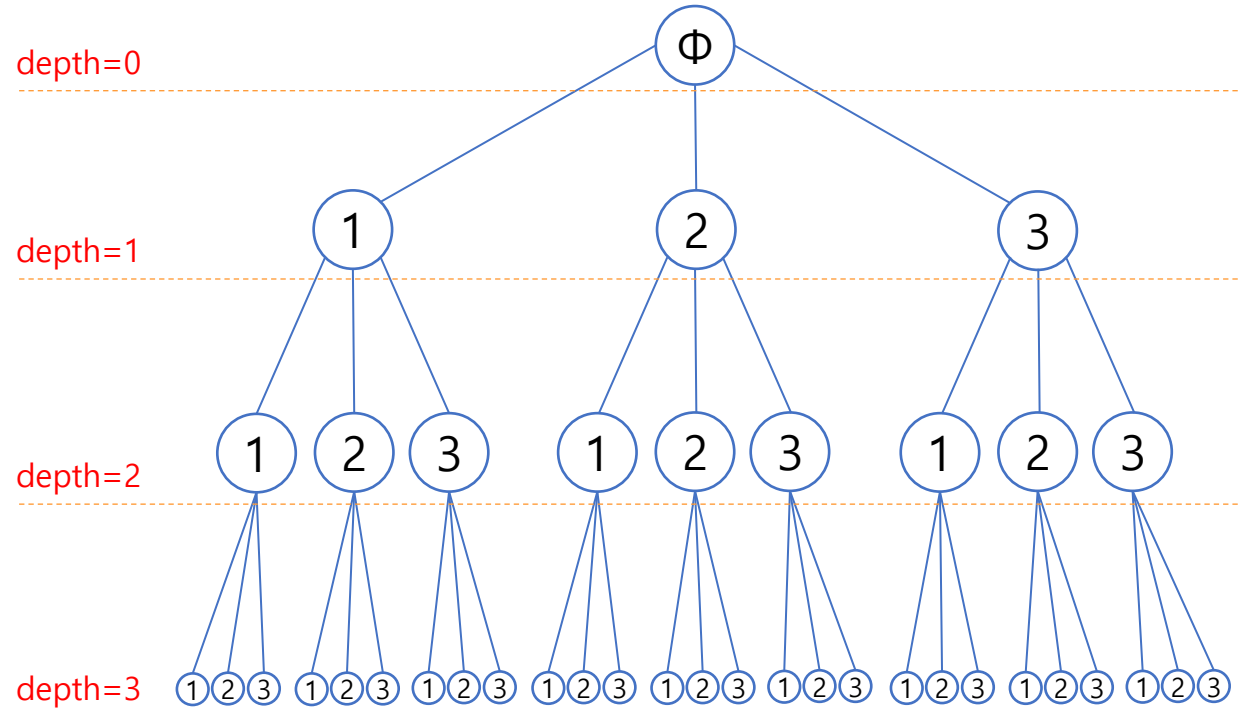
- 중복있고 순서있는 순열 조합
  - 중복 있다
    - 같은 숫자 다시 등장 가능

```
1-1-1  //
1-1-2  // OK
1-1-3  // OK
1-2-1  // OK
```

  - 순서 있다
    - 순서 다른 조합을 다른 것으로 간주

```
// 아래는 모두 다른 조합
1-2-3
2-3-1
3-2-1
```

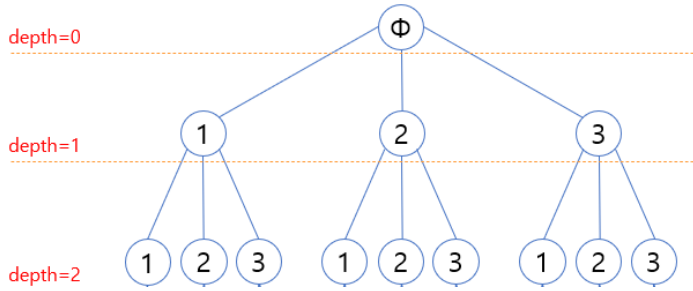
- 그래프로 표현
  - 1,2,3 중에서 뽑을 때,
  - 매 단계에서 3갈래로 갈라짐



DFS 활용 순열 조합 1

```
//중복있고 순서있는 순열 조합 만들기 (DFS구현)
#include <stdio.h>
#include <vector>
#define  MAX_N   10
using namespace std;

int n, r;
int d[] = {1,2,3,4,5,6,7,8,9,10};
vector<int> seq;

void output_seq(int depth, vector<int>& vec) {
    printf("[%d] ", depth);
    for(int a : vec)
        printf("%d-", a);
    printf("\b \n");
}
```

```
void dfs(int depth) {
    // 뽑은 개수 상관없이 작업하려면 여기에서
    output_seq(depth, seq);
    if(depth==r) {  // r개 모두 뽑았으면,
        //output_seq(depth, seq);
        return;     // 더 갈라지지 말고 돌아가!
    }
    //매 갈림길에서 똑같이 n개로 갈라짐을 구현
    for(int i=0; i<n; i++) {
        seq.push_back(d[i]); // 갈라지기 직전 노드 ①②③
        dfs(depth+1);
        seq.pop_back();
    }
}

int main() {
    // n개의 데이터 중에서 r개 뽑기
    scanf("%d %d", &n, &r);
    dfs(0);  // depth 0에서 r까지 가면 끝남
    return 0;
}
```




```
3 3
[0]
[1] 1
[2] 1-1
[3] 1-1-1
[3] 1-1-2
[3] 1-1-3
[2] 1-2
[3] 1-2-1
[3] 1-2-2
[3] 1-2-3
[2] 1-3
[3] 1-3-1
[3] 1-3-2
[3] 1-3-3
[1] 2
[2] 2-1
[3] 2-1-1
[3] 2-1-2
[3] 2-1-3
[2] 2-2
[3] 2-2-1
[3] 2-2-2
[3] 2-2-3
[2] 2-3
[3] 2-3-1
[3] 2-3-2
[3] 2-3-3
[1] 3
[2] 3-1
[3] 3-1-1
[3] 3-1-2
[3] 3-1-3
[2] 3-2
```

DFS 활용 순열 조합 2

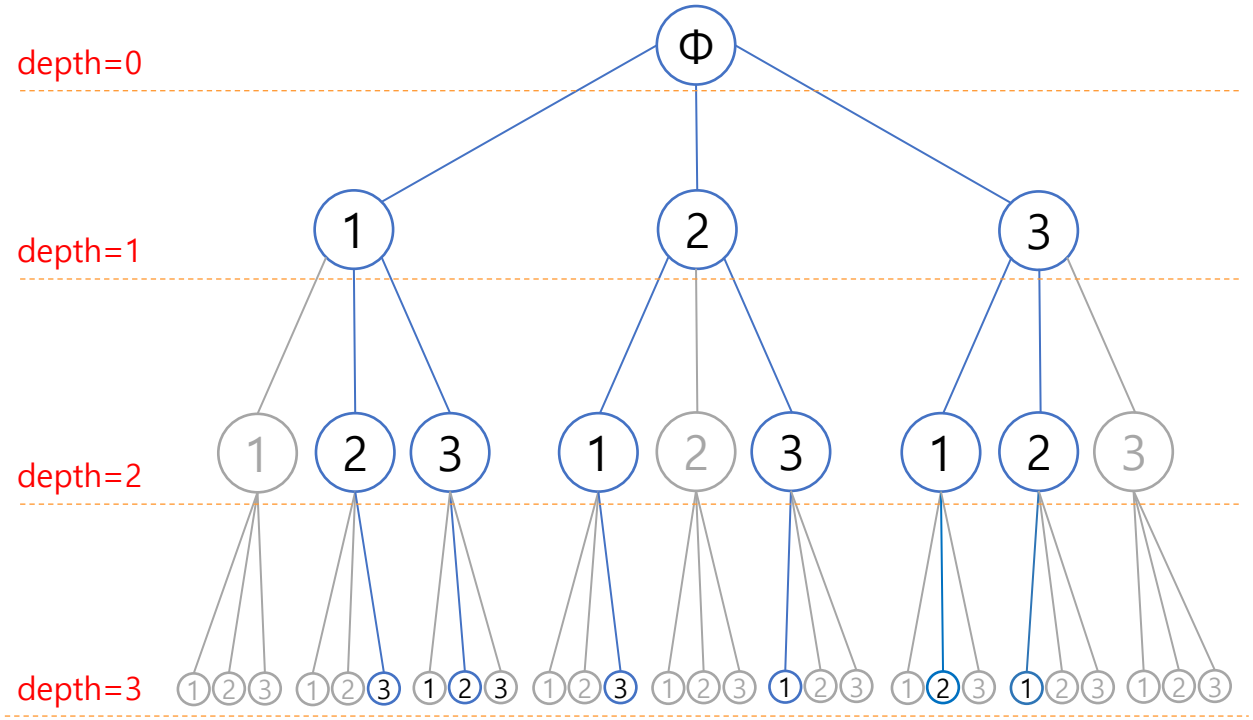
- 중복없고 순서있는 순열 조합
  - 중복 없다
    - 같은 숫자 다시 등장 불가능

```
1-1-2  // !
1-2-2  // !
1-2    // OK
1-2-3  // OK
```

  - 순서 있다
    - 순서 다른 조합을 다른 것으로 간주

```
// 아래는 모두 다른 조합
1-2-3
2-3-1
3-2-1
```

- 그래프로 표현
  - 1,2,3 중에서 뽑을 때,
  - 매 단계에서 3갈래로 갈라짐



DFS 활용 순열 조합 2

```
// 중복없고 순서있는 순열 조합 만들기 (DFS구현)
#include <stdio.h>
#include <vector>
#define  MAX_N   10
using namespace std;

int n, r;
int d[] = {1,2,3,4,5,6,7,8,9,10};
int used[MAX_N]; // 사용 여부를 저장하는 배열
vector<int> seq;

void output_seq(vector<int>& vec) {
    for(int a : vec)
        printf("%d-", a);
    printf("\b \n");
}
```

```
void dfs(int depth) {
    if(depth==r) {
        output_seq(seq);
        return;
    }
    // 매 노드에서 n개의 노드로 갈라짐을 구현
    for(int i=0; i<n; i++) {
        if(!used[i]) {
            used[i] = 1;  //사용했음을 표시
            seq.push_back(d[i]); // 사용한 노드
            dfs(depth+1);
            used[i] = 0;  //사용했음 취소
            seq.pop_back();
        }
    }
}

int main() {
    // n개의 데이터 중에서 r개 뽑기
    scanf("%d %d", &n, &r);
    dfs(0);
    return 0;
}
```

```
4 3
1-2-3
1-2-4
1-3-2
1-3-4
1-4-2
1-4-3
2-1-3
2-1-4
2-3-1
2-3-4
2-4-1
2-4-3
3-1-2
3-1-4
3-2-1
3-2-4
3-4-1
3-4-2
4-1-2
4-1-3
4-2-1
4-2-3
4-3-1
4-3-2
```

DFS 활용 순열 조합 3

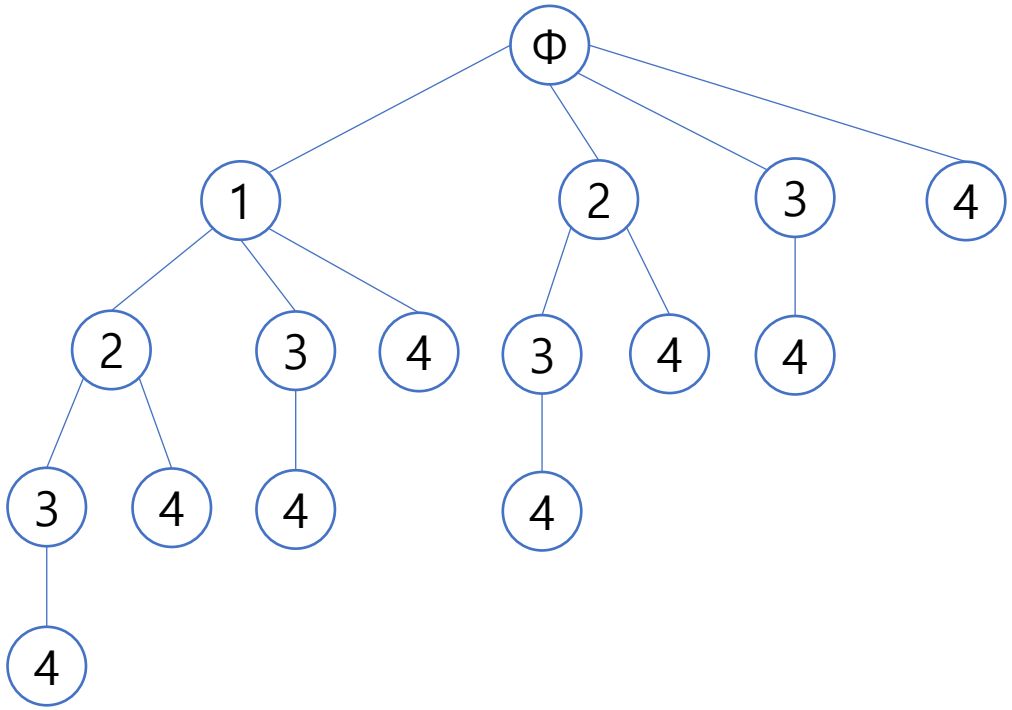
- 중복없고 순서없는 순열 조합
  - 중복 없다
    - 같은 숫자가 또 등장하지 않음

```
1-1-2  // !
1-2-2  // !
1-2    // OK
1-2-3  // OK
```

  - 순서 없다
    - 순서 다른 조합을 동일한 것으로 간주

```
// 아래는 모두 동일한 조합
1-2-3
1-3-2
3-2-1
```

- 그래프로 표현
  - 1,2,3,4 중에서 뽑을 때,
  - 자신 노드보다 큰 수로만 뻗어 나감

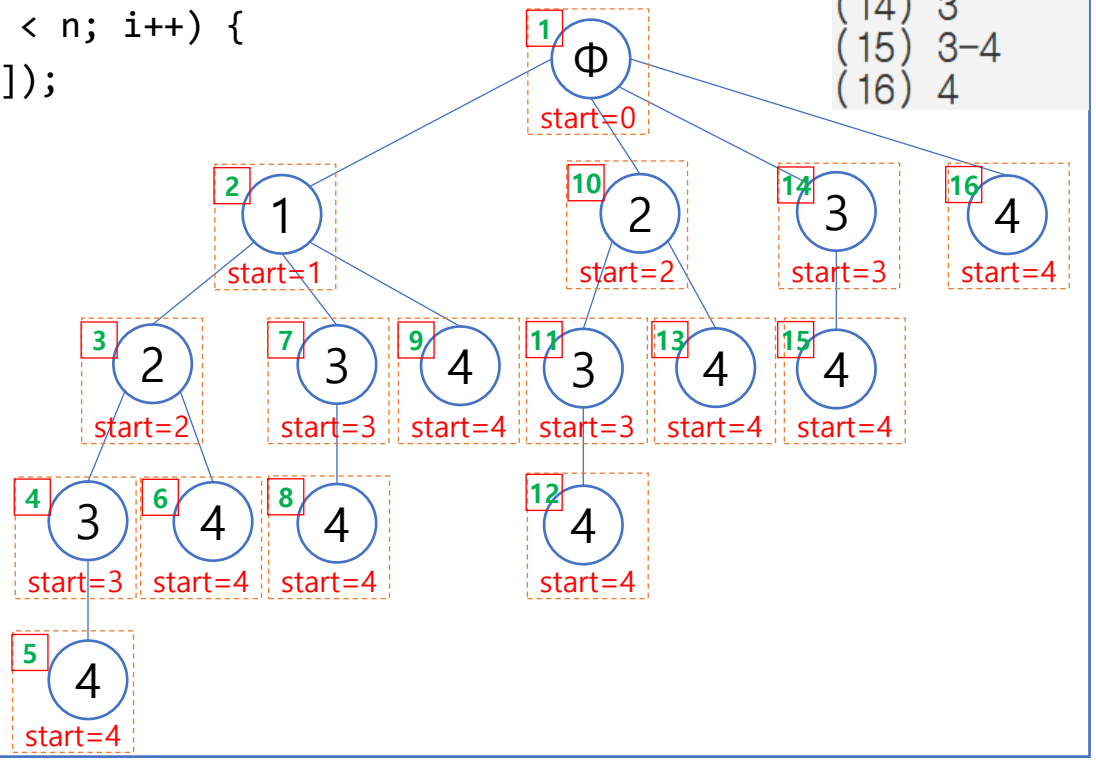


DFS 활용 순열 조합 3

```
4
( 1)
( 2) 1
( 3) 1-2
( 4) 1-2-3
( 5) 1-2-3-4
( 6) 1-2-4
( 7) 1-3
( 8) 1-3-4
( 9) 1-4
(10) 2
(11) 2-3
(12) 2-3-4
(13) 2-4
(14) 3
(15) 3-4
(16) 4
```

```
//중복없고 순서없는 순열 조합 만들기
#include <stdio.h>
#include <vector>
#define  MAX_N   10
using namespace std;

int n;
int d[] = {1,2,3,4,5,6,7,8,9,10};
int nth=0;  // 몇 번째 호출인지 기록
vector<int> seq;

void output_seq(vector<int>& vec) {
    printf("(%2d) ", nth);
    for(int a : vec)
        printf("%d-", a);
    printf("\b \n");
}
```

```
void dfs(int start) {
    nth++;  // 호출 횟수 증가
    output_seq(seq);
    // 여기에서 종료 조건 확인 및 최종 계산

    //start부터 그 이후 원소들로만 뻗어나감을 구현
    for (int i = start; i < n; i++) {
        seq.push_back(d[i]);
        dfs(i + 1);
        seq.pop_back();
    }
}

int main() {
// n개의 데이터에서 뽑기
    scanf("%d", &n);
    dfs(0);
}
```



