- 교육자료: https://gifted.datahub.pe.kr/
- 소스코드: https://gifted.datahub.pe.kr/src/
- Smart OJ: https://soj.datahub.pe.kr/

알 고 리 즘 B반

< C언어 >

충북교육연구정보원 정보영재교육원 | SW·AI교실 | 정보아카데미 강사

흥덕고등학교 교사 박정진

Online Judge 프로그램 계정 정보

순번	소속학교	이름	학년	ID	PW
1	서현중학교	김단우	중3	gifted2601	이름영타로
2	중앙초등학교	김도윤	초6	gifted2602	〃
3	주성중학교	김보영	중1	gifted2603	〃
4	솔밭초등학교	김여준	초6	gifted2504	〃
5	산남중학교	김지윤	중2	gifted2505	〃
6	한국교원대부설초등학교	이원희	초6	gifted2506	〃
7	산남중학교	임채윤	중2	gifted2507	〃
8	가경중학교	장진혁	중2	gifted2508	〃
9	오송중학교	정수현	중1	gifted2509	〃
10	오송중학교	정지현	중1	gifted2510	〃
11	각리중학교	조신후	중1	gifted2511	〃
12	금천중학교	조태운	중2	gifted2512	〃
13	청주대성초등학교	조현우	초6	gifted2513	〃
14	청주소로초등학교	홍동건	초6	gifted2514	〃
15	서전중학교	홍현진	중1	gifted2515	〃

난수의 생성과 활용

의사 랜덤

▪ 개요

- 의사 랜덤(pseudo-randomness)은 규칙이 없는 무작위적인 값을 임의로 만들어 내는 것
- 그렇게 만들어진 값들은 통계적으로 실제 랜덤과 비슷한 확률로 분포해야 함
- 의사 랜덤은 자연계의 무작위성과 다르게, 무작위 숫자들을 재현할 수 있기 때문에 통계적 처리, 알고리즘이나 오류 테스트, 서로 다른 알고리즘의 효율성 비교에 효과적으로 쓰임

▪ 선형 합동 생성법

- 무작위 수열을 만들어 내는 간단한 방법

$$X_{n+1} = (X_n * a + c) \bmod m$$

변수	범위	설명
X_0	$0 \le X_0 \le m$	seed, 시작하는 값
a	$0 < a < m$	multipier, 곱하는 값
c	$0 \le c < m$	increment, 더하는 값
m	$0 < m$	modulus, 나눌 값

- 시작 값, 곱할 값, 더할 값, 나눌 값에 따라 생성되는 값들의 순서와 무작위성이 결정된다.

의사 랜덤

- 처음 시작하는 시드 값 0, m=9, a=4, c=1 로 할 때 생성되는 선형 합동 생성 수열을 계산하는 프로그램을 작성해 보자.

$$X_{n+1} = (X_n * a + c) \bmod m$$

n	0	1	2	3	4	5	6	7	8	9	10	11
X_n	0											

```c
#include <stdio.h>

int seed = 0;
int a=4, c=1, m=9;

int next(int x) {
    int n=(x*a+c) % m;
    return n;
}

int main() {
    int x = seed;
    for(int i=0; i<=16; i++) {
        x = next(x);
        printf("%d ", x);
    }
}
```

- 다양한 시드 값, m, a, c를 선택해 만들어지는 수열에 대해 규칙성 여부, 주기 등의 특성의 분석해 보자.

의사 랜덤

- 나쁜 예: a=1, c=1, m=100 → X는 그냥 1씩 증가하는 수열 (전혀 랜덤 아님)
- 나쁜 예: a=5, c=0, m=16 → 주기가 매우 짧게 반복됨 (0이 아닌 시드라도 금방 순환)
- 좋은 예: a=1103515245, c=12345, m=2^31 (ANSI C 값)

선형 합동 생성법(LCG)에서 파라미터를 아무렇게나 고르면 주기가 짧아지거나 패턴이 눈에 보일 정도로 품질이 나빠집니다. 실제로 이론적으로 검증된 조건과, 역사적으로 검증되어 널리 쓰인 파라미터 세트들이 있어요.

1. 최대 주기를 보장하는 조건 (Hull-Dobell 정리)

$X_{n+1} = (aX_n + c) \bmod m$ 이 **모든 초기값(seed)에 대해 주기** m(가능한 최대 주기)을 갖도록 하려면, 다음 3가지 조건을 **동시에** 만족해야 합니다:

1. $\gcd(c, m) = 1$ — c와 m이 서로소
2. m의 모든 소인수 p에 대해, $a - 1$이 p로 나누어떨어짐
3. 만약 m이 4의 배수라면, $a - 1$도 4의 배수여야 함

이 세 조건을 만족하면 시드가 무엇이든 상관없이 $0 \sim m - 1$까지 모든 값을 정확히 한 번씩 순회하고 돌아옵니다. 반대로 이 조건을 어기면 주기가 m보다 훨씬 짧아지거나, 특정 시드에서 값이 금방 반복되는 문제가 생겨요.

의사 랜덤

2. $c = 0$ 인 경우 (곱셈 합동 생성법, MCG)

$c = 0$이면 조건 1(서로소)이 자동으로 깨지기 때문에 위 정리를 그대로 못 씁니다. 대신 m을 소수로 두고, a를 m의 **원시근(primitive root)**으로 고르면 주기 $m - 1$(0 제외)을 얻을 수 있어요. 실무에서는 이 방식이 더 흔하게 쓰입니다.

3. 실제로 검증되어 널리 쓰인 파라미터

직접 최적의 a, c, m을 고르는 건 스펙트럴 테스트(spectral test) 같은 정교한 수학적 검증이 필요해서 어렵습니다. 대신 역사적으로 검증된 값들을 그대로 쓰는 게 안전해요.

이름	a	c	m	비고
MINSTD (Park-Miller)	16807	0	$2^{31} - 1$ (소수)	가장 널리 인용되는 "표준" LCG
ANSI C (일부 구현)	1103515245	12345	2^{31}	오래된 `rand()` 구현에서 사용
Numerical Recipes	1664525	1013904223	2^{32}	32비트 정수 오버플로우를 이용
glibc의 예전 rand()	1103515245	12345	2^{31}	ANSI C와 동일 계열
POSIX drand48	0x5DEECE66D	0xB	2^{48}	48비트, 더 긴 주기

Source	*m*	(multiplier) *a*	(increment) *c*	output bits of seed in *rand()* or *Random(L)*
Numerical Recipes	2^{32}	1664525	1013904223	
Borland C/C++	2^{32}	22695477	1	bits 30..16 in *rand()*, 30..0 in *lrand()*
glibc (used by GCC)	2^{31}	1103515245	12345	bits 30..0
ANSI C: Watcom, Digital Mars, CodeWarrior, IBM VisualAge C/C++	2^{31}	1103515245	12345	bits 30..16
C99, C11: Suggestion in the ISO/IEC 9899	2^{31}	1103515245	12345	bits 30..16
Borland Delphi, Virtual Pascal	2^{32}	134775813	1	bits 63..32 of *(seed * L)*
Turbo Pascal	2^{32}	33797	1	
Microsoft Visual/Quick C/C++	2^{32}	214013 ($343FD_{16}$)	2531011 ($269EC3_{16}$)	bits 30..16
Microsoft Visual Basic (6 and earlier)	2^{24}	1140671485 ($43FD43FD_{16}$)	12820163 ($C39EC3_{16}$)	
RtlUniform from Native API	$2^{31} - 1$	2147483629 ($7FFFFFED_{16}$)	2147483587 ($7FFFFFC3_{16}$)	
Apple CarbonLib, C++11's `minstd_rand0`	$2^{31} - 1$	16807	0	see MINSTD
C++11's `minstd_rand`	$2^{31} - 1$	48271	0	see MINSTD
MMIX by Donald Knuth	2^{64}	6364136223846793005	1442695040888963407	
Newlib, Musl	2^{64}	6364136223846793005	1	bits 63...32
VMS's **MTH$RANDOM**, old versions of glibc	2^{32}	69069	1	
Java's java.util.Random, POSIX [ln]rand48, glibc [ln]rand48[_r]	2^{48}	25214903917 ($5DEECE66D_{16}$)	11	bits 47...16
`random0` If X_n is even then X_{n+1} will be odd, and vice versa—the lowest bit oscillates at each step.	134456 = $2^3 7^5$	8121	28411	$\frac{X_n}{134456}$
POSIX [jm]rand48, glibc [mj]rand48[_r]	2^{48}	25214903917 ($5DEECE66D_{16}$)	11	bits 47...15
POSIX [de]rand48, glibc [de]rand48[_r]	2^{48}	25214903917 ($5DEECE66D_{16}$)	11	bits 47...0
cc65	2^{23}	65793 (10101_{16})	4282663	bits 22...8
Formerly common: RANDU	2^{31}	65539	0	

▲ 다양한 LCG(선형 합동 생성식 $X_{n+1}=(a*X_n+C) \bmod m$)의 상수($a$, c, m) 조합

의사 랜덤

- 의사 난수 생성 함수

```
int rand(void);
```

  - 0부터 RAND_MAX(32767) 범위의 의사 난수를 반환
  - <stdlib.h> 헤더파일 필요

- 테스트

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

int main(void) {
    for(int i=0; i<10; i++) {
        printf("%d\n", rand());
    }

    return 0;
}
```

  - 기대와는 달리 항상 똑같은 결과에 어리둥절

의사 랜덤

- 매번 다른 결과를 얻으려면…

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
    srand(time(NULL));

    for(int i=0; i<10; i++) {
        printf("%d\n", rand());
    }

    return 0;
}
```

- 범위 난수

```c
// a:시작수, b:마지막수
int range_rand(int a, int b) {
    return rand()%(b-a+1)+a;
}

// 11부터 20 사이의 난수를 얻고 싶을 때,
range_rand(11, 20);
// 11부터 20까지는 10개의 숫자이므로
// 10으로 나눈 나머지를 구하면 10개 숫자를
// 얻을 수 있고 11부터 이므로 11을 더해야 함.
// 따라서 수식은 아래와 같이 만들어야 함.
rand() % (20-11 +1) + 11
```

의사 랜덤

- 범위 난수 테스트

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

int range_rand(int a, int b) {
    return rand()%(b-a+1)+a;
}

int main(void) {
    srand(time(NULL));
    int cnt[12]={0, };
    printf("RAND_MAX: %d\n", RAND_MAX);

    for(int i=0; i<10000; i++)
        cnt[range_rand(1, 10)]++; // [1, 10]

    for(int i=0; i<=11; i++) {
        printf("%2d: %5d\n", i, cnt[i]);
    }
}
```

```
RAND_MAX: 32767
 0:     0
 1:  1039
 2:   981
 3:  1004
 4:  1027
 5:   958
 6:  1035
 7:   951
 8:  1032
 9:  1004
10:   969
11:     0
```

- rand()의 문제점
  - RAND_MAX(32767)를 넘어서는 난수를 뽑을 수 없음.
  - 범위 제한 시 % 연산은 균등 확률을 보장하지 못함.
    - 예를들어, 0 ~ 32767 범위의 난수를 10으로 나눈 나머지는 0~7이 될 확률이 8~9 가 될 확률보다 높음.

의사 랜덤

- C++의 새로운 라이브러리 이용

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

int main() {
    // OS가 제공하는 비결정적 난수 소스
    random_device rd;
    mt19937 gen(rd());   // 이걸로 시드를 줌

    // 1부터 10까지 균등분포 정의(균등한 확률로 뽑기)
    uniform_int_distribution<int> dis(1, 10);

    for (int i = 0; i < 10; i++) {
        printf("%d ", dis());
    }
}
```

- 범위 난수

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

random_device rd;
mt19937 gen(rd());
int range_rand(int a, int b) {
    uniform_int_distribution<int> dis(a, b);
    return dis(gen);
}

int main(void) {
    int cnt[12]={0, };
    for(int i=0; i<10000; i++)
        cnt[range_rand(1, 10)]++; // [1, 10]

    for(int i=0; i<=11; i++) {
        printf("%2d: %5d\n", i, cnt[i]);
    }
}
```

Up & Down 숫자 맞추기 게임

- 이 게임은?
  - 컴퓨터가 지정된 범위 안에서 임의 숫자 하나 r을 뽑는다.
  - 사람은 이 숫자를 감으로 맞추어야 한다.
  - 사람이 숫자 p를 입력하면 컴퓨터는 딱 두 종류의 힌트만 준다.
    - r>p 이면: UP
    - r<p 이면: DOWN
  - 이 게임은 이론적으로 $\log_2$경우의수 안에 맞출 수 있다.

- 정답 r이 103인 경우 예시

```
[ 1] Guess num: 500
DOWN
[ 2] Guess num: 250
DOWN
[ 3] Guess num: 120
DOWN
[ 4] Guess num: 60
UP
[ 5] Guess num: 90
UP
[ 6] Guess num: 100
UP
[ 7] Guess num: 110
DOWN
[ 8] Guess num: 105
DOWN
[ 9] Guess num: 103
DOWN
Good job!!
```

https://gifted.datahub.pe.kr/src/gifted/random/guess_num_game.cpp

Up & Down 숫자 맞추기 게임

- 예시 프로그램

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

random_device rd;
mt19937 gen(rd());

int range_rand(int a, int b) {
    uniform_int_distribution<int> dis(a, b);
    return dis(gen);
}
```

```
int main() {
    int r = range_rand(100, 999);  // 100 ~ 999
    int limit = 10;

    int p; //게이머가 입력한 수
    // cnt가 1부터 limit이하인 동안 반복 {
    // [ ] Guess num: 이라 찍고
    // 숫자 p변수에 입력 받음

    // 정답이면 반복문 탈출
    // 정답보다 작은수를 찍었으면 UP 이라고 출력,
    // 정답보다 큰 수를 찍었으면 DOWN이라고 출력


    if(r==p)
        printf("\nGood job!\n");
    else
        printf("\nYou failed!\nAnser is %d\n", r);
}
```

- https://gifted.datahub.pe.kr/src/gifted/random/guess_num_game_update.cpp

Up & Down 숫자 맞추기 게임

```
input Game difficulty(2~9): 3
[100, 999] limit: 10
```

- 난이도 설정 업데이트

```
#include <stdio.h>
#include <random>
#include <math.h>
#define  logB(base, x)  log(x)/log(base)
using namespace std;

random_device rd;
mt19937 gen(rd());
int range_rand(int a, int b) {
    uniform_int_distribution<int> dis(a, b);
    return dis(gen);
}

int main() {
    int d;
    printf("input Game difficulty(2~9): ");
    scanf("%d", &d);
```

$$\frac{\log_c b}{\log_c a} = \log_a b \,(c > 0, c \neq 1)$$

```
    int min = pow(10, d-1);
    int max = pow(10, d)-1;
    int limit = logB(2, max-min)+1;
    printf("[%d, %d] limit: %d\n\n", min, max, limit);

    int r = range_rand(min, max);
    int p;
    for(int cnt=1; cnt<=limit; cnt++) {
        printf("[%2d] Guess num: ", cnt);
        scanf("%d", &p);

        if(r == p)   break;
        else if(r>p) printf("UP\n");
        else         printf("DOWN\n");
    }
    if(r==p)
        printf("\nGood job!\n");
    else
        printf("\nYou failed!\nAnser is %d\n", r);
}
```

숫자 야구

- 숫자 야구란?
  - 원제는 Bulls and Cows
  - 본질은 숫자 맞추기 게임
  - 사용되는 숫자는 0에서 9까지 서로 다른 숫자이다.
  - 야구의 정규 이닝이 9회까지 밖에 없기 때문에 기회를 9회로 제한한다.
  - 출제자는 3자리의 숫자를 임의로 정한다.
  - 맞추는 사람은 숫자를 불러서 결과를 확인하고 그 결과를 토대로 다른 숫자로 다음 시도를 진행한다.
  - 숫자는 맞지만 위치가 틀렸을 때는 볼.
  - 숫자와 위치가 전부 맞으면 스트라이크.
  - 숫자와 위치가 전부 틀리면 아웃
  - 물론 무엇이 볼이고 스트라이크인지는 알려주지 않는다.
- 예시 (0 7 6 이 정답인 경우)

1	4	8	5	아웃
2	3	1	0	0S 1B
3	2	0	6	1S 1B
4	0	6	7	1S 2B
5	0	7	6	3S 0B

숫자 야구

- 기본설계
  - 게임 출력 예시

```
[ 1] Guess num: 1 2 3   S: 1, B: 0
[ 2] Guess num: 4 5 6   S: 0, B: 1
[ 3] Guess num: 1 4 5   S: 0, B: 1
[ 4] Guess num: 2 5 6   S: 0, B: 1
[ 5] Guess num: 5 2 7   S: 1, B: 1
[ 6] Guess num: 5 8 2   S: 1, B: 0
[ 7] Guess num: 5 7 9   S: 2, B: 0
[ 8] Guess num: 5 7 3
Good job!
```

```c
#define LIMIT  9

int main(void) {
    int rand[3];    // 컴퓨터가 찍은 숫자
    int pick[3];    // 입력한 숫자
    int strike=0, ball=0;
    uniq_rand_3num(rand);   // 컴퓨터가 숫자 세 개 찍기

    int cnt=1;
    while(cnt <= LIMIT) {
        printf("[%2d] Guess num: ", cnt);
        get_3num(pick);   // 사용자에게 숫자 세 개 입력 받기

        strike = count_strike(rand, pick); // 스트라이크 세기
        ball   = count_ball(rand, pick);   // 볼 카운트 세기

        if(strike==3)
            break;
        else if(strike==0 && ball==0)
            printf("\tOut\n");
        else
            printf("\tS: %d, B: %d\n", strike, ball);
        cnt++;
    }
}
```

- https://gifted.datahub.pe.kr/src/gifted/random/baseball_game1.cpp

```
#include <stdio.h>
#include <time.h>
#include <random>
#include <conio.h>
#include <ctype.h>
#define  LIMIT  9
using namespace std;

mt19937 gen(time(NULL));

int range_rand(int a, int b) {
    uniform_int_distribution<int> dis(a, b);
    return dis(gen);
}
// rand 배열에 난수 3개 채우기(겹치지 않는 난수는 옵션)
void uniq_rand_3num(int rand[]) {









    //printf("%d %d %d\n", rand[0], rand[1], rand[2]);
}
```

```
// 숫자 3개 입력 받기, 숫자가 아닌 문자가 입력되면 무시
// 이미 입력된 숫자
void get_3num(int pick[]) {









}

int count_strike(int rand[], int pick[]) {
    int strike=0;
    for(int i=0; i<3; i++) {
        if(rand[i]==pick[i]) {
            strike++;
            // strike로 판정한 숫자를
            // ball 카운트 할 때 또 다시 세는 것을 예방
            pick[i]=-1;
        }
    }
    return strike;
}
```

답: 2 3 2
입력: 2 2 1

- https://gifted.datahub.pe.kr/src/gifted/random/baseball_game1.cpp

```
int count_ball(int rand[], int pick[]) {
    int ball=0;









    return ball;
}

int main(void) {
    int rand[3];    // 컴퓨터가 찍은숫자
    int pick[3];    // 입력한 숫자
    int strike=0, ball=0;
    uniq_rand_3num(rand);   // 컴퓨터가 숫자 세 개 찍기

    int cnt=1;
    while(cnt <= LIMIT) {
        printf("[%2d] Guess num: ", cnt);
        get_3num(pick);     // 사용자에게 숫자 세 개 입력 받기
```

```
        strike = count_strike(rand, pick);
        ball   = count_ball(rand, pick);

        if(strike==3)
            break;
        else if(strike==0 && ball==0)
            printf("\tOut\n");
        else
            printf("\tS: %d, B: %d\n", strike, ball);

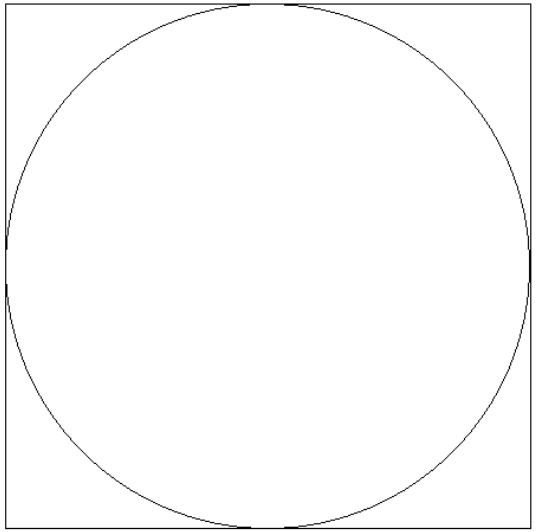
        cnt++;
    }

    if(strike >= 3) {
        if(cnt <= 2)      puts("\nIt's miracle!!!");
        else if(cnt <= 5) puts("\nPerfect!!!");
        else if(cnt <= 9) puts("\nGood job!");
    }
    else {
        printf("\nRetry again\nRight anser is ");
        printf("%d %d %d\n", rand[0], rand[1], rand[2]);
    }
    return 0;
}
```

몬테카를로 알고리즘

- 개요
  - 몬테카를로 알고리즘은 폴란드계 미국인 수학자 스타니스와프 울람이 제안한 알고리즘
  - 이 알고리즘은 원하는 결과값의 정확한 값을 얻는 방법이 아니고, 난수를 이용하여 어떤 함수의 답을 확률적으로 근접하게 계산하는 방식

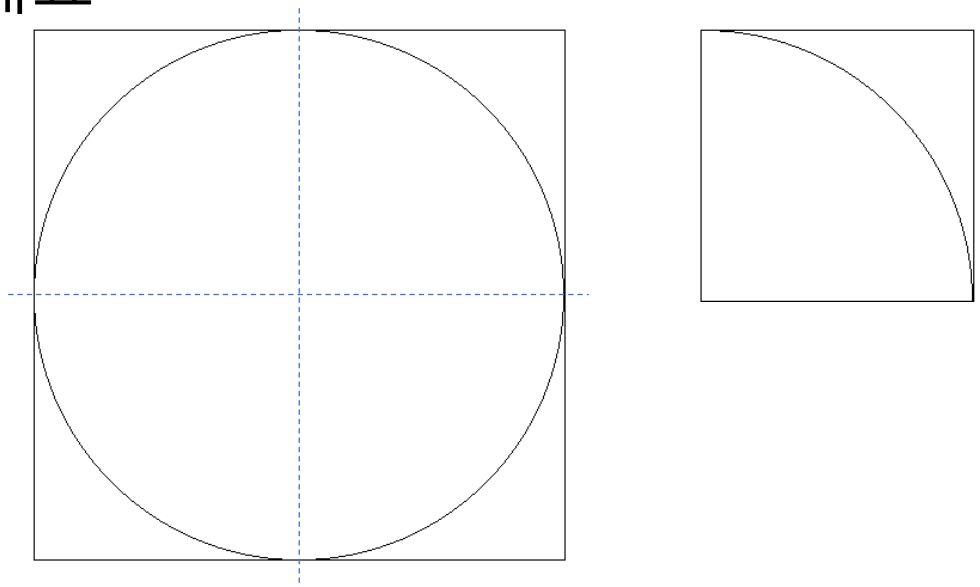
- 원주율 구하기 문제에 적용
  - [2 x PI x 반지름 = 원의 둘레] 이므로,
  - PI = 원의 둘레/(2 x 반지름)



  - 정확한 원의 둘레의 길이를 구할 수 없으므로 이 식을 가지고는 원하는 PI값을 구할 수 없다.

몬테카를로 원주율 시뮬레이션

- 개요



- 반지름이 r인 원과 원을 둘러싼 정사각형을 4등분하면 반지름이 r인 정사각형과 그 안에 원의 4등분 된 부채꼴이 만들어 진다.

- 원의 넓이는 π x r x r 이므로 이 부채꼴의 넓이는 $\pi r^2/4$
- 이때 이 도형을 향해 다트를 던진다고 상상해보면 부채꼴 내부에 다트가 맞을 확률은 $\pi r^2 / 4r^2 = \pi/4$
- 즉 100개의 다트를 던졌을 때 이 중 80개의 다트가 부채꼴 내부에 맞았다면, $\pi/4 = 80/100$ 이므로 $\pi = 3.2$ 이다.
- 이제 다트 n개를 던졌다고 가정하고, 이 중 m개의 다트가 부채꼴 내부에 맞았다고 일반화를 하면 ,
- $\pi/4 = m/n \rightarrow \pi = m/n*4$

몬테카를로 원주율 시뮬레이션 (C버전)

```
#include <stdio.h>
#include <time.h>
#include <random>
#define  MAX  100000000
using namespace std;

mt19937 gen(time(NULL));
uniform_real_distribution<double> dis(0, 1);

int main() {
    int in_cnt=0, tot_cnt=0;

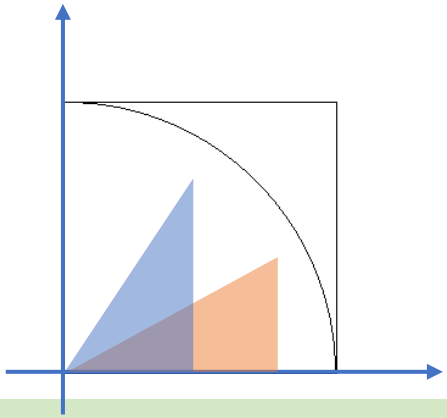
    for(int i=0; i<MAX; i++) {
        double x = dis(gen);
        double y = dis(gen);
        tot_cnt++;

        if(x*x + y*y <= 1)
            in_cnt++;

        if(tot_cnt % 10000000 == 0) {
            double pi = (double)in_cnt/tot_cnt*4;
            printf("[%9d tested] pi: %lf\n", tot_cnt, pi);
        }
    }
}
```

```
[ 10000000 tested] pi: 3.142229
[ 20000000 tested] pi: 3.142092
[ 30000000 tested] pi: 3.142001
[ 40000000 tested] pi: 3.141867
[ 50000000 tested] pi: 3.141779
[ 60000000 tested] pi: 3.141840
[ 70000000 tested] pi: 3.141799
[ 80000000 tested] pi: 3.141683
[ 90000000 tested] pi: 3.141558
[100000000 tested] pi: 3.141551

Process returned 0 (0x0)   execution time : 11.260 s
Press any key to continue.
```



몬테카를로 원주율 시뮬레이션

p5* File Edit Sketch Help

Auto-refresh Golden almandine

Sketch Files

index.html
sketch.js
style.css

sketch.js

```
function setup() {
  createCanvas(340, 360);
}

let m=10;   // 여백
let r=300;  // 반지름
let v=r*r/100;  // draw() 당 실험 갯수
let t=0;  // 모든 점 개수
let c=0;  // 부채꼴 안 점 개수
// 램덤 점 좌표 저장 배열
let d=Array.from(Array(r), () => new Array(r).fill(0))

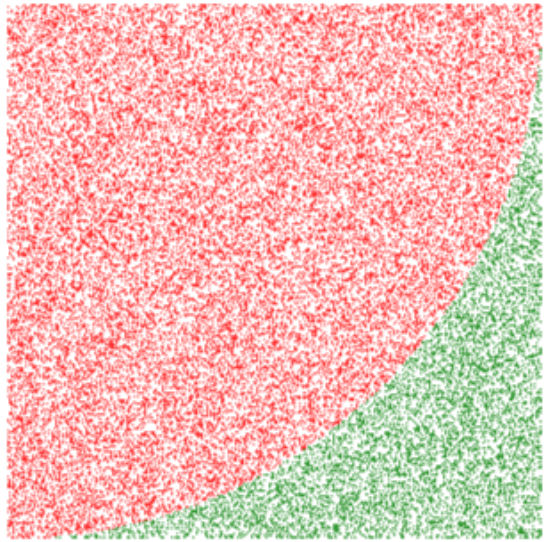
let xt, yt, PI_t;

function draw() {
  background(255);
  fill(0);

  for(var i=1; i<=v; i++) {
    xt = Math.floor(Math.random()*r);
    yt = Math.floor(Math.random()*r);
    t++;  // 점 개수 증가

    if(xt*xt + yt*yt < r*r) {  // 부채꼴 안쪽인가?
      d[xt][yt]=1;
      c++;  // 부채꼴 안쪽 점 증가
```

Preview



Total=39600 Count=31178

Monte Carlo PI: 3.1492929292929293

https://gifted.datahub.pe.kr/src/gifted/random/pi_simul.js.cpp

몬테카를로 원주율 시뮬레이션

```
// p5.js
function setup() {
  createCanvas(340, 360);
  background(255);
}

let m=10;       // 여백
let r=300;      // 반지름
let v=r*r/100;  // draw() 당 실험 갯수
let t=0;        // 모든 점 개수
let c=0;        // 부채꼴 안 점 개수
let xr, yr, PI_t;

function draw() {
  for(var i=1; i<=v; i++) {
    // 판정은 반드시 "연속값"(0~1 사이 실수)으로 해야 함.
    // Math.floor로 정수 좌표를 먼저 만들고 나서 판정하면
    // 원의 경계 부분에서 정수 격자 칸의 왼쪽 아래 꼭짓점만 검사하게
    // 되어 원 안쪽으로 과다 계산되는 "구조적 편향"이 생김
    xr = Math.random();
    yr = Math.random();
    t++;  // 점 개수 증가
    // 부채꼴 안쪽인가? (반지름 1인 단위원 기준으로 판정)
    if(xr*xr + yr*yr < 1) {
      stroke('#ff0000');
      c++;  // 부채꼴 안쪽 점 증가
    }
    else {
      stroke('#008000');
    }

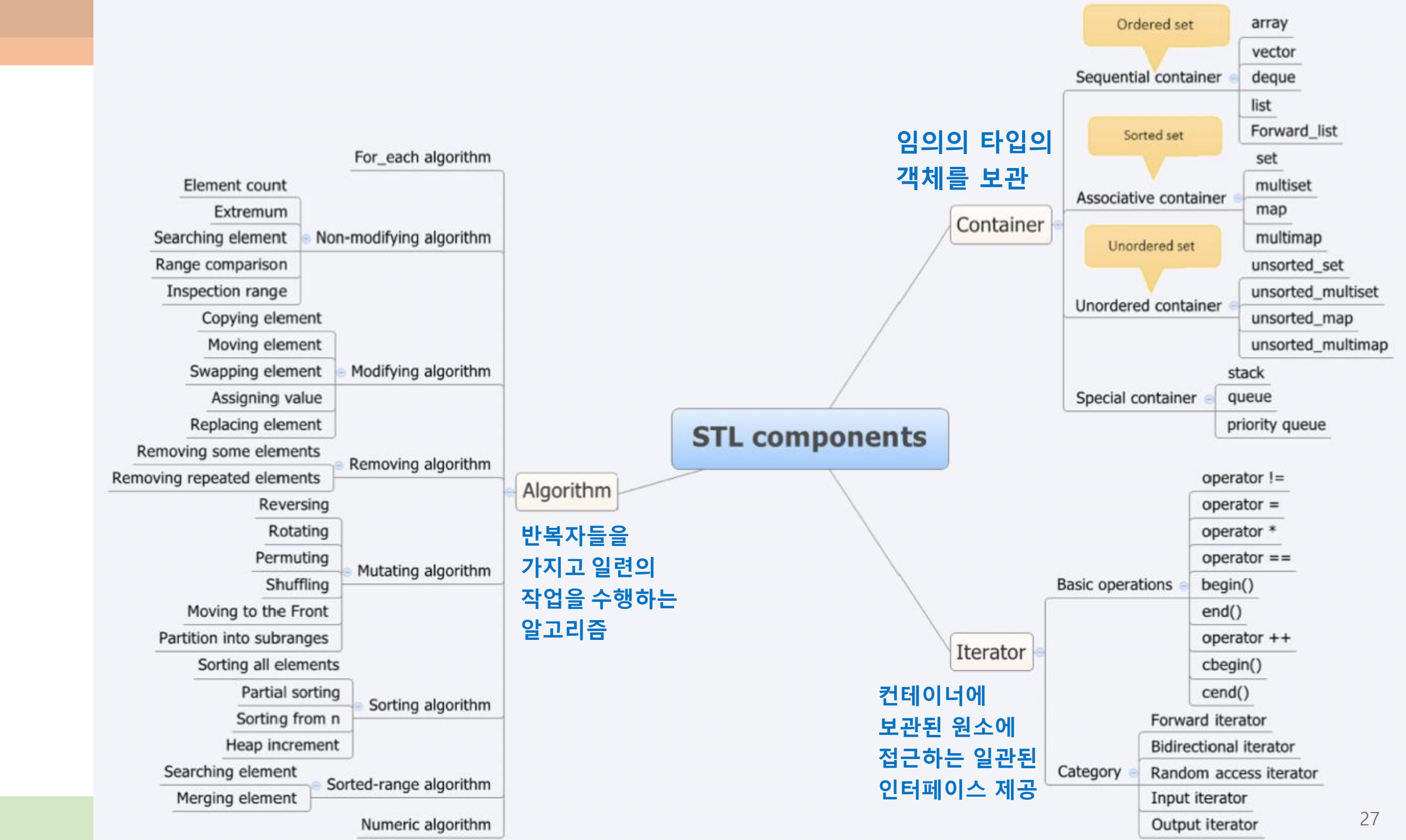
    // 화면에 찍을 때만 정수 픽셀 좌표로 변환
    point(m + Math.floor(xr*r), m + Math.floor(yr*r));
  }

  // 텍스트 영역만 지우고 다시 그리기(숫자가 겹쳐 보이지 않도록)
  fill(255);
  noStroke();
  rect(0, m+r+5, width, 40);
  fill(0);

  text("Total="+t, m, m+r+15);
  text("Count="+c, m+100, m+r+15);
  PI_t = c/t*4;
  text("Monte Carlo PI: " + PI_t, m, m+r+35);
}
```

STL

(Standard Template Library)


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

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 (iterator)

- 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 동작 시키는데 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()

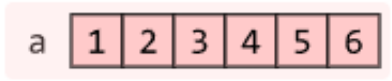
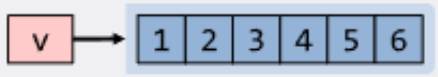
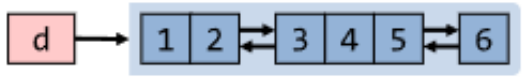
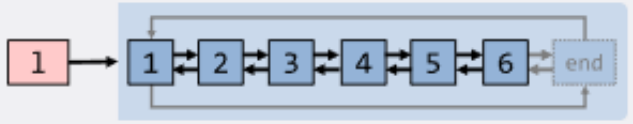
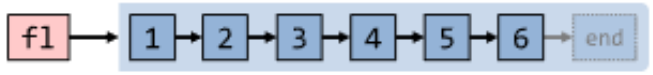
수치 알고리즘	STL에 직접 속하지 않고 C++라이브러리로 분류되는 알고리즘으로 수치적 해석을 위해 사용	#include <numeric>
		accumulate()

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, oper...

```
#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(7) = 678;  // 0-base 7th 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';
}
```

```
인덱스 2 에 위치한 원소 : 4
data 배열의 크기 = 6
예외 발생 : array::at: __n (which is 7) >= _Nm (which is 6)
data: 1 88 4 5 5 6
```

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
```

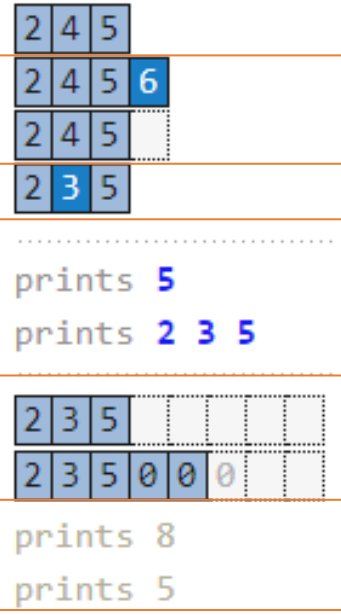
vector container

- 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

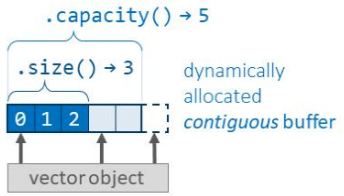
Construct A New Vector Object

```
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

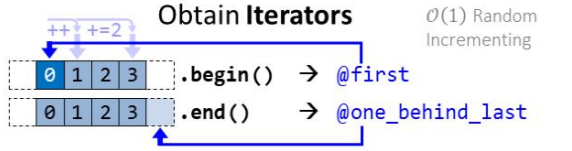
```
vector w {7,4,2};   // vector<int>
```

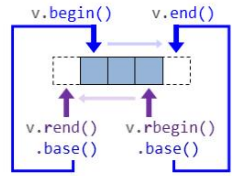
Typical Memory Layout



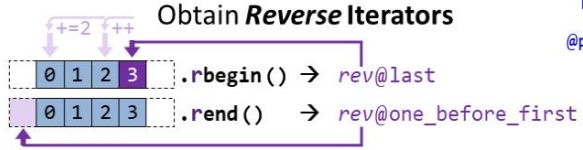
Obtain Iterators

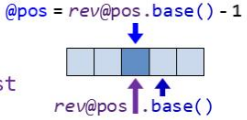
$\mathcal{O}(1)$ Random Incrementing





Obtain *Reverse* Iterators



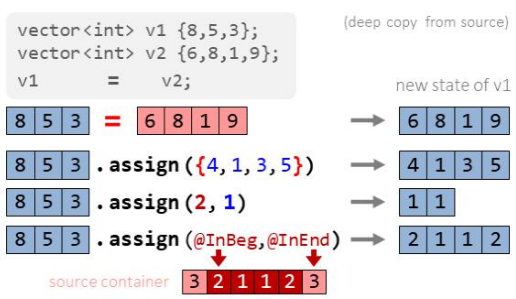


Assign New Content To An Existing Vector

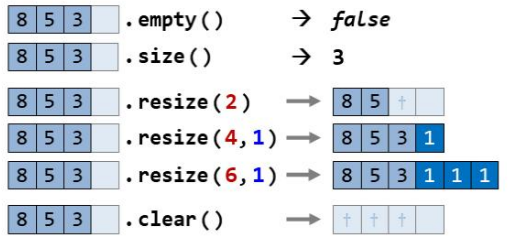
```
vector<int> v1 {8,5,3};
vector<int> v2 {6,8,1,9};
v1     =    v2;
```

(deep copy from source)

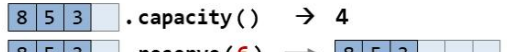
new state of v1



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!