DFS 활용 순열 조합 3

- 호출순서
  - dfs(0) 호출

  - start = 0

  for(i=0; i<n; i++)
  - 노드 d[0] (1) 을 푸시
  - dfs(1) 호출

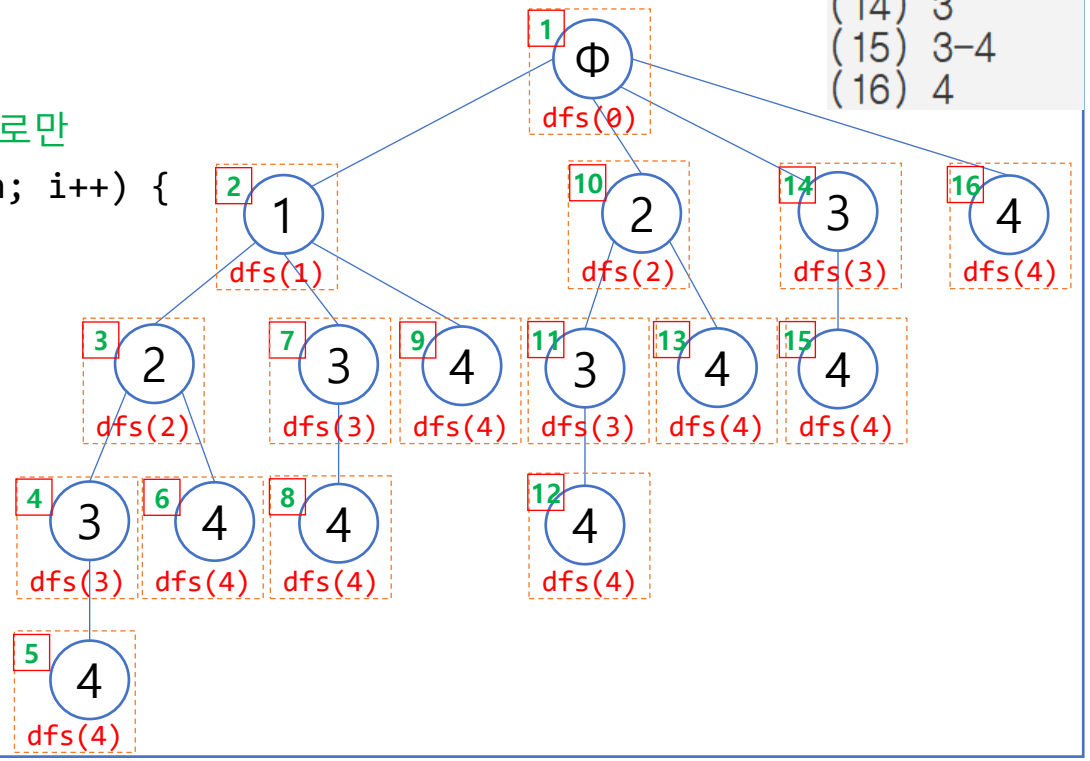
  - start = 1

  for(i=1; i<n; i++)
  - 노드 d[1] (2) 푸시
  - dfs(2) 호출

    :

```
int n;
int d[] = {1,2,3,4,5,6,7,8,9,10};
int nth=0;  // 몇 번째 호출인지 기록

void dfs(int start) {
    nth++;  // 호출 횟수 증가
    output_seq(seq);

    //start부터 그 이후 원소들로만
    for (int i = start; i < n; i++) {
        seq.push_back(d[i]);
        dfs(i + 1);
        seq.pop_back();
    }
}

int main() {
scanf("%d", &n);
    dfs(0);
}
```

```
4
( 1)
( 2) 1
( 3) 1-2
( 4) 1-2-3
( 5) 1-2-3-4
( 6) 1-2-4
( 7) 1-3
( 8) 1-3-4
( 9) 1-4
(10) 2
(11) 2-3
(12) 2-3-4
(13) 2-4
(14) 3
(15) 3-4
(16) 4
```



리모콘

- 문제

  컴퓨터실에서 수업중인 정보 선생님은 냉난방기의 온도를 조절하려고 한다. 냉난방기가 멀리 있어서 리모컨으로 조작하려고 하는데, 리모컨의 온도 조절 버튼은 다음과 같다.

  1) 온도를 1도 올리는 버튼
  2) 온도를 1도 내리는 버튼
  3) 온도를 5도 올리는 버튼
  4) 온도를 5도 내리는 버튼
  5) 온도를 10도 올리는 버튼
  6) 온도를 10도 내리는 버튼

  이와 같이 총 6개의 버튼으로 목표 온도에 도달해야 한다. 현재 설정 온도와 변경하고자 하는 목표 온도가 주어지면 이 버튼들을 이용하여 목표 온도로 변경하고자 한다.

  이때 버튼 누름의 최소 횟수를 구하시오.

  예를들어 7도에서 34도로 변경하는 경우,

  7 → 17 → 27 → 32 → 33 → 34

  이렇게 총 5번 누르면 된다.

- 입력

  현재온도 a와 목표온도 b가 입력된다.

  $(0 \le a, b \le 40)$

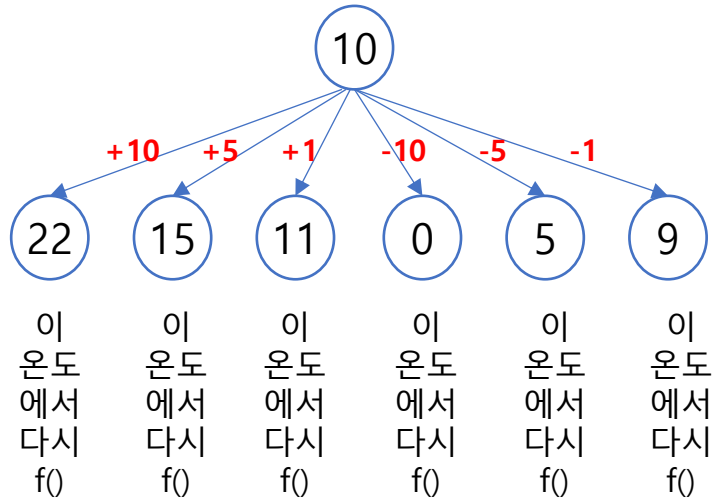
- 출력

  최소한의 버튼 사용으로 목표 온도가 되는 버튼 누름의 횟수를 출력한다.

입력 예	출력 예
7 34	5

리모콘 (초기설계)

- 초기 설계
  - 버튼을 누를 때 마다 변화하는 모든 온도에 대하여 전체 탐색 수행
  - 탐색함수 f() 는 6갈래로 뻗어 나감



  - 재귀호출 해법 가능
  - 계산량: $O(6^{res})$

```
int adj[] = { 10, 5, 1, -10, -5, -1 };

void f(int cnt, int temp) {
    // 지금까지 알아낸 리모컨 최다 조작 횟수를 최대 탐색 깊이로 설정
    if(cnt > res) return;

    // 온도 범위를 넘어서는 경우 탐색 중단
    if(temp < TEMP_LOW)  return;
    if(temp > TEMP_HIGH) return;

    if(temp == target) {
        if(cnt < res) { // 지금까지 찾아낸 횟수보다 작은 방식이면
            res = cnt;  // 결과에 현재 횟수를 기록
            output();
        }
        // 목표 채널에 도달한 뒤에는 더이상 탐색을 진행할 필요 없음
        // 왜냐하면 무조건 버튼 조작횟수가 늘어날 것이므로
        return;
    }

    for(int i=0; i<6; i++) {
        v.push_back(adj[i]);
        dfs(cnt +1, temp +adj[i]);
        v.pop_back();
    }
}
```

리모콘 (초기설계)

```
#include <stdio.h>
#include <math.h>
#include <vector>
#define  TEMP_LOW    0
#define  TEMP_HIGH  40
using namespace std;

int start, target, res;
int count=0;
int adj[] = { 10, 5, 1, -10, -5, -1 };
vector<int> v;
void f(int cnt, int temp);

void output() {
    for(int x : v)  // 버튼 누른 순서 모두 출력
        printf("%3d,", x);
    printf("\b (%d)\n", res);  // 누른 횟수 출력
}

int main(void)  {
    scanf("%d %d", &start, &target);
    //최초 답: +1 또는 -1 버튼만으로 도달하는 방법
    res = abs(target-start);
    f(0, start);
    printf("f() call count : %d\n", count);
    printf("%d\n", res);
    return 0;
}
```

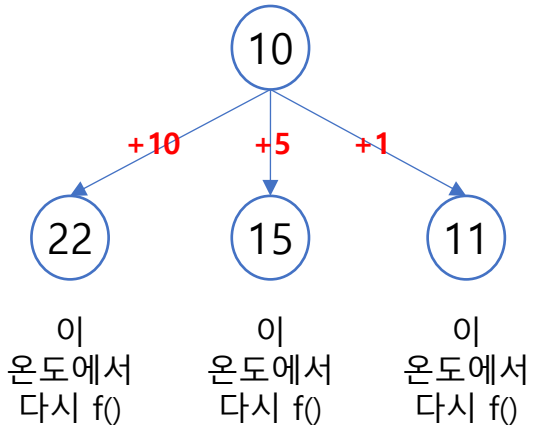
```
void f(int cnt, int temp) {
    count++;
    // 지금까지 알아낸 리모컨 최다 조작 횟수를 최대 탐색 깊이로 설정
    if(cnt > res) return;

    // 온도 범위를 넘어서는 경우 탐색 중단
    if(temp < TEMP_LOW || temp > TEMP_HIGH) return;

    if(temp == target) {
        if(cnt < res) { // 지금까지 찾아낸 횟수보다 작은 방식이면
            res = cnt;  // 결과에 현재 횟수를 기록
            output();
        }
        // 목표 채널에 도달한 뒤에는 더이상 탐색을 진행할 필요 없음
        // 왜냐하면 무조건 버튼 조작횟수가 늘어날 것이므로
        return;
    }

    for(int i=0; i<6; i++) {
        v.push_back(adj[i]);
        f(cnt +1, temp +adj[i]);
        v.pop_back();
    }
}
```

리모콘 (초기설계)

```
7 34
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5, -1 (26)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5, -1 (25)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5, -1 (24)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5, -1 (23)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5, -1 (22)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5, -1 (21)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5, -1 (20)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5, -1 (19)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5, -1 (18)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5, -1 (17)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5, -1 (16)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5, -1 (15)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5, -1 (14)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5, -1 (13)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5, -1 (12)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5, -1 (11)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5, -1 (10)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5, -1 (9)
 10, 10, 10,  1,  1,  1, -5, -1 (8)
 10, 10, 10,  1,  1,-10,  5 (7)
 10, 10, 10,  1,  1, -5 (6)
 10, 10,  5,  1,  1 (5)
f() call count : 35311
5

Process returned 0 (0x0)   execution time : 3.484 s
Press any key to continue.
```

리모콘 (개선설계)

- 개선 설계
  - 현재 온도보다 목표 온도가 큰 경우
    - 온도 올리는 버튼 3개 사용
  - 현재 온도보다 목표 온도가 작은 경우
    - 온도 내리는 버튼 3개 사용
  - 탐색 함수 f()는 3갈래로 뻗어 나감



  - 계산량: $O(3^{res})$로 감소

```
void f(int cnt, int temp) {
    count++;
    // 지금까지 알아낸 리모컨 최다 조작 횟수를 최대 탐색 깊이로 설정
    if(cnt > res) return;

    // 온도 범위를 넘어서는 경우 탐색 중단
    if(temp < TEMP_LOW || temp > TEMP_HIGH) return;

    if(temp == target) {
        if(cnt < res) { // 지금까지 찾아낸 횟수보다 작은 방식이면
            res = cnt;  // 결과에 현재 횟수를 기록
            output();
        }
        return;
    }

    for(int i=0; i<6; i++) {
        // 목표 온도가 지금 온도보다 더 높은데, 조정 온도가 0이하이면,
        if(temp < target && adj[i] <= 0)
            continue;  // 건너뜀
        // 목표 온도가 지금 온도보다 더 낮은데, 조정 온도가 0이상이면,
        if(temp > target && adj[i] >= 0)
            continue;  // 건너뜀
        v.push_back(adj[i]);
        f(cnt +1, temp +adj[i]);
        v.pop_back();
    }
}
```

리모콘 (개선설계)

```
#include <stdio.h>
#include <math.h>
#include <vector>
#define  TEMP_LOW    0
#define  TEMP_HIGH  40
using namespace std;

int start, target, res;
int count=0;
int adj[] = { 10, 5, 1, -10, -5, -1 };
vector<int> v;
void f(int cnt, int temp);

void output() {
    for(int x : v)  // 버튼 누른 순서 모두 출력
        printf("%3d,", x);
    printf("\b (%d)\n", res);  // 누른 횟수 출력
}

int main(void)  {
    scanf("%d %d", &start, &target);
    //최초 답: +1 또는 -1 버튼만으로 도달하는 방법
    res = abs(target-start);
    f(0, start);
    printf("f() call count : %d\n", count);
    printf("%d\n", res);
    return 0;
}
```

```
void f(int cnt, int temp) {
    count++;
    // 지금까지 알아낸 리모컨 최다 조작 횟수를 최대 탐색 깊이로 설정
    if(cnt > res) return;

    // 온도 범위를 넘어서는 경우 탐색 중단
    if(temp < TEMP_LOW || temp > TEMP_HIGH) return;

    if(temp == target) {
        if(cnt < res) { // 지금까지 찾아낸 횟수보다 작은 방식이면
            res = cnt;  // 결과에 현재 횟수를 기록
            output();
        }
        // 목표 채널에 도달한 뒤에는 더이상 탐색을 진행할 필요 없음
        // 왜냐하면 무조건 버튼 조작횟수가 늘어날 것이므로
        return;
    }

    for(int i=0; i<6; i++) {
        if(temp < target && adj[i] <= 0)
            continue;
        if(temp > target && adj[i] >= 0)
            continue;
        v.push_back(adj[i]);
        f(cnt+1, temp+adj[i]);
        v.pop_back();
    }
}
```

리모콘 (개선설계)

```
7 34
 10, 10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5,  1,  1 (26)
 10, 10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5,  1,  1 (25)
 10, 10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5,  1,  1 (24)
 10, 10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5,  1,  1 (23)
 10, 10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5,  1,  1 (22)
 10, 10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5,  1,  1 (21)
 10, 10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5,  1,  1 (20)
 10, 10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5,  1,  1 (19)
 10, 10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5,  1,  1 (18)
 10, 10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5,  1,  1 (17)
 10, 10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5,  1,  1 (16)
 10, 10, 10,-10, 10,-10, 10,-10, 10,-10, 10,-10,  5,  1,  1 (15)
 10, 10, 10,-10, 10,-10, 10,-10, 10,-10, 10, -5,  1,  1 (14)
 10, 10, 10,-10, 10,-10, 10,-10, 10,-10,  5,  1,  1 (13)
 10, 10, 10,-10, 10,-10, 10,-10, 10, -5,  1,  1 (12)
 10, 10, 10,-10, 10,-10, 10,-10,  5,  1,  1 (11)
 10, 10, 10,-10, 10,-10, 10, -5,  1,  1 (10)
 10, 10, 10,-10, 10,-10,  5,  1,  1 (9)
 10, 10, 10,-10, 10, -5,  1,  1 (8)
 10, 10, 10,-10,  5,  1,  1 (7)
 10, 10, 10, -5,  1,  1 (6)
 10, 10,  5,  1,  1 (5)
f() call count : 2863
5

Process returned 0 (0x0)   execution time : 7.851 s
Press any key to continue.
```

리모콘

- 개선 설계2
  - 느린 이유
    - 버튼 누르는 모든 가능성을 다 뒤져야 답을 얻게 됨
  - 해결책
    - 버튼 1번에 해결 가능한가? 아니라면,
    - 2번 눌러서 가능한가? 또 아니라면,
    - 3번 눌러서 가능한가? 또 아니라면,

      :
  - 너비우선 탐색(BFS)의 적용

- 너비우선탐색 알고리즘

  1) 시작 정점 k를 처리하고 큐에 삽입, k를 방문한 것으로 표시

  2) 큐가 빌 때까지 다음을 반복 :

  ① 큐에서 첫 번째 항목 삭제

  ② 삭제된 항목과 이웃하는 방문되지 않은 모든 정점을 처리하고 큐에 삽입

  ③ 삽입된 모든 정점에 방문을 표시

- 그래프가 아니므로(즉, 회로가 없으므로) 방문표시는 하지 않아도 됨
  - 이미 방문 된 노드일 수가 없음

리모콘 BFS 풀이

```
#include <stdio.h>
#include <queue>
using namespace std;

typedef struct {
    int temp;
    int change;
    int cnt;
}node;

queue <node> Q;
int adj[] = { 10, 5, 1, -10, -5, -1 };
```