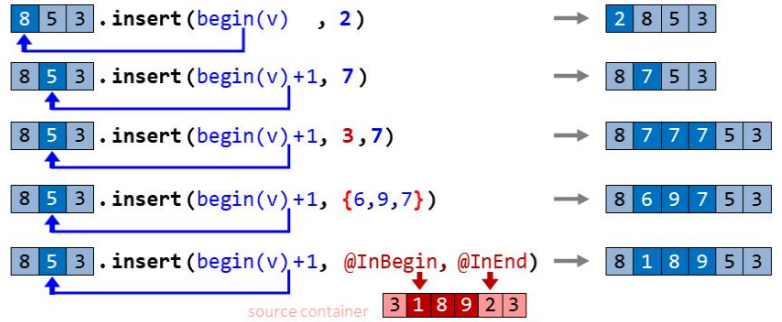
Append Elements

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

Insert Elements at Arbitrary Positions

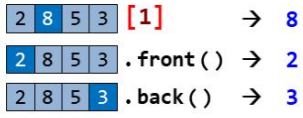
$\mathcal{O}(n)$ Worst Case

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

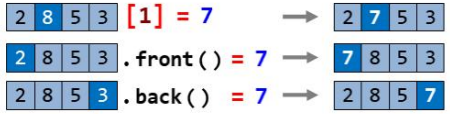


Get Element Values

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



Change Element Values



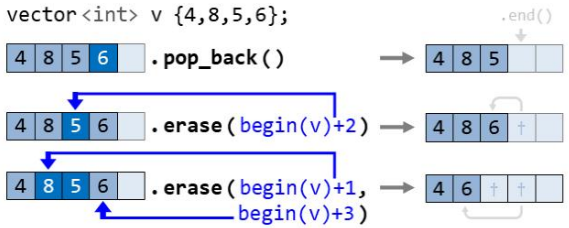
Out of Bounds Access



Erase Elements

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

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



Shrink The Capacity

(might be inefficient)

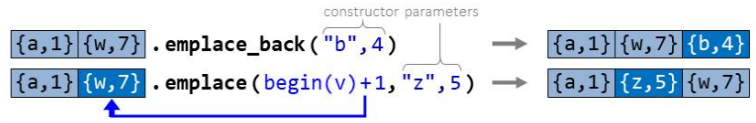
Erasing, resizing or clearing will not shrink the capacity!

```
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}};
```

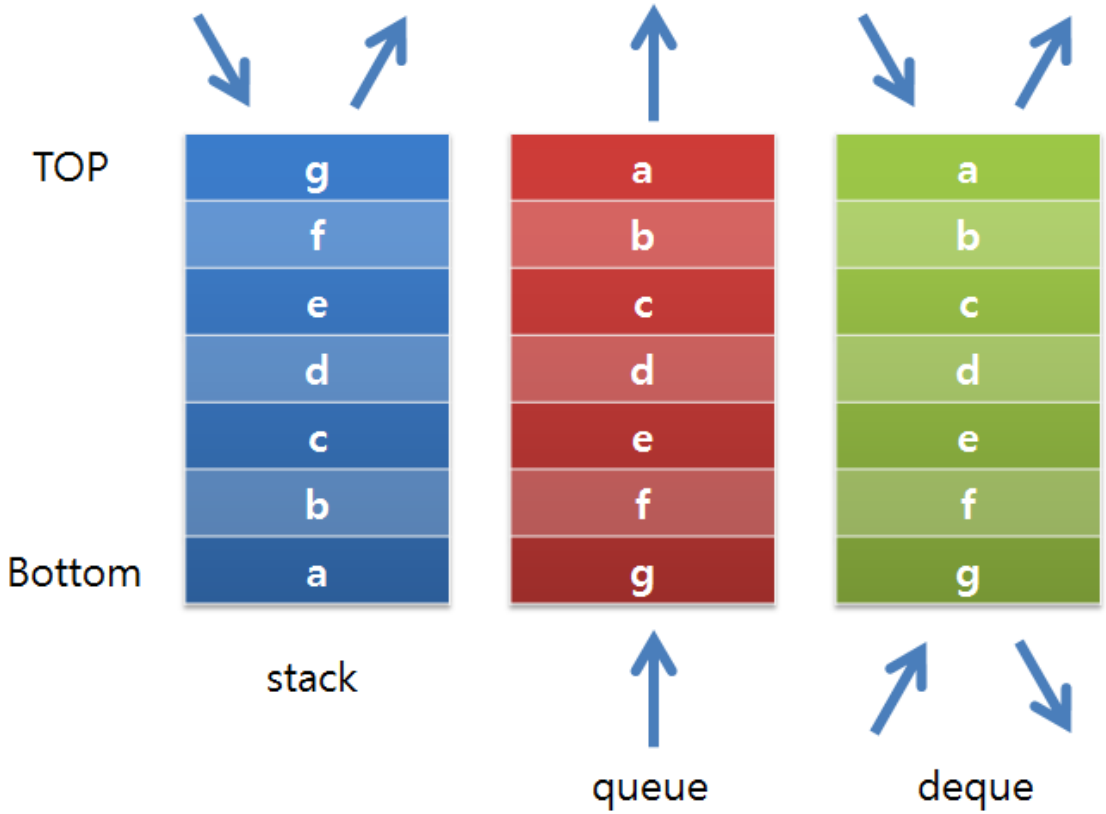


v1.1

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

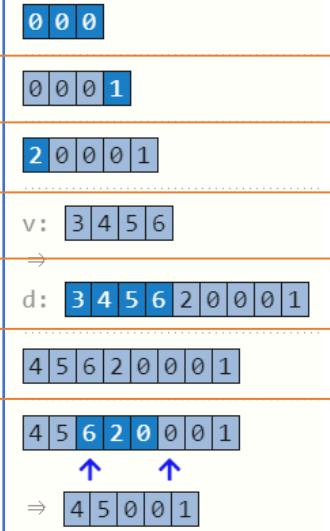
- deque의 사용 예시

```
#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)
```

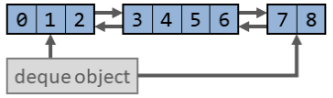
C++17 value type deducable from argument type

```
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



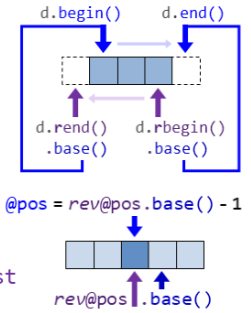
Obtain **Iterators**

$\mathcal{O}(1)$ 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
                   Invalid Index!
2 8 5 3 .at(6)   → Throws Exception
                   std::out_of_range
```

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}};
{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}
```

constructor parameters

v1.1

C++ 템플릿 인스턴스화(Template Instantiation)

- 예제 코드

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

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;
}
```

1. 흔한 오해

> "템플릿 함수는 하나만 컴파일되고, 모든 타입이 그 하나의 함수를 공유해서 쓴다"

틀렸습니다. 실제로는:

> 템플릿은 함수를 찍어내는 **틀(붕어빵 틀)**이고, 실제 함수(붕어빵)는 **호출되는 타입마다 컴파일 시점에 각각 새로 만들어진다.**

이 과정을 **템플릿 인스턴스화(template instantiation)**라고 부릅니다.

```
int main() {
    deque<int> di {1, 2, 3};
    deque<string> ds {"a", "b"};
    deque<double> dd {1.1, 2.2};

    print_deque(di);   // T = int 로 호출
    print_deque(ds);   // T = string 로 호출
    print_deque(dd);   // T = double 로 호출
}
```

```
[ 1 2 3 ]
[ a b ]
[ 1.1 2.2 ]
```

C++ 템플릿 인스턴스화(Template Instantiation)

▪ 컴파일된 실행 파일 뜯어보기

리눅스에서는 `nm` 명령어로 실행 파일 안에 어떤 함수(심볼)들이 실제로 들어있는지 확인할 수 있습니다.

```bash
g++ -std=c++17 -o demo demo.cpp
nm -C demo | grep print_deque
```

실제 출력 결과

```
0000000000002e53 W void print_deque<int>(std::deque<int,
std::allocator<int> >&)
0000000000002f87 W void
print_deque<std::__cxx11::basic_string<...> >
(std::deque<std::__cxx11::basic_string<...>, ...>&)
00000000000030ba W void print_deque<double>(std::deque<double,
std::allocator<double> >&)
```

핵심 관찰: `print_deque` 라는 이름의 함수가 실행 파일 안에 **3개** 들어있습니다!

순번	심볼 이름	메모리 주소
1	`print_deque<int>(...)`	`0x2e53`
2	`print_deque<string>(...)`	`0x2f87`
3	`print_deque<double>(...)`	`0x30ba`

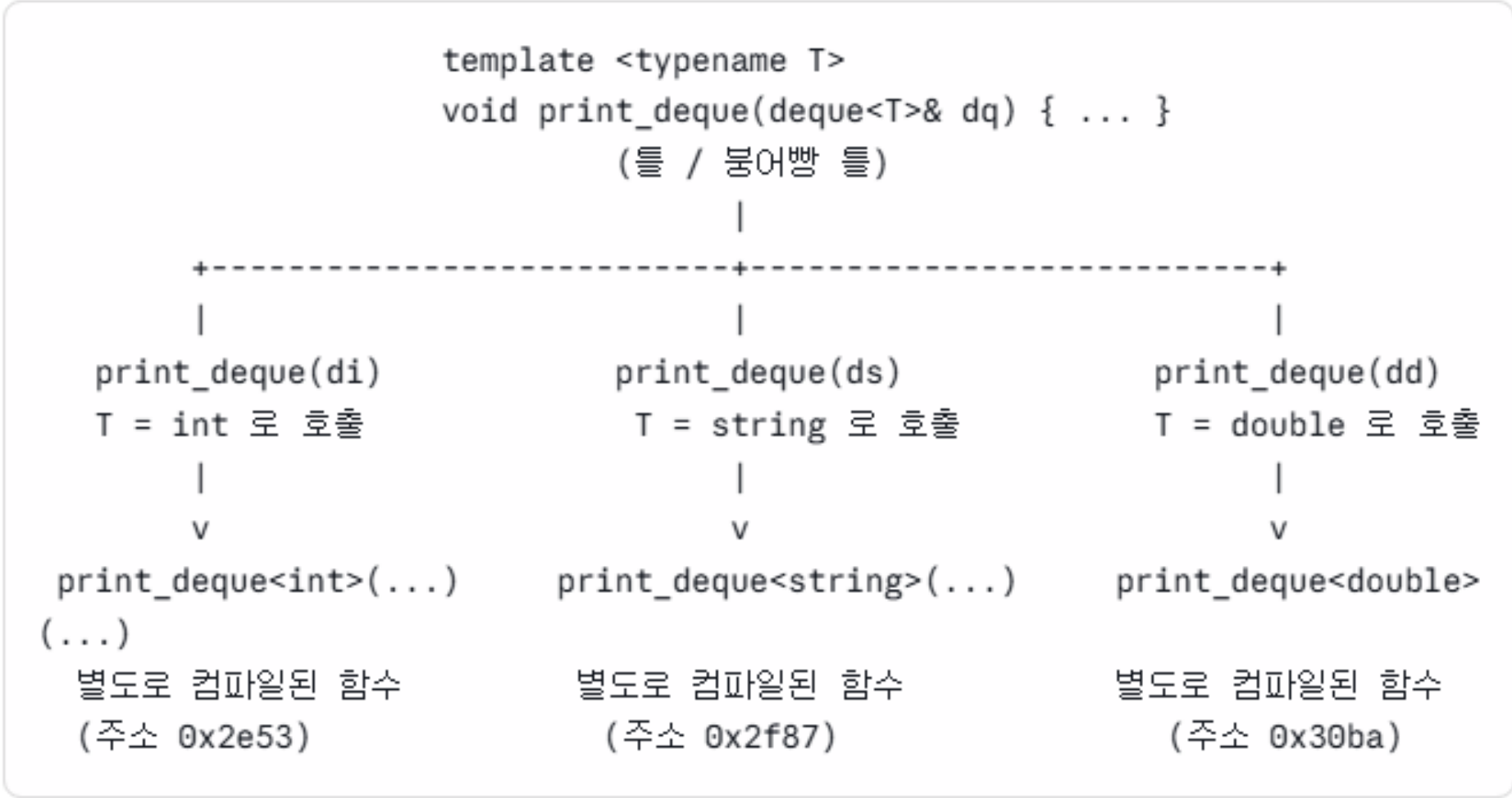
셋 다 이름은 `print_deque` 지만, 컴파일러가 붙인 **완전히 다른 심볼**이고 **서로 다른 주소에 위치한 별개의 기계어 코드**입니다. 코드를 한 번만 작성했지만, 컴파일러는 마치 우리가 이렇게 세 함수를 직접 작성한 것처럼 처리한 것입니다:

```cpp
// 컴파일러가 자동으로 "찍어낸" 버전들 (개념적 이해용)
void print_deque(deque<int>& dq) { deque<int>::iterator itr; ...
void print_deque(deque<string>& dq) { deque<string>::iterator it
void print_deque(deque<double>& dq) { deque<double>::iterator it
```

> 만약 `deque<float>` 로는 한 번도 호출하지 않았다면,
> `print_deque<float>` 는 **아예 만들어지지도 않습니다.** 템플릿은 "쓰인 만큼만" 인스턴스화됩니다.

C++ 템플릿 인스턴스화(Template Instantiation)

- 정리된 다이어그램

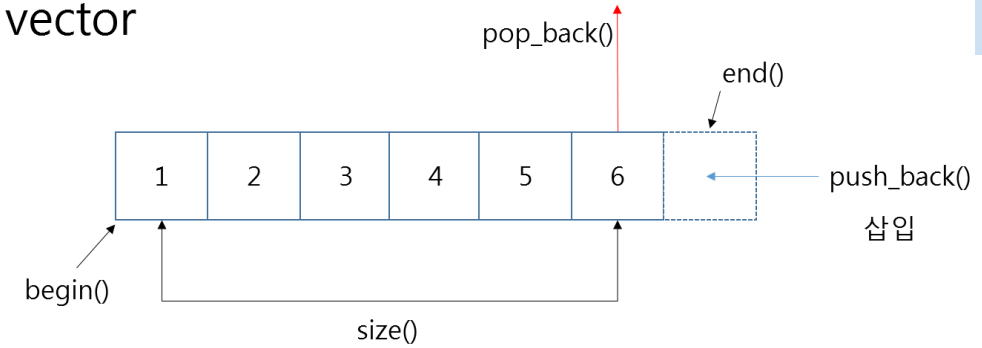


STL iterator

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(int 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;
}

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

```
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(ary));
    cout << "initial vector sequence\n";
    print_elements(vec.begin(), vec.end());

    cout << "vector sort\n";
    sort(vec.begin(), vec.end());
    print_elements(vec.begin(), vec.end());

    cout << "vector sort by asc_order()\n";
    sort(vec.begin(), vec.end(), asc_order);
    print_elements(vec.begin(), vec.end());

    cout << "deque sort by desc_order()\n";
    sort(vec.begin(), vec.end(), desc_order);
    print_elements(vec.begin(), vec.end());
}
```

```
initial array sequence
[ 5 3 1 2 4 3 7 ]
initial vector sequence
[ 5 3 1 2 4 3 7 ]
vector sort
[ 1 2 3 3 4 5 7 ]
vector sort by asc_order()
[ 1 2 3 3 4 5 7 ]
deque sort by desc_order()
[ 7 5 4 3 3 2 1 ]
```

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(ary));
    cout << "initial vector sequence\n";
    print_elements(vec.begin(), vec.end());

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

    cout << "vector sort by reverse iterator\n";
    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";
    //파란부분 실제호출: operator<<(cout, *begin)
    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
    }
};
 :
```

```
  :
  :

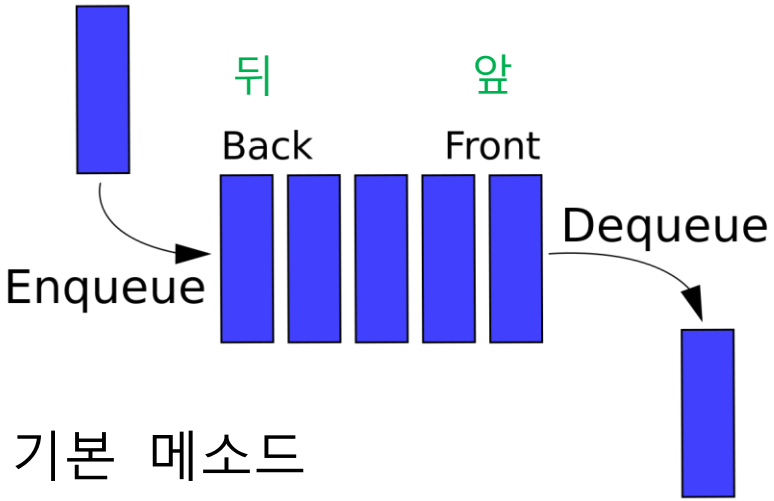
int main() {
    vector<noco> nc;
    nc.push_back({1, 90});
    nc.push_back({2, 80});
    nc.push_back({3, 70});
    nc.push_back({4, 60});
    nc.push_back({3, 75});
    nc.push_back({3, 85});
    nc.push_back({4, 85});
    nc.push_back({4, 65});

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

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

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();  // 원소 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)

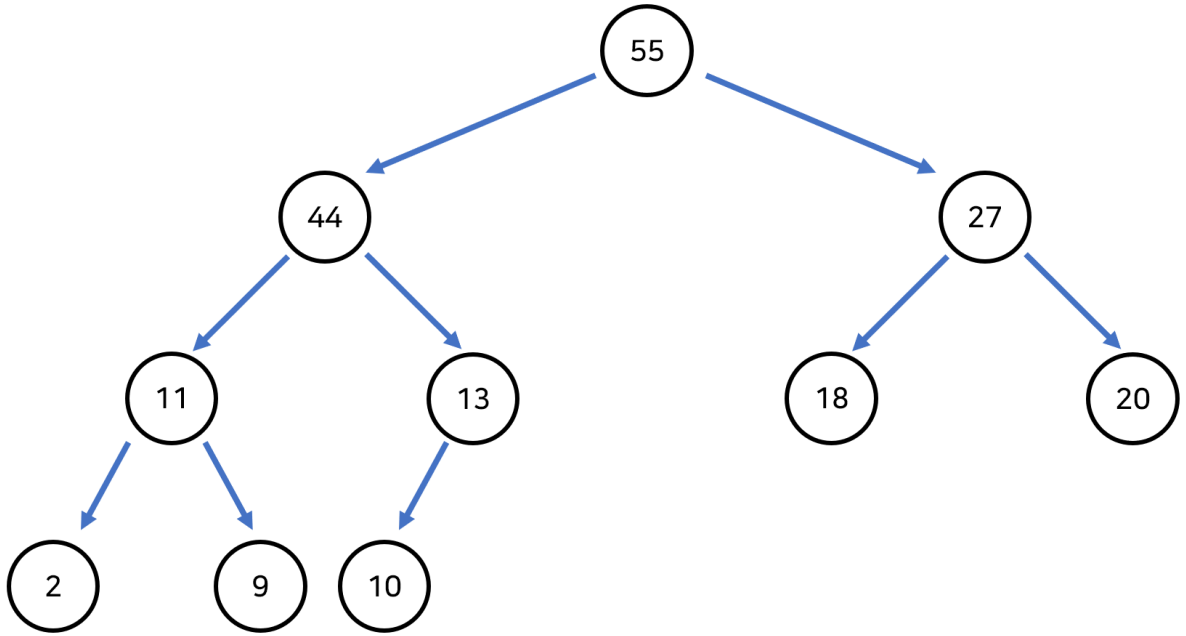
- 정의
  - 영단어 힙(heap)은 ' 무엇인가를 차곡차곡 쌓아올린 더미 ' 라는 뜻
  - 힙은 완전 이진 트리*로 구현된 자료구조
  - 가장 크거나 작은 값을 찾아내는 연산을 빠르게 하기위해 고안된 자료구조

- 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

***완전 이진트리**(complete binary tree)란?
마지막 레벨을 제외하고 모든 레벨이 완전히 채워져 있는 트리이다.

힙(heap)

- 최대힙의 예

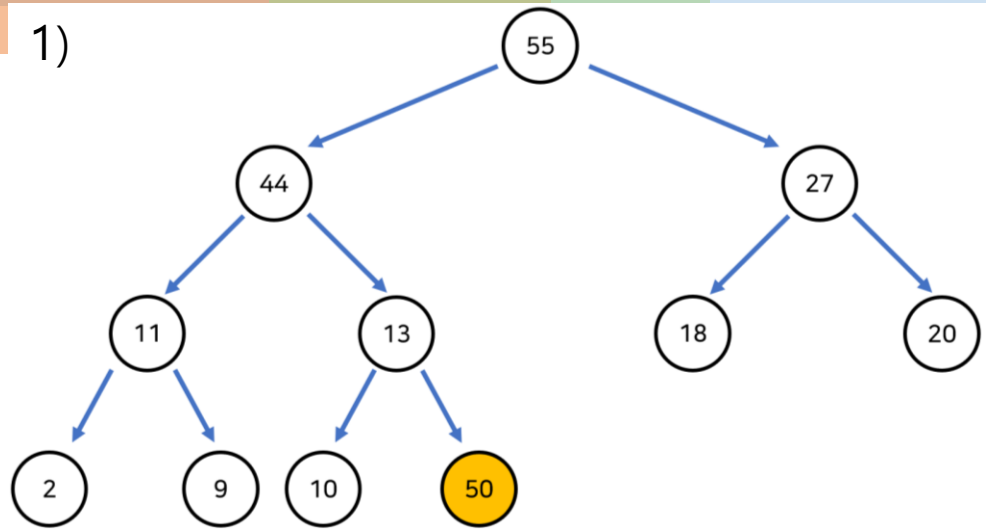


- Root의 값이 가장 크다.
- 또한 모든 서브트리에 대해서도 같다.

- 자료의 추가

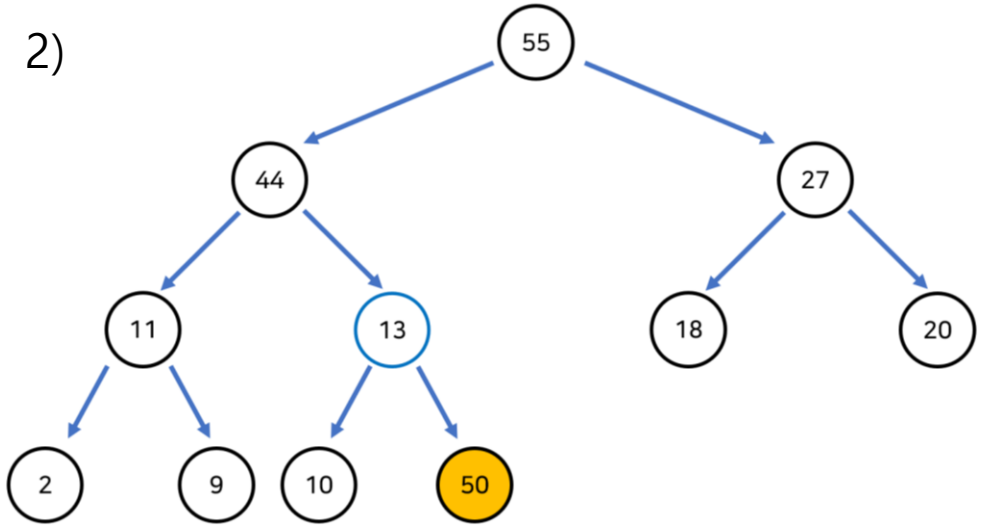
1) 새로운 노드를 트리의 맨 뒤에 추가한다. (완전 이진 트리의 형태를 깨면 안됨)
2) 추가된 노드와 부모 노드를 비교하여 자식 노드가 크다면 서로의 위치를 바꾼다.
3) 2번 작업을 부모 노드가 더 클 때까지 반복한다.

1)



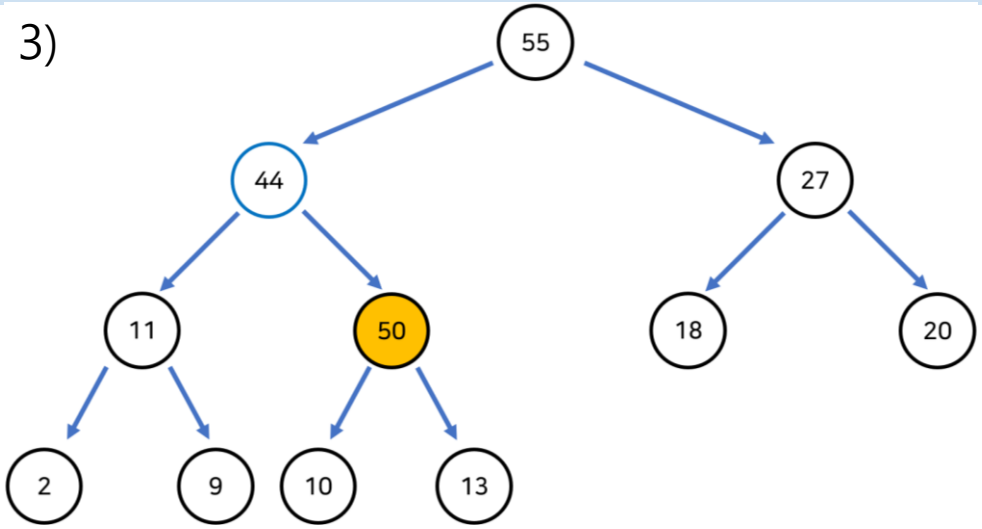
MaxHeap에 50의 값을 가진 노드를 추가하려는 상황이다. 그러면 이 노드는 트리의 맨 뒤인 13의 rightChild로 들어가게 된다.

2)



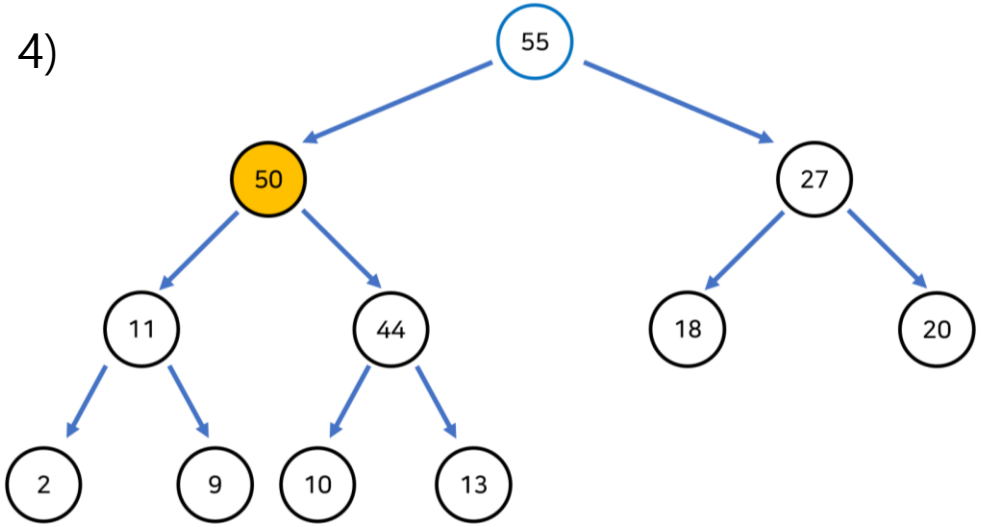
50의 부모노드인 13과 비교한다.
50이 크므로 둘의 자리를 바꿔준다.

3)



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

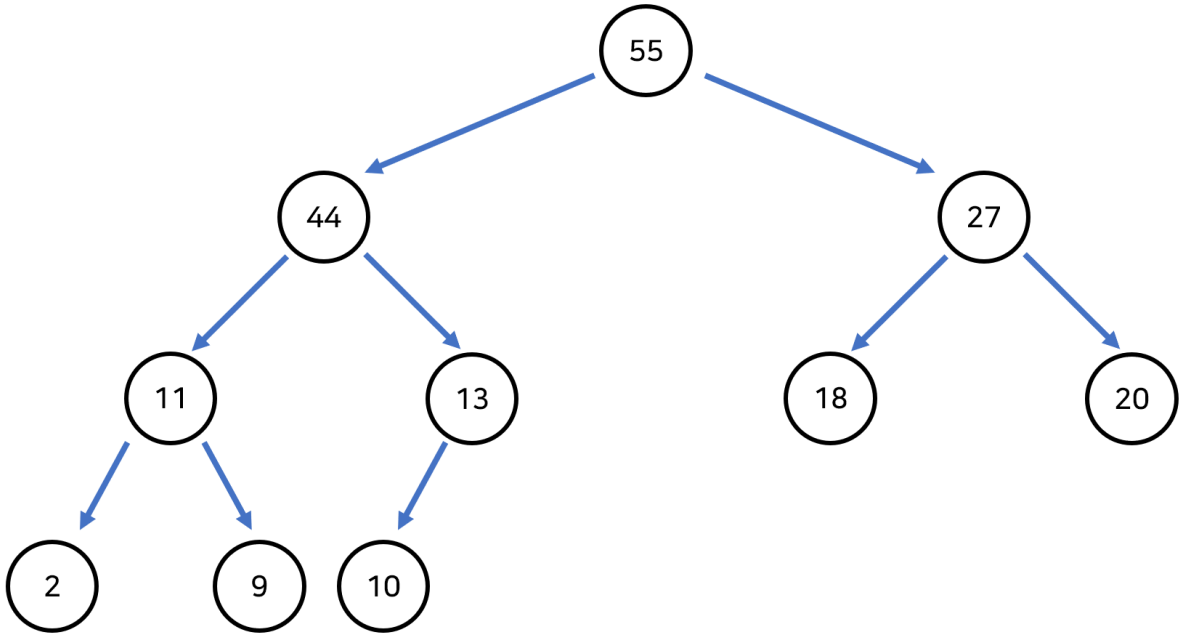
4)



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

힙(heap)

- 자료의 삭제

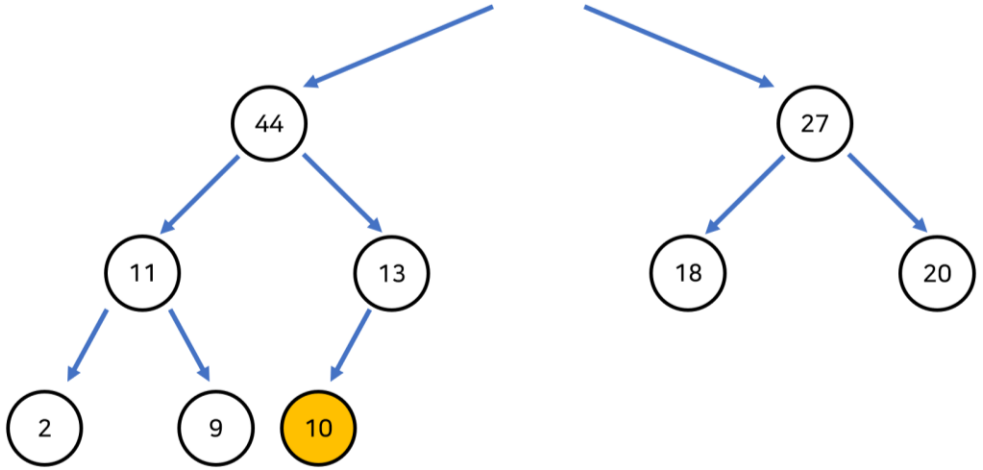


- 자료가 삭제되는 경우는 맨 위에 있는 Root노드가 빠지는 경우밖에 없다.
- 그렇게되면 다시 힙의 형태를 갖추어야 …

- 삭제 알고리즘

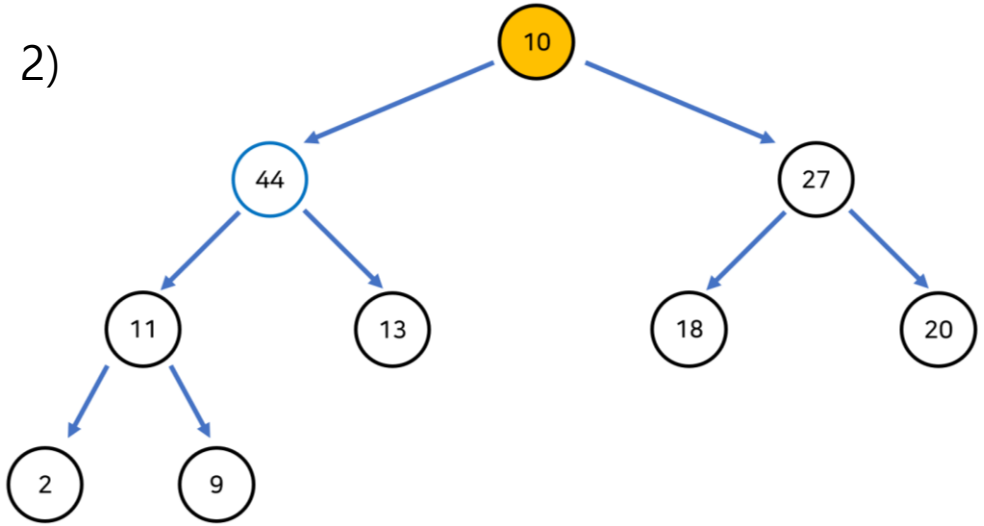
1) 맨 뒤에 있는 노드를 Root자리로 옮긴다.
2) 자식 노드 중 값이 더 큰 노드와 비교하여 자식 노드가 값이 더 크다면 위치를 바꾼다.
3) 2번의 작업을 자식 노드보다 자신이 클 때까지 반복한다.

1)



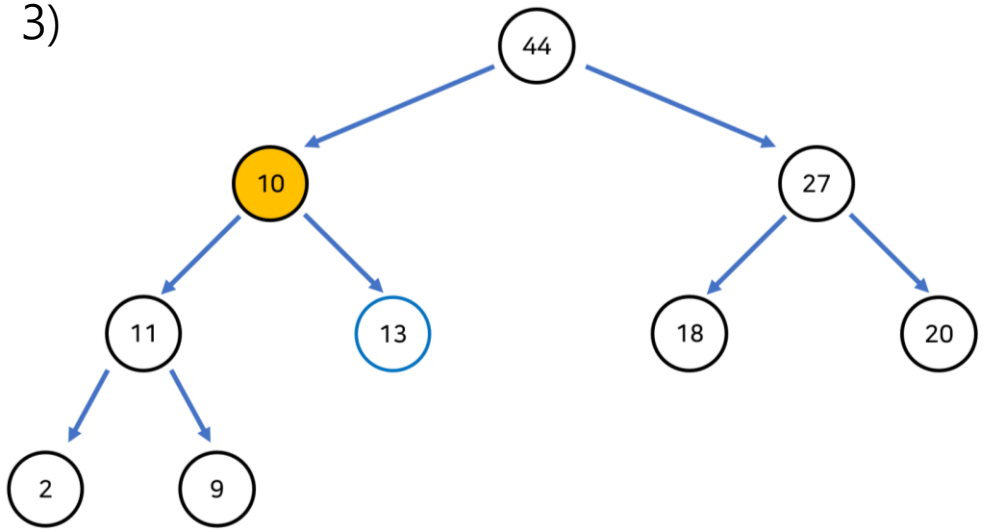
Root노드가 빠진 상황이다. 이렇게 되면 맨뒤에 있던 노드인 10을 Root로 올린다.

2)



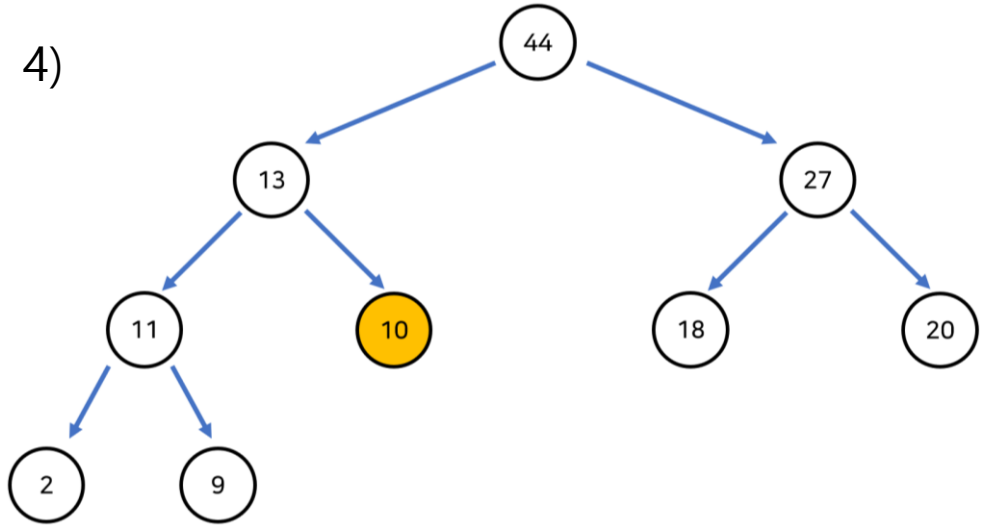
Root로 올리고 자식들과 비교를 시작하게 되는데, 자식 둘 중에 44가 더 크므로 44와 비교를 한다. 그런데 44가 10보다 크므로 44와 자리를 바꾼다.

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>, greater<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

- operator< 오버로딩 방법

만약 'node내림차순, cost오름차순' 으로 설계하려면?

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

using namespace std;

struct noco {
    int node;
    int cost;

    // 'cost오름차순, node오름차순' 으로 설계
    bool operator<(noco nc) const {
    //C++에서 우선순위 큐는 기본 최대힙(오름차순)으로 구현
    //되어있으므로 반대 논리값을 리턴해야 함.
        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;
}
```

문제: 창고

우선순위 큐부터 배우고 올 것.

- 문제

정올이는 N개의 상자를 가지고 있다. 이 중 K개의 상자를 창고에 보관해 두고, 필요할 때 사용하려 한다.

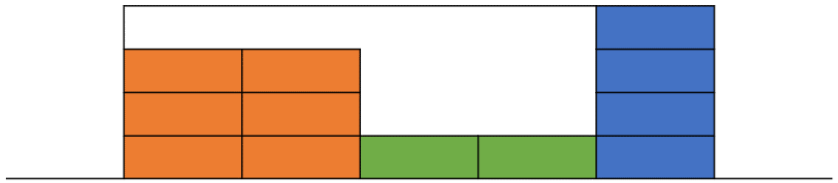
각 상자에는 1~N번까지 번호가 붙여져 있으며, (Wi 너비 * Hi 높이)인 직사각형 모양을 하고 있다.

상자에 들어있는 자재들은 너무 무겁기에, 상자 위에 상자를 쌓는 것은 불가능하다. 또한, 눕히면 자재들이 부서질 위험이 있어, 상자는 돌리지 않고 현재 상태 그대로 창고에 순서대로 보관해야 한다.

K개의 상자를 보관하기 위해서, 창고는 전체 K개의 상자의 너비합에 해당하는 너비와, K개의 상자의 높이 중 최댓값을 가지는 높이를 가지도록 만들 예정이다.

이때, 창고가 사용하는 공간의 크기를 최소화하기 위해, 충북이는 창고가 사용하는 넓이(너비 x 높이)를 최소화하도록 K개의 상자를 선택할 것이다.

그러나 충북이는 어떠한 상자들을 선택해야 창고 넓이가 최소가 되는지 구하기 어려워하고 있다. 충북이를 대신해, 지어야 하는 창고의 최소 넓이를 알려주자.



3개의 상자를 담은 5*4 창고 예시

- 출처: 2023 충북정올 학교예선 중등부 5번 문제

문제: 창고

- 입력형식

  첫 줄에 정수 N, K가 공백을 사이에 두고 주어진다. N은 상자의 수, K는 보관해야 하는 상자의 수를 의미한다.

  다음 N개의 줄 중 i번째 줄에는 i번째 상자의 너비와 높이 Wi, Hi가 공백을 사이에 두고 주어진다.

  - $1 \le K \le N \le 100{,}000$
  - $1 \le W_i, H_i \le 1{,}000{,}000$

- 출력형식

  K개의 상자를 선택해, 지어야 하는 창고의 최소 넓이를 출력한다.

- 입력과 출력의 예

입력 예1	출력 예1
4 3 2 3 2 1 1 4 4 5	20

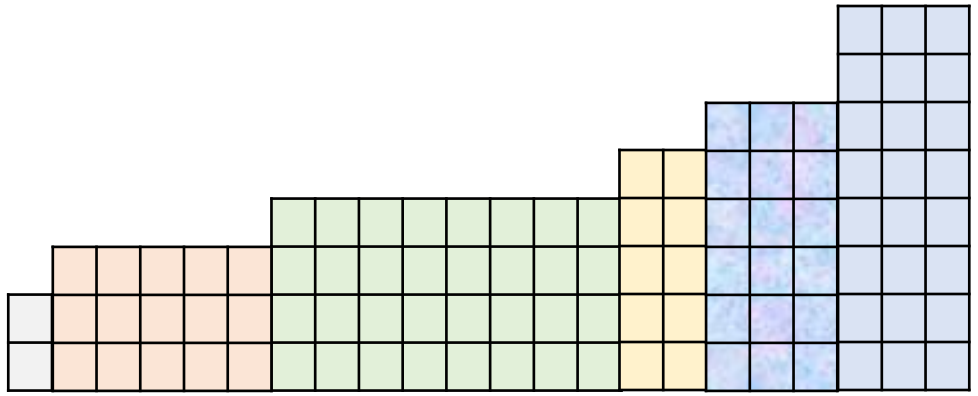
입력 예2	출력 예2
4 1 5 4 3 6 1 19 2 10	18

- 출처: 2023 충북정올 학교예선 중등부 5번 문제

문제: 창고

- 고찰
  - 어떤 상자를 선택해야 창고의 너비가 최소가 될 것인가?
  - width * height 가 최소가 되려면 너비와 높이 둘 다 작아야 유리하다.
  - 일단 높이를 기준으로 정하여 정렬하고 작은 것들 우선 선택해본다.
  - 하지만 이것이 최상의 답이라는 보장이 없으므로 선택된 상자 중 최악의 너비를 제거하고 남은 상자들을 추가해 보다 나은 답을 계속 찾는다.

- 최악의 너비 찾기
  - 우선순위큐는 최대힙을 구성하므로 상자의 너비들을 이 큐에 넣어 놓으면, 자동으로 최상단에 가장 큰 너비가 위치한다.
  - 1순위 교체 대상인 가장 큰 너비를 빠르게 찾을 수 있다.



```
// 창고 정답 풀이

#include <iostream>
#include <queue>
#include <algorithm>
#include <limits.h>

#define llong long long
using namespace std;

struct box {
    int w, h;
    // box 구조체의 정렬 방법 정의
    bool operator<(const box& b) {
        if(h != b.h)  return h < b.h;  //h 오름차순
        else return w < b.w;  //w 오름차순
    }
} box[1000000];

int n, k;
llong res=LLONG_MAX, width;

// 선택된 상자의 너비를 보관할 예정
priority_queue <int> pq;
```

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

    for(i=0; i<n; i++)
        scanf("%d %d", &box[i].w, &box[i].h);
    sort(box, box + n);

    for(i=0; i<k-1; i++) { // 높이가 작은거부터 k-1개의 상자 선택하여
        width += box[i].w; // 너비 합
        pq.push(box[i].w); // 너비 푸시
    }

    //k번(인덱스는 k-1임) 상자부터 마지막 상자까지 한 개씩 교환해가며 너비계산
    for(i=k-1; i<n; i++) {
        width += box[i].w;  // i번 상자 너비 추가
        pq.push(box[i].w);  // i번 상자 너비 푸시
        //상자의 높이 기준 오름차순 정렬을 해 놓은 상태이므로,
        //i번 앞쪽에 배치된 모든 상자의 높이는 box[i].h 보다 작을 수밖에 없다.
        res = min(res, width * box[i].h);
        // 선택된 상자 중 너비가 가장 큰 상자 제거
        width -= pq.top();
        pq.pop();
    }
    printf("%lld", res);
    return 0;
}
```

```
// 아래와 같이 main() 함수를 수정하여 작동을 추적해 보자.
int main() {
    int i;
    scanf("%d %d", &n, &k);

    for(i=0; i<n; i++)
        scanf("%d %d", &box[i].w, &box[i].h);
    sort(box, box + n);

    puts("\n정렬 이후:");
    for(i=0; i<n; i++)
        printf("(%d,%d)\n", box[i].w, box[i].h);
    putchar('\n');

    for(i=0; i<k-1; i++) { // 높이가 작은거부터 k-1개의 상자 선택하여
        width += box[i].w; // 너비 합
        pq.push(box[i].w); // 너비 푸시
        printf("(%d,%d) 선택\n", box[i].w, box[i].h);
    }

    //k번(인덱스는 k-1임) 상자부터 마지막 상자까지 한 개씩 교환해가며 너비계산
    for(i=k-1; i<n; i++) {
        width += box[i].w;  // i번 상자 너비 추가
        pq.push(box[i].w);  // i번 상자 너비 푸시
        //상자의 높이 기준 오름차순 정렬을 해 놓은 상태이므로,
        //i번 앞쪽에 배치된 모든 상자의 높이는 box[i].h 보다 작을수 밖에 없다.
        printf("(%d,%d) 추가 선택시 너비: %d, 넓이: %d\n\n",
                box[i].w, box[i].h, width, width*box[i].h);
        res = min(res, width * box[i].h);
        // 선택된 상자 중 너비가 가장 큰 상자 제거
        width -= pq.top();
        printf("가장 큰 상자 너비 %d 제거, 현재 너비 %d\n", pq.top(), width);
        pq.pop();
    }
    printf("%lld", res);
    return 0;
}
```

창고 해답 작동 추적하기

- 작동 추적
  - 5개 상자 입력, 창고에 3개 보관

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

정렬 이후:
(5,2)
(4,3)
(3,4)
(2,5)
(1,6)
```

높이 기준
오름차순
정렬됨

```
(5,2) 선택
(4,3) 선택
(3,4) 추가 선택시 너비: 12, 넓이: 48

가장 큰 상자 너비 5 제거, 현재 너비 7
(2,5) 추가 선택시 너비: 9, 넓이: 45

가장 큰 상자 너비 4 제거, 현재 너비 5
(1,6) 추가 선택시 너비: 6, 넓이: 36

가장 큰 상자 너비 3 제거, 현재 너비 3
36
```

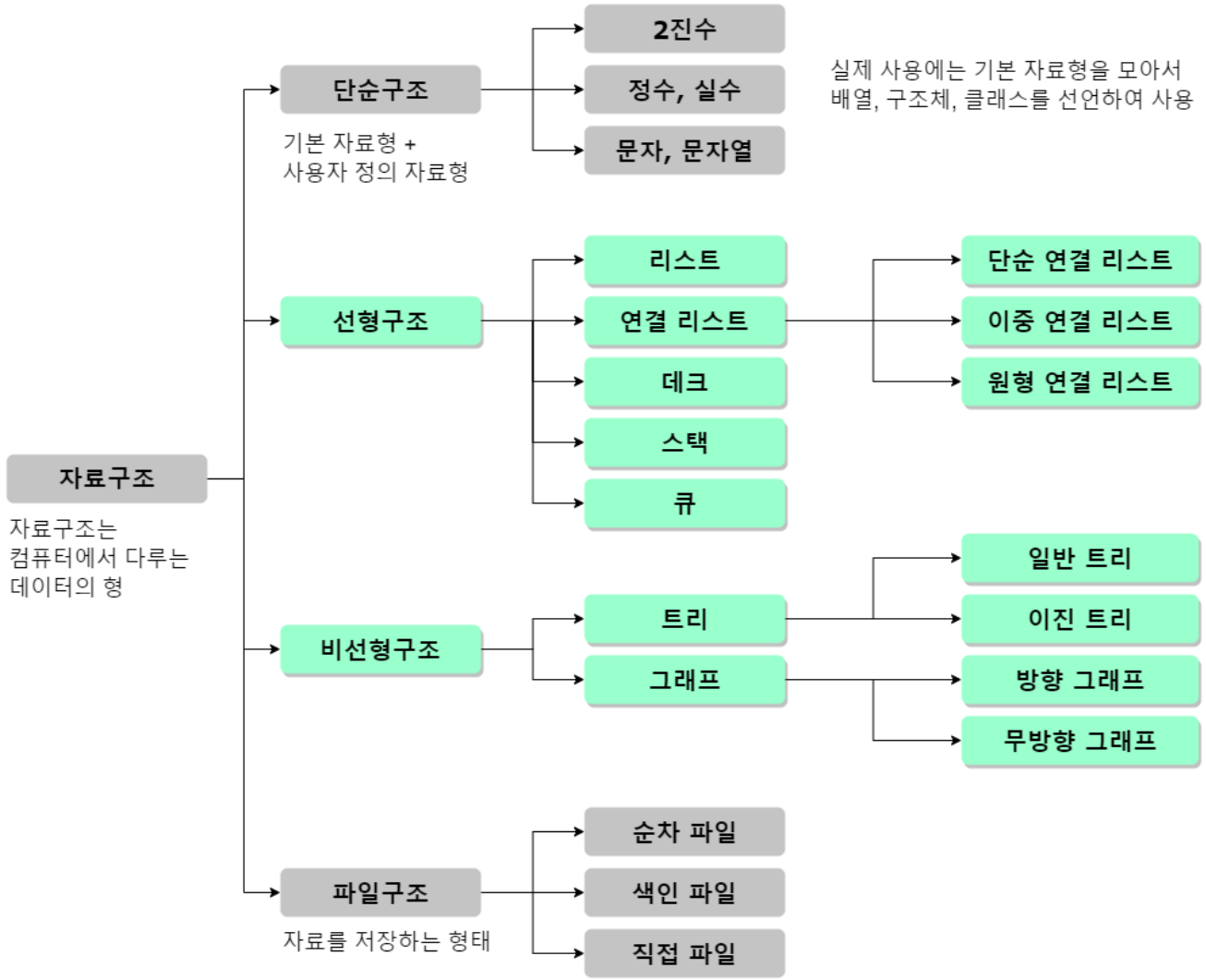
k-1 개 선택한 상태에서

자료구조

선형구조와 비선형구조

자료구조

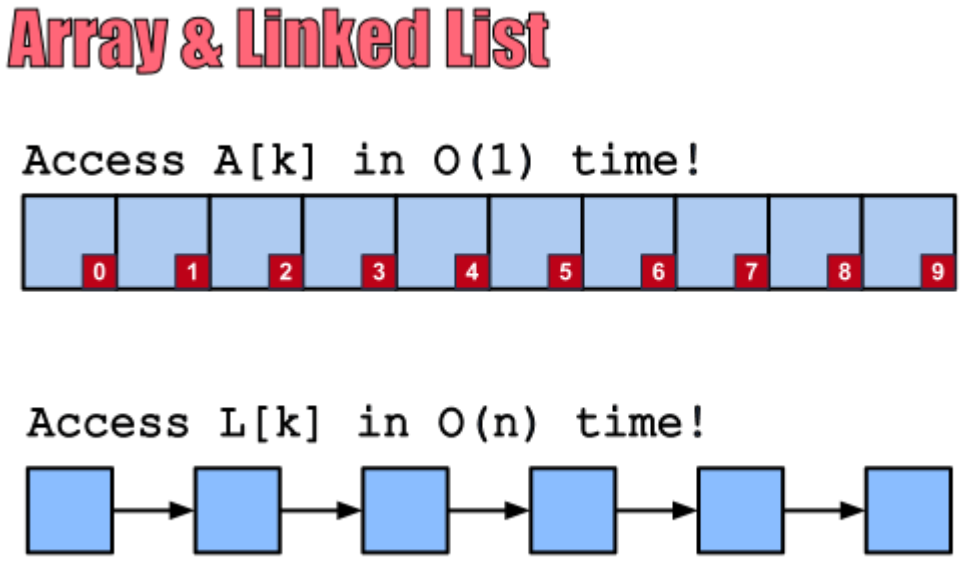
- 자료구조(data structure)
  - 전산학에서 자료를 효율적으로 이용할 수 있도록 컴퓨터에 저장하는 방법이다.
  - 신중히 선택한 자료구조는 보다 효율적인 알고리즘을 사용할 수 있게 한다.

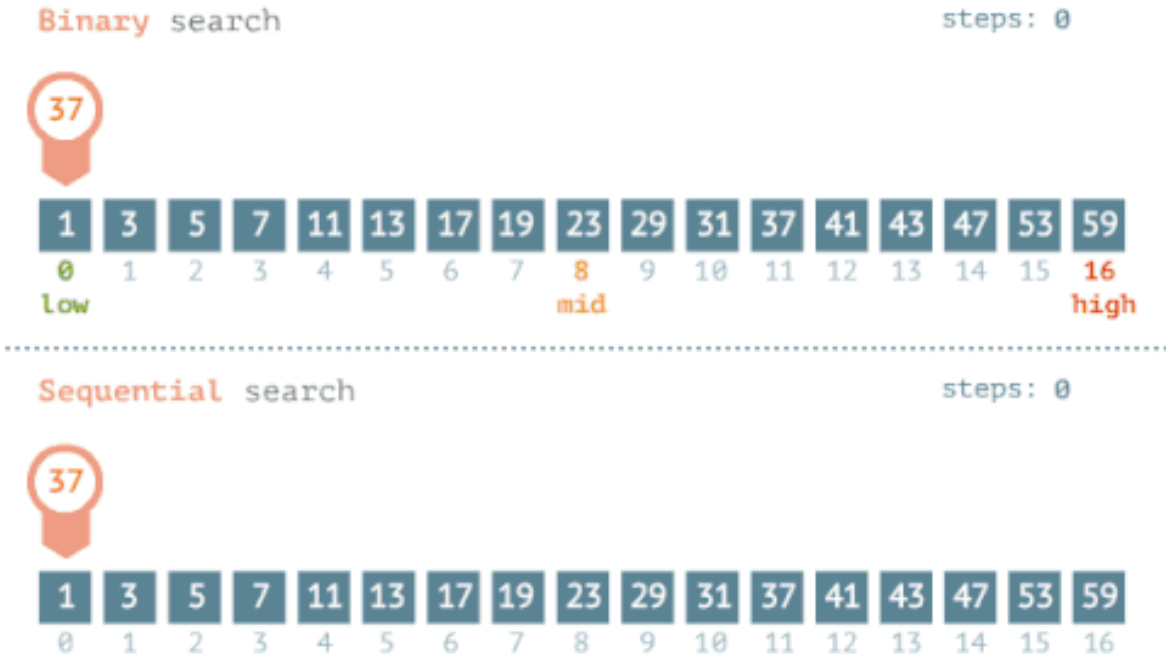


선형구조

- 선형구조란?
  - 자료를 구성하는 데이터를 순차적으로 나열시킨 형태를 의미

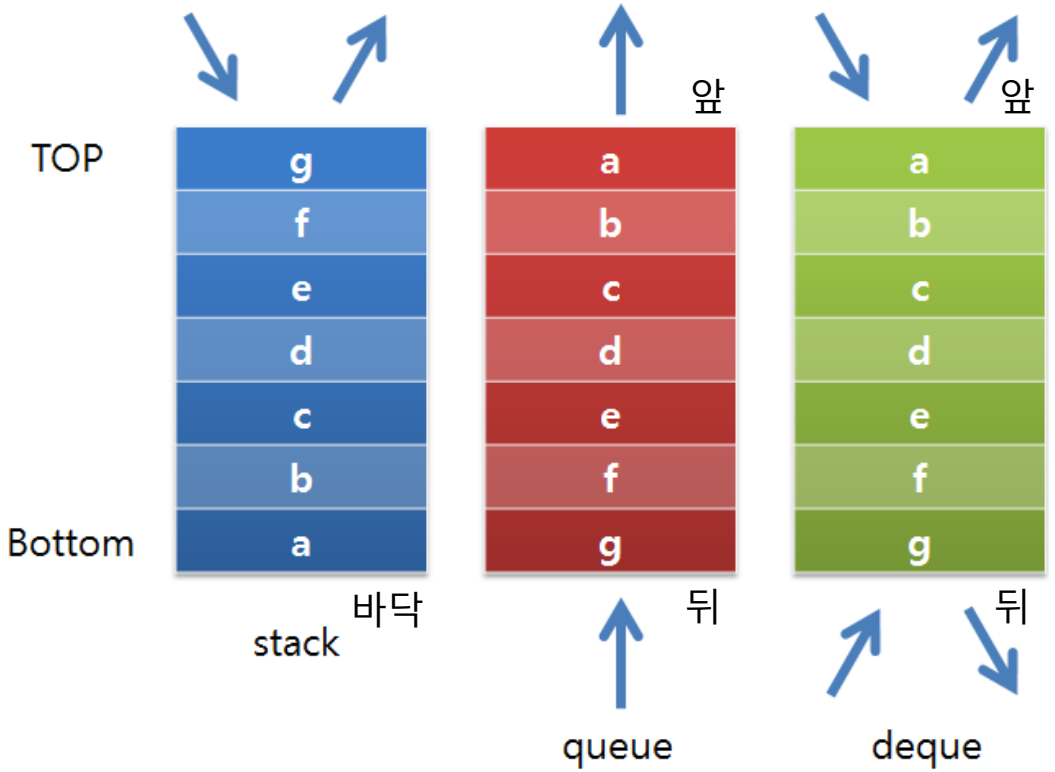
- 탐색법
  - 순차탐색
  - 이분탐색





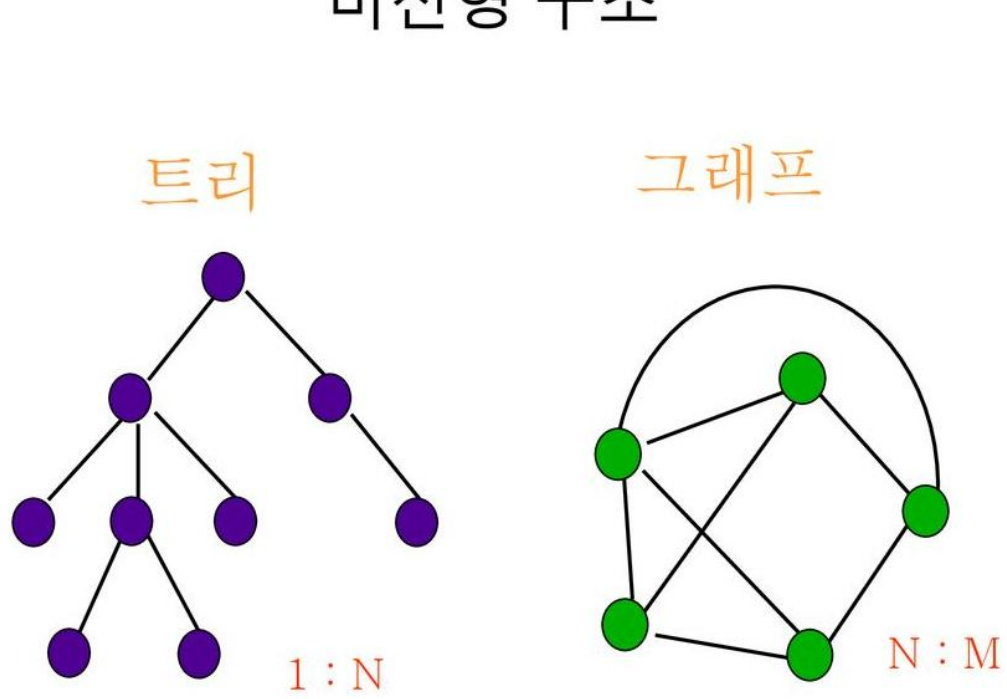
선형구조

- 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



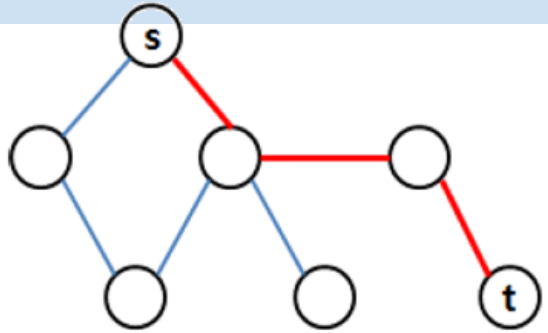
비선형구조

- 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)

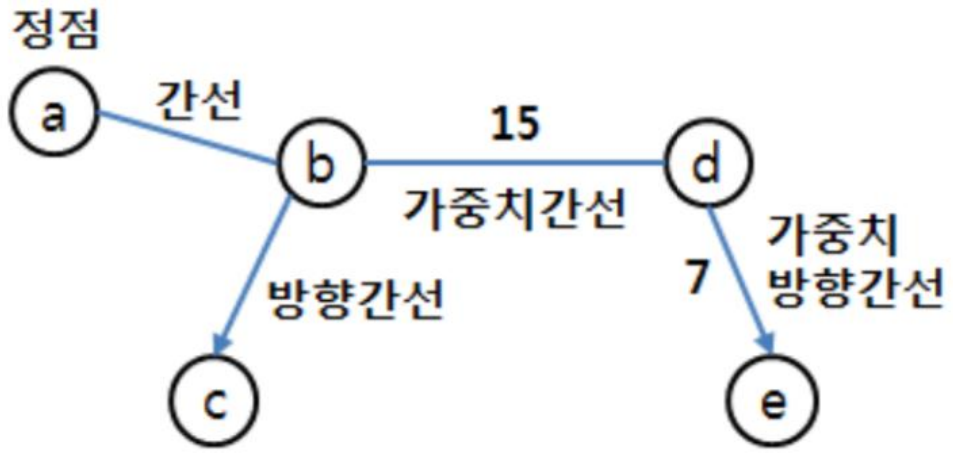


- 그래프 중에 회로가 없는 그래프를 트리라고 한다.

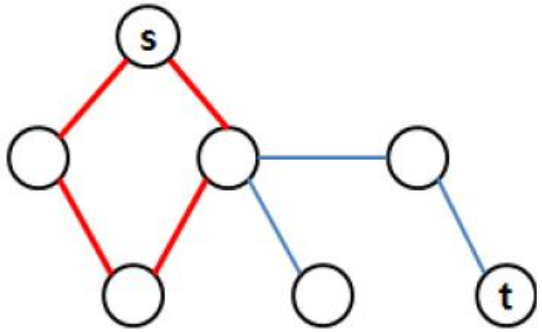
비선형구조 - 그래프



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

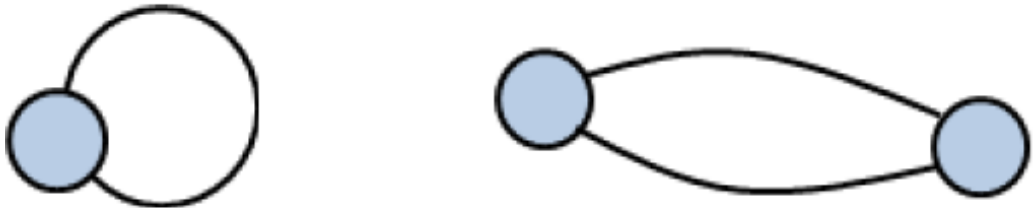


- 경로(path)
  - 임의의 정점 s에서 임의의 정점 t로 이동할 때, s에서 t로 이동하는데 사용한 정점들을 연결하고 있는 간선들의 순서로된 집합
- 회로(cycle)
  - 그래프에서 임의의 정점 s에서 같은 정점 s로의 경로들



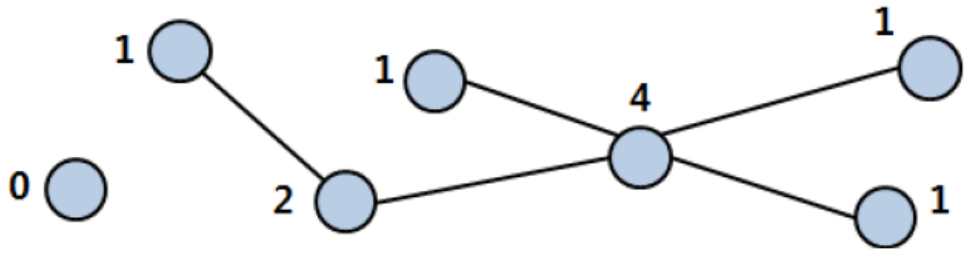
비선형구조 - 그래프

- 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



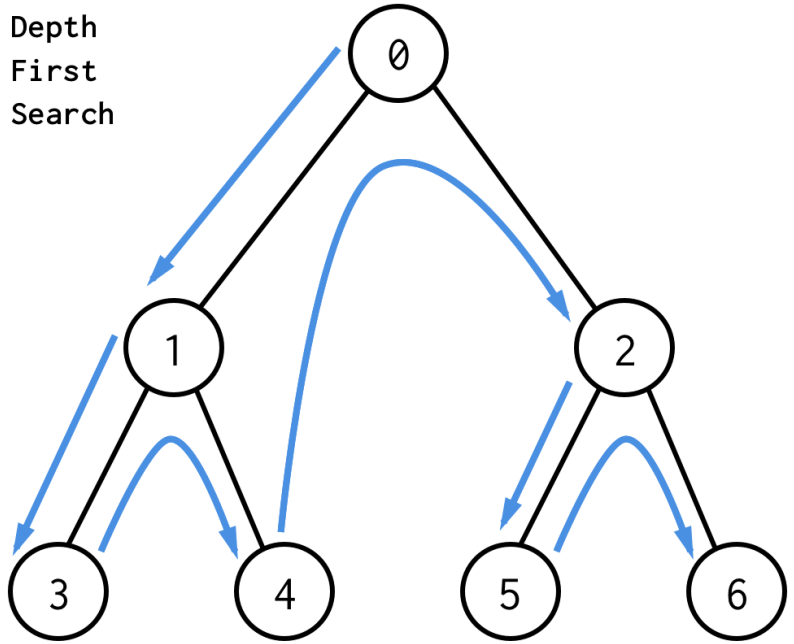
각 정점에서의 차수

비선형 탐색

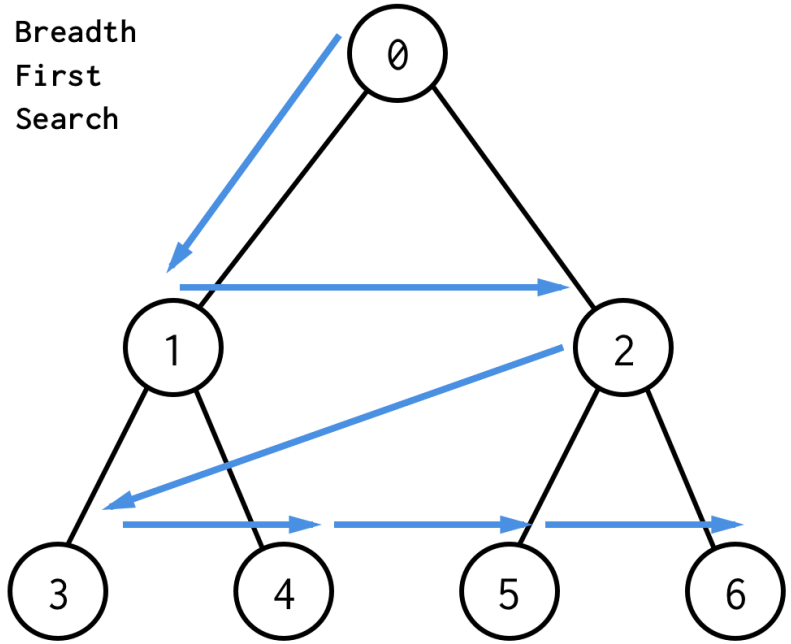
비선형구조의 전체탐색(DFS vs BFS)

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

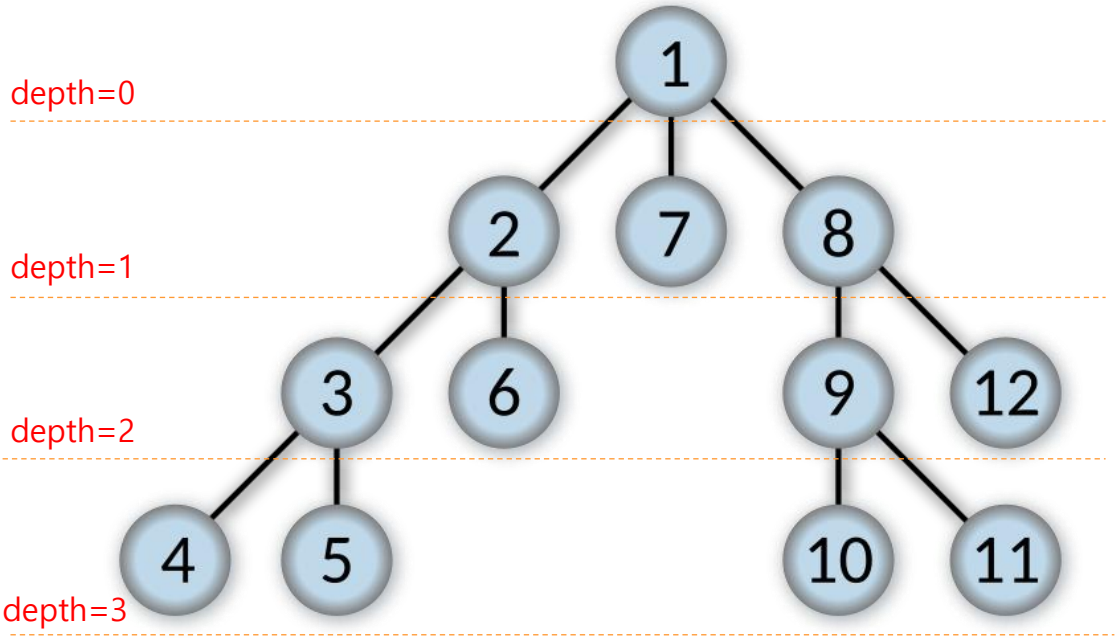
탐색

- 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

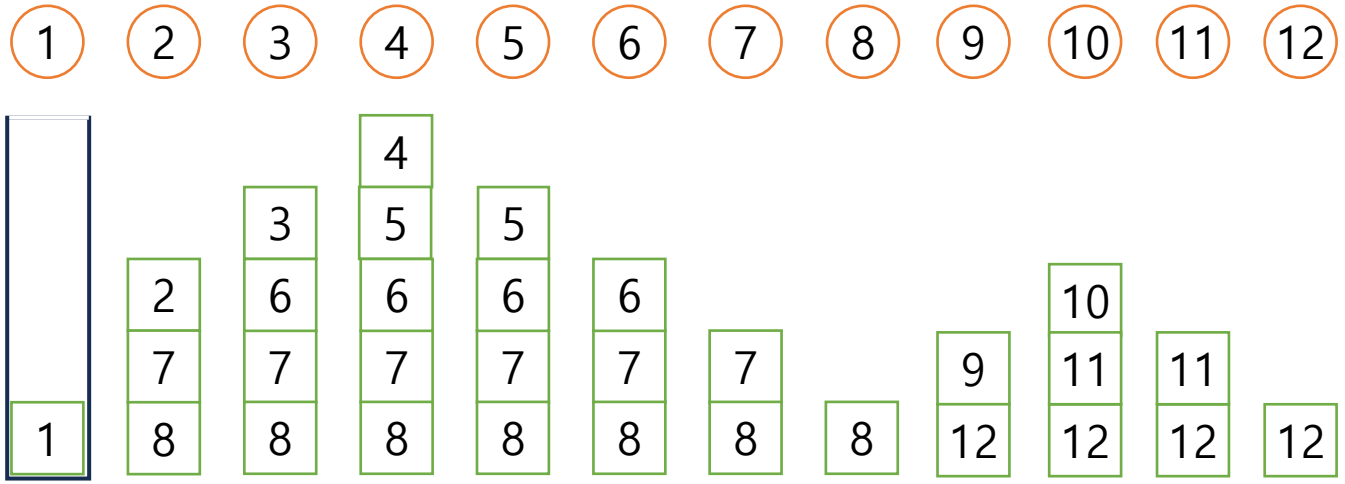
- 너비우선탐색(BFS)
  - 최대한 넓게 이동한 다음, 더 이상 갈 수 없을 때 아래로 이동
  - 루트 노드(혹은 다른 임의의 노드)에서 시작해서 인접한 노드를 먼저 탐색하는 방법
  - 시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회 방법

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

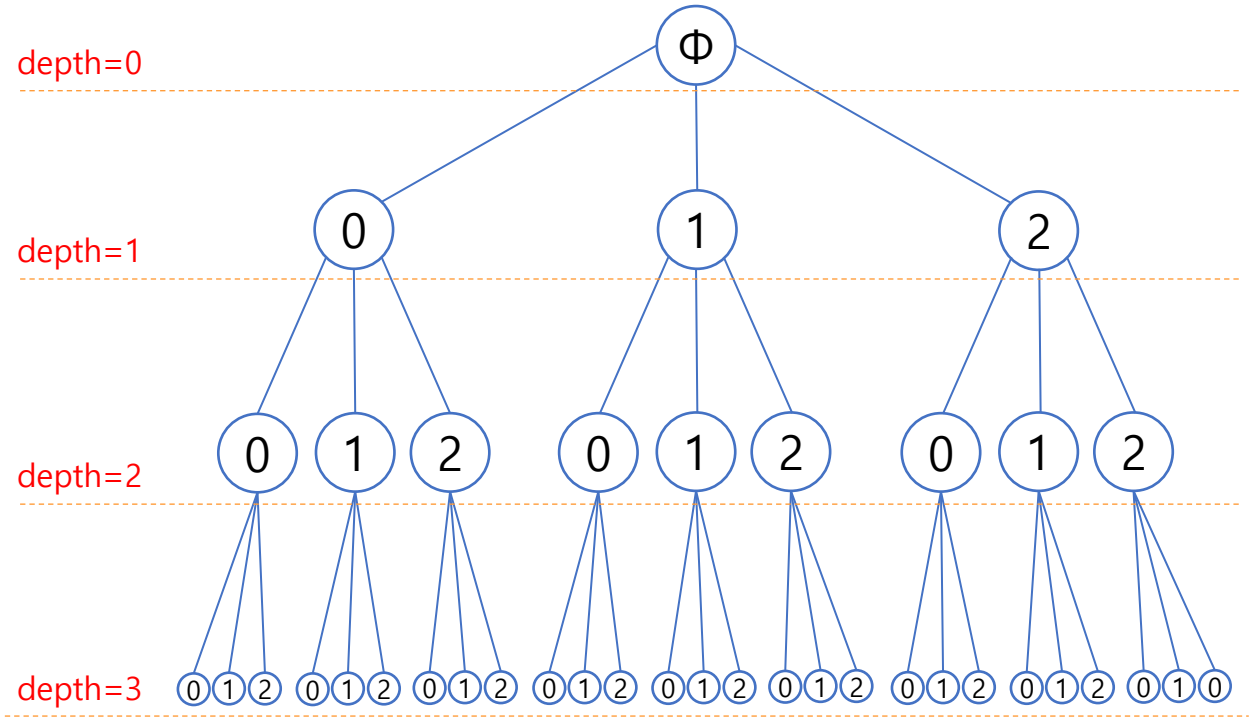
- 중복있고 순서있는 순열 조합
  - 중복 있다
    - 같은 숫자 다시 등장 가능