- 너비우선탐색 알고리즘
  1) 시작 정점 k를 처리하고 큐에 삽입
  2) 큐가 빌 때까지 다음을 반복 :
     ① 큐에서 첫 번째 항목 삭제
     ② 삭제된 항목과 이웃하는 방문 되지 않은 모든 정점을 처리하고 큐에 삽입
     ③ 삽입된 모든 정점에 방문을 표시

```
int main(void)  {
    int start, target;
    scanf("%d %d", &start, &target);
    node n;
    Q.push({start, 0, 0});

    printf("(cnt)chang,tepm\n");
    while(! Q.empty()) {
        n = Q.front();  // 큐에서 첫번째 항목 획득
        Q.pop();        // 삭제
        printf("(%3d)  %3d, %3d\n", n.cnt, n.change, n.temp);

        if(n.temp == target)
            break;

        for(int i=0; i<6; i++) {
            // 목표 온도가 지금 온도보다 더 높은데, 조정 온도가 0이하이면,
            if(n.temp < target && adj[i] <= 0)
                continue;  // 건너뜀
            // 목표 온도가 지금 온도보다 더 낮은데, 조정 온도가 0이상이면,
            if(n.temp > target && adj[i] >= 0)
                continue;  // 건너뜀
            Q.push({n.temp+adj[i], adj[i], n.cnt+1});
        }
    }
    printf("%d\n", n.cnt);;
    return 0;
}
```

```
queue <node> Q;
int adj[] = { 10, 5, 1, -10, -5, -1 };

int main(void)  {                   // BFS를 이용한 풀이
    int start, target;
    scanf("%d %d", &start, &target);
    node n;
    Q.push({start, 0, 0});

    printf("(cnt)chang,tepm\n");
    while(! Q.empty()) {
        n = Q.front();
        Q.pop();
        printf("(%3d)  %3d, %3d\n", n.cnt, n.change, n.temp);

        if(n.temp == target)
            break;

        for(int i=0; i<6; i++) {
            // 목표 온도가 더 높은데, 조정 온도가 0이하이면,
            if(n.temp < target && adj[i] <= 0)
                continue;  // 건너뜀
            // 목표 온도가 더 낮은데, 조정 온도가 0이상이면,
            if(n.temp > target && adj[i] >= 0)
                continue;  // 건너뜀
            Q.push({n.temp+adj[i], adj[i], n.cnt+1});
        }
    }
    printf("%d\n", n.cnt);;
    return 0;
}
```

```
1 8
(cnt)chang,tepm
(  0)    0,   1
(  1)   10,  11
(  1)    5,   6
(  1)    1,   2
(  2)  -10,   1
(  2)   -5,   6
(  2)   -1,  10
(  2)   10,  16
(  2)    5,  11
(  2)    1,   7
(  2)   10,  12
(  2)    5,   7
(  2)    1,   3
(  3)   10,  11
(  3)    5,   6
(  3)    1,   2
(  3)   10,  16
(  3)    5,  11
(  3)    1,   7
(  3)  -10,   0
(  3)   -5,   5
(  3)   -1,   9
(  3)  -10,   6
(  3)   -5,  11
(  3)   -1,  15
(  3)  -10,   1
(  3)   -5,   6
(  3)   -1,  10
(  3)   10,  17
(  3)    5,  12
(  3)    1,   8
3
```

실행결과

- 1도 에서 시작
- 8도 도달이 목표
- 버튼을 한번 누를 때 마다 세 갈래로 갈라짐
- 1회차

  +10, +5 +1

- 2회차

  +10 (-10, -5, -1)

  +5 (+10, +5, +1)

  +1 (+10, +5, +1)

- 3회차

리모콘2 (채널 빨리 바꾸기)

- 문제

  스마트 TV 한 대를 구매하였다. 당연히 채널을 조정할 수 있는 리모콘도 함께 들어있었다. 그런데 리모콘의 버튼이 아래와 같이 총 6개만 존재하였다.

  [채널-6], [채널-4], [채널–1], [채널+3], [채널+5], [채널+9]

  리모컨의 채널 조정은 존재하는 채널로만 이동 가능한데 TV채널은 1번부터 40번까지만 존재한다. 그래서 만약 1번 채널에서 [채널-6] 버튼을 누르면 무시될 것이다. 또한 38번 채널에서 [채널+9] 버튼을 누른다면 무시될 것이다.

  위와 같은 조건에서 33번 채널에서 40번 채널로 이동하는 방법으로는,

  ① [채널+5], [채널-1], [채널+3]

  ② [채널+3], [채널-1], [채널+5]

  ③ [채널-1], [채널+5], [채널+3]

  ④ [채널-1], [채널+3], [채널+5]

  ⑤ [채널-1], [채널-1], [채널+9]

  위 방법들 중 한 가지 방법으로 목표 채널로 이동할 수 있다.

  (버튼 조작 중 채널 범위를 넘어서는 [채널+5], [채널+3], [채널-1] 등의 방법은 안됨)

  이왕이면 빠르게 채널을 바꾸는 것이 좋을 것이다.

  현재 채널과 목표 채널이 주어졌을 때, 최소 버튼 조작으로 목표 채널로 이동한다면 몇 번 만에 가능한지, 그리고 최소 버튼 조작의 방법이 몇 가지 존재하는지 알아내는 프로그램을 제작하시오.

- 출처: 2023 충북정올 학교예선 초등 5번 문제

리모콘2 (채널 빨리 바꾸기)

위 예시에서는 33번 채널에서 40번 채널로 최소 3번의 버튼 조작으로 이동이 가능하며, 3번의 버튼 조작으로 이동하는 방법은 총 5가지 존재한다.

- 입력

  첫 번재 줄에 공백으로 구분하여 현재 채널(C)과 목표 채널(T)이 입력된다.

  (1 ≤ C ≤ 40, 1 ≤ T ≤ 40)

- 출력

  첫 번째 줄에 목표 채널로 이동하기 위해 필요한 최소 버튼 조작 횟수를 자연수로 출력한다.

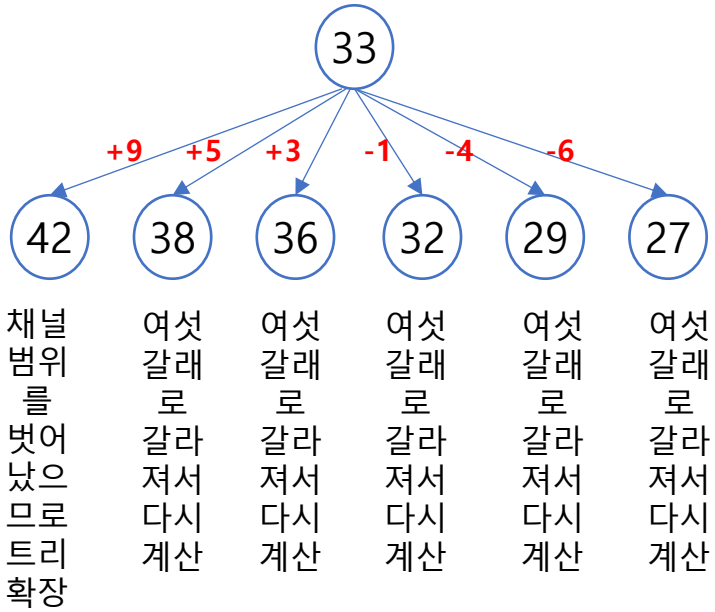
  두 번째 줄에 최소 버튼 조작 횟수로 이동하는 방법의 가짓수를 자연수로 출력한다.

- 입출력의 예

입력 예	출력 예
31 39	2 4
33 40	3 5

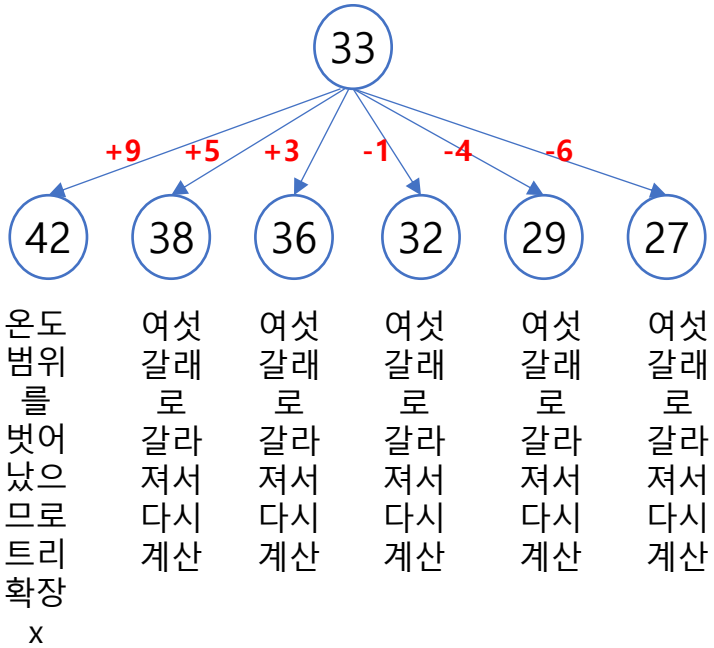
리모콘2

- 풀이 아이디어1



- 깊이 우선으로 탐색
- DFS로 구현
- 계산량: $O(6^d)$, d=최대탐색 깊이

- 풀이 아이디어2



- 너비 우선으로 탐색
- BFS로 구현
- 계산량: $O(6^{res})$, res=답을 찾았을 때 깊이

DFS를 이용한 방법

```
// MAX가 10으로 설정되어 있으므로
// 최대 리모콘 조작횟수 10회까지 탐색한다.
#include <stdio.h>
#include <vector>

#define  CH_LOW    1
#define  CH_HIGH   40
#define  MAX       10     // 최대 탐색 깊이
using namespace std;

int a, b;
int res = MAX;
int methods = 0;  // 몇 가지 방법이 있는가?
int adj[] = { 9, 5, 3, -6, -4, -1 };

int main(void)  {
    scanf("%d %d", &a, &b);
    dfs(0, a);
    printf("%d\n%d\n", res, methods);
    return 0;
}
```

```
void dfs(int cnt, int ch) {
    //리모컨 최다 조작 횟수를 최대 탐색 깊이로 설정
    if(cnt > MAX)    return;
    // 채널 범위를 넘어가는 경우 서브노드 탐색 중단
    if(ch < CH_LOW || ch > CH_HIGH)  return;

    if(ch == b) {
        if(cnt < res) { // 지금까지 찾아낸 횟수보다 작은 방식이면
            res = cnt;  // 결과에 현재 횟수를 기록
            methods = 1;  // 새로운 최소조작법을 찾았으므로
        }
        else if(cnt == res)
            methods ++;  // 최소 조작법과 동일 횟수를 찾았으므로
        // 목표 채널에 도달한 뒤에는 더이상 탐색을 진행할 필요 없음
        // 왜냐하면 무조건 버튼 조작횟수가 늘어날 것이므로…
        return;
    }

    for(int i=0; i<6; i++) {
        dfs(cnt+1, ch+adj[i]);
    }
}
```

```
// 리모컨의 누른 버튼을 추적하는 변형
#include <stdio.h>
#include <vector>

#define  CH_LOW    1
#define  CH_HIGH  40
#define  MAX      10     // 최대 탐색 깊이
using namespace std;

int a, b;
int res = MAX;
int methods = 0;
int adj[] = { 9, 5, 3, -6, -4, -1 };
vector<int> v;
void dfs(int cnt, int ch);

void output() {
    for(int x : v)  // 버튼 누른 순서 모두 출력
        printf("%3d,", x);
    printf("\b (%d)\n", res);  // 누른 횟수 출력
}

int main(void)  {
    scanf("%d %d", &a, &b);
    dfs(0, a);
    printf("%d\n%d\n", res, methods);
    return 0;
}
```

```
void dfs(int cnt, int ch) {
    //리모컨 최다 조작 횟수를 최대 탐색 깊이로 설정
    if(cnt > MAX)    return;
    // 채널 범위를 넘어가는 경우 서브노드 탐색 중단
    if(ch < CH_LOW || ch > CH_HIGH)  return;

    if(ch == b) {
        if(cnt < res) { // 지금까지 찾아낸 횟수보다 작은 방식이면
            res = cnt;  // 결과에 현재 횟수를 기록
            methods = 1;  // 새로운 최소조작법을 찾았으므로
            output();
        }
        else if(cnt == res) {
            methods++;  // 최소 조작법과 동일 횟수를 찾았으므로
            output();
        }

        // 목표 채널에 도달한 뒤에는 더이상 탐색을 진행할 필요 없음
        // 왜냐하면 무조건 버튼 조작횟수가 늘어날 것이므로
        return;
    }

    for(int i=0; i<6; i++) {
        v.push_back(adj[i]);
        dfs(cnt+1, ch+adj[i]);
        v.pop_back();
    }
}
```

BFS를 이용한 방법

```
#include <stdio.h>
#include <vector>
#include <queue>
using namespace std;

#define  CH_LOW    1
#define  CH_HIGH  40

typedef struct node {
    int ch;     // channel
    int cnt;    // count
};

int main(void)  {             // BFS를 이용한 풀이
    int adj[] = { 9, 5, 3, -6, -4, -1 };
    int min_cnt = -1;

    int start, target;
    scanf("%d %d", &start, &target);

    queue<node> Q;
    Q.push({start, 0});
```

```
    while(! Q.empty()) {
        node n = Q.front();
        Q.pop();

        // 채널 범위를 벗어나면 해당 서브노드 탐색 중단
        if(n.ch < CH_LOW || n.ch > CH_HIGH)
            continue;

        if(n.ch == target) {  // 목표 채널에 도달하면
            min_cnt = n.cnt;
            break;
        }

        for(int i=0; i<6; i++) {
            Q.push({n.ch+adj[i], n.cnt+1});
        }
    }

    printf("%d\n", min_cnt);
    return 0;
}
```

```
// 누른 버튼을 추적하는 변형
#include <stdio.h>
#include <vector>
#include <queue>

#define  CH_LOW    1
#define  CH_HIGH   40
#define  INT_MAX   0x7fffffff

using namespace std;

typedef struct {
    int ch;     // channel
    int cnt;    // count
    vector<int> btns;   // button history
} node;

int main(void)  {               // BFS를 이용한 풀이
    int adj[] = { 9, 5, 3, -6, -4, -1 };
    int min_cnt = INT_MAX;
    int methods = 0;

    int start, target;
    scanf("%d %d", &start, &target);

    queue<node> Q;
    Q.push({start, 0, vector<int>()});

    while(! Q.empty()) {
        node n = Q.front();
        Q.pop();

        // 채널 범위를 벗어나면 해당 서브노드 탐색 중단
        if(n.ch < CH_LOW || n.ch > CH_HIGH)
            continue;

        // 지금 계산 중인 조작횟수가 최소 조작횟수를 넘기면 스톱
        if(n.cnt > min_cnt)
            break;

        if(n.ch == target) {  // 목표 채널에 도달하면
            min_cnt = n.cnt;
            methods++;

            /* printf("[%d]: ", min_cnt);

            for(int x: n.btns)
                printf("%3d, ", x);
            printf("\b \n"); */
        }

        for(int i=0; i<6; i++) {
            vector<int> btns(begin(n.btns), end(n.btns));
            btns.push_back(adj[i]);
            Q.push({n.ch+adj[i], n.cnt+1, btns});
        }
    }

    printf("%d\n", min_cnt);
    printf("%d\n", methods);
    return 0;
}
```

테이블의 최소 합

- 문제

  n*n개의 수가 주어진다. (1<=n<=10)

  이때 겹치지 않는 각 열과 각 행에서 수를 하나씩 뽑는다.

  (즉, 총 n개의 수를 뽑을 것이다, 그리고 각 수는 100 이하의 값이다.)

  이 n개의 수의 합을 구할 때 최솟값을 구하시오.

- 입력

  첫 줄에 n이 입력된다. 다음 줄부터 n+1 줄까지 n개씩의 정수가 입력된다.

- 출력

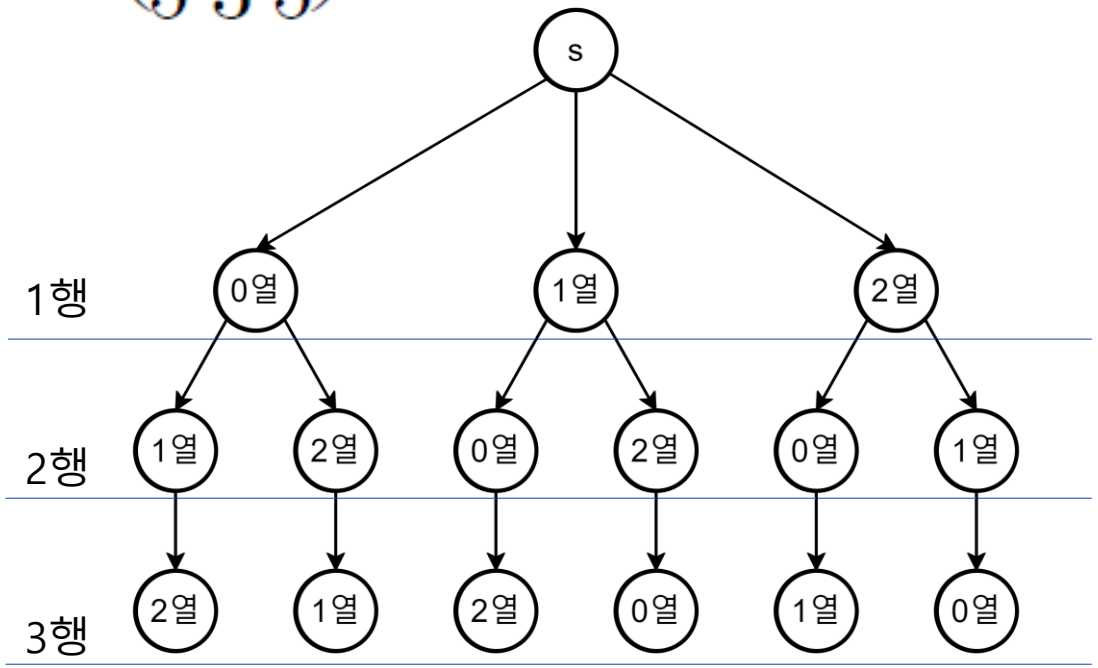
  구한 최소 합을 출력한다.