```
0-0-0  //
0-0-1  // OK
0-0-2  // OK
0-1-0  // OK
```

  - 순서 있다
    - 순서 다른 조합을 다른 것으로 간주

```
// 아래는 모두 다른 조합
0-1-2
1-2-0
2-1-0
```

- 그래프로 표현
  - 0,1,2 중에서 뽑을 때,
  - 매 단계에서 3갈래로 갈라짐



DFS 활용 순열 조합 1

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

int n, r;
int d[] = {0,1,2,3,4,5,6,7,8,9};

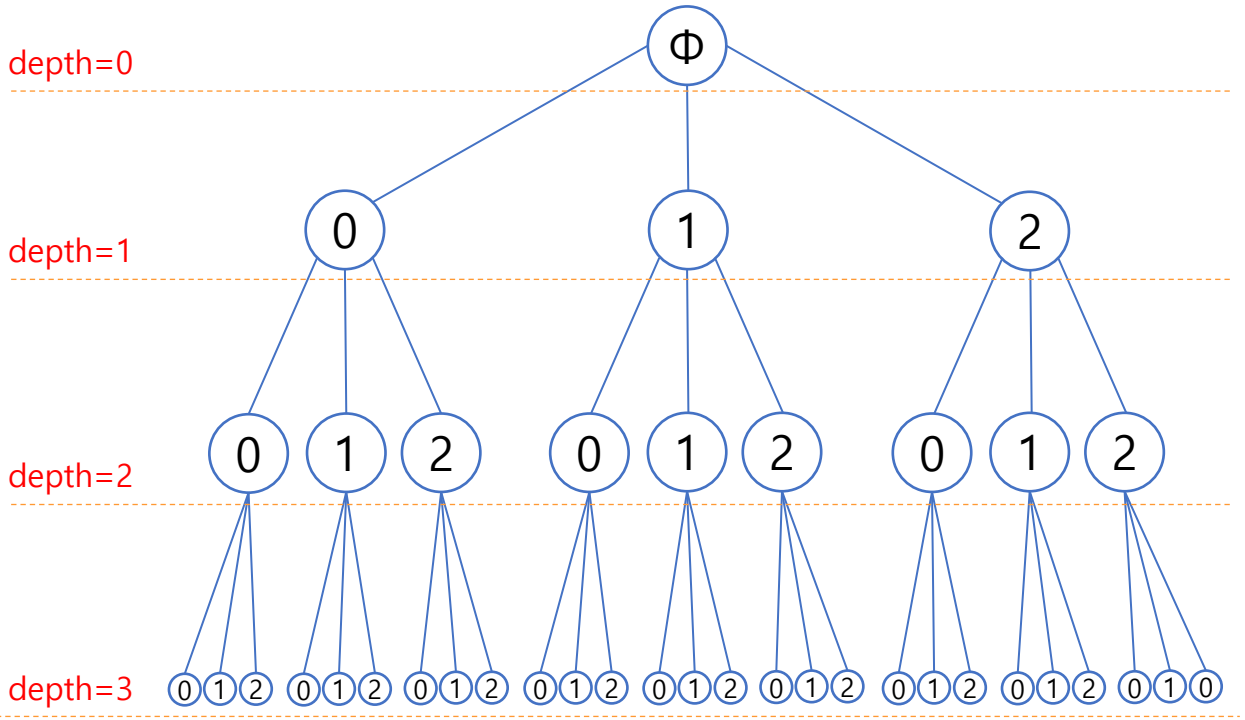
void output(int depth, vector<int>& vec) {
    printf("[%d] ", depth);
    for(int a : vec)
        printf("%d-", a);
    printf("\b \n");
}
```

```
void dfs(int depth, vector<int>& v) {
    // 뽑은 개수 상관없이 작업하려면 여기에서
    output(depth, v);
    if(depth==r) {  // r개 모두 뽑았으면,
        //output(depth, v);
        return;     // 더 갈라지지 말고 돌아가!
    }
    //매 갈림길에서 똑같이 n개로 갈라짐을 구현
    for(int i=0; i<n; i++) {
        v.push_back(d[i]); // d[i] 선택 후,
        dfs(depth+1, v);   // dfs 탐색 재실행
        v.pop_back();      // dfs가 탐색 종료 후 d[i]제거
    }
}
int main() {
    // n개의 데이터 중에서 r개 뽑기
    scanf("%d %d", &n, &r);
    vector<int> v;
    dfs(0, v);  // depth 0에서 r까지 가면 끝남
    return 0;
}
```

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

DFS 활용 순열 조합 1

- 실행 분석



- 소스코드

```
void dfs(int depth, vector<int>& v) {
    // depth는 뽑은 개수를 의미
    if(depth==r) {  // r개 모두 뽑았으면,
        output(depth, v);
        return;     // 더 갈라지지 말고 돌아가!
    }
    //매 갈림길에서 똑같이 n개로 갈라짐을 구현
    for(int i=0; i<n; i++) {
        // depth 레벨에서 d[i] 선택 후,
        v.push_back(d[i]);
        // depth+1 레벨 dfs 시작
        dfs(depth+1, v);
        // dfs() 종료되면 = 백트랙하면
        v.pop_back();
    }
}
```

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

합이 k가 되는 수열 만들기(중복허용)

- 문제

  1에서 9사이 서로 다른 자연수 n개가 주어진다.

  주어진 숫자 만을 사용하여 길이가 r인 수열을 만든다.

  (만드는 수열은 중복을 허용하고, 순서가 다르면 다른 것으로 본다)

  생성한 모든 수열 중에서, 수열에 포함된 숫자들의 합이 정확히 k가 되는 경우의 수를 구하여라.

  만약 숫자 1 2 3 를 이용하여 2자리 수열을 만들어 합이 3이 되는 경우를 수를 구한다면,

  (1,3) (2,2) (3,1)

  이렇게 3가지가 존재한다.

- 입력

  n

  a_1 a_2 a_3 ... a_n

  r

  k

  n: 사용할 수 있는 수의 개수 ($1 \le n \le 9$)

  a_1, a_2, ..., a_n: 사용할 수 있는 n개의 자연수($1 \le a_i \le 9$)

  r: 수열의 길이($1 \le r \le 5$)

  k: 만들고자 하는 합

- 출력

  조건을 만족하는 수열의 개수를 출력한다.

입력 예	출력 예
3 1 2 3 2 4	3

합이 k가 되는 수열 만들기

```
#include <stdio.h>
#include <vector>
#define MAX_N 10
using namespace std;

int n, r, k;
int d[MAX_N];
int ans = 0;

int main() {
    scanf("%d", &n);
    for(int i=0; i<n; i++) {
        scanf("%d", &d[i]);
    }
    scanf("%d\n%d", &r, &k);

    vector<int> v;
    dfs(0, 0, v);
    printf("%d\n", ans);
    return 0;
}
```

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

void dfs(int depth, int sum, vector<int>& v) {
    if(depth == r) {
        if(sum == k) {
            //output(v);
            ans++;
        }
        return;
    }

    for(int i=0; i<n; i++) {
        v.push_back(d[i]);
        dfs(depth+1, sum + d[i], v);
        v.pop_back();
    }
}
```

```
4
1 2 3 4
3
7
1-2-4
1-3-3
1-4-2
2-1-4
2-2-3
2-3-2
2-4-1
3-1-3
3-2-2
3-3-1
4-1-2
4-2-1
12
```

DFS 활용 순열 조합 2

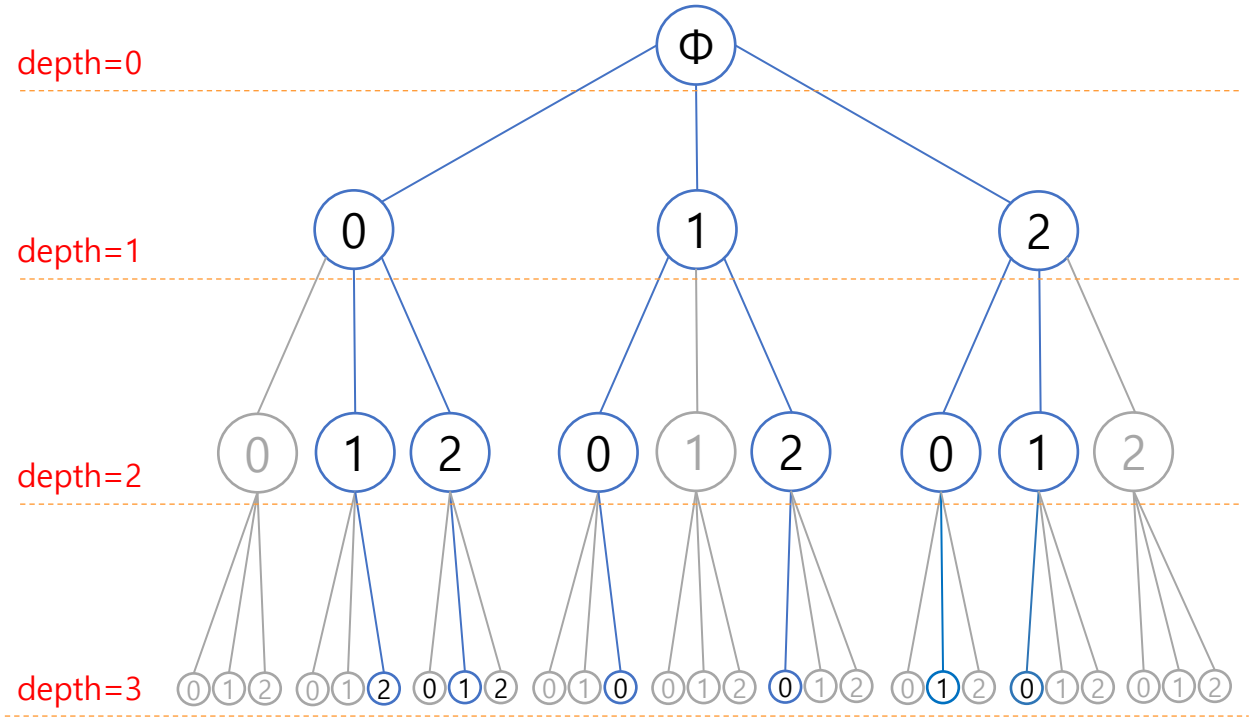
- 중복없고 순서있는 순열 조합
  - 중복 없다
    - 같은 숫자 다시 등장 불가능

```
0-0-1  // !
0-1-1  // !
0-1    // OK
0-1-2  // OK
```

  - 순서 있다
    - 순서 다른 조합을 다른 것으로 간주

```
// 아래는 모두 다른 조합
0-1-2
1-2-0
2-1-0
```

- 그래프로 표현
  - 0,1,2 중에서 뽑을 때,
  - 매 단계에서 3갈래로 갈라짐



DFS 활용 순열 조합 2

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

int n, r;
int    d[] = {0,1,2,3,4,5,6,7,8,9};
int used[] = {0,0,0,0,0,0,0,0,0,0}; // 사용 여부를 저장

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

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

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

```
4 3
[3] 0-1-2
[3] 0-1-3
[3] 0-2-1
[3] 0-2-3
[3] 0-3-1
[3] 0-3-2
[3] 1-0-2
[3] 1-0-3
[3] 1-2-0
[3] 1-2-3
[3] 1-3-0
[3] 1-3-2
[3] 2-0-1
[3] 2-0-3
[3] 2-1-0
[3] 2-1-3
[3] 2-3-0
[3] 2-3-1
[3] 3-0-1
[3] 3-0-2
[3] 3-1-0
[3] 3-1-2
[3] 3-2-0
[3] 3-2-1
```

숫자 카드로 수열 만들기 I

- 문제

  1부터 9사이 숫자가 적힌 카드 n개가 주어졌을 때, 이 숫자 카드를 나열하여 만들어 낼 수 있는 r자릿수의 숫자 중

  2의 배수와 3의 배수가 몇 개 인지 출력하는 프로그램을 작성하시오.

  예를 들어, 2 3 4 가 적힌 세 개의 카드를 나열하여 만들어 낼 수 있는 2자리 숫자는

  23, 24, 32, 34, 42, 43 이 가능하고,

  2의 배수는 4개(24, 32, 34, 42)이고,

  3의 배수는 2개(24, 42)이다.

- 입력

  n

  a_1 a_2 a_3 ... a_n

  r

  n: 주어진 숫자 카드의 개수 n (1 ≤ n ≤ 9)

  a_1, a_2, ..., a_n: 카드에 적힌 숫자 목록(1 ≤ a_i ≤ 9)

  r: 숫자의 자릿수(1 ≤ r ≤ 8)

- 출력

  첫 번째 줄에 2의 배수가 몇 개 만들어지는지 출력하고,

  두 번째 줄에 3의 배수가 몇 개 만들어지는지 출력한다.

입력 예	출력 예
3 2 3 4 2	4 2

숫자 카드로 수열 만들기 I

```
#include <stdio.h>
#define  MAX_N   10
using namespace std;

int n, r;
int d[MAX_N];
int used[MAX_N]; // 사용 여부를 저장하는 배열
int cnt2=0, cnt3=0; // 배수 카운터
void dfs(int depth, int num);

int main() {
    scanf("%d", &n);
    for(int i=0; i<n; i++) {
        scanf("%d", &d[i]);
    }
    scanf("%d", &r);

    dfs(0, 0);
    printf("%d\n%d\n", cnt2, cnt3);
    return 0;
}
```

```
void dfs(int depth, int num) {
    if(depth==r) {
        printf("%d %d%d\n", num, num%2==0, num%3==0);
        if(num%2==0) cnt2++;
        if(num%3==0) cnt3++;
        return;
    }

    // 매 노드에서 n개의 노드로 갈라짐을 구현
    for(int i=0; i<n; i++) {
        if(!used[i]) {
            used[i] = 1;  // 사용했음을 표시
            num = num*10+d[i]; //d[i]를 추가하여 만든 수
            dfs(depth+1, num);
            num = (num-d[i])/10; //d[i] 추가전으로 복원
            used[i] = 0;  // 사용했음 표시 취소
        }
    }
}
```

```
4
1 2 4 5
2
12 11
14 10
15 01
21 01
24 11
25 00
41 00
42 11
45 01
51 01
52 10
54 11
6
8
```

숫자 카드로 수열 만들기 II

- 문제

  아래 그림과 같이 책상 위에 숫자 카드 9장이 놓여 있다.



  이 카드 중 일부 n개를 선택한 뒤, 이 카드를 나열하여 만들어 낼 수 있는 모든 n 자릿수의 숫자들을 상상해 보자. 이 숫자들을 오름차순 정렬하였을 때 k 번 째에 해당하는 수가 무엇인지 알아내는 프로그램을 작성하시오.

  예를 들어 5 2 3 이렇게 3장의 숫자 카드를 나열하여 만들어낼 수 있는 숫자들을 오름차순으로 나열하면 235, 253, 325, 352, 523, 532 이고, 이 중 6번째 숫자는 532이다.

입력 예	출력 예
3 2 3 5 6	532

- 입력

  첫 번째 줄에 선택된 숫자 카드의 개수 n이 입력된다. (1 ≤ n ≤ 9)

  두 번째 줄에 선택된 카드에 적힌 숫자들이 공백으로 분리되어 입력된다.

  세 번째 줄에 k가 입력된다.

- 출력

  선택된 숫자 카드를 나열하여 만들어 낼 수 있는 수들을 오름차순 정렬하였을 때 k번째에 해당하는 수를 출력한다.

  만약 k번째에 해당하는 수가 없다면 아무것도 출력하지 마시오.

숫자 카드로 수열 만들기 II

```
#include <iostream>
#include <vector>
#define  MAX_N    10
using namespace std;

int n, k, cnt=0;
int    d[MAX_N];
int used[MAX_N];
int ans=0;
bool dfs(int depth, int num) ;

int main() {  // n개의 데이터 중에서 r개 뽑기
    cin >> n;
    for(int i=0; i<n; i++)
        cin >> d[i];
    cin >> k;

    if(dfs(0, 0)) cout << ans;
    return 0;
}
```

```
bool dfs(int depth, int num) {
    if(depth==n) {
        cnt++;
        if(cnt == k) {
            ans = num;
            return true;
        }
    }

    for(int i=0; i<n; i++) {
        if(!used[i]) {
            used[i] = 1;
            if(dfs(depth+1, num*10+d[i]))
                return true;
            used[i] = 0;
        }
    }
    return false;
}
```

DFS 활용 순열 조합 3

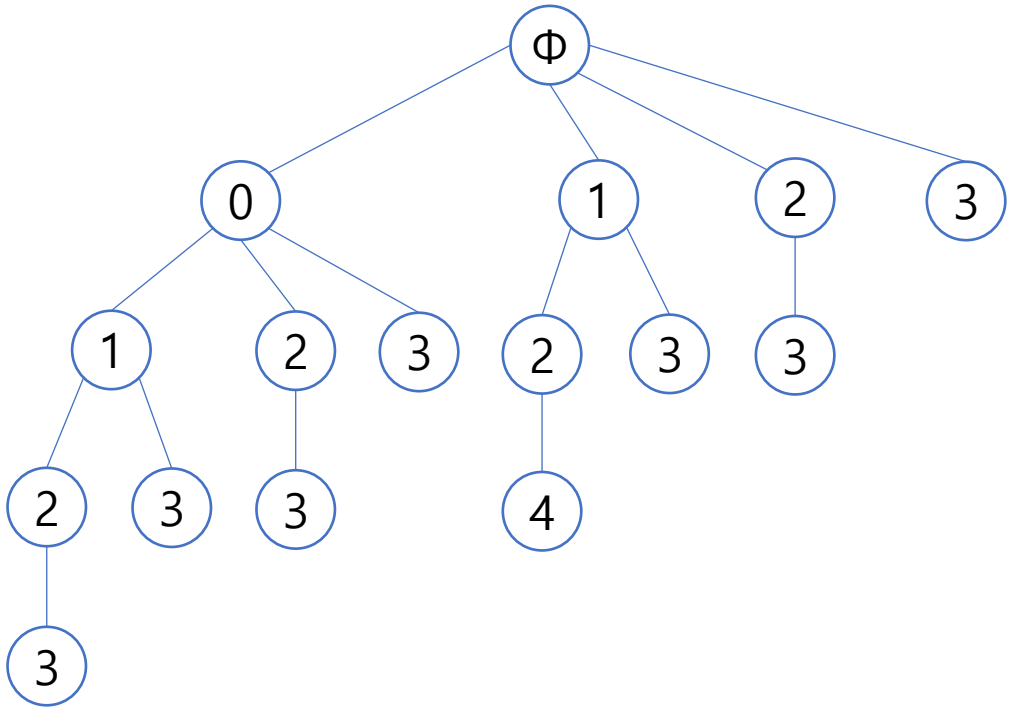
- 중복없고 순서없는 순열 조합
  - 중복 없다
    - 같은 숫자가 또 등장하지 않음

```
0-0-1  // !
0-1-1  // !
0-1    // OK
0-1-2  // OK
```

  - 순서 없다
    - 순서 다른 조합을 동일한 것으로 간주

```
// 아래는 모두 동일한 조합
0-1-2
0-2-1
2-1-0
```

- 그래프로 표현
  - 0,1,2,3 중에서 뽑을 때,
  - 자신 노드보다 큰 수로만 뻗어 나감



DFS 활용 순열 조합 3

```
4 3
( 1)
( 2) 0
( 3) 0-1
( 4) 0-1-2
( 5) 0-1-2-3
( 6) 0-1-3
( 7) 0-2
( 8) 0-2-3
( 9) 0-3
(10) 1
(11) 1-2
(12) 1-2-3
(13) 1-3
(14) 2
(15) 2-3
(16) 3
```

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

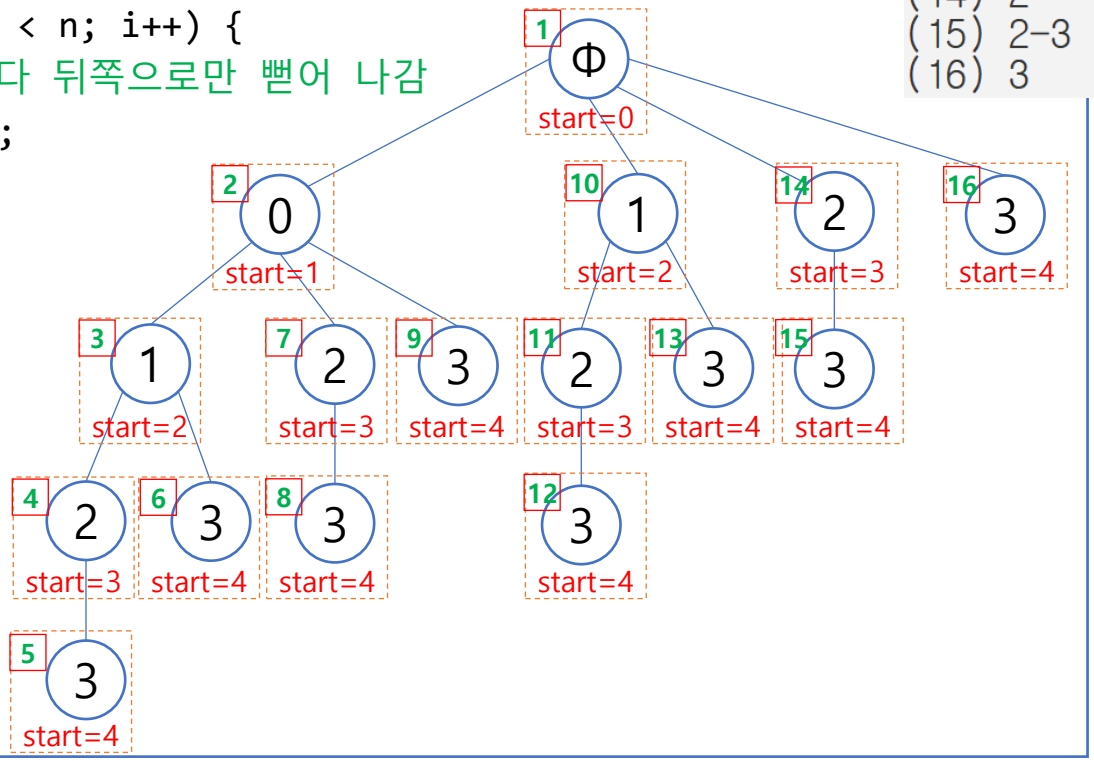
int n;
int d[] = {0,1,2,3,4,5,6,7,8,9};
int nth=0;  // 몇 번째 호출인지 기록
vector<int> v;

void output() {
    printf("(%2d) ", nth);
    for(int a : v)
        printf("%d-", a);
    printf("\b \n");
}
```

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

    //start부터 그 이후 원소들로만 뻗어나감을 구현
    for (int i = start; i < n; i++) {
        //i번 선택 후 그보다 뒤쪽으로만 뻗어 나감
        v.push_back(d[i]);
        dfs(i + 1);
        v.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] (0) 을 푸시
  - dfs(1) 호출
  - start = 1

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

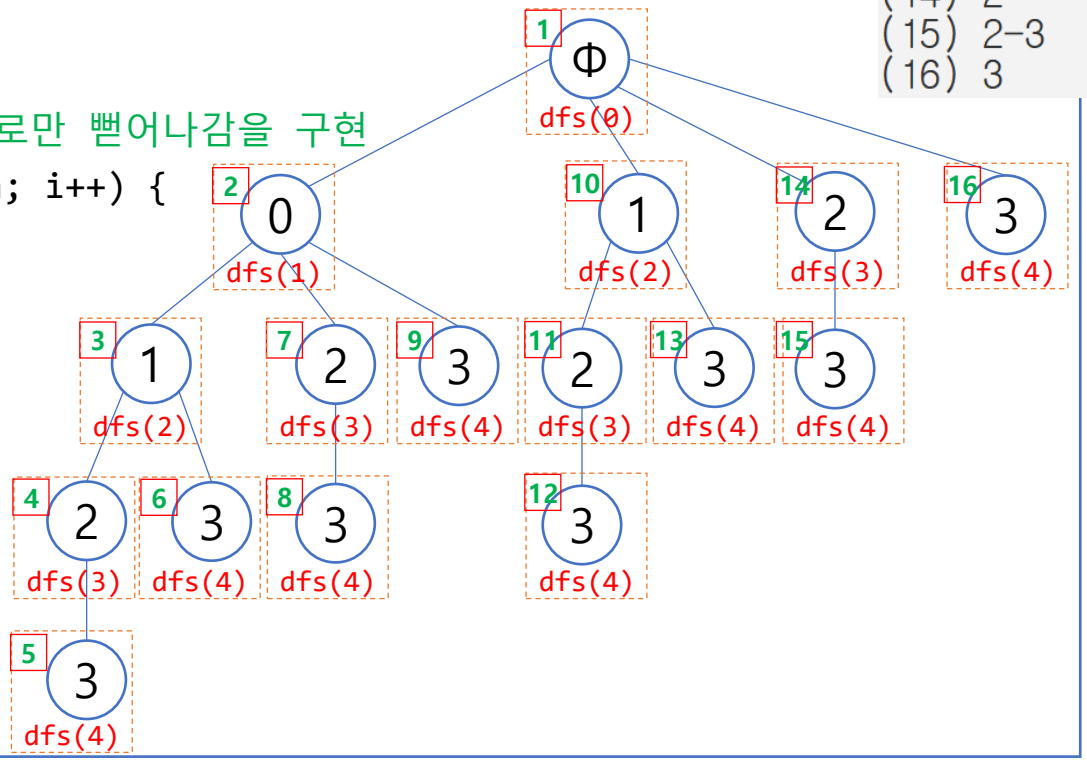
    :

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

void dfs(int start, vector<int> &v) {
    nth++;  // 호출 횟수 증가
    output(v);

    //start부터 그 이후 원소들로만 뻗어나감을 구현
    for (int i = start; i < n; i++) {
        v.push_back(d[i]);
        dfs(i + 1, v);
        v.pop_back();
    }
}
int main() {
    scanf("%d", &n);
    vector<int> v;
    dfs(0);
}
```

```
4 3
( 1)
( 2) 0
( 3) 0-1
( 4) 0-1-2
( 5) 0-1-2-3
( 6) 0-1-3
( 7) 0-2
( 8) 0-2-3
( 9) 0-3
(10) 1
(11) 1-2
(12) 1-2-3
(13) 1-3
(14) 2
(15) 2-3
(16) 3
```



공평한 배분

- 문제

  금광 앞에 있는 살고 있는 지겸이와 재경이에게 금광에서 나온 금을 최대한 공평하게 나누어 주고자 한다.

  금덩이의 개수 n과, 각 금덩이의 무게가 주어졌을 때, 두 사람이 가장 공평하게 나누어 갖는 방법을 계산하는 프로그램을 작성하시오.

  예를 들어 5g, 1g, 4g, 11g, 8g 이렇게 5개의 금덩이가 주어진다면,

  한 사람이 14g (5g+1g+8g), 나머지 한 사람이 15g(4g+11g)으로 나누었을 때 그 차이가 1g으로 가장 공평하다.

- 입력

  첫 번째 줄에는 금덩이의 개수가 입력된다.(1 ≤ n ≤ 20)

  두 번째 줄에는 금덩이들의 무게가 공백으로 구분되어 입력된다.(1 ≤ 금덩이 무게 ≤ 1000)

- 출력

  가장 공평하게 나누었을 때의 차이를 출력한다.

- 입력과 출력의 예

입력 예	출력 예
5 5 1 4 11 8	1

공평한 배분

- 전체 코드

```
#include <iostream>
#include <vector>
#include <cmath>
#define  MAX_N  20
using namespace std;

int n;
int d[MAX_N];
int tot=0, ans=1e9;
vector<int> v;
void dfs(int start, int sum);

int main() {
    cin >> n;
    for(int i=0; i<n; i++) {
        cin >> d[i];
        tot += d[i];
    }

    dfs(0, 0);
    cout << ans;
}
```

```
void dfs(int start, int sum) {









}
```

```
5
5 1 4 11 8
29,  0() vs 29
19,  5(5,) vs 24
17,  6(5,1,) vs 23
 9, 10(5,1,4,) vs 19
13, 21(5,1,4,11,) vs 8
29, 29(5,1,4,11,8,) vs 0
 7, 18(5,1,4,8,) vs 11
 5, 17(5,1,11,) vs 12
21, 25(5,1,11,8,) vs 4
 1, 14(5,1,8,) vs 15
11,  9(5,4,) vs 20
11, 20(5,4,11,) vs 9
27, 28(5,4,11,8,) vs 1
 5, 17(5,4,8,) vs 12
 3, 16(5,11,) vs 13
19, 24(5,11,8,) vs 5
 3, 13(5,8,) vs 16
27,  1(1,) vs 28
19,  5(1,4,) vs 24
 3, 16(1,4,11,) vs 13
19, 24(1,4,11,8,) vs 5
 3, 13(1,4,8,) vs 16
 5, 12(1,11,) vs 17
11, 20(1,11,8,) vs 9
11,  9(1,8,) vs 20
21,  4(4,) vs 25
 1, 15(4,11,) vs 14
17, 23(4,11,8,) vs 6
 5, 12(4,8,) vs 17
 7, 11(11,) vs 18
 9, 19(11,8,) vs 10
13,  8(8,) vs 21
1
```

공평한 배분

- 전체 코드

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

int N;
int D[20];
int tot=0;
int ans=1e9;

void dfs(int d, int sum, vector<int> &v);

int main() {
    cin >> N;

    for(int i=0; i<N; i++) {
        cin >> D[i];
        tot += D[i];
    }

    vector<int> v;
    dfs(0, 0, v);
    cout << ans << endl;
    return 0;
}
```

```
void dfs(int d, int sum, vector<int> &v) {
    int diff = abs(sum - (tot-sum));
    //printf("%3d  ", diff);
    //cout << sum << " (";
    //for(int i: v)
    //    cout << i << ", ";
    //cout << ") vs " << (tot-sum) << endl;

    if(diff < ans) {
        ans = diff;
    }

    for(int i=d; i<N; i++) {
        v.push_back(D[i]);
        dfs(i+1, sum+D[i], v);
        v.pop_back();
    }
}
```

```
5
5 1 4 11 8
 29  0 () vs 29
 19  5 (5, ) vs 24
 17  6 (5, 1, ) vs 23
  9  10 (5, 1, 4, ) vs 19
 13  21 (5, 1, 4, 11, ) vs 8
 29  29 (5, 1, 4, 11, 8, ) vs 0
  7  18 (5, 1, 4, 8, ) vs 11
  5  17 (5, 1, 11, ) vs 12
 21  25 (5, 1, 11, 8, ) vs 4
  1  14 (5, 1, 8, ) vs 15
 11  9 (5, 4, ) vs 20
 11  20 (5, 4, 11, ) vs 9
 27  28 (5, 4, 11, 8, ) vs 1
  5  17 (5, 4, 8, ) vs 12
  3  16 (5, 11, ) vs 13
 19  24 (5, 11, 8, ) vs 5
  3  13 (5, 8, ) vs 16
 27  1 (1, ) vs 28
 19  5 (1, 4, ) vs 24
  3  16 (1, 4, 11, ) vs 13
 19  24 (1, 4, 11, 8, ) vs 5
  3  13 (1, 4, 8, ) vs 16
  5  12 (1, 11, ) vs 17
 11  20 (1, 11, 8, ) vs 9
 11  9 (1, 8, ) vs 20
 21  4 (4, ) vs 25
  1  15 (4, 11, ) vs 14
 17  23 (4, 11, 8, ) vs 6
  5  12 (4, 8, ) vs 17
  7  11 (11, ) vs 18
  9  19 (11, 8, ) vs 10
 13  8 (8, ) vs 21
1
```

리모콘1

- 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 ≤ a, b ≤ 40)

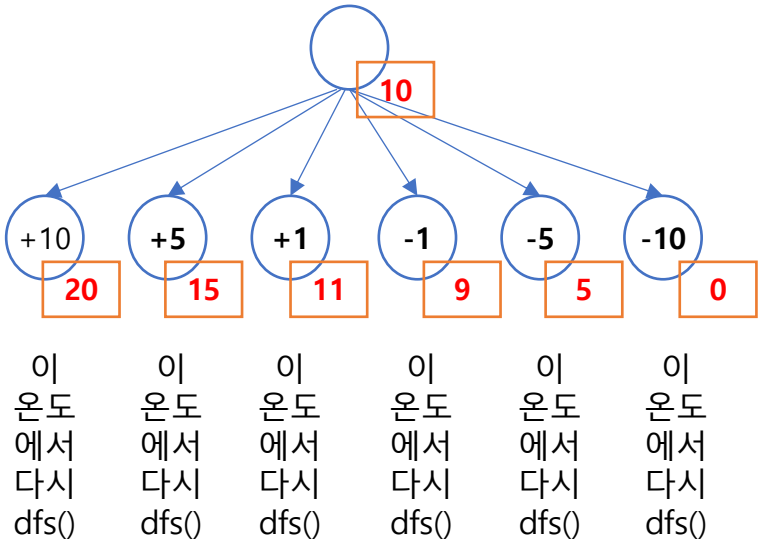
- 출력

  최소한의 버튼 사용으로 목표 온도가 되는 버튼 누름의 횟수를 출력한다.

입력 예	출력 예
7 34	5

리모콘1 (초기설계)

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



  - 재귀호출 해법 가능
  - 계산량: $O(6^{res})$

```
#define  TEMP_LOW    0
#define  TEMP_HIGH  40

int adj[] = { 10, 5, 1, -10, -5, -1 }; //리모컨 버튼으로 바꿈

void dfs(int cnt, int temp) {  // 현재 온도를 추가로 전달
    // 온도 범위를 넘어서는 경우 탐색 중단
    if(temp < TEMP_LOW || temp > TEMP_HIGH) return;

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

    //매 갈림길에서 똑같이 n개로 갈라짐을 구현
    for(int i=0; i<6; i++) {
        seq.push_back(d[i]);
        dfs(cnt+1, temp+adj[i]);
        seq.pop_back();
    }
}
```

리모콘1 (초기설계)

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

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

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

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

```
void dfs(int cnt, int temp) {
    call_cnt++;









}
```

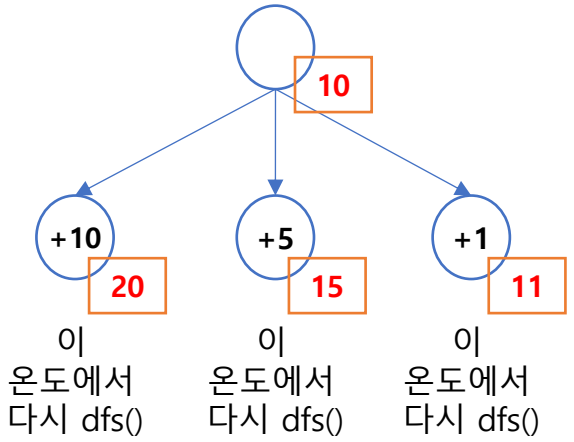
리모콘1 (초기설계)

```
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,  5, -1 (25)
 10, 10, 10,  1,  1,  1,-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,  5, -1 (23)
 10, 10, 10,  1,  1,  1,-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,  5, -1 (21)
 10, 10, 10,  1,  1,  1,-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,  5, -1 (19)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10, 10, -5, -1 (18)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10,-10,  5, -1 (17)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10, 10, -5, -1 (16)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10,-10,  5, -1 (15)
 10, 10, 10,  1,  1,  1,-10, 10,-10, 10, -5, -1 (14)
 10, 10, 10,  1,  1,  1,-10, 10,-10,  5, -1 (13)
 10, 10, 10,  1,  1,  1,-10, 10, 10, -5, -1 (12)
 10, 10, 10,  1,  1,  1,-10, 10,-10,  5, -1 (11)
 10, 10, 10,  1,  1,  1,-10, 10, -5, -1 (10)
 10, 10, 10,  1,  1,  1,-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.
```

리모콘1 (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 dfs()는 3갈래로 뻗어 나감



  - 계산량: $O(3^{res})$로 감소

```
void dfs(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]);
        dfs(cnt+1, temp +ajd[i]);
        v.pop_back();
    }
}
```

리모콘1 (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.
```

리모콘1

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

      :

  - 너비우선 탐색(BFS)의 적용

- 너비우선탐색 알고리즘

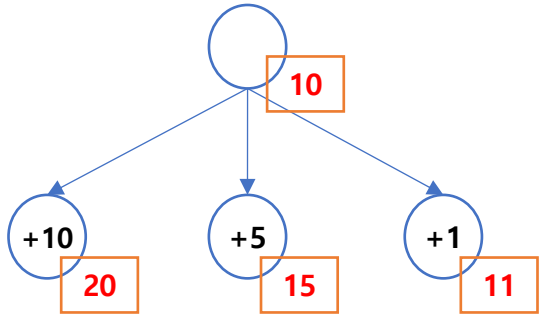
  1) 시작 정점 k를 큐에 삽입

  2) 큐가 빌 때까지 다음을 반복 :

  ① 큐에서 첫 번째 노드를 꺼내어 삭제

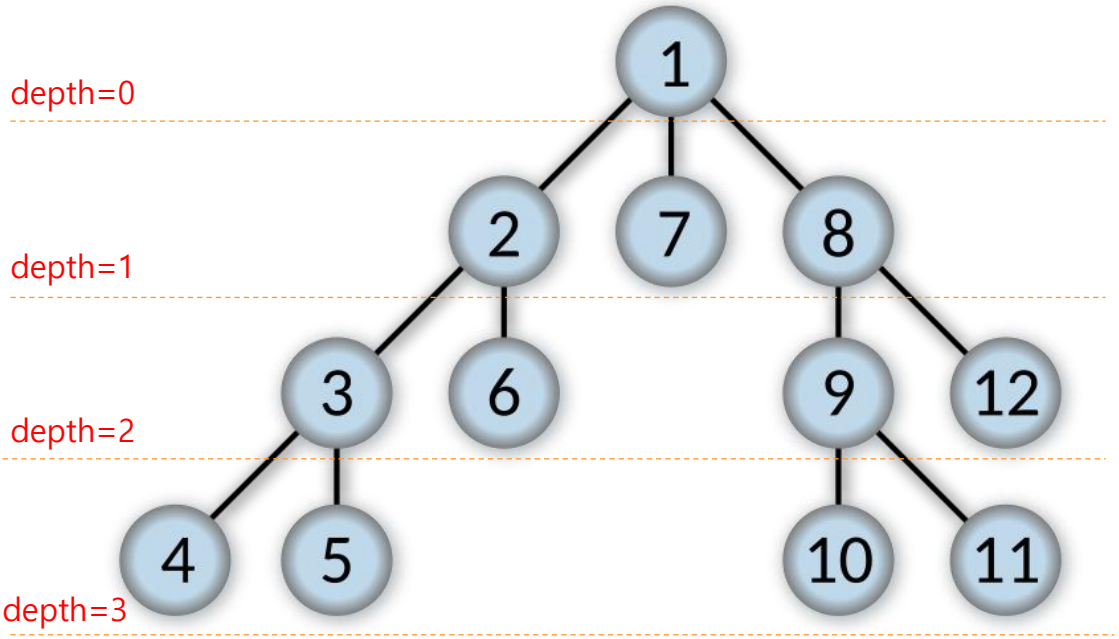
  ② ①에서 꺼낸 노드를 처리

  ③ ①에서 꺼낸 노드와 이웃하는 모든 노드를 큐에 삽입

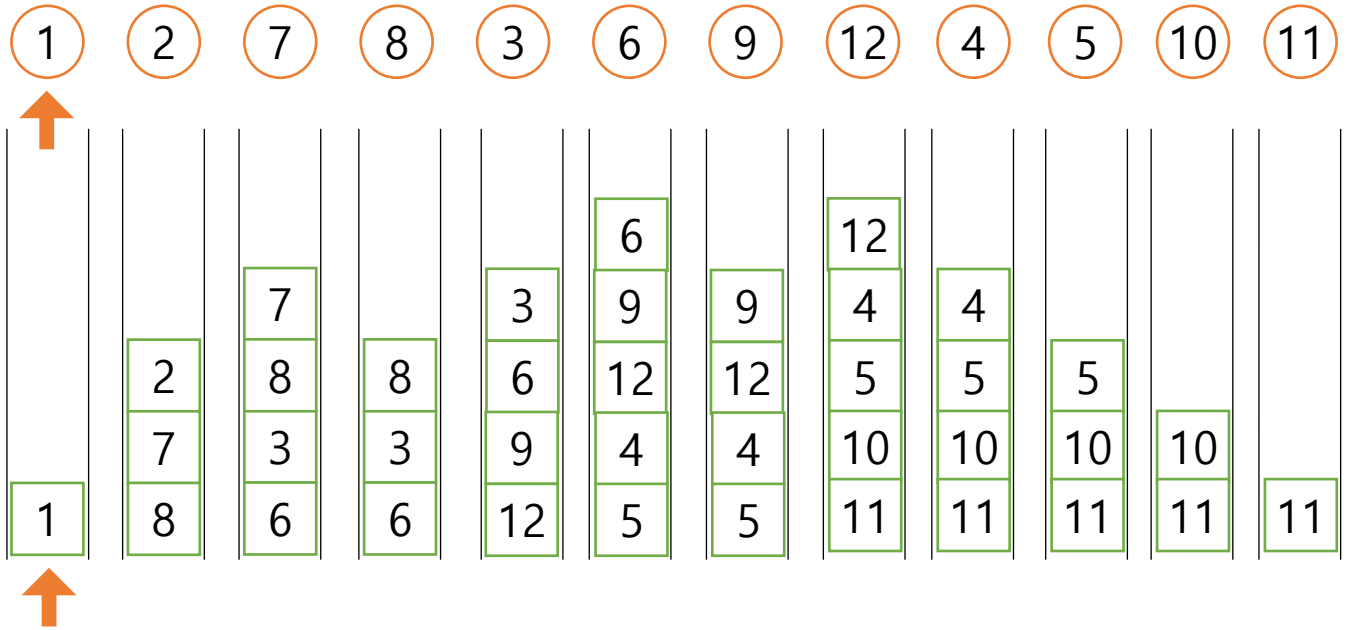


트리구조의 너비우선탐색(BFS)

- 트리 구조



- 큐을 이용한 BFS 순회



- bfs(1)
- bfs(2),bfs(7),bfs(8)
- dfs(3),dfs(6),dfs(9),dfs(12)
- dfs(4),dfs(5),dfs(10),dfs(11)

리모콘1 BFS 풀이

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

struct node {
    int cnt;
    int pre_temp;
    int change;
    int cur_temp;
} node;

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

- **너비우선탐색 알고리즘**
  1) 시작 정점 k를 큐에 삽입
  2) 큐가 빌 때까지 다음을 반복 :
     ① 큐에서 첫 번째 노드를 꺼내어 삭제
     ② ①에서 꺼낸 노드를 처리
     ③ ①에서 꺼낸 노드와 이웃하는 모든 노드를 큐에 삽입

```
int main(void)  {
    int start, target;
    scanf("%d %d", &start, &target);
    printf("(cnt) pre,chg,cur\n");

    Q.push({0, 0, 0, start});
    while(! Q.empty()) {









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

```
int main(void)  {
    int start, target;
    scanf("%d %d", &start, &target);
    printf("(cnt) pre,chg,cur\n");

    Q.push({0, 0, 0, start});
    while(! Q.empty()) {
        node n = Q.front();  // 큐에서 첫번째 항목 획득
        Q.pop();             // 삭제
        printf("(%3d) %3d,%3d,%3d\n",
                n.cnt, n.pre_temp, n.change, n.cur_temp);

        if(n.cur_temp == target) {
            ans = n.cnt;
            break;
        }

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

```
1 8
(cnt) pre,chg,cur
(  0)   0,  0,  1
(  1)   1, 10, 11
(  1)   1,  5,  6
(  1)   1,  1,  2
(  2)  11,-10,  1
(  2)  11, -5,  6
(  2)  11, -1, 10
(  2)   6, 10, 16
(  2)   6,  5, 11
(  2)   6,  1,  7
(  2)   2, 10, 12
(  2)   2,  5,  7
(  2)   2,  1,  3
(  3)   1, 10, 11
(  3)   1,  5,  6
(  3)   1,  1,  2
(  3)   6, 10, 16
(  3)   6,  5, 11
(  3)   6,  1,  7
(  3)  10,-10,  0
(  3)  10, -5,  5
(  3)  10, -1,  9
(  3)  16,-10,  6
(  3)  16, -5, 11
(  3)  16, -1, 15
(  3)  11,-10,  1
(  3)  11, -5,  6
(  3)  11, -1, 10
(  3)   7, 10, 17
(  3)   7,  5, 12
(  3)   7,  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.

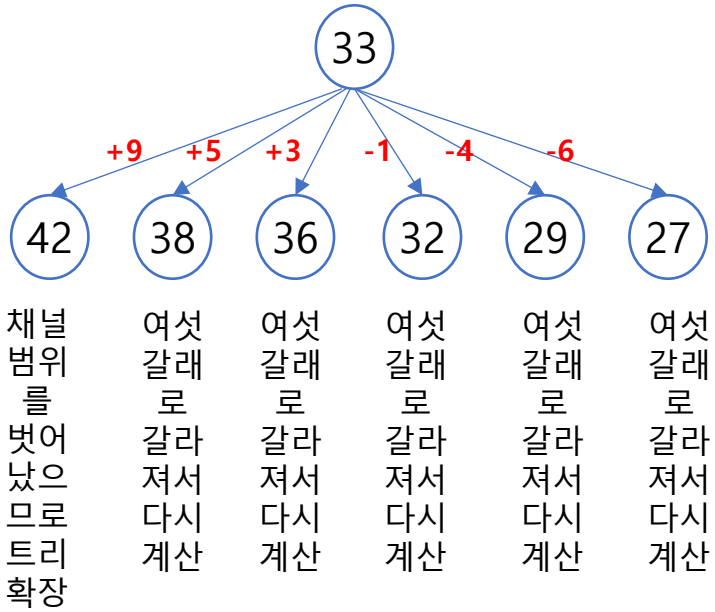
  두 번째 줄에 최소 버튼 조작 횟수로 이동하는 방법의 가짓수를 자연수로 출력한다.

- 입출력의 예

입력 예	출력 예
31 39	2 4
33 40	3 5

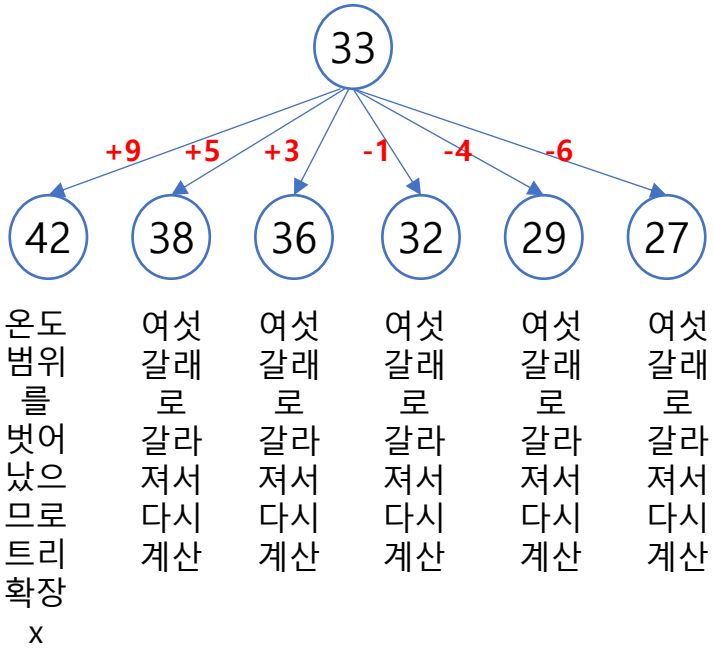
리모콘2

- 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;
}
```

거스름돈Ⅱ (순한맛)

- 문제

  N가지 종류의 화폐가 있다. 이 화폐들을 최소한으로 이용해서 거스름돈 M원을 만들려고 한다.

  이 때 각 화폐는 몇 개라도 사용할 수 있으며, 사용한 화폐의 구성은 같지만 순서만 다른 것은 같은 경우로 구분한다.

  예를 들어 2원, 3원 단위의 화폐가 있을 때, 15원을 만들기 3원을 5개 사용하는 것이 가장 최소한의 화폐 개수이다.

  ※ DFS 탐색으로도 문제를 해결할 수 있도록 낮이도를 낮춤.

- 입력

  첫째 줄에 N, M이 주어진다.

  (1 <= N <= 100, 1 <= M <= 10,000)

  이후의 N개의 줄에는 각 화폐의 가치가 주어진다.

  화폐의 가치는 10,000보다 작거나 같은 자연수 이다.

- 출력

  첫째 줄에 M원을 만들기 위해 필요한 최소 화폐 개수 출력한다.

  불가능할 때는 -1을 출력한다.

입력 예	출력 예
2 15 2 3	5

거스름돈II (DFS + BFS)

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

int N, M, ans=1e9;
int d[100];
vector<int> v;
void dfs(int cnt, int sum);

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

int main(void)  {
    scanf("%d %d", &N, &M);
    for(int i=0; i<N; i++)
        scanf("%d", &d[i]);

    dfs(0, 0);
    if(ans!=1e9) printf("%d\n", ans);
    else         printf("%d\n", -1);
    return 0;
}
```

```
void dfs(int cnt, int sum) {









}

struct node {
    int cnt, coin, sum;
};

void bfs() {
    printf("(cnt), coin, sum\n");







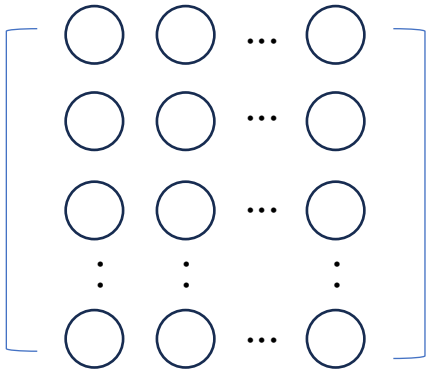


}
```

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

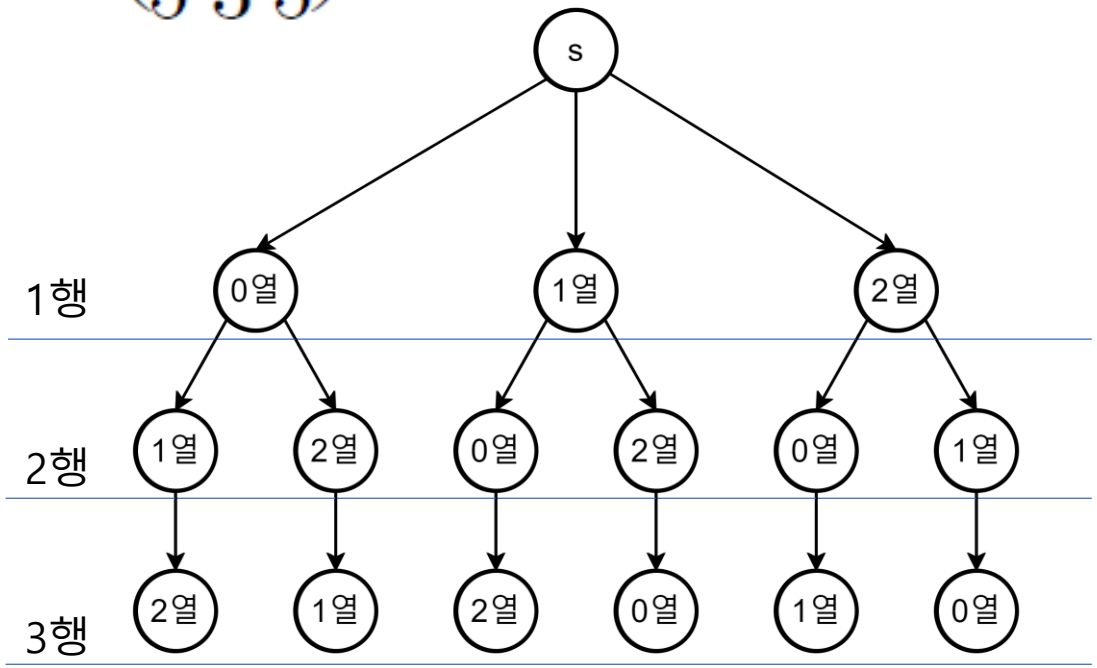
구한 최소 합을 출력한다.

입력 예	출력 예
3 1 5 3 2 4 7 5 3 5	8

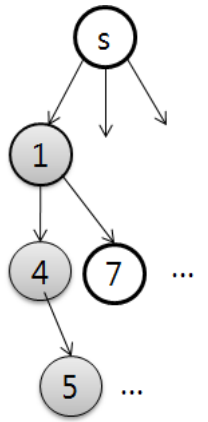
테이블의 최소 합

- 전체탐색

$$\begin{pmatrix} 1 & 5 & 3 \\ 2 & 4 & 7 \\ 5 & 3 & 5 \end{pmatrix}$$

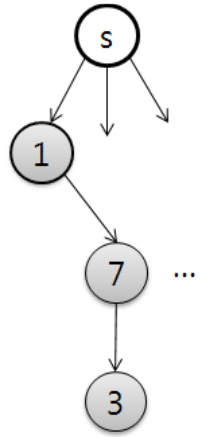


- 처음으로 구한 해 (10)



$$\begin{pmatrix} \textcircled{1} & 5 & 3 \\ 2 & \textcircled{4} & 7 \\ 5 & 3 & \textcircled{5} \end{pmatrix}$$

- 두번째로 구한 해 (11)



$$\begin{pmatrix} \textcircled{1} & 5 & 3 \\ 2 & 4 & \textcircled{7} \\ 5 & \textcircled{3} & 5 \end{pmatrix}$$

테이블의 최소 합 DFS (기본설계)

```
#include <stdio.h>
#include <vector>
#include <limits.h>
#define MAX_N   10
using namespace std;

int n;
int col_used[MAX_N];
int m[MAX_N][MAX_N];
int min_sol=INT_MAX;
vector<int> v;

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() {
    printf("seq: ");
    for(int a : seq)
        printf("%d-", a);
    printf("\b [%d]\n", min_sol);
}
```

```
// row행까지 더한 결과 sum인 상태
void dfs(int row, int sum) {









}

int main() {
    input();
    dfs(0, 0);
    printf("%d\n", min_sol);
    return 0;
}
```

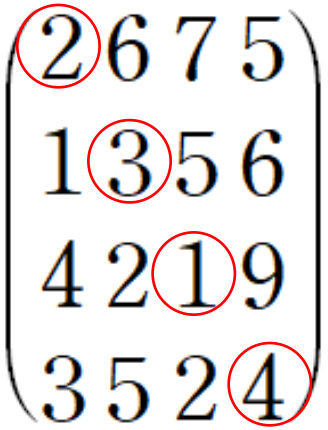
```
3
1 5 3
2 4 7
5 3 5
seq: 1-4-5 [10]
seq: 1-7-3 [10]
seq: 5-2-5 [10]
seq: 5-7-5 [10]
seq: 3-2-3 [8]
seq: 3-4-5 [8]
8
```

테이블의 최소 합

- 탐색배제1: 찾은 답보다 좋을 것
  - 배제 조건

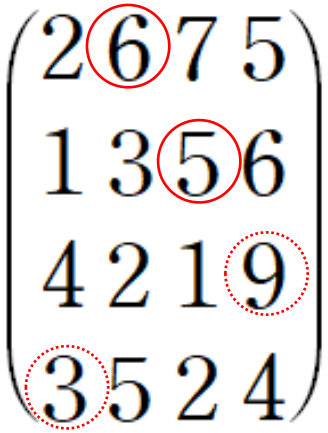
현재까지의 합 > 지금까지의 최소 합

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;
}
```

```
3
1 5 3
2 4 7
5 3 5
seq: 1-4-5 [10]
seq: 3-2-3 [8]
8
```

테이블의 최소 합

- 탐색배제2: 탐욕법으로 사전 스캔
  - ① 1행에서 가장 작은 수를 택하고 다음 행으로 진행한다.
  - ② 다음 행에서 아직까지 선택되지 않은 열 중 가장 작은 수를 택하고 다음 행으로 진행한다.
  - ③ 아직 마지막 행을 마치지 않았으면 2단계로 간다.
  - ④ 지금까지 선택한 수들의 합을 처음 해로 한다.