입력 예	출력 예
3 1 5 3 2 4 7 5 3 5	8

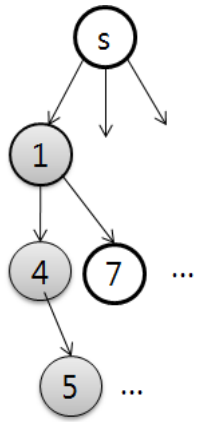
테이블의 최소 합

- 전체탐색

$$\begin{pmatrix} 1 & 5 & 3 \\ 2 & 4 & 7 \\ 5 & 3 & 5 \end{pmatrix}$$

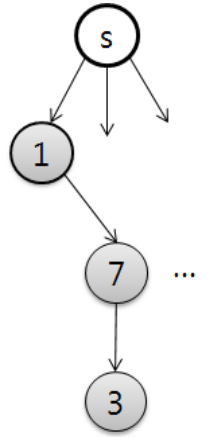


- 처음으로 구한 해 (10)



$$\begin{pmatrix} \textcircled{1} & 5 & 3 \\ 2 & \textcircled{4} & 7 \\ 5 & 3 & \textcircled{5} \end{pmatrix}$$

- 두번째로 구한 해 (11)



$$\begin{pmatrix} \textcircled{1} & 5 & 3 \\ 2 & 4 & \textcircled{7} \\ 5 & \textcircled{3} & 5 \end{pmatrix}$$

테이블의 최소 합 DFS 소스코드

```
#include <stdio.h>
#include <vector>
#define MAX_INT 0x7fffffff
#define MAX_N   10
using namespace std;

int n;
int col_used[MAX_N];
int m[MAX_N][MAX_N];
int min_sol=MAX_INT;
vector<int> seq;

void input(void) {
    scanf("%d", &n);
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            scanf("%d", &m[i][j]);
}

void output_seq(int sum) {
    printf("seq: ");
    for(int a : seq)
        printf("%d-", a);
    printf("\b [%d]\n", sum);
}
```

```
// row행까지 더한 결과 sum인 상태
void dfs(int row, int sum) {
    if(row==n) {              // 마지막 행에 도달하면…
        output_seq(sum);      // 선택된 열 순서 출력
        if(sum < min_sol)     // 더 작은 해를 찾으면…
            min_sol = sum;

        return;       //row가 n행에 도달했으니 탐색중지
    }
    // row의 행의 모든 컬럼에 대하여 DFS,
    for(int c=0; c<n; c++) {
        if(!col_used[c]) {       // 사용된 적이 없으면,
            col_used[c] = 1;     // c열 사용됨을 표시
            seq.push_back(c);
            dfs(row+1, sum+m[row][c]);  // 다음 행 탐색
            seq.pop_back();
            col_used[c] = 0;     // 백트랙시 사용됨 해제
        }
    }
}

int main() {
    input();
    dfs(0, 0);
    printf("%d\n", min_sol);
    return 0;
}
```

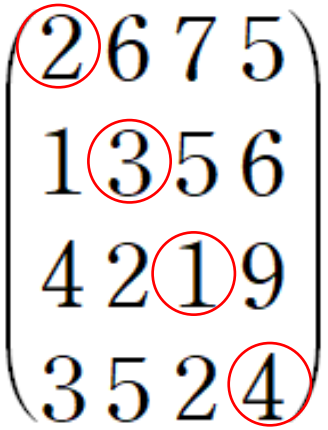
```
3
1 5 3
2 4 7
5 3 5
seq: 1-2-3 [10]
seq: 1-3-2 [11]
seq: 2-1-3 [12]
seq: 2-3-1 [17]
seq: 3-1-2 [8]
seq: 3-2-1 [12]
8
```

테이블의 최소 합

- 탐색배제: 찾은 답보다 좋을 것
  - 배제 조건

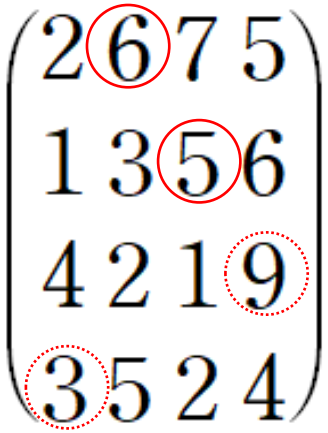
현재까지의 합 > 지금까지의 최소 합

첫 번째로 찾은 답



2+3+1+4=10

n 번째 답 계산 중



6+5 (stop)

```
void solve(int row, int sum) {
   //현재까지의 합 > 지금까지의 최소 합
   if(sum>min_sol)
      return;

    if(row==n) {
        if(score<min_sol)
            min_sol = score;
        output_seq();
        printf("[%d]\n", min_sol);
        return;
    }
    for(int c=0; c<n; c++) {
        if(col_check[c]==0) {
            col_check[c]=1;
            seq.push_back(c);
            solve(row+1, score+m[row][c]);
            seq.pop_back();
            col_check[c]=0;
        }
    }
    return;
}
```

케이블 재사용

- 문제

  초고속 인터넷 제공을 위해 각 가정집까지 광케이블을 포설하는 통신업체 KKT는 최근 급격한 원자잿값 상승으로 수익이 급감하고 있었다. 그래서 이를 타개하기 위해 기존에는 그냥 버렸던 자투리 광케이블을 모두 수거한 뒤 이를 이어 붙여서 재사용하는 방식으로 비용 절감을 하기로 하였다.

  자투리 광케이블이 N개 있었다면 각각의 길이는 Li이며 이 중 1개 이상을 선택하여 이를 단독으로 사용하거나 이어 붙여서 길이가 L인 광케이블을 만들어낼 수 있다. 이렇게 재사용한 광케이블의 길이 L이 공사에 필요한 길이 T이상이 되면 공사에 사용이 가능하다. 필요한 광케이블 길이와 재사용된 광케이블 길이 차이의 최솟값을 구하는 프로그램을 작성하시오

- 출처: 2024 시도상업경진대회 비즈니스프로그래밍 8번

케이블 재사용

- 입력

(1) 첫 번째 줄에는 자투리 광케이블의 개수 N이 입력된다.
(2 ≤ N ≤ 22)

(2) 두 번째 줄에는 자투리 광케이블의 길이 Li가 공백으로 분리되어 N개 입력된다.
(1 ≤ Li ≤ 1000)

(3) 세 번째 공사에 필요한 광케이블의 길이 T가 입력된다.
(1 ≤ T ≤ 20000)

- 출력

필요한 광케이블의 길이 T와 자투리를 이어 붙여 만든 광케이블의 길이 L과의 차이의 최솟값을 출력한다.

단, 이어 붙여 만든 광케이블의 길이가 필요한 광케이블의 길이 이상이어야 공사가 가능하다.

- 입력과 출력의 예

입력 예	출력 예
4 1 2 3 9 5	0

케이블 재사용

- 전체 코드

```
#include <stdio.h>
#include <limits.h>
#include <vector>
using namespace std;

int N, T;
int L[22];
int min_diff = INT_MAX;
vector<int> seq;
void dfs(int, int);

int main() {
    scanf("%d", &N);

    for (int i = 0; i < N; i++) {
        scanf("%d", &L[i]);
    }
    scanf("%d", &T);

    // DFS 탐색 시작
    dfs(0, 0);
    printf("%d\n", min_diff);

    return 0;
}
```

```
void output_combi(vector<int>& seq, int len) {
    for(int a : seq)
        printf("%d-", a);
    printf("\b [%d]\n", len);
}

void dfs(int start, int cur_len) {
    output_combi(seq, cur_len);
    // 현재 길이가 T 이상이면 최소 차이 갱신
    if (_______________) {
        if (cur_len - T < min_diff) {
            min_diff = __________;
        }
        printf("stop! length over\n");
        return;
    }

    // 모든 자투리 광케이블을 탐색
    for (int i = start; i < N; i++) {
        seq.push_back(L[i]);
        dfs(i + 1, ____________);
        seq.pop_back();
    }
}
```

```
4
1 2 3 9
5
 [0]
1 [1]
1-2 [3]
1-2-3 [6]
stop! length over
1-2-9 [12]
stop! length over
1-3 [4]
1-3-9 [13]
stop! length over
1-9 [10]
stop! length over
2 [2]
2-3 [5]
stop! length over
2-9 [11]
stop! length over
3 [3]
3-9 [12]
stop! length over
9 [9]
stop! length over
0
```

비선형구조의 자료 구현

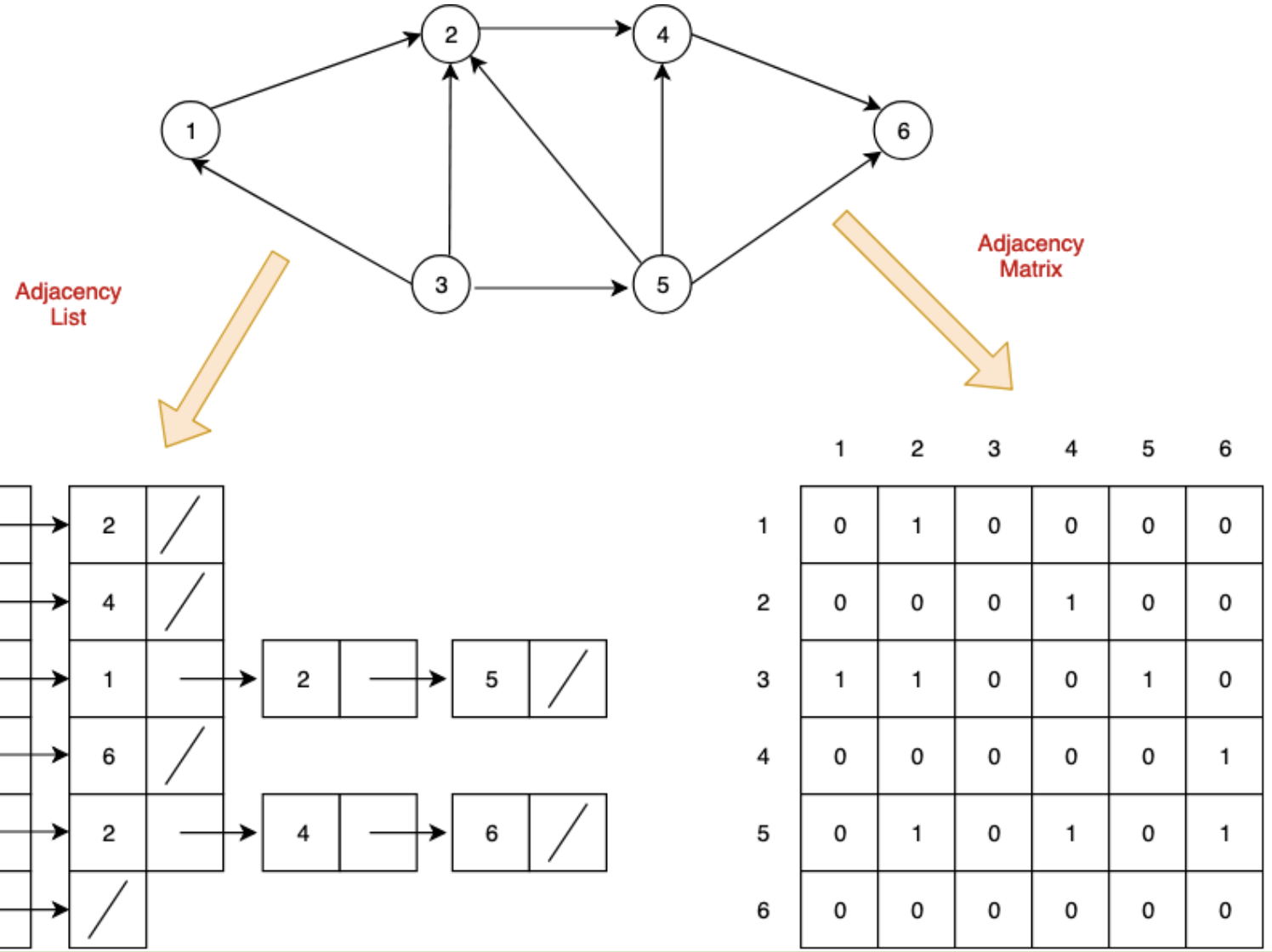
- 그래프의 자료 구현
  - 인접리스트

    (adjacency list)

  - 인접행렬

    (adjacency matrix)

  - 기타방법 …

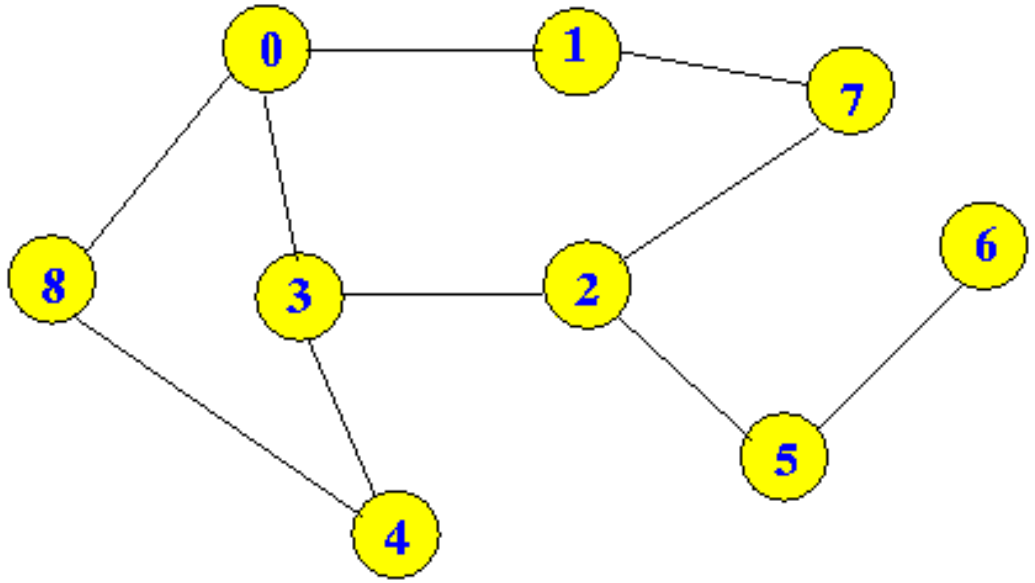


깊이우선탐색(DFS)

- 그래프의 순회

  트리와 달리 그래프는 사이클이 존재함

  방문정보를 유지하여 재방문을 막아야 함



- 깊이우선탐색 알고리즘

  : go deep(before going wide)

```
def dfs(k):
```

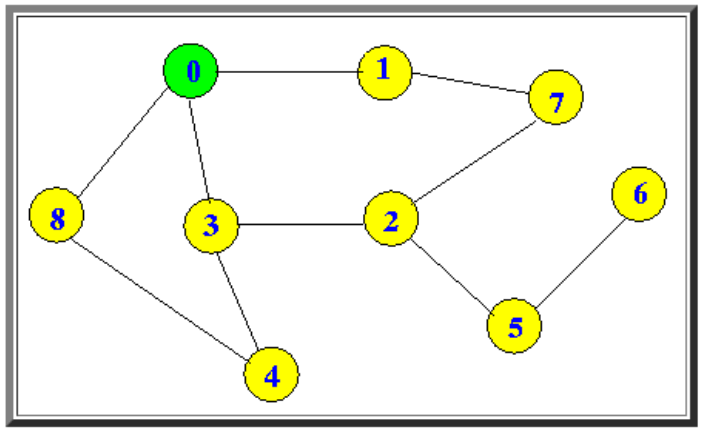
1) 정점 k를 처리하고 방문한 것으로 표시
2) 정점 k와 연결된 모든 정점에 대하여 방문한적이 없으면 그 정점에서 dfs, 완료되면 되돌아오기(백트랙)

image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

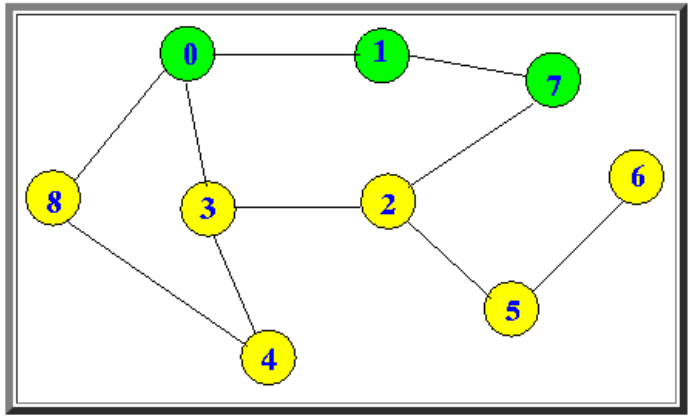
깊이우선탐색(DFS)

- 탐색순서

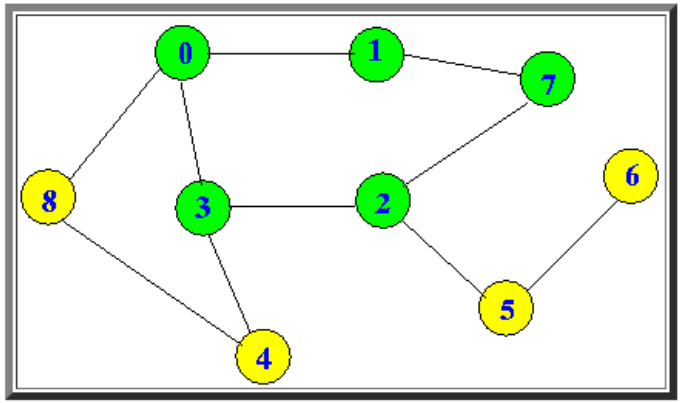
① dfs(0):



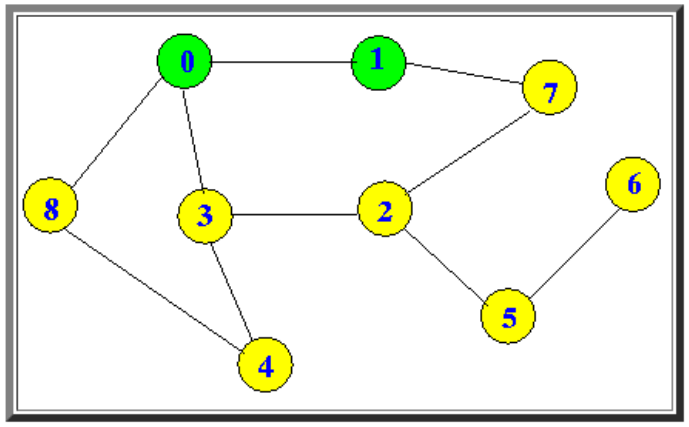
- ~~dfs(1) → dfs(0)~~ (because node 0 is "visited"); dfs(1) → dfs(7)