```
// 탐욕법 사전스캔
int greedy_chk[MAX_N];
void greedy_ans() {
    min_sol=0;

    printf("\nGreedy seq: ");
    for(int r=0; r<n; r++) {  // 모든 행에 대하여
        int min=MAX_INT, k;
        for(int c=0; c<n; c++) {  // 모든 열에 대해
            // 사용한적 없는 열이면서 최소값이면
            if(!greedy_chk[c] && min>m[r][c]) {
                min=m[r][c];
                k=c;
            }
        }
        min_sol+=min;
        greedy_chk[k]=1;
        printf("%d-", k);
    }
    printf("\b [%d]\n", min_sol);
}
```

```
4
8 7 6 2
5 7 9 8
2 8 9 5
4 8 6 1
```

테이블의 최소 합 – 탐색배제1+2 전체 소스코드

```
#include <stdio.h>
#include <deque>
#include <limits.h>
#define MAX_N   10
using namespace std;

deque<int> seq;
int m[MAX_N][MAX_N];
int col_chk[MAX_N];
int n, min_sol=INT_MAX;
int counter=0;

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() {
    printf("seq: ");
    for(int a : seq)
        printf("%d-", a);
    printf("\b ");
}
```

```
void solve(int row, int score) {
   //현재까지의 합 > 지금까지의 최소 합
   // 아래 주석처리시 전체 탐색으로 작동
   if(score>min_sol) return;

    counter++;
    if(row==n) {     // 마지막 행에 도달하면…
        if(score<min_sol)
            min_sol = score;

        output_seq(); // 선택된 열 순서 출력
        printf("[%d]\n", min_sol);
        return;
    }

    // row의 행의 모든 컬럼에 대하여
    for(int c=0; c<n; c++) {
        if(!col_chk[c]) {
            col_chk[c] = 1;  // c열 사용됨 표시
            seq.push_back(c);
            // 다음행 탐색
            solve(row+1, score+m[row][c]);
            seq.pop_back();
            col_chk[c] = 0;  // c열 사용됨 해제
        }
    }
}
```

```
// 탐욕법 사전스캔
int greedy_chk[MAX_N];
void greedy_ans() {
    min_sol=0;

    printf("\nGreedy seq: ");
    for(int r=0; r<n; r++) {  // 모든 행에 대하여
        int min=MAX_INT, k;
        for(int c=0; c<n; c++) {  // 모든 열에 대해
            // 사용한적 없는 열이면서 최소값이면
            if(!greedy_chk[c] && min>m[r][c]) {
                min=m[r][c];
                k=c;
            }
        }
        min_sol+=min;
        greedy_chk[k]=1;
        printf("%d-", k);
    }
    printf("\b [%d]\n", min_sol);
}

int main() {
    input();
    // 아래 주석처리시 탐색배제1만 적용됨
    greedy_ans();

    solve(0, 0);
    printf("min sum: %d\n", min_sol);
    printf("counter: %d\n", counter);
    return 0;
}
```

테이블의 최소 합 – 탐색배제 성능비교

- 성능 비교 테스트 데이터

입력 1	입력 2	입력 3
3 12 76 2 52 77 37 13 67 16	5 93 61 92 56 94 18 32 17 10 64 20 98 85 32 82 1 45 66 77 78 52 11 94 26 57	7 88 51 24 88 94 50 60 14 55 1 23 12 84 91 26 44 81 97 33 82 30 3 71 12 99 16 92 48 87 5 14 93 28 92 56 4 14 92 96 48 41 77 94 32 43 16 1 52 51

- 탐색 횟수 비교 결과

알고리즘	입력 1	입력 2	입력 3
전체탐색	16	326	13700
탐색배제1	10	97	486
탐색배제2	10	79	330

- 배제된 공간 비율

알고리즘	입력 1	입력 2	입력 3
탐색배제1	37.5%	70.25%	99.06%
탐색배제2	37.5%	75.77%	99.08%

케이블 재사용

- 문제

  초고속 인터넷 제공을 위해 각 가정집까지 광케이블을 포설하는 통신업체 KKT는 최근 급격한 원자재값 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;
void dfs(int, int, vector<int> &);

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

    for (int i = 0; i < N; i++) {
        scanf("%d", &L[i]);
    }
    scanf("%d", &T);

    // DFS 탐색 시작
    vector<int> v;
    dfs(0, 0, v);
    printf("%d\n", min_diff);

    return 0;
}
```

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

void dfs(int start, int cur_len, vector<int> &v) {
    output_combi(v, 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 = _______; i < _____; i++) {
        v.push_back(L[i]);
        dfs(i + 1, _____________, v);
        v.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
```

그래프 구조의 자료 표현

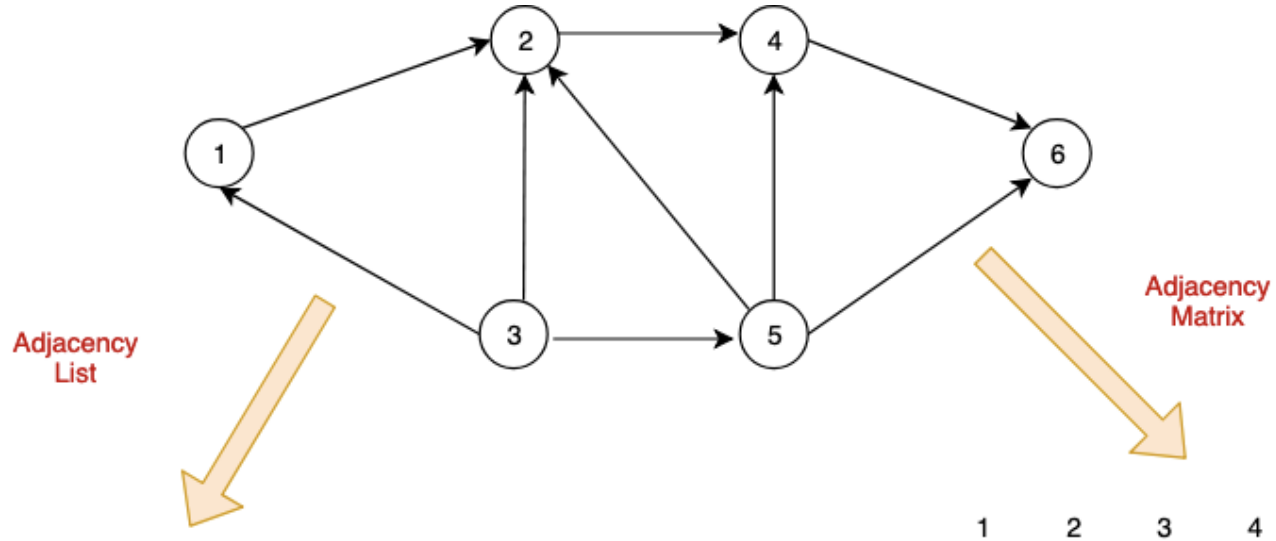
- 그래프의 자료 구현
  - 인접리스트

    (adjacency list)

  - 인접행렬

    (adjacency matrix)

  - 기타방법 …



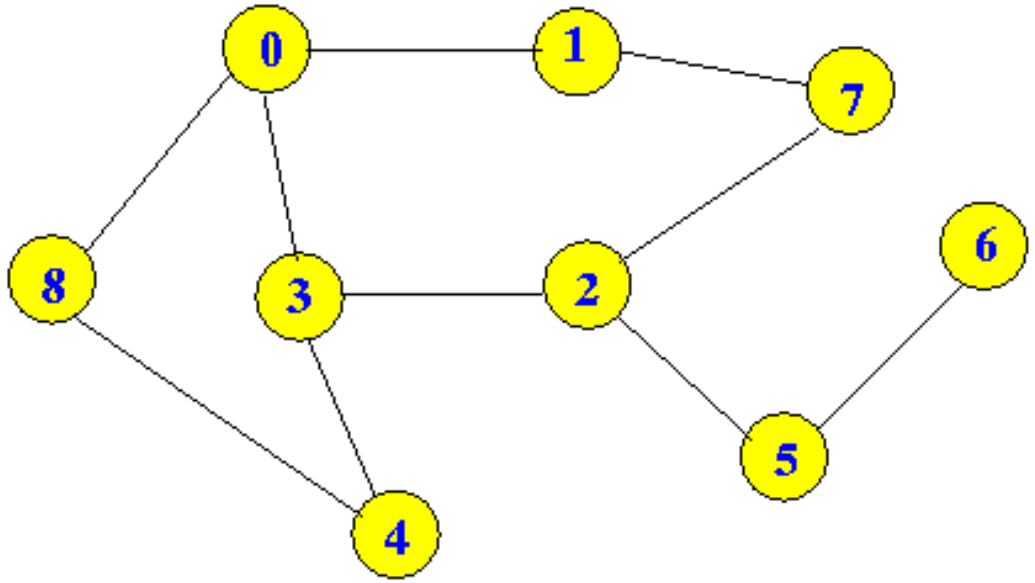
	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	0	1	0	0
3	1	1	0	0	1	0
4	0	0	0	0	0	1
5	0	1	0	1	0	1
6	0	0	0	0	0	0

깊이우선탐색(DFS)

- 그래프의 순회

  트리와 달리 그래프는 사이클이 존재함

  방문정보를 유지하여 재방문을 막아야 함



- 깊이우선탐색 알고리즘

  : go deep(before going wide)

```
def dfs(k):
```

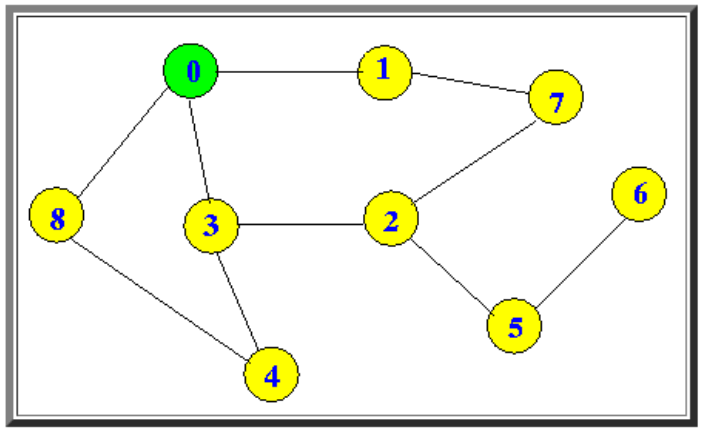
1) 정점 k를 처리하고 방문한 것으로 표시
2) 정점 k와 연결된 모든 정점에 대하여 방문한적이 없으면 그 정점에서 dfs, 완료되면 되돌아오기(백트랙)

image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

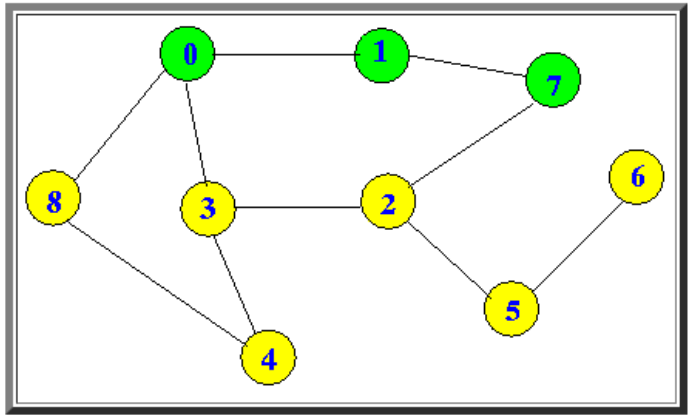
깊이우선탐색(DFS)

- 탐색순서

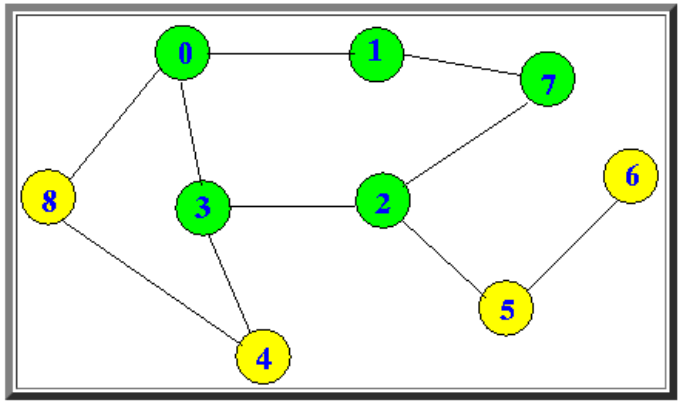
① dfs(0):



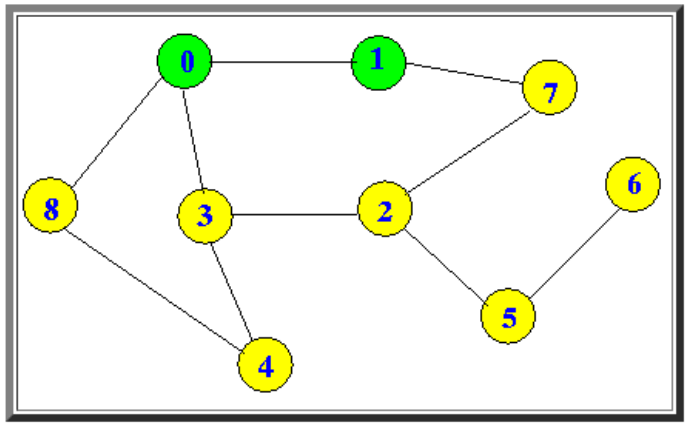
- ~~dfs(1) → dfs(0)~~ (because node 0 is "visited"); dfs(1) → dfs(7)



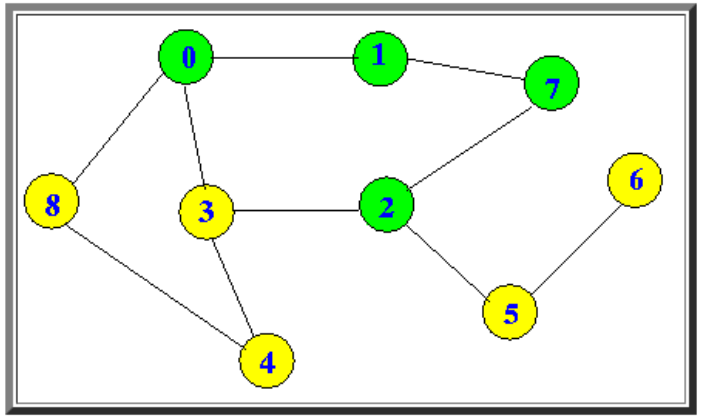
- dfs(2) → dfs(3)



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)

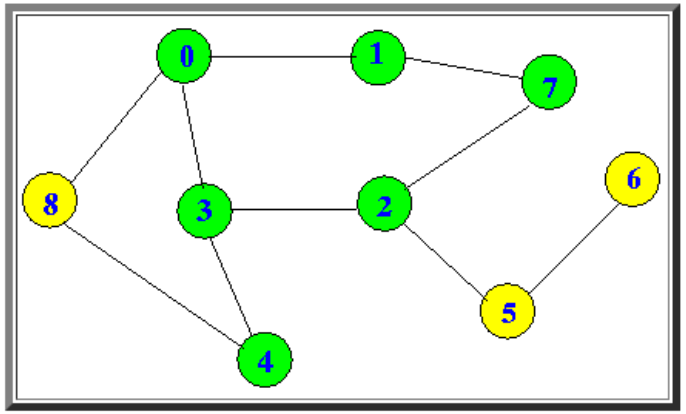
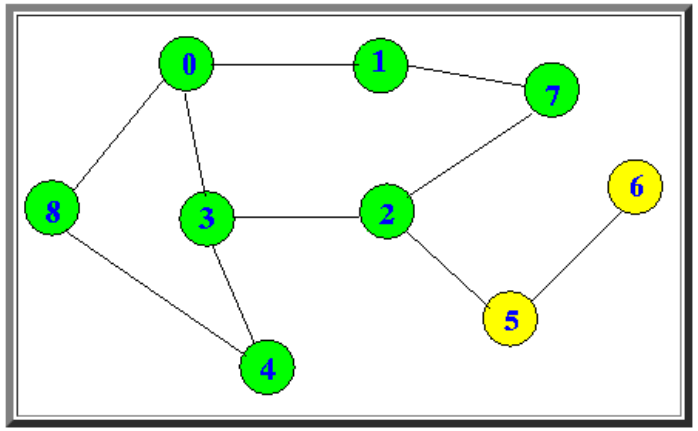


image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

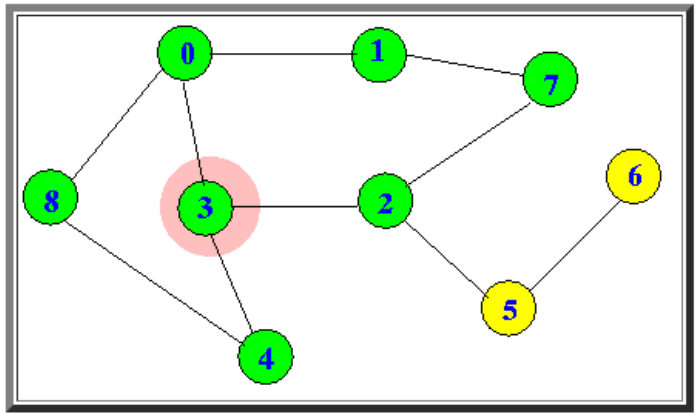
깊이우선탐색(DFS)

- 탐색순서

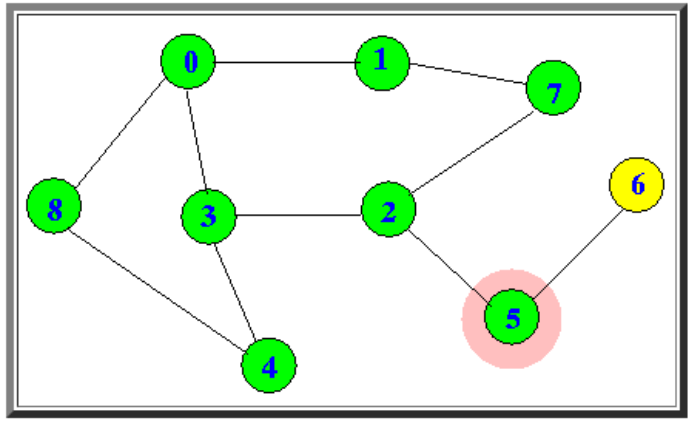
- ~~dfs(4) → dfs(3)~~; dfs(4) → dfs(8)



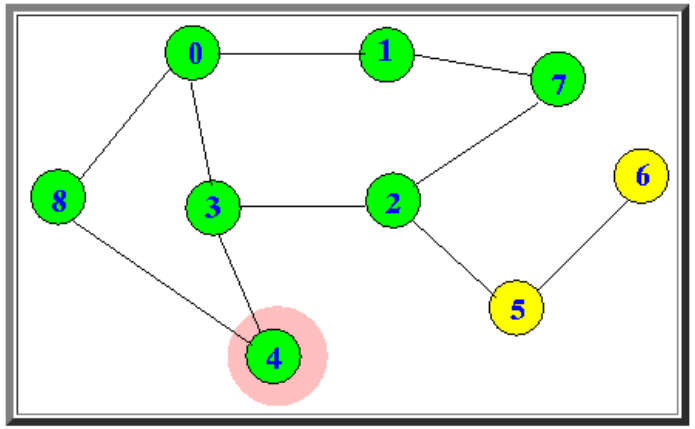
- return to dfs(3)



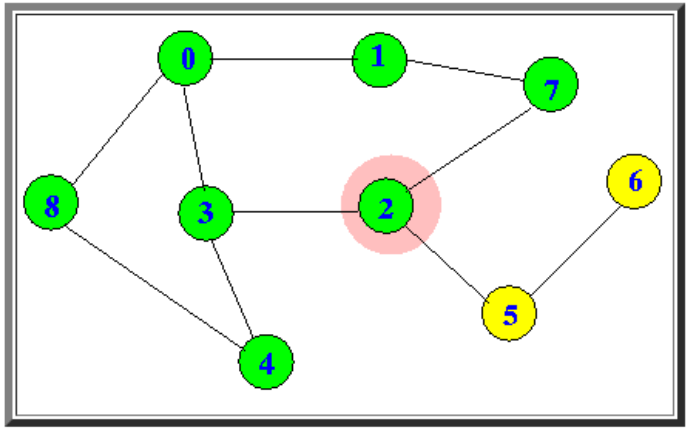
- ~~dfs(2) → dfs(3)~~; dfs(2) → dfs(5)



- ~~dfs(8) → dfs(0)~~; ~~dfs(8) → dfs(4)~~; return to dfs(4)



- return to dfs(2)



- ~~dfs(5) → dfs(2)~~; dfs(5) → dfs(6)

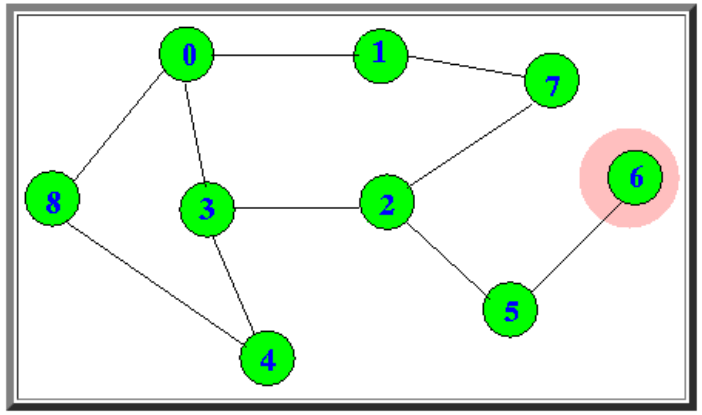
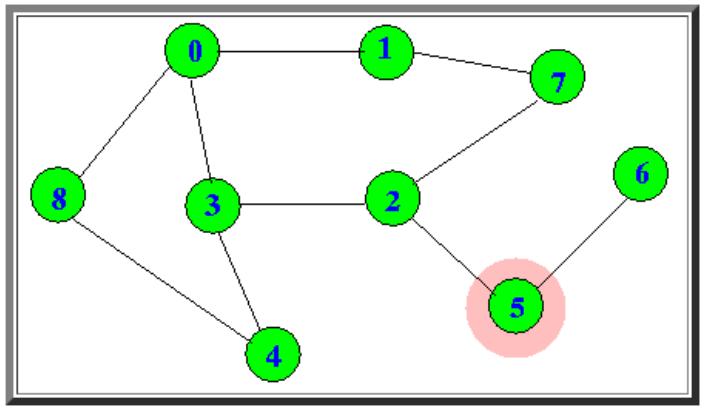


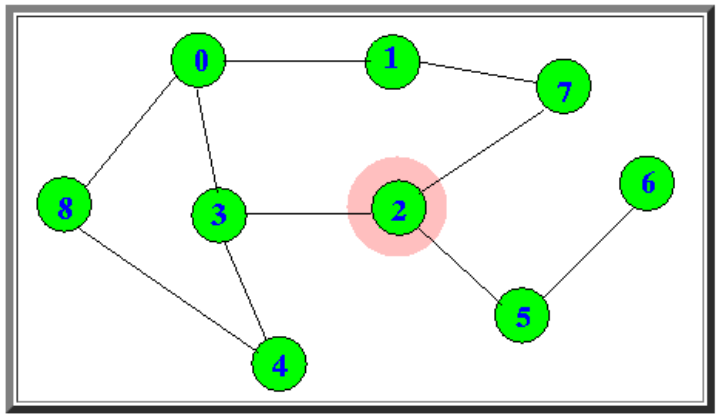
image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

깊이우선탐색(DFS)

- 탐색순서
  - ~~dfs(6) → dfs(5)~~; return to dfs(5)



  - return to dfs(2)



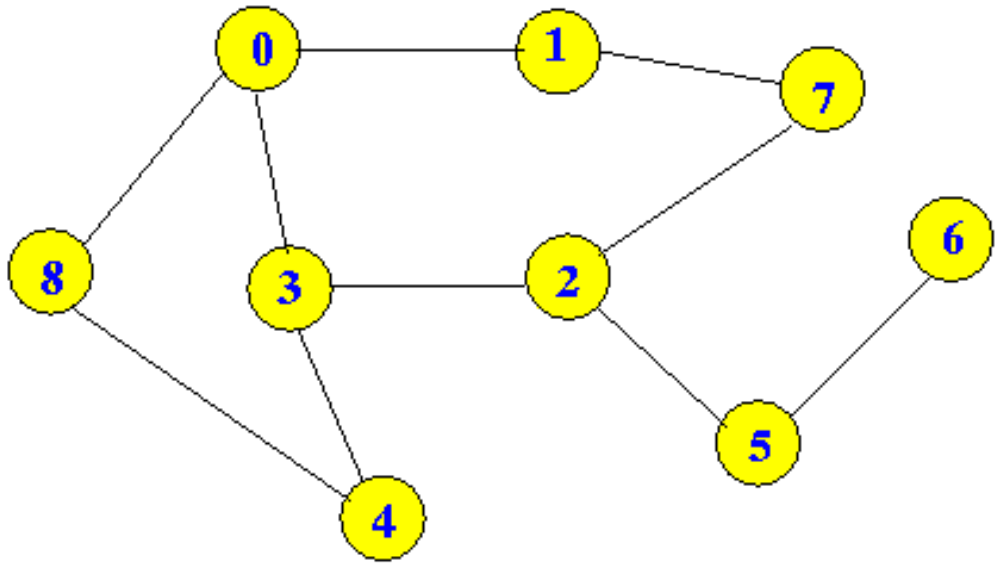
  - return to dfs(7)
  - return to dfs(1)
  - return to dfs(0)

    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);
    }
    else
      printf("(%d) already visited.\n", G[k][i]);
  }
  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_basic_src_quiz.cpp

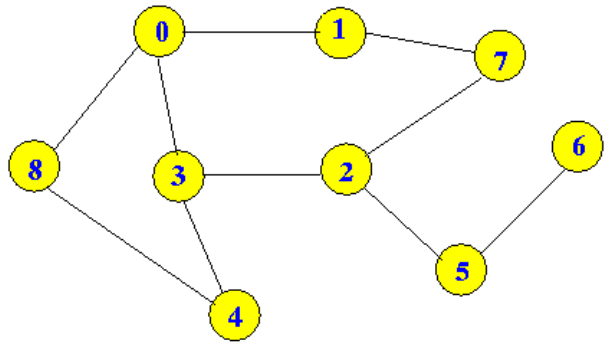
```
#include <stdio.h>
#include <vector>
using namespace std;
int n, m;
vector<vector<int>> G;
vector<bool> visited;

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 idx  [0][1][2]\n");
  for(int a=0; a<G.size(); a++) {
    printf("G[%d]:", 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);
    }
    else
       printf("(%d) already visited.\n", G[k][i]);
  }
  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.
(0) already visited.
dfs(7) started.
(1) already visited.
dfs(2) started.
dfs(3) started.
(0) already visited.
(2) already visited.
dfs(4) started.
(3) already visited.
dfs(8) started.
(0) already visited.
(4) already visited.
return to dfs(4).
return to dfs(3).
return to dfs(2).
dfs(5) started.
(2) already visited.
dfs(6) started.
(5) already visited.
return to dfs(5).
return to dfs(2).
(7) already visited.
return to dfs(7).
return to dfs(1).
return to dfs(0).
(3) already visited.
(8) already visited.
```

깊이우선탐색(DFS) 전체 소스코드

dfs_basic_src_full.cpp

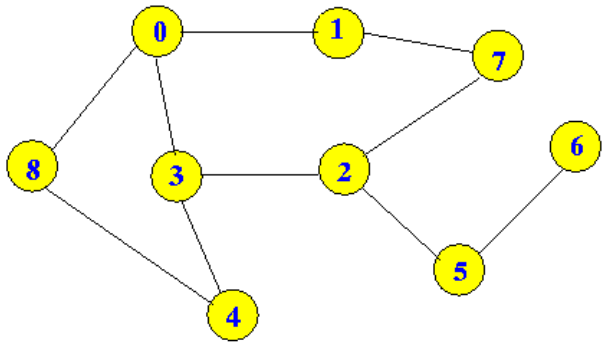
```cpp
#include <stdio.h>
#include <vector>
using namespace std;
int n, m;
vector<vector<int>> G;
vector<bool> visited;

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 idx  [0][1][2]\n");
  for(int a=0; a<G.size(); a++) {
    printf("G[%d]:", a);
    for(int i : G[a])
      printf("%3d", i);
    printf("\n");
  }
  printf("\n");
}
```

```cpp
// 정점 k에서 dfs
void dfs(int k) {
 // 정점 k를 방문하였음을 출력
  printf("dfs(%d) started.\n", k);
  // 나중에 다시오지 않기 위해 방문을 표시하고,
  visited[k] = true;