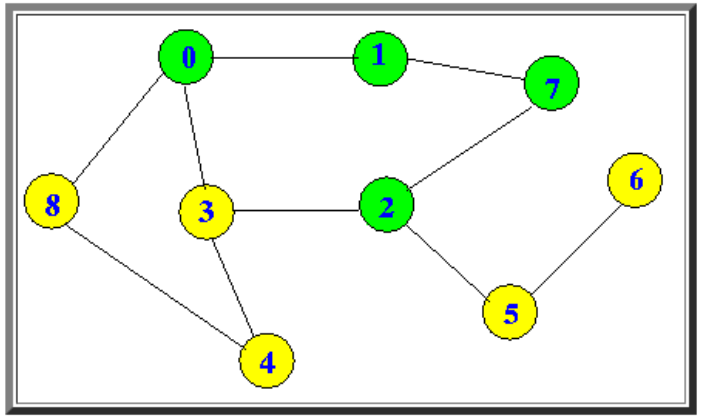
- dfs(2) → dfs(3)



② dfs(0) → dfs(1)



- ~~dfs(7) → dfs(1)~~; dfs(7) → dfs(2)



- ~~dfs(3) → dfs(0)~~; ~~dfs(3) → dfs(2)~~; dfs(3) → dfs(4)

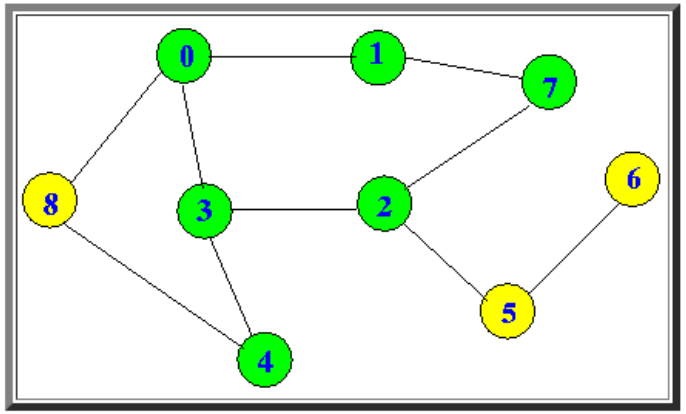
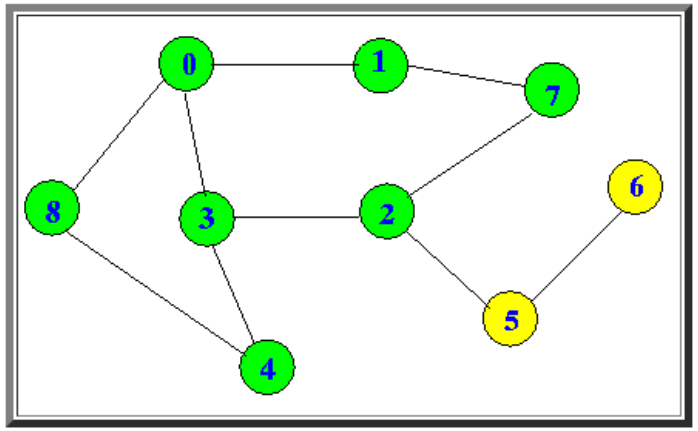


image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

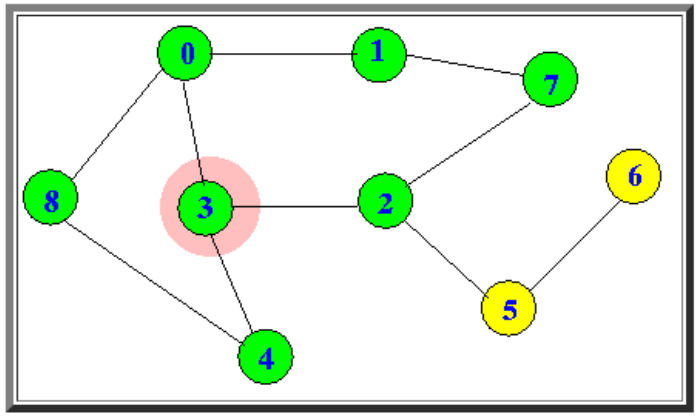
깊이우선탐색(DFS)

- 탐색순서

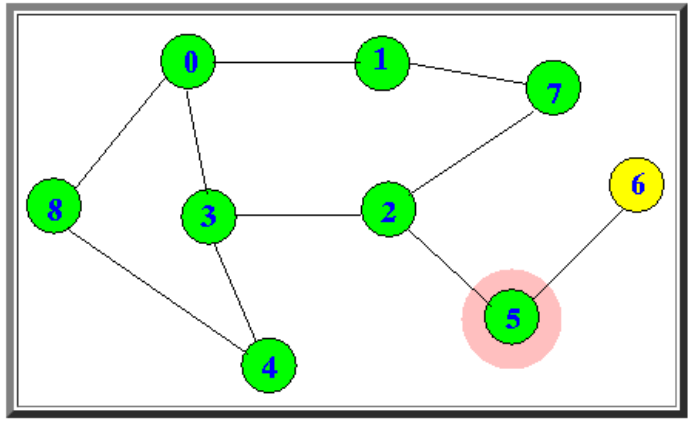
- ~~dfs(4) → dfs(3)~~; dfs(4) → dfs(8)



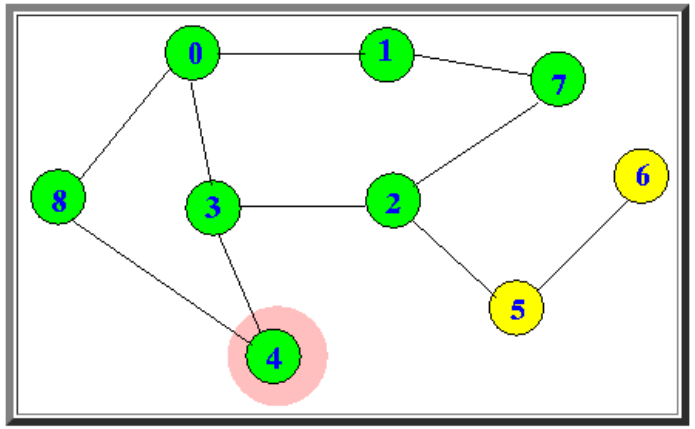
- return to dfs(3)



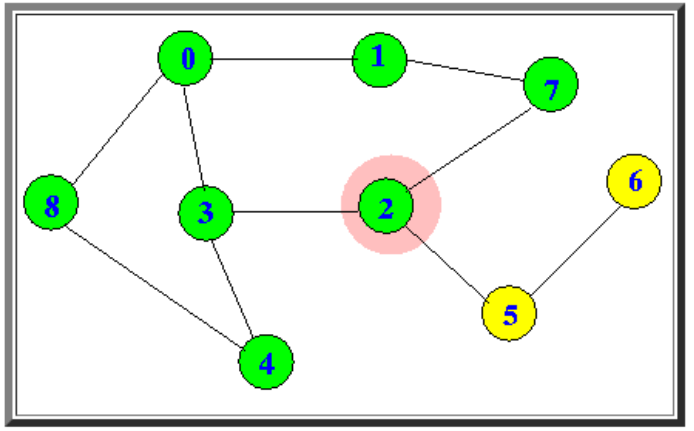
- ~~dfs(2) → dfs(3)~~; dfs(2) → dfs(5)



- ~~dfs(8) → dfs(0)~~; ~~dfs(8) → dfs(4)~~; return to dfs(4)



- return to dfs(2)



- ~~dfs(5) → dfs(2)~~; dfs(5) → dfs(6)

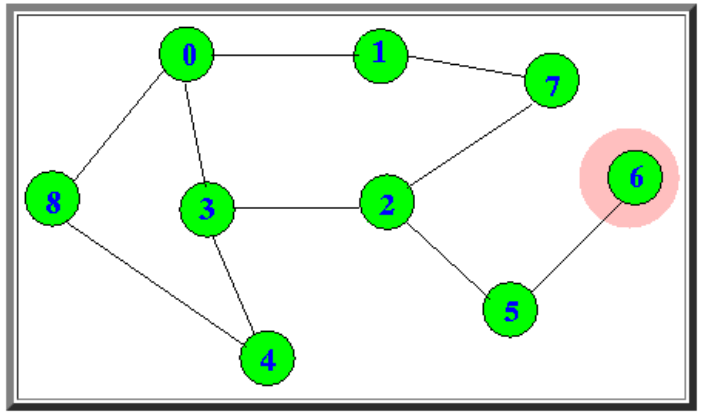
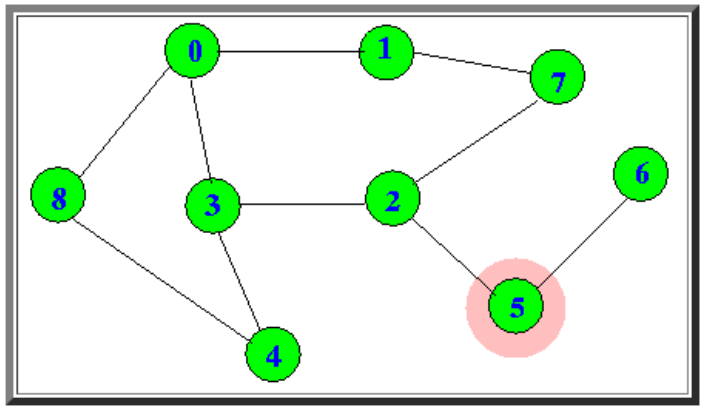


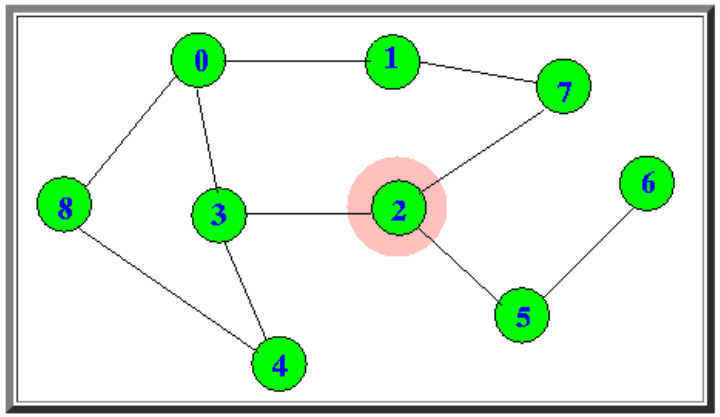
image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

깊이우선탐색(DFS)

- 탐색순서
  - ~~dfs(6) → dfs(5)~~; return to dfs(5)



  - return to dfs(2)



  - return to dfs(7)

  - return to dfs(1)

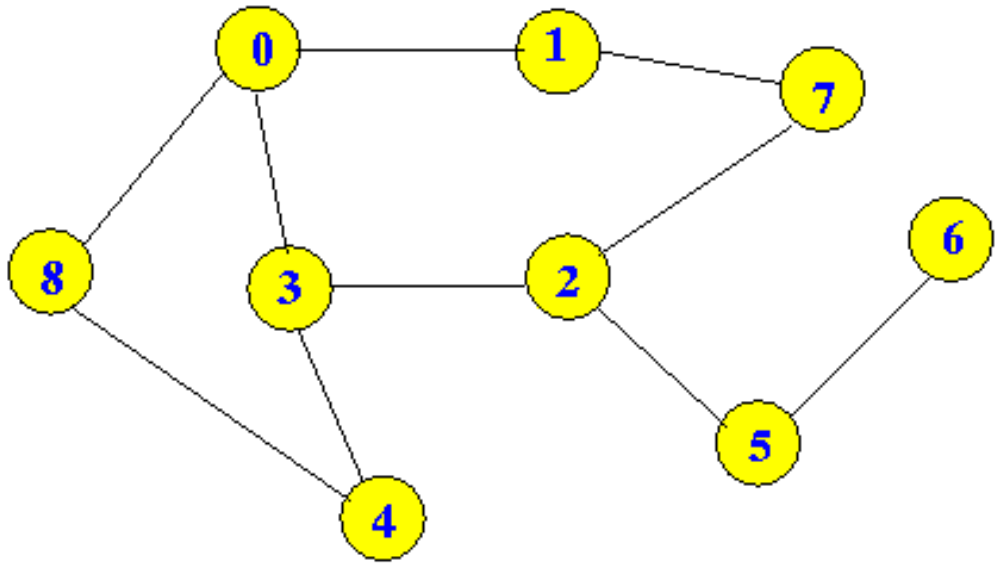
  - return to dfs(0)

    DONE

image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

깊이우선탐색(DFS)

- vector를 인접리스트로 활용

그래프	데이터
	8 10 0 1 0 3 0 8 1 7 2 3 2 5 3 4 2 7 4 8 5 6


```
int n, m;
vector<vector<int>> G;
vector<bool> visited;

void input_G() {
   scanf("%d %d", &n, &m);
   // 정점이 0부터 시작하므로 n+1
   G.resize(n+1); // 공간확보
   visited.assign(n+1, 0);

   for(int i=0; i<m; i++) {
      int a, b;
      scanf("%d %d", &a, &b);
      // 양방향 연결이므로 a→b, a←b
      G[a].push_back(b);
      G[b].push_back(a);
   }
}

int main() {
   input_G();
}
```

```
 idx  [0][1][2]
G[0]:  1  3  8
G[1]:  0  7
G[2]:  3  5  7
G[3]:  0  2  4
G[4]:  3  8
G[5]:  2  6
G[6]:  5
G[7]:  1  2
G[8]:  0  4

visited[0]: 0
visited[1]: 0
visited[2]: 0
visited[3]: 0
visited[4]: 0
visited[5]: 0
visited[6]: 0
visited[7]: 0
visited[8]: 0
```

깊이우선탐색(DFS)

- DFS 알고리즘 구현(인접리스트)
  - DFS 알고리즘

  def dfs(k):

  1) 정점 k를 처리하고 방문한 것으로 표시
  2) 정점 k와 연결된 모든 정점에 대하여 방문한 적이 없으면 그 정점에서 dfs, dfs 완료되면 되돌아오기(백트랙)

dfs함수가 종료되면 호출 위치로 알아서 되돌아오므로 딱히 백트랙을 구현할 필요는 없음.

```
int n, m;
vector<vector<int>> G;
vector<bool> visited;

// 정점 k에서 dfs
void dfs(int k) {
  // 정점 k를 방문하였음을 출력
  printf("dfs(%d) started.\n", k);
  // 나중에 다시오지 않기 위해 방문을 표시하고,
  ?

  // 정점 k와 연결된 모든 정점에 대하여
  for (int i=0; i<G[k].size(); i++) {
    // k와 연결된 i번째 정점에 방문한 적 없으면,
    if (?) {
      // 그 정점에서 다시 dfs 시작
      ?
      // dfs 완료하고 돌아왔다고 메시지 출력
      printf("return to dfs(%d).\n", k);
    }
  }
  return; //연결된 모든 정점을 방문완료하여 되돌아가기
}
```

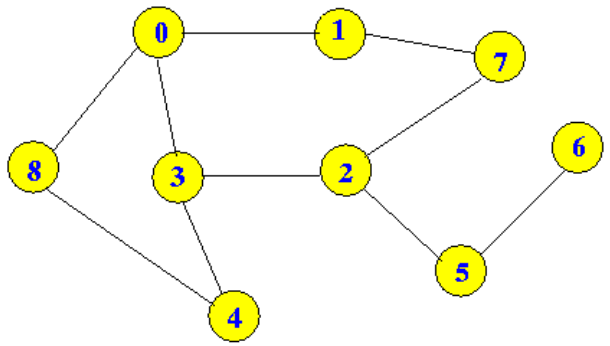
```
 idx  [0][1][2]
G[0]:  1  3  8
G[1]:  0  7
G[2]:  3  5  7
G[3]:  0  2  4
G[4]:  3  8
G[5]:  2  6
G[6]:  5
G[7]:  1  2
G[8]:  0  4

visited[0]: 0
visited[1]: 0
visited[2]: 0
visited[3]: 0
visited[4]: 0
visited[5]: 0
visited[6]: 0
visited[7]: 0
visited[8]: 0
```

깊이우선탐색(DFS) 전체 소스코드

dfs_full_src_quiz.cpp

```
#include <stdio.h>
#include <vector>
using namespace std;
int n, m;
vector<vector<int>> G;
vector<bool> visited;

void input_G() {
  scanf("%d %d", &n, &m);
  G.resize(n+1);  // 정점이 0부터 시작하므로 n+1
  visited.assign(n+1, 0);
  for(int i=0; i<m; i++) {
    int a, b;
    scanf("%d %d", &a, &b);
    G[a].push_back(b);
    G[b].push_back(a);
  }
}
void output_G() {
  printf("\n");
  for(int a=0; a<G.size(); a++) {
    printf("%2d:", a);
    for(int i : G[a])
      printf("%3d", i);
    printf("\n");
  }
  printf("\n");
}
```

```
// 정점 k에서 dfs
void dfs(int k) {
  // 정점 k를 방문하였음을 출력
  printf("dfs(%d) started.\n", k);
  // 나중에 다시오지 않기 위해 방문을 표시하고,
  ____________;

  // 정점 k와 연결된 모든 정점에 대하여
  for (int i=0; i<__________; i++) {
    // 정점k의 i번째 정점에 방문한 적이 없으면,
    if (____________) {
      // 그 정점에서 다시 dfs 시작
      ____________;
      // dfs 완료하고 돌아왔다고 메시지 출력
      printf("return to dfs(%d).\n", k);
    }
  }
  return;
}

int main() {
  input_G();
  output_G();
  dfs(0);
}
```




```
 idx  [0][1][2]
G[0]:  1  3  8
G[1]:  0  7
G[2]:  3  5  7
G[3]:  0  2  4
G[4]:  3  8
G[5]:  2  6
G[6]:  5
G[7]:  1  2
G[8]:  0  4

dfs(0) started.
dfs(1) started.
dfs(7) started.
dfs(2) started.
dfs(3) started.
dfs(4) started.
dfs(8) started.
return to dfs(4).
return to dfs(3).
return to dfs(2).
dfs(5) started.
dfs(6) started.
return to dfs(5).
return to dfs(2).
return to dfs(7).
return to dfs(1).
return to dfs(0).
```

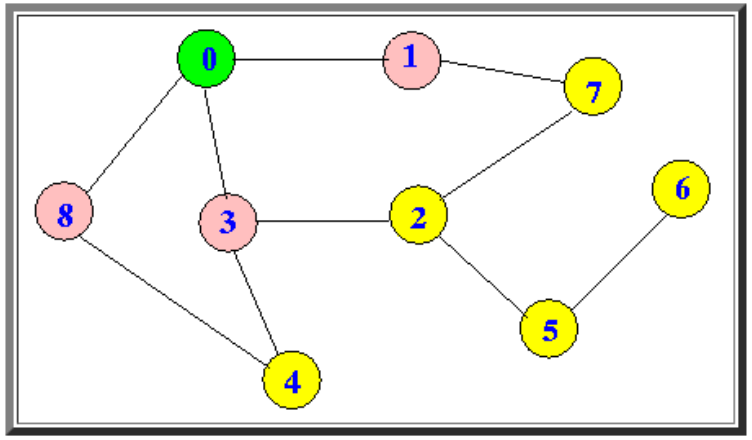
너비우선탐색(BFS)

- 너비우선탐색 알고리즘

  : Breadth First Search

1. 정점 k를 처리하고 큐에 삽입, k를 방문한 것으로 표시
2. 큐가 빌 때까지 다음을 반복:
   1) 큐에서 첫 번째 항목 삭제
   2) 삭제된 항목과 이웃하는 모든 정점에 대하여 방문하지 않은 정점이라면,
      ① 그 정점을 처리하고 큐에 삽입
      ② 그 정점에 방문을 표시

- 너비 우선의 의미
  - Visit ***all*** **the neighbors** first:



  - ***Only*** then visit the ***neighbors'*** **neighbors**:

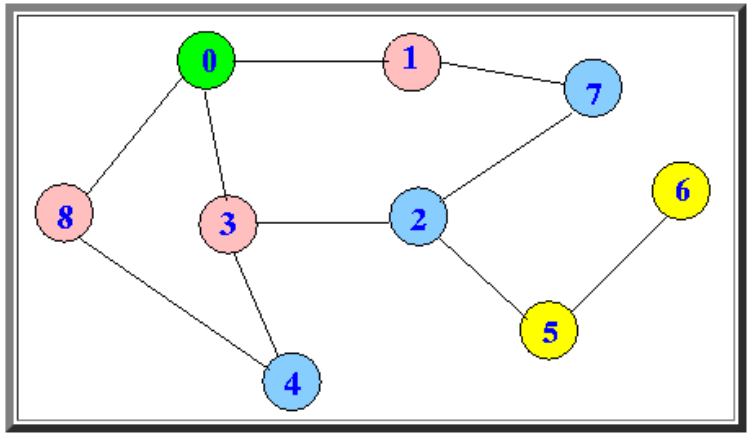
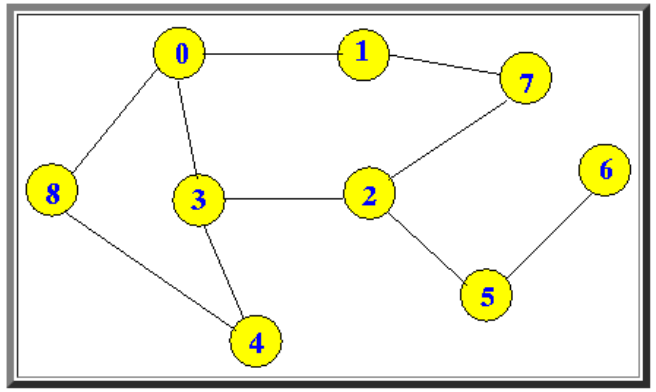


image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

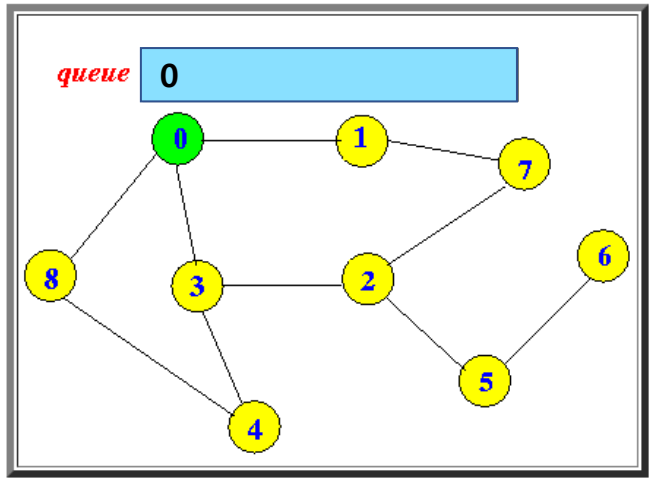
1. 정점 k를 처리하고 큐에 삽입, k를 방문한 것으로 표시
2. 큐가 빌 때까지 다음을 반복:
   1) 큐에서 첫 번째 항목 삭제
   2) 삭제된 항목과 이웃하는 모든 정점에 대하여 방문하지 않은 정점이라면,
      ① 그 정점을 처리하고 큐에 삽입
      ② 그 정점에 방문을 표시

너비우선탐색(BFS)

① Graph:

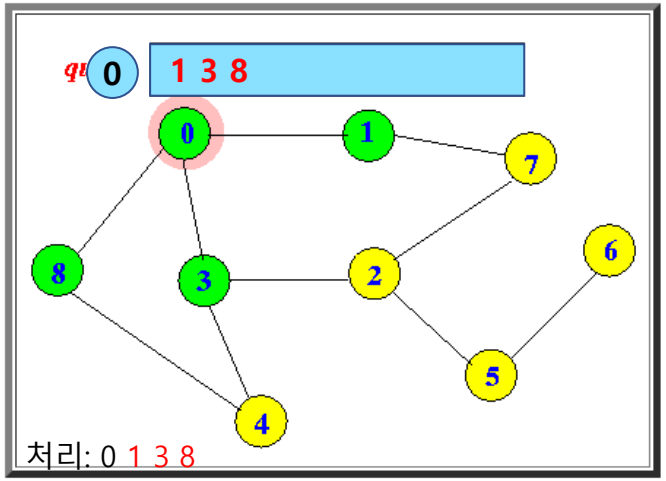


② Initial state:

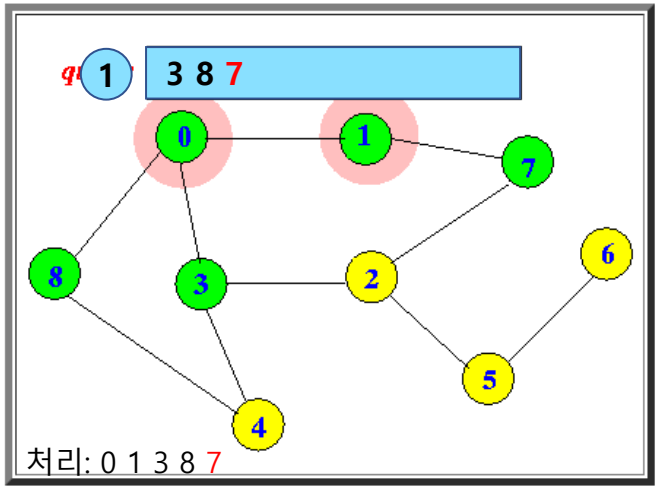


처리: 0

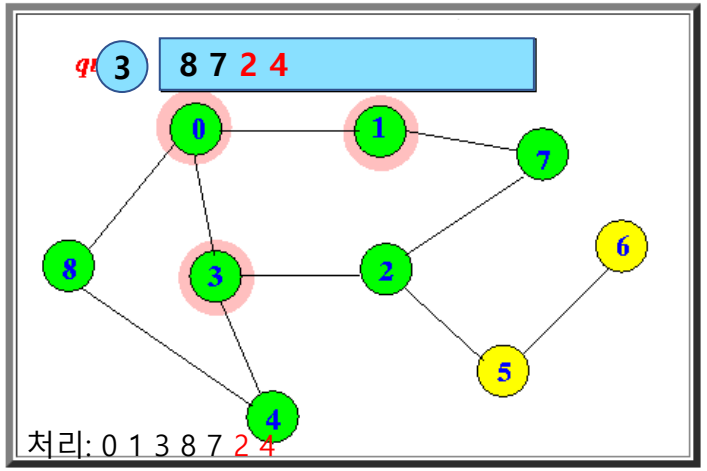
- State *after* processing 0



- State *after* processing 1



- State *after* processing 3



- State *after* processing 8

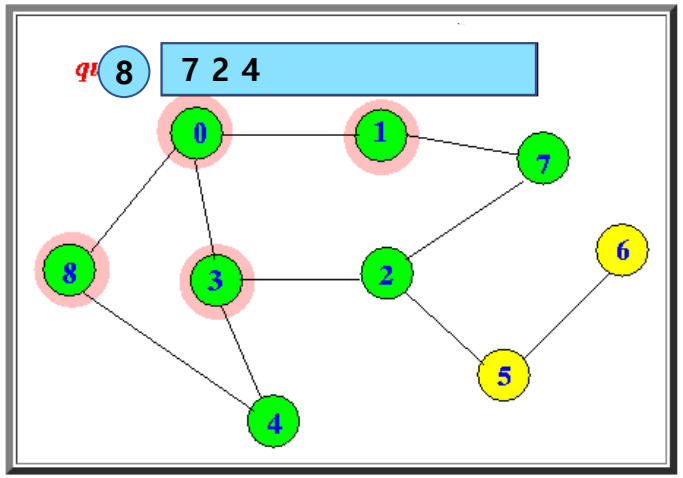
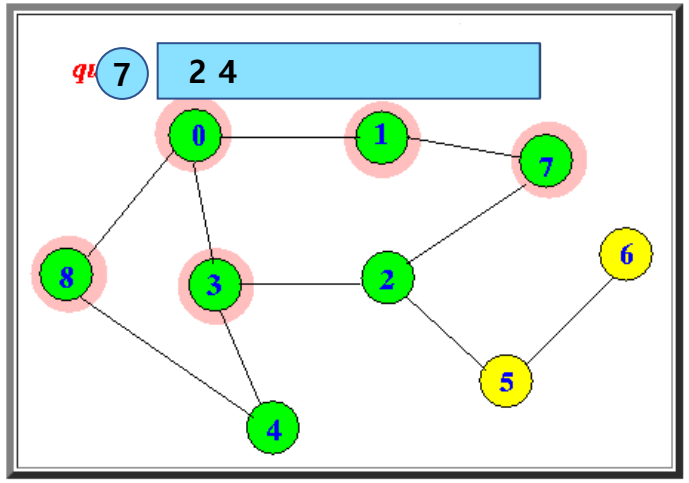


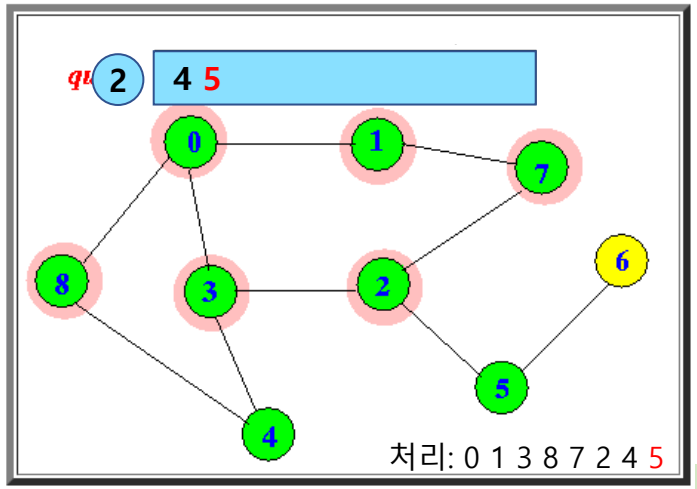
image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

너비우선탐색(BFS)

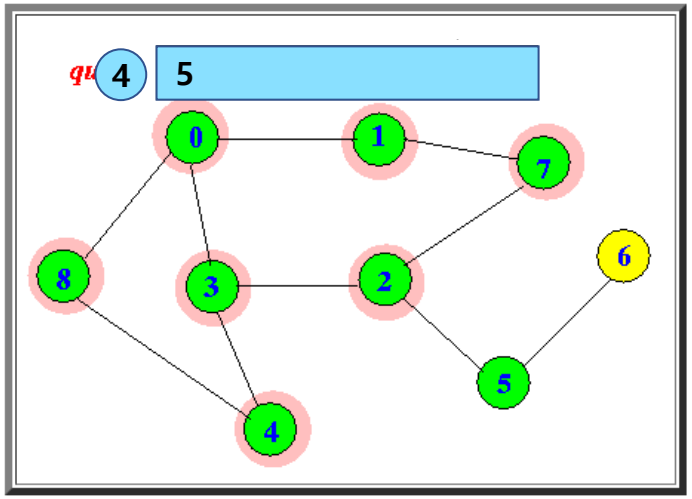
① State *after* processing 7



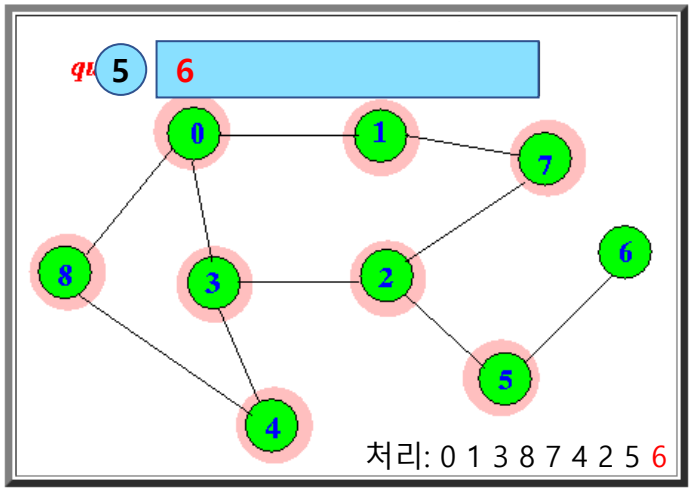
② State *after* processing 2



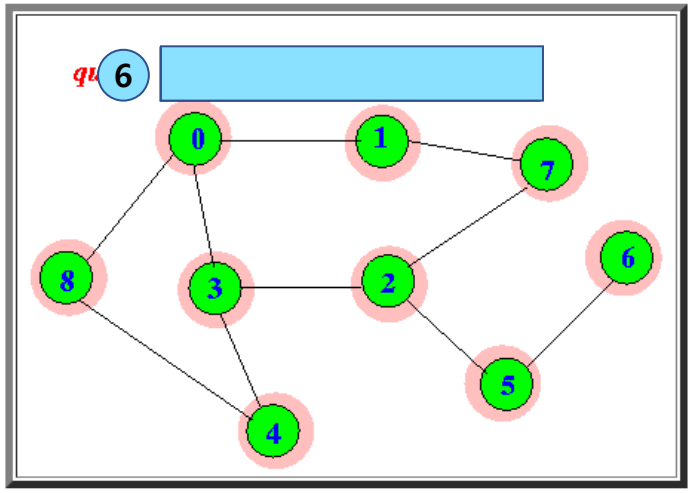
▪ State *after* processing 4



▪ State *after* processing 5



▪ State *after* processing 6



▪ DONE

(The queue has become empty)

image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

너비우선탐

- 너비우선탐색 알고리즘

  : Breadth First Search

  1. 정점 k를 처리하고 큐에 삽입, k를 방문한 것으로 표시
  2. 큐가 빌 때까지 다음을 반복:
     1) 큐에서 첫 번째 항목 삭제
     2) 삭제된 항목과 이웃하는 모든 정점에 대하여 방문하지 않은 정점이라면,
        ① 그 정점을 처리하고 큐에 삽입
        ② 그 정점에 방문을 표시

```
int n, m;
vector<vector<int>> G;
vector<bool> visited;

// 정점 k에서 bfs 시작
void bfs(int k) {
  queue<int> Q;   // 방금 방문한 이웃의 정점을 넣어 놓는 큐

  printf("bfs: (%d)\n", k);
  Q.push(k);          // 시작 정점을 Q에 삽입
  visited[k]=1;       // 시작 정점 방문했다고 표시

  while(     ?     ) {     // 큐에 내용물 있으면 계속반복
    int cur=Q.front();     // 큐의 첫번째 원소
    Q.pop();               // 빼냄(삭제)
    printf("%d deleted.\nbfs: ", cur);

    // 방금 전에 삭제한 정점과 이웃하는 모든 정점에 대하여
    for(int i=0; i<G[cur].size(); i++) {
      // 방문한 적이 없으면
      if(          ?          ) {
        printf("(%d) ", G[cur][i]);
            ?
            ?
      }
    }
    printf("\n");
  }
}
```

```
 idx  [0][1][2]
G[0]:  1  3  8
G[1]:  0  7
G[2]:  3  5  7
G[3]:  0  2  4
G[4]:  3  8
G[5]:  2  6
G[6]:  5
G[7]:  1  2
G[8]:  0  4
```