  // 정점 k와 연결된 모든 정점에 대하여,
  for (int i=0; i<G[k].size(); i++) {
    // 정점k의 i번째 정점에 방문한 적이 없으면,
    if (!visited[G[k][i]]) {
      // 그 정점에서 다시 dfs 시작
      dfs(G[k][i]);
      // dfs 완료하고 돌아왔다고 메시지 출력
      printf("return to dfs(%d).\n", k);
    }
    else
      printf("(%d) already visited.\n", G[k][i]);
  }
  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.
(0) already visited.
dfs(7) started.
(1) already visited.
dfs(2) started.
dfs(3) started.
(0) already visited.
(2) already visited.
dfs(4) started.
(3) already visited.
dfs(8) started.
(0) already visited.
(4) already visited.
return to dfs(4).
return to dfs(3).
return to dfs(2).
dfs(5) started.
(2) already visited.
dfs(6) started.
(5) already visited.
return to dfs(5).
return to dfs(2).
(7) already visited.
return to dfs(7).
return to dfs(1).
return to dfs(0).
(3) already visited.
(8) already visited.
```

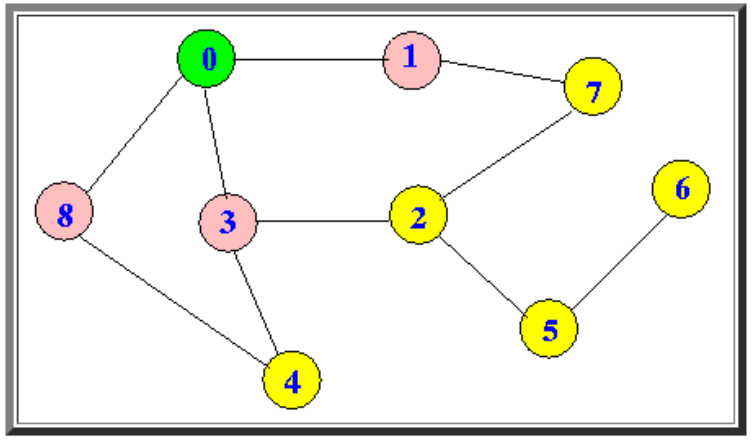
너비우선탐색(BFS)

- 너비우선탐색 알고리즘

  : Breadth First Search

1. 정점 k를 큐에 삽입하고 k를 방문한 것으로 표시
2. 큐가 빌 때까지 다음을 반복:
   1) 큐에서 첫 번째 항목 삭제와 처리
   2) 삭제된 정점과 이웃하는 모든 정점에 대하여 방문하지 않은 정점이라면,
      ① 그 정점을 큐에 삽입
      ② 그 정점에 방문을 표시

- 너비 우선의 의미
  - Visit ***all*** **the neighbors** first:



  - ***Only*** then visit the ***neighbors'*** **neighbors**:

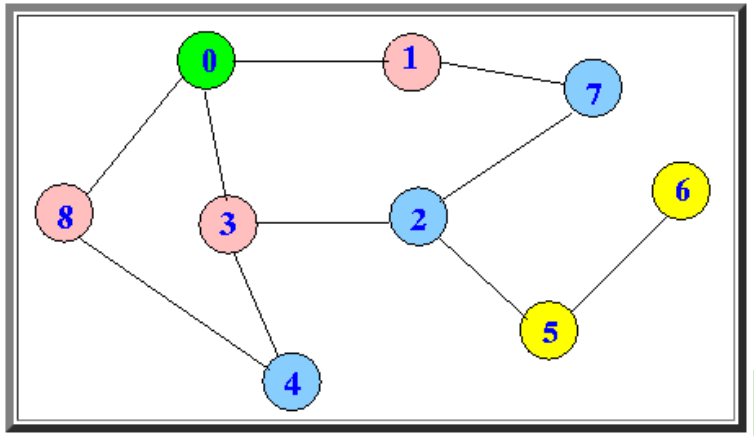
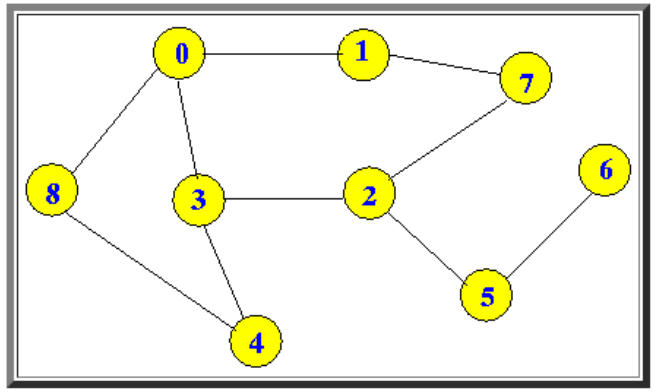


image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

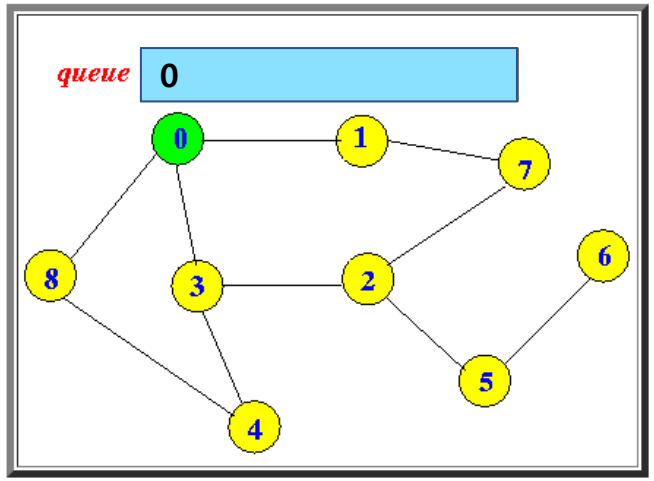
1. 정점 k를 처리하고 큐에 삽입, k를 방문한 것으로 표시
2. 큐가 빌 때까지 다음을 반복:
   1) 큐에서 첫 번째 항목 삭제
   2) 삭제된 항목과 이웃하는 모든 정점에 대하여 방문하지 않은 정점이라면,
      ① 그 정점을 처리하고 큐에 삽입
      ② 그 정점에 방문을 표시

너비우선탐색(BFS)

① Graph:

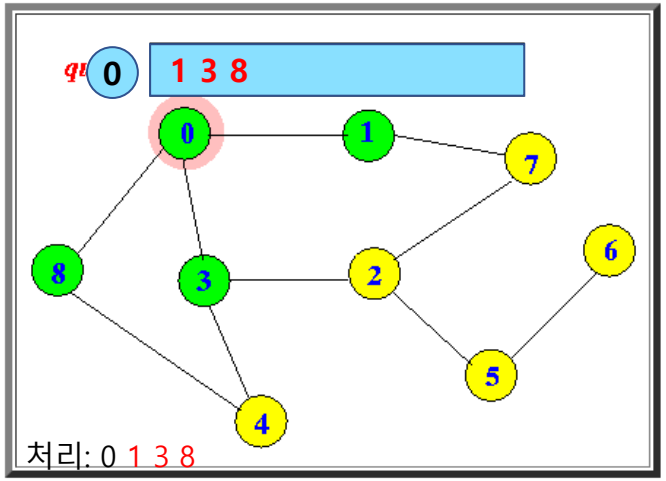


② Initial state:

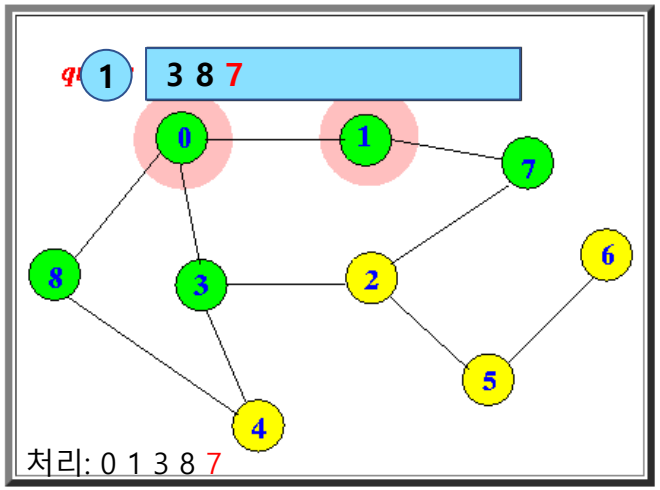


처리: 0

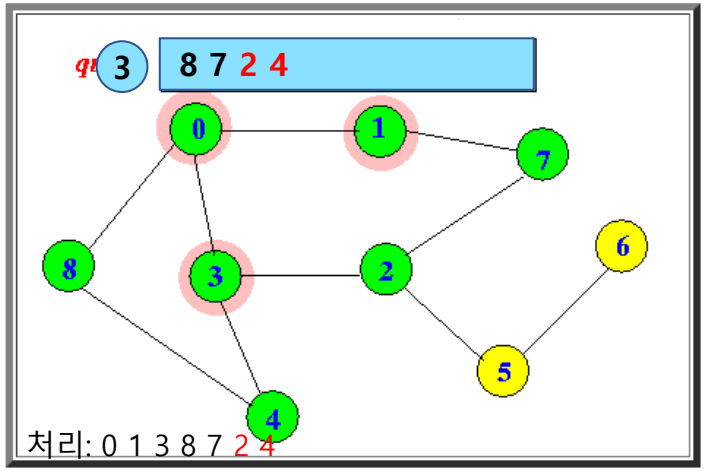
■ State *after* processing 0



■ State *after* processing 1



■ State *after* processing 3



■ State *after* processing 8

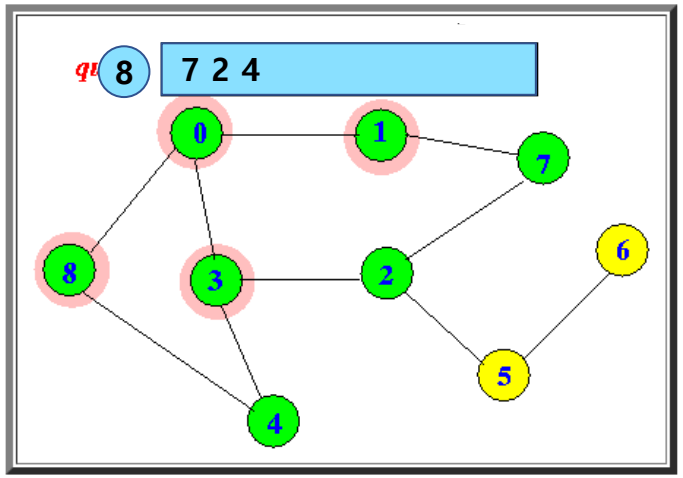
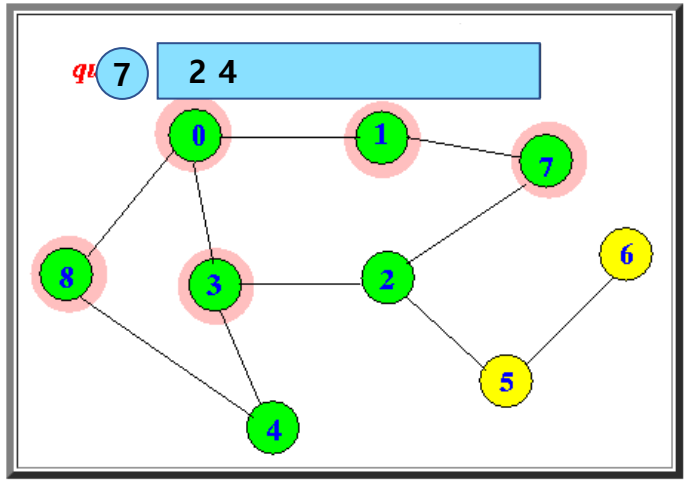


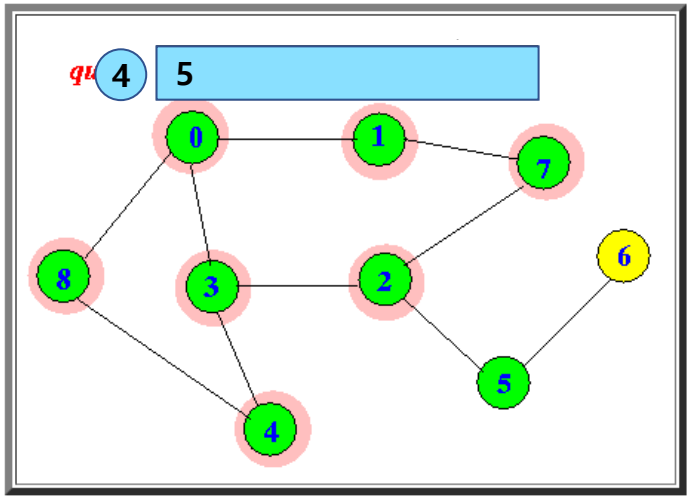
image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

너비우선탐색(BFS)

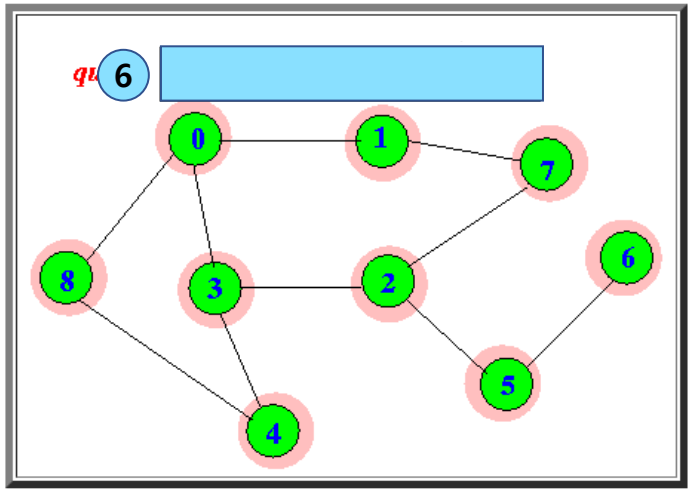
① State *after* processing 7



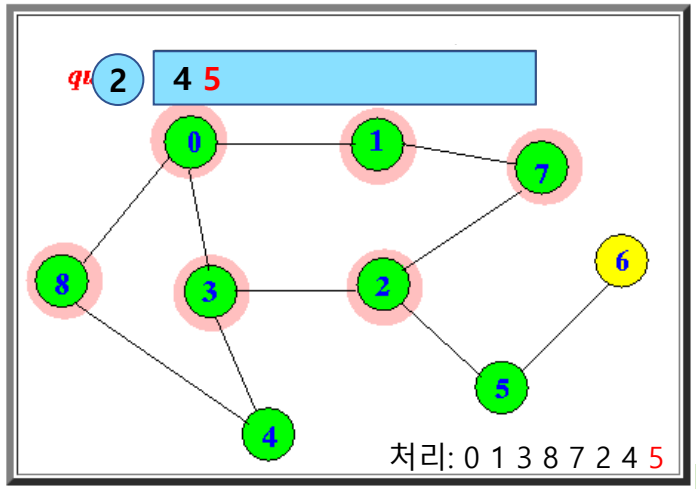
② State *after* processing 4



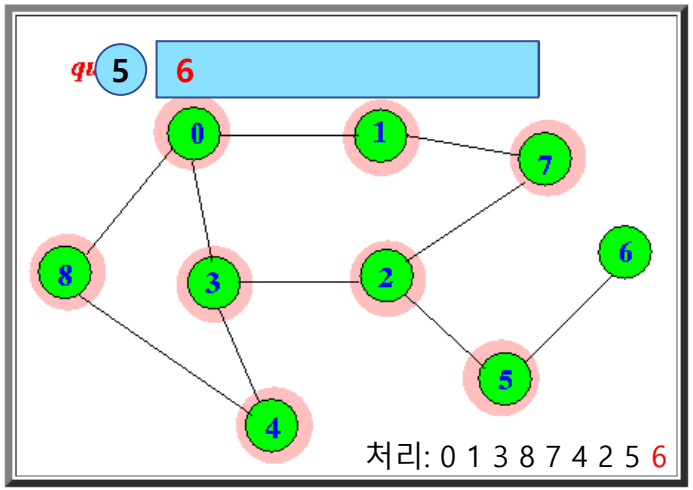
State *after* processing 6



State *after* processing 2



State *after* processing 5



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_basic_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

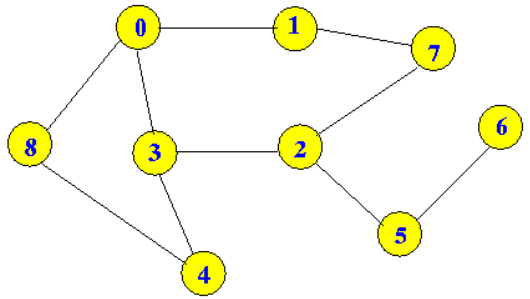
```
// 정점 k에서 bfs
void bfs(int k) {
  queue<int> Q;

  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("bfs:(%d) and deleted\n", cur);

    // 삭제한 정점과 이웃하는 모든 정점에 대하여
    for(int i=0; i<_____________; i++) {
      // 방문한 적이 없으면
      if(!visited[___________]) {
        Q.push(_________);
        visited[_________]=1;
      }
    }
  }
}
```




```
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

Q:[0]
bfs:(0) and deleted
Q:[1,3,8]
bfs:(1) and deleted
Q:[3,8,7]
bfs:(3) and deleted
Q:[8,7,2,4]
bfs:(8) and deleted
Q:[7,2,4]
bfs:(7) and deleted
Q:[2,4]
bfs:(2) and deleted
Q:[4,5]
bfs:(4) and deleted
Q:[5]
bfs:(5) and deleted
Q:[6]
bfs:(6) and deleted
```

너비우선탐색(BFS) 전체 소스코드

bfs_basic_src_full.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

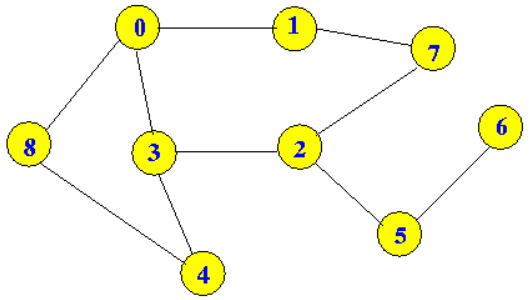
```
// 정점 k에서 bfs
void bfs(int k) {
  queue<int> Q;

  Q.push(k);          // 시작 정점을 Q에 삽입
  visited[k]=1;       // 시작 정점 방문했다고 표시

  while(!Q.empty()) {
    output_Q(Q);

    int cur=Q.front();    // 큐의 첫번째 원소
    Q.pop();              // 빼냄(삭제)
    printf("bfs:(%d) and deleted\n", cur);

    // 삭제한 정점과 이웃하는 모든 정점에 대하여
    for(int i=0; i<G[cur].size(); i++) {
      // 방문한 적이 없으면
      if(!visited[G[cur][i]]) {
        Q.push(G[cur][i]);
        visited[G[cur][i]]=1;
      }
    }
  }
}
```




```
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);
  }
}

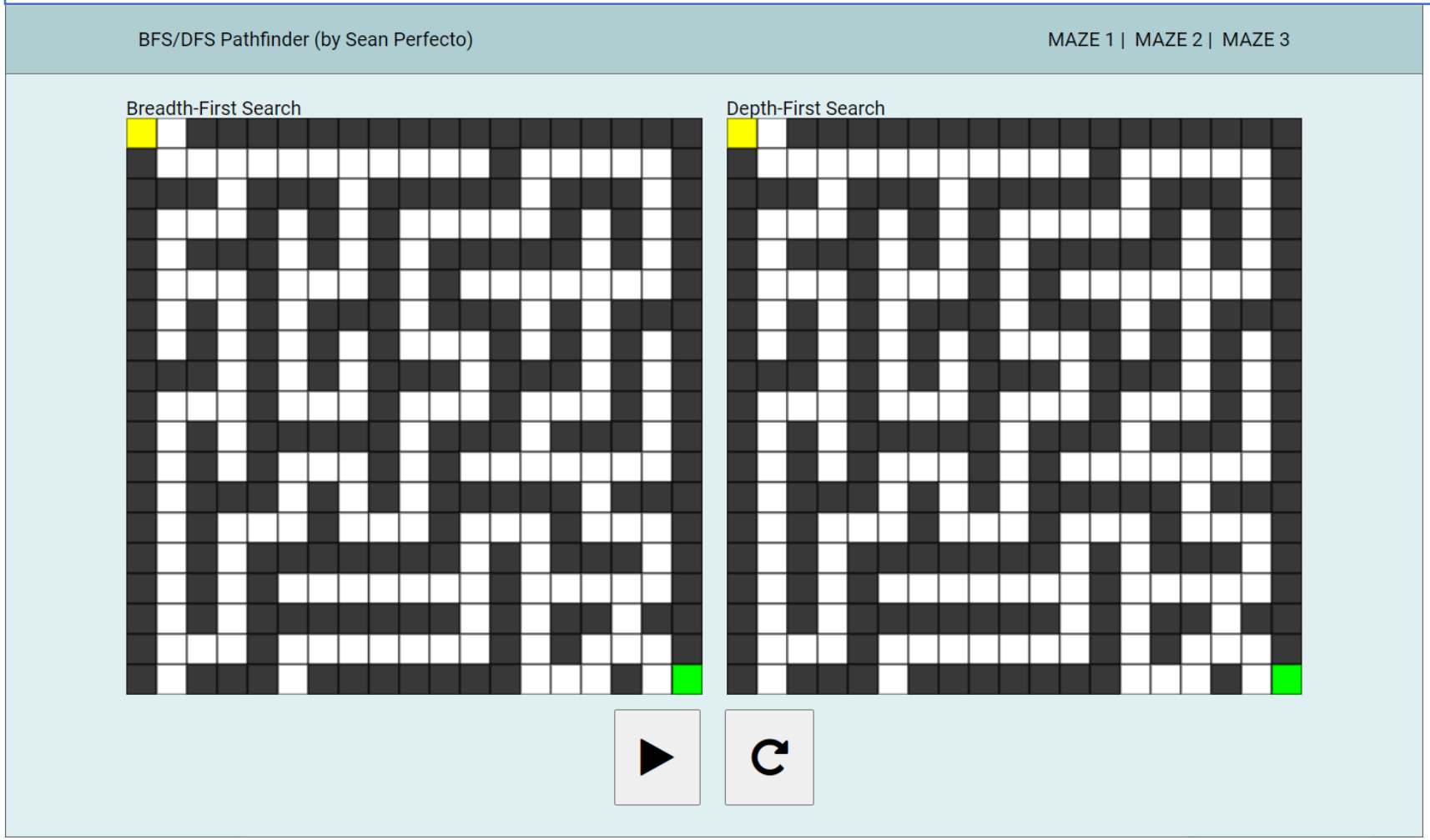
int main() {
  input_G();
  output_G();
  bfs(0);
}
```

```
 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

Q:[0]
bfs:(0) and deleted
Q:[1,3,8]
bfs:(1) and deleted
Q:[3,8,7]
bfs:(3) and deleted
Q:[8,7,2,4]
bfs:(8) and deleted
Q:[7,2,4]
bfs:(7) and deleted
Q:[2,4]
bfs:(2) and deleted
Q:[4,5]
bfs:(4) and deleted
Q:[5]
bfs:(5) and deleted
Q:[6]
bfs:(6) and deleted
```

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

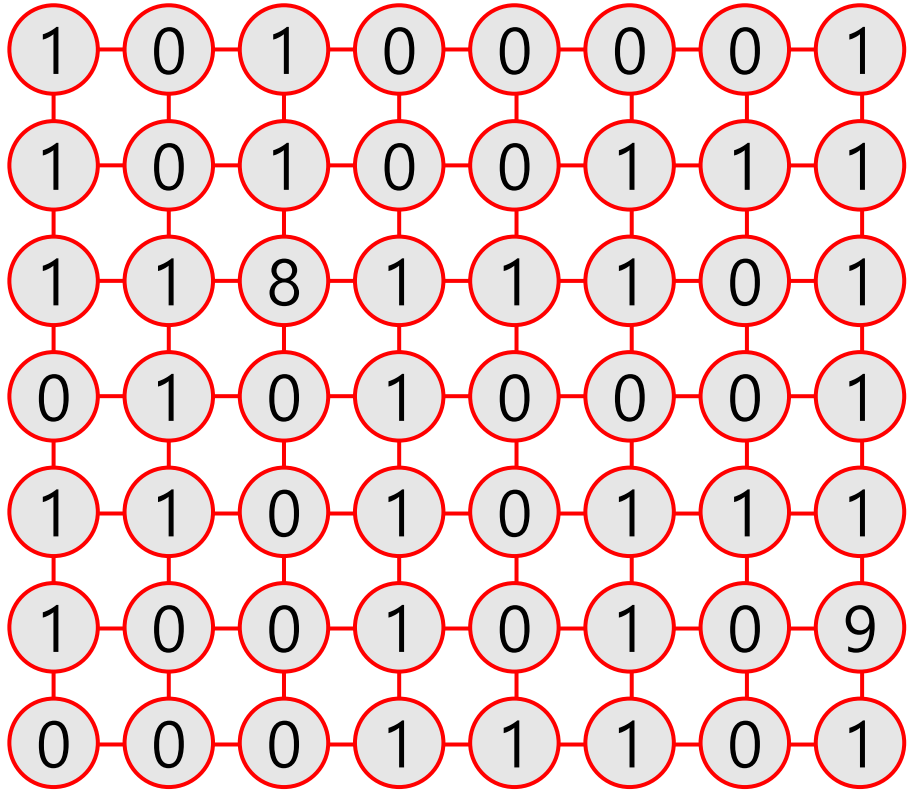
  첫 번째 줄에 최소 이동 칸의 개수를 출력한다.

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 <limits.h>
#include <algorithm>
#include <queue>
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

- 방향 탐색 간략화 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

- 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
    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=INT_MAX;
// 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 큐에 삽입하고, k를 방문한 것으로 표시
2. 큐가 빌 때까지 다음을 반복:
   1) 큐에서 첫 번째 항목 삭제와 처리
   2) 삭제된 정점과 이웃하는 모든 정점에 대하여 방문하지 않은 정점이라면,
      ① 그 정점을 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(시작행sa, 시작열sb, 거리dist)
void bfs(int sa, int sb, int dist) {
    queue<vertex> q;
    [          ?          ]  // 큐에 현재 시작 삽입

    while(!q.empty()) {          // 큐가 빌 때까지 반복
        [          ?          ]  // 큐의 첫 번째 항목 꺼내기
                                 // 큐에서 첫 번째 항목 삭제

        [          ?          ]  // 거리 표시
        // 목적지에 도달하면 성공 출력하고 탈출
        if(v.a==Ga && v.b==Gb) {
            output("BFS search success", v.d);
            [          ?          ]  // 답 저장
        }

        for(int i=0; i<4; i++) {
            [                                                  ?                                                  ]
        }
    }
    output("BFS search fail", -1); // 여기까지 오면 탐색 실패임
}
```

미로 탈출 - BFS로 구현

```
// bfs(시작행sa, 시작열sb, 거리dist)
void bfs(int sa, int sb, int dist) {
    queue<vertex> q;
    q.push({sa, sb, dist});  // 큐에 현재 시작 삽입

    while(!q.empty()) {    // 큐가 빌 때까지 반복
        vertex v = q.front();  // 큐의 첫 번째 항목 꺼내기
        q.pop();               // 큐에서 첫 번째 항목 삭제

        M[v.a][v.b]=v.d;  // 거리 표시
        // 목적지에 도달하면 성공 출력하고 탈출
        if(v.a==Ga && v.b==Gb) {
            output("BFS search success", v.d);
            min_dist = v.d;      // 답 저장
            return;
        }

        for(int i=0; i<4; i++) {
            int na=v.a+da[i];
            int nb=v.b+db[i];
            if(safe(na, nb) && M[na][nb]==-1) {
                q.push({na, nb, v.d+1});  // 다음 정점를 큐에 삽입
                M[na][nb]=-1;  // 방문했음 표시(벽으로 간주)
            }
        }
    }
    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

```
// bfs(시작행sa, 시작열sb, 거리dist)
void bfs(int sa, int sb, int dist) {
    queue<vertex> q;
    q.push({sa, sb, dist});  // 큐에 현재 시작 삽입

    while(!q.empty()) {          // 큐가 빌 때까지 반복
        vertex v = q.front();  // 큐의 첫 번째 항목 꺼내기
        q.pop();               // 큐에서 첫 번째 항목 삭제

        if(v.d>dist)  // 거리가 늘어나면
            output("BFS searched new distance");

        M[v.a][v.b]=dist=v.d; // 거리 표시
        // 목적지에 도달하면 성공 출력하고 탈출
        if(v.a==Ga && v.b==Gb) {
            output("BFS search success", v.d);
            min_dist = v.d;    // 답 저장
            return;
        }
```

```
[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
```

3. maze_bfs_show_process.cpp

미로 탈출 DFS + BFS 전체 소스코드

```
//written by akapo@naver.com
#include <stdio.h>
#include <limits.h>
#include <algorithm>
#include <queue>
using namespace std;

struct vertex {
    int a, b, d;  // a행, b열, 거리d
};

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
int min_dist=INT_MAX;

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(시작행sa, 시작열sb, 거리dist)
void bfs(int sa, int sb, int dist) {
    queue<vertex> q;
    q.push({sa, sb, dist});  // 큐에 현재 시작 삽입

    while(!q.empty()) {   // 큐가 빌 때까지 반복
        vertex v = q.front(); // 큐의 첫 번째 항목 꺼내기
        q.pop();              // 큐에서 첫 번째 항목 삭제

        M[v.a][v.b]=v.d;  // 거리 표시
        // 목적지에 도달하면 성공 출력하고 탈출
        if(v.a==Ga && v.b==Gb) {
            output("BFS search success", v.d);
            min_dist = v.d;    // 답 저장
            return;
        }

        for(int i=0; i<4; i++) {
            int na=v.a+da[i];
            int nb=v.b+db[i];
            if(safe(na, nb) && M[na][nb]==-1) {
                q.push({na, nb, v.d+1});  // 다음 위치를 큐에 삽입
                M[na][nb]=-1;  // 방문했음 표시(벽으로 간주)
            }
        }
    }
    output("BFS search fail", -1); // 여기까지 오면 탐색 실패임
}
```

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

최단경로 이동하기

- 문제

N행 M열로 구성된 사각형 모양의 게임판에서 출발지에서 목적지까지 이동하는 최단거리를 구하고, 최단거리로 이동하는 모든 방법의 갯수를 계산하는 프로그램을 작성하시오.

게임판에서 이동은 상하좌우로만 이동 가능하며, 대각선으로는 이동할 수 없다.

예를 들어 's'는 출발지 'g'는 목적지 'o'는 이동 가능한 칸, 'x'는 이동 불가능한 칸이라고 하자. 만약 아래 그림과 같이 지도가 입력되면, 출발지에서 목적지까지 이동하는 최단거리는 6이고 최단거리로 이동하는 방법은 총 4가지가 존재한다.

<예시>

지도 입력

s	o	o	o
o	o	x	o
o	o	o	g

이동 방법1

1	2	3	4
		X	5
			6

이동 방법2

1	2		
	3	X	
	4	5	6

이동 방법3

1			
2	3	X	
	4	5	6

이동 방법4

1			
2		X	
3	4	5	6

- 출처: 2023 전국상업경진대회 비즈니스프로그래밍 부문 시도예선 8번

최단경로 이동하기

- 입력형식
  - 첫 번째 줄에 게임판의 크기가 N행 M열이 자연수로 입력된다.

    (4 <= N, M <= 50)
  - 두 번째 줄부터 N행 M열의 지도 정보가 입력된다('s'는 출발지 'g'는 목적지 'o'는 이동 가능한 칸, 'x'는 이동 불가능한 칸을 의미, 알파벳은 모두 소문자임)

- 출력형식
  - 첫 번째 줄에 출발지에서 목적지까지 이동하는 최단거리를 정수로 출력한다. (출발지에서 목적지로 가는 길이 없는 입력은 주어지지 않는다)
  - 두 번째 줄에 최단거리로 이동하는 방법의 가짓수를 출력한다.

- 입력과 출력의 예

입력 예2	출력 예2
3 4 sooo ooxo oog	4

- 출처: 2023 전국상업경진대회 비즈니스프로그래밍 부문 시도예선 8번

원하는 물의 양 얻기(water pouring puzzle)

- 문제

  세 개의 컵 A, B, C가 주어지는데 각 a리터, b리터, c리터의 물을 담을 수 있다.

  컵 C에 물을 가득 가득 채운 상태에서 출발하여 세 개의 컵의 물을 이리 저리 옮겨 담아 정확히 원하는 d리터를 물을 최소 몇 번의 교환 만에 얻어 낼 수 있는지 알아내는 프로그램을 작성하시오. 단, 넘치게 물을 부어서 물을 버리는 것은 불허한다.

  예를 들어 컵 A, B, C의 용량이 3L, 7L, 11L 일 때 5L의 물을 얻으려면 C컵에 물을 가득 채운 상태에서 아래와 같이 3번만 교환하면 된다.

  ① 11L 컵에서 3L 컵으로 물을 부어 8L만 남기고

  ② 3L 컵의 물을 7L 비커로 옮기고

  ③ 11L 컵에서 3L 컵로 물을 부으면 11L 비커에 5L만 남게 된다.

- 입력형식

  1) 첫 번째 줄에 세 컵의 용량 a, b, c가 공백으로 분리되어 자연수로 입력된다. .
  (1 ≤ a, b, c ≤ 200)

  2) 두 번째 줄에 원하는 물의 양 d가 자연수로 입력된다. (1 ≤ d ≤ 200)

- 출력형식

  1) d리터를 물을 얻는데 필요한 최소의 교환 횟수를 출력한다.

  2) 만약, 아무리 교환해도 d리터의 물을 얻어낼 수 없다면 –1을 출력한다.

- 입력과 출력의 예

입력 예1	출력 예1
3 7 11 5	3

원하는 물의 양 얻기(기본설계)

3cup_water_pouring_quiz.cpp

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

struct state {
    int A, B, C, cnt;
};

int send[] = {0, 0, 1, 1, 2, 2};
int recv[] = {1, 2, 0, 2, 0, 1};
//A B의 무게만 있으면 C의 무게가 고정되므로 2개로만 체크 가능
bool visited[201][201];
// A B C 물의 양을 저장하는 배열
int cups[3];
int d; // 원하는 물의 양

int BFS();

int main() {
    cin >> cups[0] >> cups[1] >> cups[2];
    cin >> d;
    cout << BFS() << endl;
}
```

```cpp
int BFS() {
    queue<state> q;
    // 처음 상태: A=0, B=0, C=가득, 횟수=0
    q.push({0, 0, cups[2], 0});
    visited[0][0] = true;

    while (!q.empty()) {









    }
}
```

```
int BFS() {
     queue<state> q;
    // 처음 상태: A=0, B=0, C=가득, 횟수=0
    q.push({0, 0, cups[2], 0});
    visited[0][0] = true;

    while (!q.empty()) {
        state cur = q.front();
        q.pop();

        if (cur.A==d || cur.B==d || cur.C==d) {  // 원하는 물의 양을 얻었으면
            return cur.cnt;
        }

        for (int k = 0; k < 6; k++) { // A->B, A->C, B->A, B->C, C->A, C->B 6개의 케이스로 이동
            int next[] = { cur.A, cur.B, cur.C };
            next[recv[k]] += next[send[k]];
            next[send[k]] = 0;
            if (next[recv[k]] > cups[recv[k]]) { // 대상 물통의 용량보다 물이 많아 넘칠 때
                // 초과하는 만큼 다시 이전 물통에 넣어줌
                next[send[k]] = next[recv[k]] - cups[recv[k]];
                next[recv[k]] = cups[recv[k]]; // 대상 물통은 최대로 채워줌
            }

            if (!visited[next[0]][next[1]]) { // A와 B의 물의 양을 통하여 방문 배열 체크
                visited[next[0]][next[1]] = true;

                state neo = {next[0], next[1], next[2], cur.cnt+1};
                q.push(neo);
            }
        }
    }
}
```

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

struct task {
    char from, to;
    int  cnt, A, B, C;
};

struct state {
    int A, B, C, cnt;
    vector<task> his;
};

int send[] = {0, 0, 1, 1, 2, 2};
int recv[] = {1, 2, 0, 2, 0, 1};
//A B의 무게만 있으면 C의 무게가 고정되므로 2개로만 체크 가능
bool visited[201][201];
// A B C 물의 양을 저장하는 배열
int cups[3];
char cups_name[3] = {'A','B','C'};
int d; // 원하는 물의 양
state BFS();

int main() {
    cin >> cups[0] >> cups[1] >> cups[2];
    cin >> d;
    state s = BFS();
    cout << s.cnt << endl;

    vector<task> tv = s.his;
    for(task t : tv)
      printf("(%2d) %c->%c [%d, %d, %d]\n",
              t.cnt, t.from, t.to, t.A, t.B, t.C);
}
```

water_pouring_sol.cpp