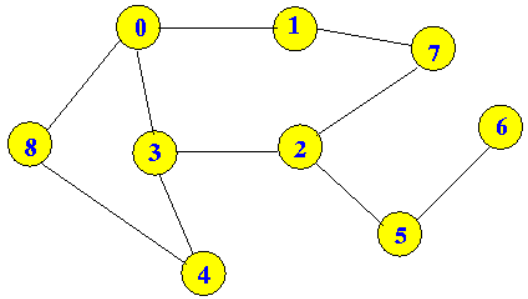
너비우선탐색(BFS) 전체 소스코드

bfs_full_src_quiz.cpp

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

int n, m;
vector<vector<int>> G;
vector<bool> visited;

// 현재 큐의 모습을 출력
void output_Q(queue<int> Q) {
  printf("Q:[");
  while(!Q.empty()) {
    int cur=Q.front();
    printf("%3d", cur);
    Q.pop();
  }
  printf("], ");
}
```

1. 정점 k를 처리하고 큐에 삽입, k를 방문한 것으로 표시
2. 큐가 빌 때까지 다음을 반복:
   1) 큐에서 첫 번째 항목 삭제
   2) 삭제된 항목과 이웃하는 모든 정점에 대하여 방문하지 않은 정점이라면,
      ① 그 정점을 처리하고 큐에 삽입
      ② 그 정점에 방문을 표시

```
// 정점 k에서 bfs
void bfs(int k) {
  queue<int> Q;

  printf("bfs: (%d)\n", k);
  Q.push(k);        // 시작 정점을 Q에 삽입
  visited[k]=1;     // 시작 정점 방문했다고 표시

  while(!Q.empty()) {
    output_Q(Q);

    int cur=              // 큐의 첫번째 원소
    Q.    ();             // 빼냄(삭제)
    printf("%d deleted.\nbfs: ", cur);

    // 삭제한 정점과 이웃하는 모든 정점에 대하여
    for(int i=0; i<              ; i++) {
      // 방문한 적이 없으면
      if(!visited[           ]) {
        printf("(%d) ", G[cur][i]);
        Q.push(        );
        visited[      ]=1;
      }
    }
    printf("\n");
  }
}
```

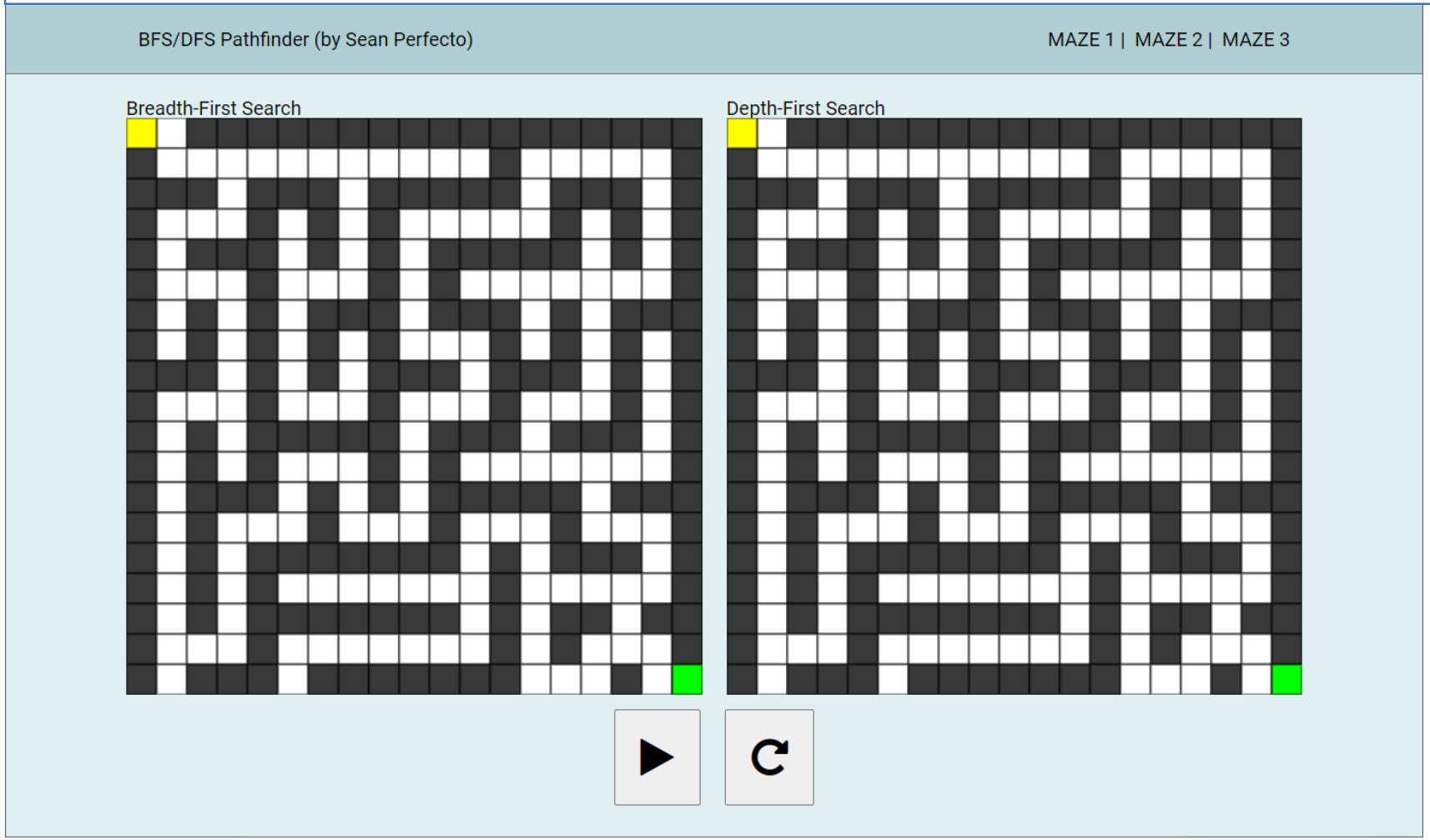



```
void output_G() {
  printf("\n");
  for(int a=0; a<=n; a++) {
    printf("%2d:", a);
    for(int i : G[a])
      printf("%3d", i);
    printf("\n");
  }
  printf("\n");
}

void input_G() {
  scanf("%d %d", &n, &m);
  // 정점이 0부터 시작하므로 n+1
  G.resize(n+1);
  visited.assign(n+1, 0);

  for(int i=0; i<m; i++) {
    int a, b;
    scanf("%d %d", &a, &b);
    G[a].push_back(b);
    G[b].push_back(a);
  }
}

int main() {
  input_G();
  output_G();
  bfs(0);
}
```

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

bfs: (0)
Q:[ 0], 0 deleted.
bfs: (1) (3) (8)
Q:[ 1 3 8], 1 deleted.
bfs: (7)
Q:[ 3 8 7], 3 deleted.
bfs: (2) (4)
Q:[ 8 7 2 4], 8 delete
bfs:
Q:[ 7 2 4], 7 deleted.
bfs:
Q:[ 2 4], 2 deleted.
bfs: (5)
Q:[ 4 5], 4 deleted.
bfs:
Q:[ 5], 5 deleted.
bfs: (6)
Q:[ 6], 6 deleted.
bfs:
```

미로 탈출 (BFS vs DFS)

https://seanperfecto.github.io/BFS-DFS-Pathfinder/



미로 탈출

- 문제

  정진이는 벽과 길로 만들어진 N행 M열(세로 N칸, 가로 M칸) 크기의 미로에 갇혀 있다.

  미로에서는 한 번에 한 칸씩만 이동할 수 있다. 벽은 0으로, 길은 0아닌 수로 표시된다.

  정진이가 미로에서 탈출하기 위해 이동하여야 하는 최소 칸수를 구하시오. 시작 칸과 마지막 칸도 이동거리에 포함시킨다.

입력 예	출력 예
7 8 1 0 1 0 0 0 0 1 1 0 1 0 0 1 1 1 1 1 8 1 1 1 0 1 0 1 0 1 0 0 0 1 1 1 0 1 0 1 1 1 1 0 0 1 0 1 0 9 0 0 0 1 1 1 0 1	11

- 입력

  첫 번째 줄에 두 정수 N, M이 주어진다.

  (4 ≤ N, M ≤ 200)

  다음 N개의 줄에는 각각 M개의 정수로 미로의 정보가 주어진다. 숫자는 각각 다음을 의미한다.

  (0: 벽, 1: 길, 8: 시작위치, 9: 도착위치)

- 출력

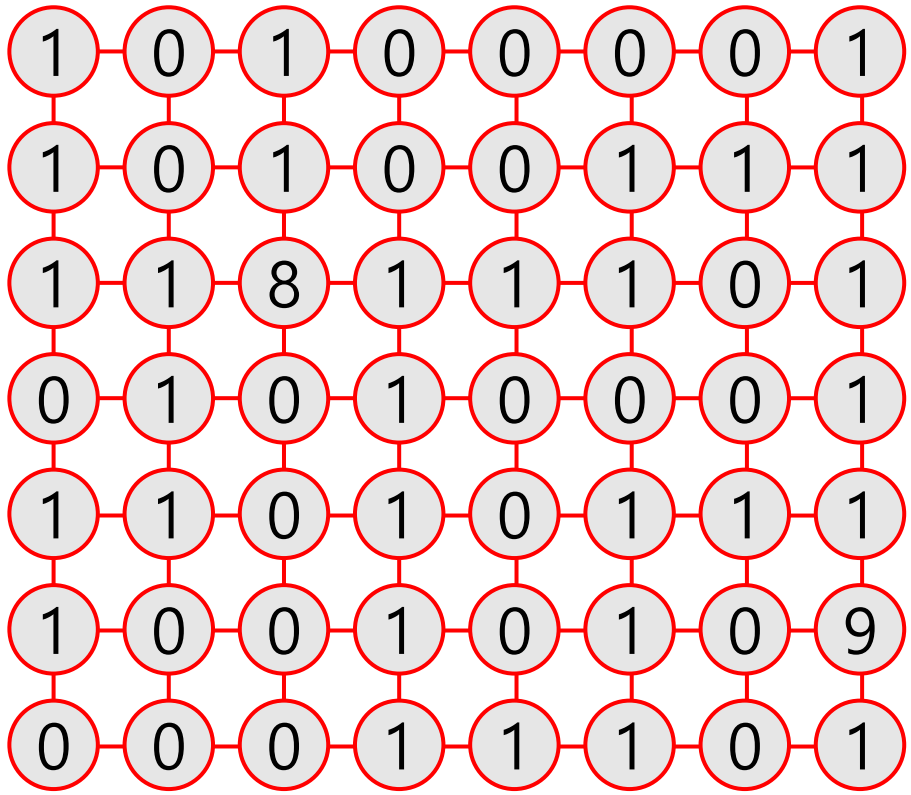
  첫 번째 줄에 최소 이동 칸의 개수를 출력한다.

미로 탈출

- 지도 데이터

```
7 8
1 0 1 0 0 0 0 1
1 0 1 0 0 1 1 1
1 1 8 1 1 1 0 1
0 1 0 1 0 0 0 1
1 1 0 1 0 1 1 1
1 0 0 1 0 1 0 9
0 0 0 1 1 1 0 1
```

- 그래프로 해석
  - 지도데이터를 그래프로보고 탐색기법 적용



미로 탈출 (초기설계)

```
#include <stdio.h>
#include <algorithm>
#include <queue>
#define  MAX_INT 0x7fffffff
using namespace std;

typedef struct {
    int a, b;    // a행 b열
} vertex;

int n, m;         // n행 m열
int M[201][201]; // 지도 정보
int Sa, Sb;  //출발지 a행 b열
int Ga, Gb;  //목적지 a행 b열

// 이동할 네 가지 방향 정의, 아래와 같이 적으면
// 아래, 오른쪽, 위쪽, 왼쪽 순서로 탐색하게 됨.
int da[] = {1, 0,-1, 0}; // 행(세로) 방향
int db[] = {0, 1, 0,-1}; // 열(가로) 방향

bool safe(int a, int b) {  // a행 b열
    return (0<=a && a<n) && (0<=b && b<m);
}
```

```
void input() {
   // (n행 m열)
   scanf("%d %d", &n, &m);
   // 2차원 리스트의 맵 정보 입력 받기
   for(int a=0; a<n; a++) {
       for(int b=0; b<m; b++) {
          int c;
          scanf("%d", &c);

          if(c==8) // 출발지 설정
             Sa=a, Sb=b;
          else if(c==9) // 목적지 설정
             Ga=a, Gb=b;

          // 모든 길을 -1로 변경
          // a행 b열에 삽입
          if(c>=1)
             M[a][b]=-1;
          else
             M[a][b]=0;
       }
   }
}
```

```
// 현재 맵에서 탐색 상태를 출력
void output(const char* title, int dist) {
    // 제목출력
    printf("\n[%s] (%d)\n", title, dist);

    for(int a = 0; a < n; a++) {
        for(int b = 0; b < m; b++) {
            printf("%3d", M[a][b]);
        }
        printf("\n");
    }
}

void dfs(int a, int b, int d) {
}

int main(void) {
    input();
    output("initial state",-1);
    // dfs 또는 bfs 하나의 함수만 호출할 것!
    dfs(Sa, Sb, 1);  // Sa행 Sb열에서 DFS시작
    //bfs(Sa, Sb, 1);  //Sa행 Sb열에서 BFS시작
    output("last state", -1);
    return 0;
}
```

미로 탈출 - DFS로 구현

- DFS 알고리즘

```
def dfs(k):
1)정점 k를 처리하고 방문한 것으로 표시
2)k와 연결된 모든 정점에 대하여
   방문한적이 없으면 그 정점에서 dfs,
   방문이 완료되면 되돌아 오기
```

- 초기맵 상태

```
Sa, Sb = (2, 2)
Ga, Gb = (5, 7)
```

	0	1	2	3	4	5	6	7
0	-1	0	-1	0	0	0	0	-1
1	-1	0	-1	0	0	-1	-1	-1
2	-1	-1	**-1**	-1	-1	-1	0	-1
3	0	-1	0	-1	0	0	0	-1
4	-1	-1	0	-1	0	-1	-1	-1
5	-1	0	0	-1	0	-1	0	**-1**
6	0	0	0	-1	-1	-1	0	-1

- DFS 구현

```
// a행, b열에서 dfs, 현재까지 계산한 거리는 d
void dfs(int a, int b, int d) {
    ?          //① a행 b열에 거리 기록
    //② 방문한 것으로 표시 이거 안해도 됨?
    //④ 목적지에 도착하면 맵상태와 도달거리 출력
    ?

    //③ 4방향으로 dfs
    // da, db배열을 이용하여 for문으로 간략화 가능
    ?
    // ↓
    // →
    // ↑
    // ←
}
```

미로 탈출 - DFS로 구현

- 결과 고찰

```
7 8
1 0 1 0 0 0 0 1
1 0 1 0 0 1 1 1
1 1 8 1 1 1 0 1
0 1 0 1 0 0 0 1
1 1 0 1 0 1 1 1
1 0 0 1 0 1 0 9
0 0 0 1 1 1 0 1

[initial state] (-1)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 -1 -1 -1
-1 -1 -1 -1 -1 -1  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[DFS success] (13)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 -1 -1 -1
-1 -1  1  2 -1 -1  0 -1
 0 -1  0  3  0  0  0 -1
-1 -1  0  4  0 10 11 12
-1  0  0  5  0  9  0 13
 0  0  0  6  7  8  0 -1

[last state] (-1)
 5  0  3  0  0  0  0 16
 4  0  2  0  0 17 16 15
 3  2  1  2 19 18  0 14
 0  3  0  3  0  0  0 13
 5  4  0  4  0 10 11 12
 6  0  0  5  0  9  0 13
 0  0  0  6  7  8  0 -1
```

- 엥? 최적해가 아닌데…
- 왜 최적해를 못 찾지?
- 원래 전체탐색이 진행되어야 하는데…

- DFS 구현

```
// a행, b열에서 dfs, 현재까지 계산한 거리는 d
void dfs(int a, int b, int d) {
    M[a][b] = d;  //① a행 b열에 거리 기록
    //② 방문한 것으로 표시 이거 안해도 됨?
    //④ 목적지에 도착하면 맵상태와 도달거리 출력
    if(a==Ga && b==Gb) {
        output("DFS success", d);
        return;
    }

    //③ 4방향으로 dfs
    // da, db배열을 이용하여 for문으로 간략화 가능
    if(safe(a+1, b) && M[a+1][b]==-1) // ↓
        dfs(a+1, b, d+1);
    if(safe(a, b+1) && M[a][b+1]==-1) // →
        dfs(a, b+1, d+1);
    if(safe(a-1, b) && M[a-1][b]==-1) // ↑
        dfs(a-1, b, d+1);
    if(safe(a, b-1) && M[a][b-1]==-1) // ←
        dfs(a, b-1, d+1);
}
```

미로 탈출 - DFS로 구현

- 이전 버전

```
// a행, b열에서 dfs, 현재까지 계산한 거리는 d
void dfs(int a, int b, int d) {
    M[a][b] = d;  //① a행 b열에 거리 기록
    //② 방문한 것으로 표시 이거 안해도 됨?
    //④ 목적지에 도착하면 맵상태와 도달거리 출력
    if(a==Ga && b==Gb) {
        output("DFS success", d);
        return;
    }

    //③ 4방향으로 dfs
    // da, db배열을 이용하여 for문으로 간략화 가능
    if(safe(a+1, b) && M[a+1][b]==-1) // ↓
        dfs(a+1, b, d+1);
    if(safe(a, b+1) && M[a][b+1]==-1) // →
        dfs(a, b+1, d+1);
    if(safe(a-1, b) && M[a-1][b]==-1) // ↑
        dfs(a-1, b, d+1);
    if(safe(a, b-1) && M[a][b-1]==-1) // ←
        dfs(a, b-1, d+1);
}
```

- 방향 탐색 배열이용 업데이트

```
// 이동할 네 가지 방향 정의, 아래와 같이 적으면
// 아래, 오른쪽, 위쪽, 왼쪽 순서로 탐색하게 됨.
int da[] = {1, 0,-1, 0}; // 행(세로) 방향
int db[] = {0, 1, 0,-1}; // 열(가로) 방향

// a행, b열에서 dfs, 현재까지 계산한 거리는 d
void dfs(int a, int b, int d) {
    M[a][b] = d;
    if(a==Ga && b==Gb) {
        output("DFS success", d);
        return;
    }

    //③ 4방향으로 dfs
    for(int i=0; i<4; i++) {
        int na = a+da[i], nb = b+db[i];
        if(safe(na, nb) && M[na][nb]==-1) {
            dfs(na, nb, d+1);
        }
    }
}
```

미로 탈출 - DFS로 구현

- DFS 구현 업데이트

```
// 최소거리(최적해)를 저장하기 위한 변수 셋팅
            ?
// a행, b열에서 dfs, 현재까지 계산한 거리는 d
void dfs(int a, int b, int d) {
    M[a][b] = d;
    if(a==Ga && b==Gb) {
        // 더 짧은 방법을 찾으면 최소거리 업데이트
            ?
        output("DFS success", d);
        return;
    }
    //③ 4방향으로 dfs 방법 업데이트, 백트랙시 길 복원
                    ?
}
```

- 결과 확인

```
[initial state] (-1)
 -1  0 -1  0  0  0  0 -1
 -1  0 -1  0  0 -1 -1 -1
 -1 -1 -1 -1 -1 -1  0 -1
  0 -1  0 -1  0  0  0 -1
 -1 -1  0 -1  0 -1 -1 -1
 -1  0  0 -1  0 -1  0 -1
  0  0  0 -1 -1 -1  0 -1

[DFS success] (11)
 -1  0 -1  0  0  0  0 -1
 -1  0 -1  0  0  5  6  7
 -1 -1  1  2  3  4  0  8
  0 -1  0 -1  0  0  0  9
 -1 -1  0 -1  0 -1 -1 10
 -1  0  0 -1  0 -1  0 11
  0  0  0 -1 -1 -1  0 -1

[DFS success] (13)
 -1  0 -1  0  0  0  0 -1
 -1  0 -1  0  0 -1 -1 -1
 -1 -1  1  2 -1 -1  0 -1
  0 -1  0  3  0  0  0 -1
 -1 -1  0  4  0 10 11 12
 -1  0  0  5  0  9  0 13
  0  0  0  6  7  8  0 -1

[last state] (-1)
 -1  0 -1  0  0  0  0 -1
 -1  0 -1  0  0 -1 -1 -1
 -1 -1  1 -1 -1 -1  0 -1
  0 -1  0 -1  0  0  0 -1
 -1 -1  0 -1  0 -1 -1 -1
 -1  0  0 -1  0 -1  0 -1
  0  0  0 -1 -1 -1  0 -1
```

미로 탈출 – DFS 최종 구현

```
int min_dist=MAX_INT;
// a행, b열에서 dfs, 현재까지 계산한 거리는 d
void dfs(int a, int b, int d) {
    M[a][b] = d;

    // 목적지에 도달하면,
    if(a==Ga && b==Gb) {
        if(min_dist > d)
            min_dist=d;
        output("DFS search success", d);
        return;
    }

    int noway=0;  // 현재 위치에서 이동 불가능 방향 개수
    for(int i=0; i<4; i++) {
        int na = a+da[i];
        int nb = b+db[i];

        if(safe(na, nb) && M[na][nb]==-1) {
            dfs(na, nb, d+1);
            M[na][nb]=-1;  // 백트랙할 경우 길복원
        }
        else
            noway++; // 이동불가 카운터 증가

        if(noway==4) // 4방향 모두 길이 막혀있으면,
            output("DFS search fail", -1);
    }
}
```

```
[initial state] (-1)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 -1 -1 -1
-1 -1 -1 -1 -1 -1  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[DFS search success] (13)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 -1 -1 -1
-1 -1  1  2 -1 -1  0 -1
 0 -1  0  3  0  0  0 -1
-1 -1  0  4  0 10 11 12
-1  0  0  5  0  9  0 13
 0  0  0  6  7  8  0 -1

[DFS search fail] (-1)
-1  0 -1  0  0  0  0 16
-1  0 -1  0  0 -1 -1 15
-1 -1  1  2 -1 -1  0 14
 0 -1  0  3  0  0  0 13
-1 -1  0  4  0 10 11 12
-1  0  0  5  0  9  0 -1
 0  0  0  6  7  8  0 -1

[DFS search fail] (-1)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 17 16 15
-1 -1  1  2 19 18  0 14
 0 -1  0  3  0  0  0 13
-1 -1  0  4  0 10 11 12
-1  0  0  5  0  9  0 -1
 0  0  0  6  7  8  0 -1
```

```
[DFS search success] (11)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0  5  6  7
-1 -1  1  2  3  4  0  8
 0 -1  0 -1  0  0  0  9
-1 -1  0 -1  0 -1 -1 10
-1  0  0 -1  0 -1  0 11
 0  0  0 -1 -1 -1  0 -1

[DFS search fail] (-1)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0  5  6  7
-1 -1  1  2  3  4  0  8
 0 -1  0 19  0  0  0  9
-1 -1  0 18  0 12 11 10
-1  0  0 17  0 13  0 -1
 0  0  0 16 15 14  0 -1

[DFS search fail] (-1)
-1  0 -1  0  0  0  0  8
-1  0 -1  0  0  5  6  7
-1 -1  1  2  3  4  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[DFS search fail] (-1)
-1  0  3  0  0  0  0 -1
-1  0  2  0  0 -1 -1 -1
-1 -1  1 -1 -1 -1  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1
```

```
[DFS search fail] (-1)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 -1 -1 -1
-1  2  1 -1 -1 -1  0 -1
 0  3  0 -1  0  0  0 -1
 5  4  0 -1  0 -1 -1 -1
 6  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[DFS search fail] (-1)
 5  0 -1  0  0  0  0 -1
 4  0 -1  0  0 -1 -1 -1
 3  2  1 -1 -1 -1  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[last state] (-1)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 -1 -1 -1
-1 -1  1 -1 -1 -1  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1
```

미로 탈출 - BFS로 구현

- BFS 알고리즘

1. 정점 k를 처리하고 큐에 삽입, k를 방문한 것으로 표시
2. 큐가 빌 때까지 다음을 반복:
   1) 큐에서 첫 번째 항목 삭제
   2) 삭제된 항목과 이웃하는 모든 정점에 대하여 방문하지 않은 정점이라면,
      ① 그 정점을 처리하고 큐에 삽입
      ② 그 정점에 방문을 표시

- 초기 맵 상태

Sa, Sb = (2, 2)
Ga, Gb = (5, 7)

	0	1	2	3	4	5	6	7
0	-1	0	-1	0	0	0	0	-1
1	-1	0	-1	0	0	-1	-1	-1
2	-1	-1	**-1**	-1	-1	-1	0	-1
3	0	-1	0	-1	0	0	0	-1
4	-1	-1	0	-1	0	-1	-1	-1
5	-1	0	0	-1	0	-1	0	**-1**
6	0	0	0	-1	-1	-1	0	-1

```
// bfs(시작행a, 시작열b, 거리d)
void bfs(int sa, int sb, int d) {
    ?                  // 시작위치에 거리 d표시, 방문표시 안함?
    queue<vertex> q;
    ?                  // 큐에 현재 위치 삽입

    while(!q.empty()) {  // 큐가 빌 때까지 반복
        ?              // 큐의 첫 번째 정점 a행
                       // 큐의 첫 번째 정점 b열
                       // 큐에서 첫 번째 항목 삭제

        // 현재 위치에서 4가지 방향으로의 위치 확인
        for(int i = 0; i < 4; i++) {
            ?          // na: next a
                       // nb: next b
            if (safe(na, nb) && M[na][nb] == -1) {
                ?      // 다음 위치에 거리 표시
                // 목적지에 도달하면 성공 출력하고 탈출
                ?

                ?      // 다음 위치를 큐에 삽입
            }
        }
    }
    output("BFS search fail", -1); // 여기까지 오면 탐색 실패임
}
```

미로 탈출 - BFS로 구현

```
// bfs(시작행a, 시작열b, 거리d)
void bfs(int sa, int sb, int d) {
    M[sa][sb] = d;      // 시작위치에 거리 d표시
    queue<vertex> q;
    q.push({sa, sb});  // 큐에 현재 위치 삽입

    while(!q.empty()) {  // 큐가 빌 때까지 반복
        int a = q.front().a; // 큐의 첫 번째 정점 a행
        int b = q.front().b; // 큐의 첫 번째 정점 b열
        q.pop();             // 큐에서 첫 번째 항목 삭제
        // 현재 위치에서 4가지 방향으로의 위치 확인
        for(int i=0; i<4; i++) {
            int na=a+da[i];  // na: next a
            int nb=b+db[i];  // nb: next b
            if (safe(na, nb) && M[na][nb] == -1) {
                M[na][nb] = M[a][b] + 1;  // 다음 위치에 거리 표시
                // 목적지에 도달하면 성공 출력하고 탈출
                if(na==Ga && nb==Gb) {
                    output("BFS search success", M[na][nb]);
                    return;
                }
                q.push({na, nb});   // 다음 위치를 큐에 삽입
            }
        }
    }
    output("BFS search fail", -1); // 여기까지 오면 탐색 실패임
}
```

- 결과 확인

```
7 8
1 0 1 0 0 0 0 1
1 0 1 0 0 1 1 1
1 1 8 1 1 1 0 1
0 1 0 1 0 0 0 1
1 1 0 1 0 1 1 1
1 0 0 1 0 1 0 9
0 0 0 1 1 1 0 1

[initial state] (-1)
 -1  0 -1  0  0  0  0 -1
 -1  0 -1  0  0 -1 -1 -1
 -1 -1 -1 -1 -1 -1  0 -1
  0 -1  0 -1  0  0  0 -1
 -1 -1  0 -1  0 -1 -1 -1
 -1  0  0 -1  0 -1  0 -1
  0  0  0 -1 -1 -1  0 -1
```

```
[BFS search success] (11)
  5  0  3  0  0  0  0  8
  4  0  2  0  0  5  6  7
  3  2  1  2  3  4  0  8
  0  3  0  3  0  0  0  9
  5  4  0  4  0 10 11 10
  6  0  0  5  0  9  0 11
  0  0  0  6  7  8  0 -1

[last state] (-1)
  5  0  3  0  0  0  0  8
  4  0  2  0  0  5  6  7
  3  2  1  2  3  4  0  8
  0  3  0  3  0  0  0  9
  5  4  0  4  0 10 11 10
  6  0  0  5  0  9  0 11
  0  0  0  6  7  8  0 -1
```

미로 탈출 - BFS로 구현

- 탐색 과정을 보여주는 업데이트

```
if (safe(na, nb) && M[na][nb] == -1) {
    int new_d = M[a][b] + 1;  // 새로운 거리 계산

    // 새롭게 계산된 거리가 지금 거리보다 멀어지면,
    if(new_d > d) { // 현재 맵상태 일단 출력하고
        output("BFS searched new distance", -1);
        d = new_d;
    }
    M[na][nb] = new_d; // 다음 위치에 거리 표시
    // 목적지에 도달하면 성공 출력하고 탈출
    if(na==Ga && nb==Gb) {
        output("BFS search success", M[na][nb]);
        return;
    }
    q.push({na, nb});   // 다음 위치를 큐에 삽입
}
```

```
[BFS searched new distance]
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 -1 -1 -1
-1 -1  1 -1 -1 -1  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[BFS searched new distance]
-1  0 -1  0  0  0  0 -1
-1  0  2  0  0 -1 -1 -1
-1  2  1  2 -1 -1  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[BFS searched new distance]
-1  0  3  0  0  0  0 -1
-1  0  2  0  0 -1 -1 -1
 3  2  1  2  3 -1  0 -1
 0  3  0  3  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[BFS searched new distance]
-1  0  3  0  0  0  0 -1
 4  0  2  0  0 -1 -1 -1
 3  2  1  2  3  4  0 -1
 0  3  0  3  0  0  0 -1
-1  4  0  4  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1
```

```
[BFS searched new distance]
 5  0  3  0  0  0  0 -1
 4  0  2  0  0  5 -1 -1
 3  2  1  2  3  4  0 -1
 0  3  0  3  0  0  0 -1
 5  4  0  4  0 -1 -1 -1
-1  0  0  5  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[BFS searched new distance]
 5  0  3  0  0  0  0 -1
 4  0  2  0  0  5  6 -1
 3  2  1  2  3  4  0 -1
 0  3  0  3  0  0  0 -1
 5  4  0  4  0 -1 -1 -1
 6  0  0  5  0 -1  0 -1
 0  0  0  6 -1 -1  0 -1

[BFS searched new distance]
 5  0  3  0  0  0  0 -1
 4  0  2  0  0  5  6  7
 3  2  1  2  3  4  0 -1
 0  3  0  3  0  0  0 -1
 5  4  0  4  0 -1 -1 -1
 6  0  0  5  0 -1  0 -1
 0  0  0  6  7 -1  0 -1

[BFS searched new distance]
 5  0  3  0  0  0  0  8
 4  0  2  0  0  5  6  7
 3  2  1  2  3  4  0  8
 0  3  0  3  0  0  0 -1
 5  4  0  4  0 -1 -1 -1
 6  0  0  5  0 -1  0 -1
 0  0  0  6  7  8  0 -1
```

https://algo.datahub.pe.kr/src/maze/bfs/2.%20maze_bfs_show_process.cpp

미로 탈출 DFS + BFS 전체 소스코드

```
//written by akapo@naver.com
#include <stdio.h>
#include <algorithm>
#include <queue>
#define  MAX_INT 0x7fffffff
using namespace std;

typedef struct {
    int a, b;
} vertex;

int n, m;
int M[201][201]; // 지도 정보
int Sa, Sb;  //출발지 a행 b열
int Ga, Gb;  //목적지 a행 b열

// 이동할 네 가지 방향 정의, 아래와 같이 적으면
// 아래, 오른쪽, 위쪽, 왼쪽 순서로 탐색하게 됨.
int da[] = {1, 0,-1, 0}; // 행 방향
Int db[] = {0, 1, 0,-1}; // 열 방향

bool safe(int a, int b) {  // a행 b열
    return (0<=a && a<n) && (0<=b && b<m);
}

void input() {
    // N, M을 공백을 기준으로 구분하여 입력 받기(n행 m열)
    scanf("%d %d", &n, &m);
    // 2차원 리스트의 맵 정보 입력 받기
    for(int a=0; a<n; a++) {
        for(int b=0; b<m; b++) {
            int c;
            scanf("%d", &c);

            if(c==8) Sa=a, Sb=b;    // 출발지 설정
            else if(c==9) Ga=a, Gb=b; // 목적지 설정

            // 출발지 목적지 포함 모든 길을 -1로 변경함
            if(c>=1) M[a][b]=-1;
            else M[a][b]=0; // a행 b열에 삽입
        }
    }
}
```

```
// 현재 맵에서 탐색 상태를 출력
void output(const char* title, int dist) {
    if(dist > 0)
        printf("\n[%s] (%d)\n", title, dist);
    else
        printf("\n[%s]\n", title);

    for(int a=0; a<n; a++) {
        for(int b = 0; b < m; b++) {
            printf("%3d", M[a][b]);
        }
        printf("\n");
    }
}
// bfs(시작행a, 시작열b, 거리d)
void bfs(int sa, int sb, int d) {
    M[sa][sb] = d;
    queue<vertex> q;
    q.push({sa, sb});

    while(!q.empty()) {  // 큐가 빌 때까지 반복하기
        int a = q.front().a;
        int b = q.front().b;
        q.pop();

        // 현재 위치에서 4가지 방향으로의 위치 확인
        for(int i=0; i<4; i++) {
            int na=a+da[i]; // na: next a
            int nb=b+db[i]; // nb: next b

            if (safe(na, nb) && M[na][nb] == -1) {
                M[na][nb] = M[a][b] + 1;

                if(na==Ga && nb==Gb) { // 목적지에 도달하면,
                    output("BFS search success", M[na][nb]);
                    return;
                }
                q.push({na, nb});
            }
        }
    }
    output("BFS search fail", -1);
}
```

```
int min_dist=MAX_INT;
// a행, b열에서 dfs, 현재까지 계산한 거리는 d
void dfs(int a, int b, int d) {
    M[a][b] = d;

    // 목적지에 도달하면,
    if(a==Ga && b==Gb) {
        if(min_dist > d)
            min_dist=d;
        output("DFS search success", d);
        return;
    }

    int noway=0;  // 현재 위치에서 이동 불가능 방향 개수
    for(int i=0; i<4; i++) {
        int na = a+da[i];
        int nb = b+db[i];

        if(safe(na, nb) && M[na][nb]==-1) {
            dfs(na, nb, d+1);
            M[na][nb]=-1;  // 백트랙할 경우 길복원
        }
        else
            noway++; // 이동불가 카운터 증가

        if(noway==4) // 4방향 모두 길이 막혀있으면,
            output("DFS search fail", -1);
    }
}

int main(void) {
    input();
    output("initial state",-1);
    // dfs 또는 bfs 하나의 함수만 호출할 것!
    // Sa행 Sb열에서 DFS시작
    dfs(Sa, Sb, 1);
    // Sa행 Sb열에서 BFS시작
    //bfs(Sa, Sb, 1);
    output("last state",-1);
    return 0;
}
```