```cpp
state BFS() {
    queue<state> q;
    // 처음 상태: A=0, B=0, C=가득, 횟수=0
    q.push({0, 0, cups[2], 0, vector<task>()});
    visited[0][0] = true;

    while (!q.empty()) {
        state cur = q.front();
        q.pop();

        if (cur.A==d || cur.B==d || cur.C==d) {  // 원하는 물의 양을 얻었으면
            return cur;
        }

        // A->B, A->C, B->A, B->C, C->A, C->B 6개의 케이스로 이동
        for (int k = 0; k < 6; k++) {
            int next[] = { cur.A, cur.B, cur.C };
            next[recv[k]] += next[send[k]];
            next[send[k]] = 0;
            if (next[recv[k]] > cups[recv[k]]) { // 대상 물통의 용량보다 물이 많아 넘칠 때
                // 초과하는 만큼 다시 이전 물통에 넣어줌
                next[send[k]] = next[recv[k]] - cups[recv[k]];
                next[recv[k]] = cups[recv[k]]; // 대상 물통은 최대로 채워줌
            }

            if (!visited[next[0]][next[1]]) { // A와 B의 물의 양을 통하여 방문 배열 체크
                vector<task> tv(cur.his.begin(), cur.his.end());
                tv.push_back({cups_name[send[k]], cups_name[recv[k]],
                              cur.cnt+1, next[0], next[1], next[2]});
                state neo = {next[0], next[1], next[2], cur.cnt+1, tv};
                q.push(neo);
                visited[next[0]][next[1]] = true;
            }
        }
    }
}
```

water pouring puzzle 시각화

- Gemini 프롬프트

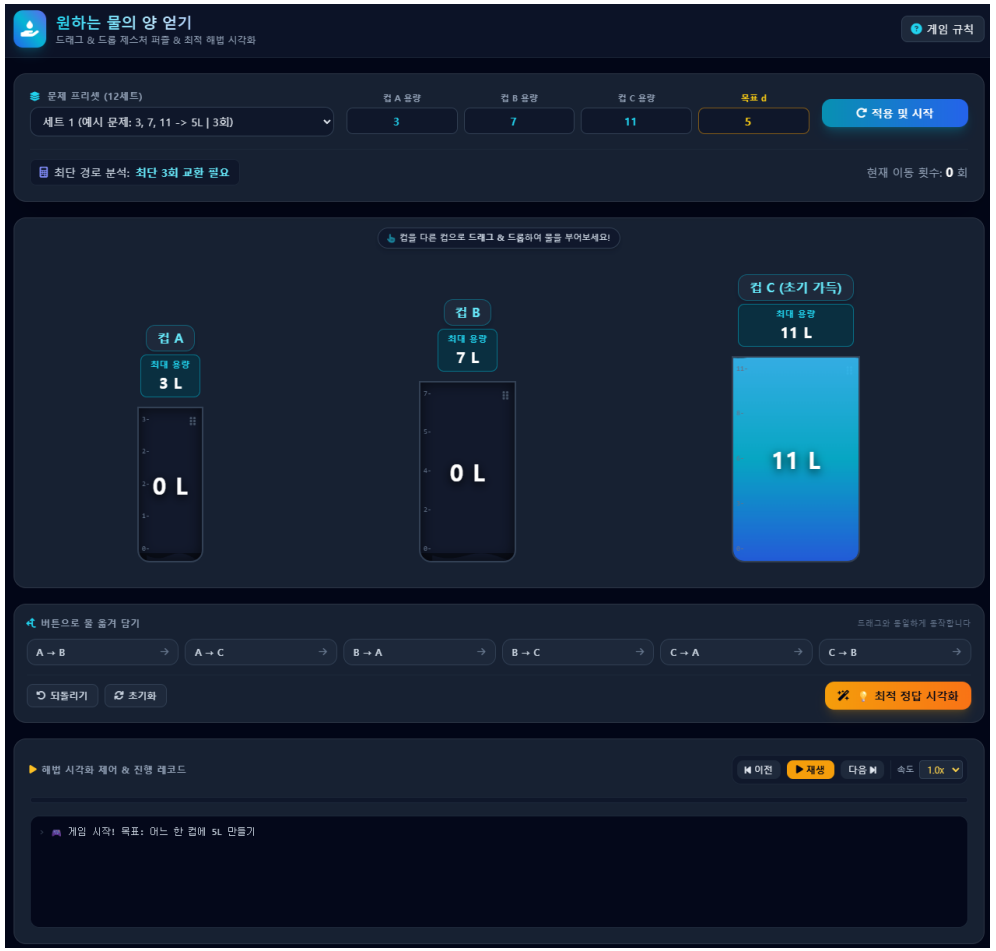
'원하는 물의 양 얻기(water pouring puzzle)' 프로그래밍 퀴즈를 사용자가 직접 플레이 가능한 게임 형태의 프로그램으로 만들어줘. 부가기능으로 답을 찾아주는 기능도 넣어줘. 답을 찾는 과정을 시각해서 보여주어야 해.
답이 존재하는 A,B,C컵의 용량과 목표d 도 10세트이상 준비해줘.

문제
세 개의 컵 A, B, C가 주어지는데 각 a리터, b리터, c리터의 물을 담을 수 있다.
컵 C에 물을 가득 가득 채운 상태에서 출발하여 세 개의 컵의 물을 이리 저리 옮겨 담아 정확히 원하는 d리터를 물을 최소 몇 번의 교환 만에 얻어 낼 수 있는지 알아내는 프로그램을 작성하시오. 단, 넘치게 물을 부어서 물을 버리는 것은 불허한다.
예를 들어 컵 A, B, C의 용량이 3L, 7L, 11L 일 때 5L의 물을 얻으려면 C컵에 물을 가득 채운 상태에서 아래와 같이 3번만 교환하면 된다.
① 11L 컵에서 3L 컵으로 물을 부어 8L만 남기고
② 3L 컵의 물을 7L 비커로 옮기고
③ 11L 컵에서 3L 컵로 물을 부으면 11L 비커에 5L만 남게 된다.
#입력형식
첫 번째 줄에 세 컵의 용량 a, b, c가 공백으로 분리되어 자연수로 입력된다. (1 ≤ a, b, c ≤ 200)
두 번째 줄에 원하는 물의 양 d가 자연수로 입력된다. (1 ≤ d ≤ 200)
#출력형식
d리터를 물을 얻는데 필요한 최소의 교환 횟수를 출력한다.
만약, 아무리 교환해도 d리터의 물을 얻어낼 수 없다면 –1을 출력한다.

water pouring puzzle 시각화

- 대화 기록
  - https://gemini.google.com/share/d/1QW2MnzIiEHy8HQiTzxCbE-dvyl7evguj?usp=sharing
- 결과물
  - https://gifted.datahub.pe.kr/src/visualizer/water_pouring_puzzle.html

- 화면



몬티홀 문제 시뮬레이터 만들기

- 문제 소개
  - https://youtu.be/CBBLZjXs-bU
- 완성된 시뮬레이터
  - https://gifted.datahub.pe.kr/src/visualizer/monty_hall_simulation.html

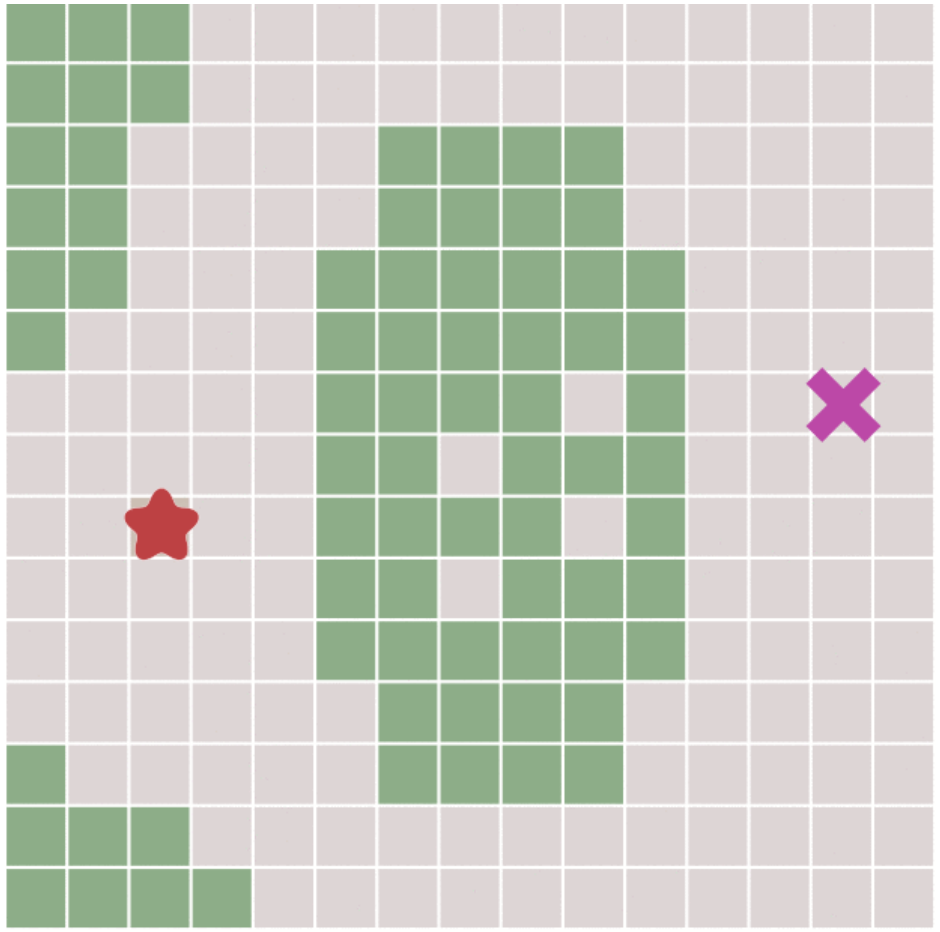
- Gemini 프롬프트

> 방금 전에 급우들과 몬티홀 딜레마에 대한 영상을 보았어.
> 선택을 바꾸는 것이 유리한지 아니면 선택을 고수하는 것이 유리한지 확인시켜주는 시각화 컴퓨터 시뮬레이션 프로그램을 작성해줘.
> 먼저 너의 구현 계획을 나에게 알려주고 내가 OK하면 제작을 시작하렴.

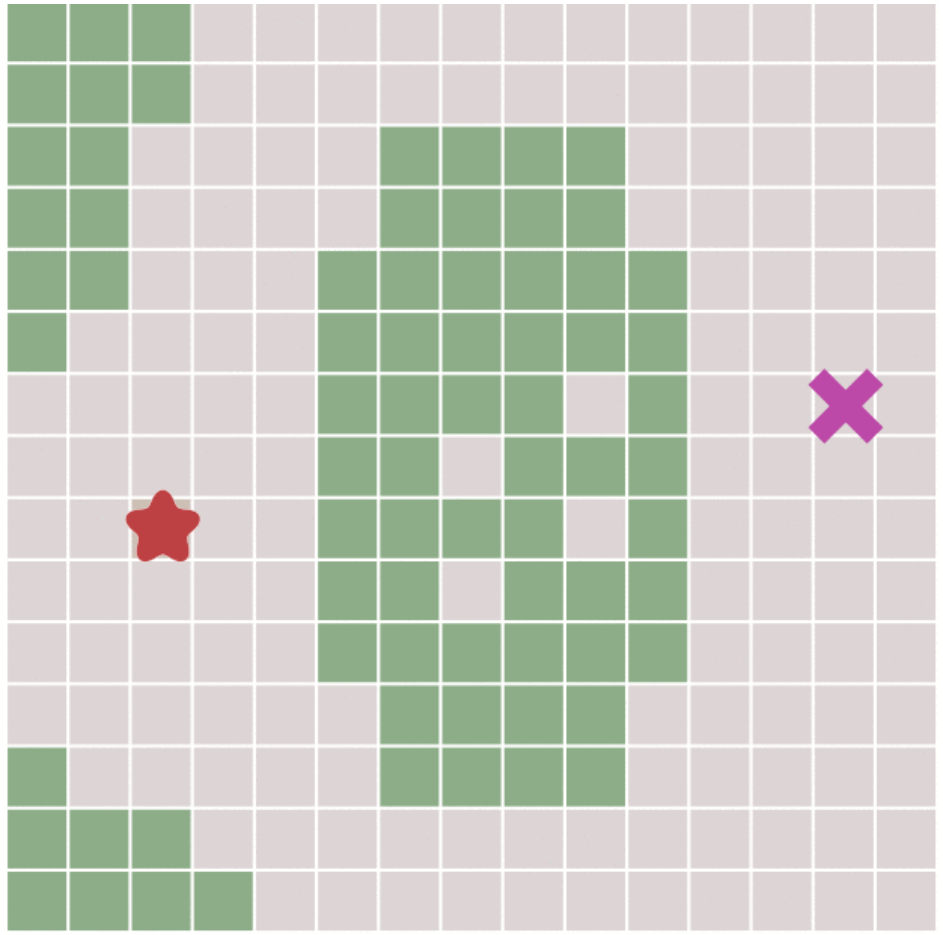
가중치 간선 맵(미로)에서 최소비용 경로 탐색하기

가중치 간선 맵(미로)?

- 동일 가중치 간선 맵



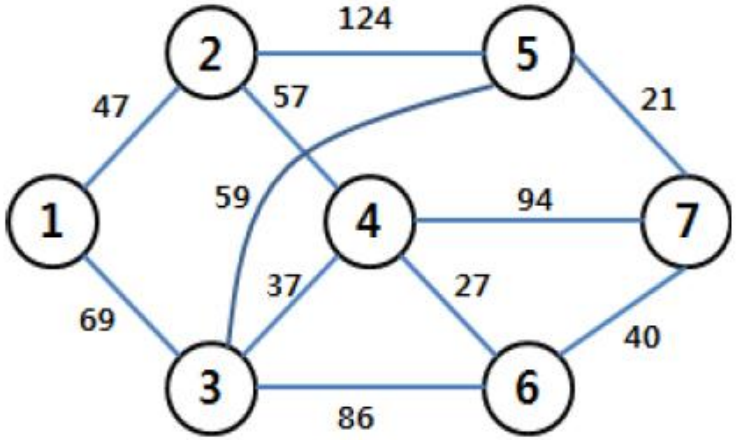
- 상이 가중치 간선 맵



연구활동 가는 길

- 문제

정올이는 GSHS에서 연구활동 교수님을 뵈러 A대학교를 가려고 한다. 출발점과 도착점을 포함하여 지역이 n개, 한 지역에서 다른 지역으로 가는 방법이 총 m개이며 GSHS는 지역 1이고 S대학교는 지역 n이라고 할 때 대학까지 최소 비용을 구하시오.



- 최소 비용이 드는 경로 : 1→3→5→7,
- 최소 비용 : 69+59+21=149

- 입력

첫 번째 줄에는 지점의 수 n과 간선의 수 m이 공백으로 구분되어 입력된다. 다음 줄부터 m줄에 걸쳐서 두 정점의 번호와 가중치 w가 입력된다(자기 간선, 멀티 간선이 있을 수 있다).

(2≤n≤15, 1≤m≤30, 1≤w≤200)

- 출력

대학까지 가는데 드는 최소 비용을 출력한다.

만약 갈 수 없다면 "-1"을 출력한다.

입력 예	출력 예
7 11 1 2 47 1 3 69 2 4 57 2 5 124 3 4 37 3 5 59 3 6 86 4 6 27 4 7 94 5 7 21 6 7 40	149

연구활동 가는 길

- 초기 설계

research_path_try1.cpp

```
#include <stdio.h>
#include <limits>
#include <iostream>
#include <vector>
#define MAX_V   15   // 최대 정점 개수
using namespace std;

int n, m;
int G[MAX_V+1][MAX_V+1];  // 연결정보를 저장하는 인접행렬
int visited[MAX_V+1];  // 방문여부를 저장하는 배열
int sol=INT_MAX;
vector<int> path;   // 방문 정점 순서를 기록하는 벡터

void input() {
  scanf("%d %d", &n, &m);
  for(int i=0; i<m; i++) {
    int s, e, w;
    scanf("%d %d %d", &s, &e, &w);
    G[s][e] = G[e][s] = w;
  }
}
```

```
void output(int W) {
    for(int v : path)
        printf("%d-", v);
    printf("\b [%d]\n", W);
}

// 정점 V에서 목적지까지 가는 거리를 계산해라,
// 현재까지 이동 비용은 W이다.
void dfs(int V, int W) {
    // 세부구현 필요

}

int main() {
  input();
  dfs(1, 0);  // 정점 1, 거리 0에서 시작
  printf("\nmin: %d\n", sol==INT_MAX ? -1 : sol);
  return 0;
}
```

연구활동 가는 길

- solve() 함수 구현 – DFS

```
def dfs(k):
1) 정점 k를 처리하고 방문한 것으로 표시
2) k와 연결된 모든 정점에 대하여
   방문한적이 없으면 그 정점에서 dfs,
   dfs가 완료되면 되돌아 오기
```

V \ i	1	2	3	4	5	6	7
1		47	69				
2	47			57	124		
3	69			37	59	86	
4		57	37			27	94
5		124	59				21
6			86	27			40
7				94	21	40	

```
// 정점 V에서 목적지까지 가는 거리를 계산해라,
// 현재까지 이동거리는 W이다.
void dfs(int V, int W) {
  path.push_back(V);
  [          ?          ] // 방문함을 표시
  // 마지막 n 위치에 도달했으면...
  if(V == n)  {
    // 더 짧은 거리를 찾았으면 업데이트 함.
    [              ?              ]
    output(W);
    return;
  }

  for(int i=1; i<=n; i++) {
    // i를 방문한 적이 없고, 현재V와 i가 연결되어 있다면,
    // (0이 아니면 연결되어 있다는 뜻)...
    [                                ]  {
    [                                ] // 정점 i에서 dfs
    [              ?                 ] // 백트랙시 방문정보 해제
    [                                ]
    [                                ]
  }
  return;
}
```

연구활동 가는 길 전체탐색(DFS) - 소스코드

```
#include <stdio.h>
#include <limits>
#include <iostream>
#include <vector>
#define MAX_V   15   // 최대 정점 개수
using namespace std;

int n, m;
int G[MAX_V+1][MAX_V+1];
int visited[MAX_V+1];
int sol=INT_MAX;
vector<int> path;

void input() {
  scanf("%d %d", &n, &m);
  for(int i=0; i<m; i++) {
    int s, e, w;
    scanf("%d %d %d", &s, &e, &w);
    G[s][e] = G[e][s] = w;
  }
}

void output(int W) {
    for(int v : path)
        printf("%d-", v);
    printf("\b [%d]\n", W);
}
```

```
void dfs(int V, int W) {
  if(V == n)  { // 마지막 n 위치에 도달했으면...
    // 더 짧은 거리를 찾았으면 업데이트 함.
    if(W < sol) sol = W;
    output(W);
    return;
  }

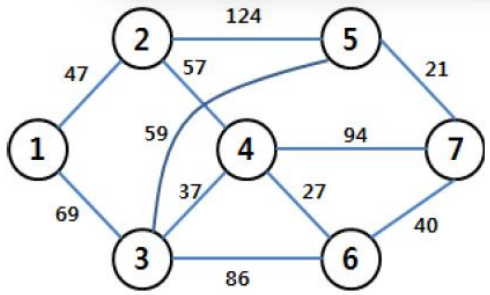
  for(int i=1; i<=n; i++) {
    // i를 방문한 적이 없고, 현재V와 i가 연결되어 있다면,
    // (0이 아니면 연결되어 있다는 뜻)...
    if(!chk[i] && G[V][i])  {
      path.push_back(i);
      visited[i]=1;
      dfs(i, W+G[V][i]);
      visited[i]=0;
      path.pop_back();
    }
  }
}

int main() {
  input();

  path.push_back(1);
  visited[1]=1;
  dfs(1, 0); // 정점 1, 거리 0에서 시작
  printf("\nmin: %d\n", sol==INT_MAX ? -1 : sol);
  return 0;
}
```

```
1-2-4-3-5-7 [221]
1-2-4-3-6-7 [267]
1-2-4-6-3-5-7 [297]
1-2-4-6-7 [171]
1-2-4-7 [198]
1-2-5-3-4-6-7 [334]
1-2-5-3-4-7 [361]
1-2-5-3-6-4-7 [437]
1-2-5-3-6-7 [356]
1-2-5-7 [192]
1-3-4-2-5-7 [308]
1-3-4-6-7 [173]
1-3-4-7 [200]
1-3-5-2-4-6-7 [376]
1-3-5-2-4-7 [403]
1-3-5-7 [149]
1-3-6-4-2-5-7 [384]
1-3-6-4-7 [276]
1-3-6-7 [195]

min: 149
```



연구활동 가는 길 전체탐색(DFS) - 소스코드

research_path_try2.cpp

```
#include <stdio.h>
#include <limits>
#include <iostream>
#include <vector>
#define MAX_V   15   // 최대 정점 개수
using namespace std;

int n, m;
int G[MAX_V+1][MAX_V+1];
int visited[MAX_V+1];
int sol=INT_MAX;
vector<int> path;

void input() {
  scanf("%d %d", &n, &m);
  for(int i=0; i<m; i++) {
    int s, e, w;
    scanf("%d %d %d", &s, &e, &w);
    G[s][e] = G[e][s] = w;
  }
}

void output(int W) {
    for(int v : path)
        printf("%d-", v);
    printf("\b [%d]\n", W);
}
```

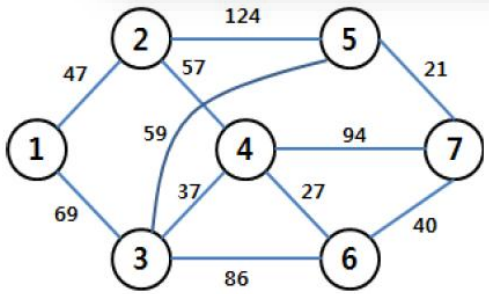
```
// 정점 V에서 목적지까지 가는 거리를 계산해라,
// 현재까지 이동거리는 W이다.
void dfs(int V, int W) {
  path.push_back(V);
  visited[V] = 1;
  // 마지막 n 위치에 도달했으면...
  if(V == n)  {
    // 더 짧은 거리를 찾았으면 업데이트 함.
    if(W < sol) sol = W;
    output(W);
    return;
  }

  for(int i=1; i<=n; i++) {
    // i를 방문한 적이 없고, 현재V와 i가 연결되어 있다면,
    // (0이 아니면 연결되어 있다는 뜻)...
    if(!visited[i] && G[V][i])  {
      dfs(i, G[V][i]+W);
      visited[i] = 0;
      path.pop_back();
    }
  }
}

int main() {
  input();
  dfs(1, 0);  // 정점 1, 거리 0에서 시작
  printf("\nmin: %d\n", sol==INT_MAX ? -1 : sol);
  return 0;
}
```

```
1-2-4-3-5-7 [221]
1-2-4-3-6-7 [267]
1-2-4-6-3-5-7 [297]
1-2-4-6-7 [171]
1-2-4-7 [198]
1-2-5-3-4-6-7 [334]
1-2-5-3-4-7 [361]
1-2-5-3-6-4-7 [437]
1-2-5-3-6-7 [356]
1-2-5-7 [192]
1-3-4-2-5-7 [308]
1-3-4-6-7 [173]
1-3-4-7 [200]
1-3-5-2-4-6-7 [376]
1-3-5-2-4-7 [403]
1-3-5-7 [149]
1-3-6-4-2-5-7 [384]
1-3-6-4-7 [276]
1-3-6-7 [195]

min: 149
```

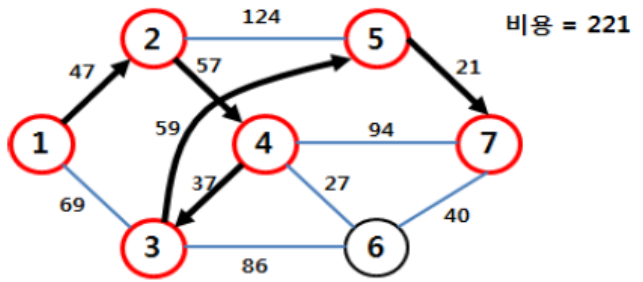


연구활동 가는 길

- 탐색의 배제1
  - 탐색배제 조건

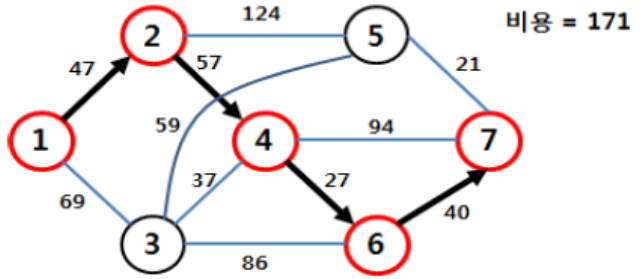
    현재 알아낸 최소 거리 >
    지금까지 구한 경로의 거리

  - 위 조건을 만족할 경우, 더 이상 탐색하지 않더라도 해를 구하는 데 전혀 문제가 없음



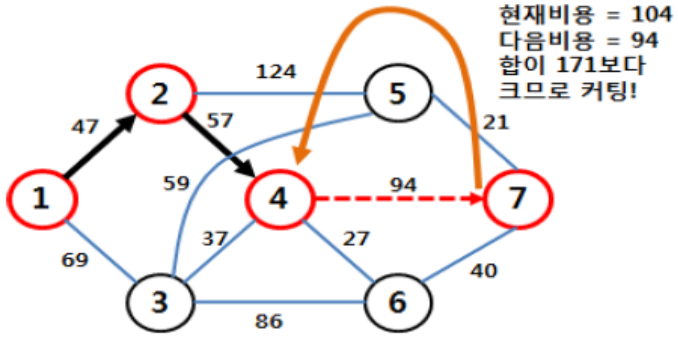
처음으로 찾게 되는 경로, 이 경로의 길이는 221이다.

현재까지 구한 최소 이동거리 = 221



다음으로 구한 경로는 171이 된다. 이 해는 지금까지의 해보다 221보다 더 좋은 경로이므로 갱신

현재까지 구한 최소 이동거리 = 171



다음 경로로 진행하는 도중에 현재까지 최소인 171보다 커지게 되므로 커팅!! 따라서 탐색영역이 배제되고 효율은 높아진다. 이러한 과정으로 마지막까지 진행.

연구활동

- 탐색의 배제1
  - 탐색배제 조건

    현재 알아낸 최소 거리 >
    지금까지 구한 경로의 거리

  - 위 조건을 만족할 경우, 더 이상 탐색하지 않더라도 해를 구하는 데 전혀 문제가 없음

```
void solve(int V, int W) {
  path.push_back(V);
  chk[V] = 1;

  // 현재 알아낸 최소 거리 > 지금까지 구한 거리
  if(W > sol) return;

  // 마지막 n 위치에 도달했으면...
  if(V == n)  {
    // 더 짧은 거리를 찾았으면 업데이트 함.
    if(W < sol) sol = W;

    output(W);
    return;
  }

  for(int i=1; i<=n; i++) {
    if(!chk[i] && G[V][i])  {
      solve(i, W+G[V][i]);
      path.pop_back();
      chk[i] = 0;
    }
  }
}
```

연구활동 가는 길 – 수행 횟수 비교

research_path_try3.cpp

```
// 기존 코드
int counter=0;
void solve(int V, int W) {
  counter++;
  path.push_back(V);
  chk[V] = 1;

  if(V == n)  {
    if(W < sol) sol = W;

    for(int i=0; i<path.size(); i++)
      printf("%d-", path[i]);
    printf("\b : [%d]\n", W);
    return;
  }

  for(int i=1; i<=n; i++) {
    if(!chk[i] && G[V][i])  {
      solve(i, W+G[V][i]);
      path.pop_back();
      chk[i] = 0;
    }
  }
}
```

```
// 탐색배제1 사용
int counter=0;
void solve(int V, int W) {
  counter++;
  path.push_back(V);
  chk[V] = 1;

  if(W > sol) return;

  if(V == n)  {
    if(W < sol) sol = W;

    for(int i=0; i<path.size(); i++)
      printf("%d-", path[i]);
    printf("\b : [%d]\n", W);
    return;
  }

  for(int i=1; i<=n; i++) {
    if(!chk[i] && G[V][i])  {
      solve(i, W+G[V][i]);
      path.pop_back();
      chk[i] = 0;
    }
  }
}
```

연구활동

- 탐색의 배제2
  - 가장 짧은 경로일 가능성이 높은 길을 탐욕법으로 하나 찾은 뒤,
  - 탐색배제1 방법을 함께 사용

V \ i	1	2	3	4	5	6	7
1		47	69				
2	47			57	124		
3	69			37	59	86	
4		57	37			27	94
5		124	59				21
6			86	27			40
7				94	21	40	

```
// 탐색배제2: 탐욕법 길찾기
int greedy_chk[MAX_V+1];
void greedy_ans(int V) {
  int W=0, t;
  sol=0;
  greedy_chk[V]=1;

  printf("\nGreedy path: %d", V);
  // 탐색 정점 V가 목적지에 도달할 때까지…
  while(V != n) {
    int min=MAX_INT;
    // 모든 G[V][i]에 대하여 최소값 탐색
    for(int i=1; i<=n; i++) {
      // 방문한적 없고, 연결되어 있고, 더 작은 값이면,
      if(!greedy_chk[i] && G[V][i] && G[V][i]<min) {
        greedy_chk[i]=1;  // 방문을 표시
        min=G[V][i]; // 새로운 최소값으로 업데이트
        t=i;  // 새로운 최소값인 정점을 저장해 둠
      }
    }
    if(V != t) { // 다음 정점으로 이동이 되면
      sol+=G[V][t]; // 거리를 누적
      V=t;  // 가장 짧은 거리의 점점으로 이동
      printf("-%d", V);
    }
    else {    // 이동불가에 빠졌으면(탐욕법으로 길 찾기 실패)
      sol=MAX_INT;
      break;  // 탈출
    }
  }
  printf(" [%d]\n", sol); // 탐욕법으로 찾은 최소경로 길이 출력
}
```

연구활동 가는 길 전체탐색(DFS) – 탐색배제1+2 전체 소스코드

```
#include <stdio.h>
#include <limits.h>
#include <vector>
#define MAX_V   15   // 최대 정점 개수
using namespace std;

int n, m;
int G[MAX_V+1][MAX_V+1];
int chk[MAX_V+1];
int sol=INT_MAX;
int counter=0;
vector<int> path;

void solve(int V, int W);
void greedy_ans(int V);

void input() {
  scanf("%d %d", &n, &m);
  for(int i=0; i<m; i++) {
    int s, e, w;
    scanf("%d %d %d", &s, &e, &w);
    G[s][e] = G[e][s] = w;
  }
}

int main() {
  input();
  greedy_ans(1);  // 여기 주석처리시 탐색배제1만 적용됨

  solve(1, 0);  // 정점 1, 거리 0에서 시작

  printf("\nPath min: %d\n", sol==MAX_INT ? -1 : sol);
  printf("Vertex counter: %d\n", counter);
  return 0;
}
```

```
void output(int W) {
    for(int v : path)
        printf("%d-", v);
    printf("\b [%d]\n", W);
}

// 정점 V에서 거리를 계산해라,
// 현재까지 이동거리는 W이다.
void solve(int V, int W) {
  counter++;
  path.push_back(V);
  chk[V] = 1;

  // 현재 알아낸 최소 거리
  //   > 지금까지 구한 최소 경로의 거리
  if(W>sol) return; // 여기도 주석처리시 전체탐색

  // 마지막 n 위치에 도달했으면...
  if(V == n)  {
    // 더 짧은 거리를 찾았으면 업데이트 함.
    if(W < sol) sol = W;
    output(W);
    return;
  }

  for(int i=1; i<=n; i++) {
    // i를 방문한 적이 없고, 현재V와 i가 연결되어 있다면,
    // (0이 아니면 연결되어 있다는 뜻)...
    if(!chk[i] && G[V][i])  {
      solve(i, G[V][i]+W);
      path.pop_back();
      chk[i] = 0;
    }
  }
}
```

```
// 탐색배제2 사용
int greedy_chk[MAX_V+1];
void greedy_ans(int V) {
  int W=0, t;
  sol=0;
  greedy_chk[V]=1;

  printf("\nGreedy path: %d", V);
  // 탐색 정점 V가 목적지에 도달할 때까지…
  while(V != n) {
    int min=MAX_INT;
    // 모든 G[V][i]에 대하여 최소값 탐색
    for(int i=1; i<=n; i++) {
      // 방문한적 없고, 연결되어 있고, 더 작은 값이면,
      if(!greedy_chk[i] && G[V][i] && G[V][i]<min) {
        greedy_chk[i]=1;  // 방문을 표시
        min=G[V][i]; // 새로운 최소값으로 업데이트
        t=i;  // 새로운 최소값인 정점을 저장해 둠
      }
    }
    if(V!=t) {  // 다음 정점으로 이동하면
      sol+=G[V][t]; // 거리를 누적
      V=t; // 가장 짧은 거리의 점점으로 이동
      printf("-%d", V);
    }
    else {    // 이동불가에 빠짐(길찾기 실패)
      sol=MAX_INT;
      break;  // 탈출
    }
  }
  // 탐욕법으로 찾은 최소경로 길이 출력
  printf(" [%d]\n", sol);
}
```

연구활동 가는 길 – 탐색의 배제

- 성능 비교 테스트 데이터

입력 1	입력 2	입력 3	입력 4
5 8 1 2 2 1 3 1 1 4 3 2 5 2 2 3 1 4 5 3 3 5 2 1 5 14	7 11 1 2 47 1 3 69 2 4 57 2 5 124 3 4 37 3 5 59 3 6 86 4 6 27 4 7 94 5 7 21 6 7 40	8 13 1 2 47 1 3 69 1 7 50 2 4 57 2 5 124 3 4 107 3 5 59 3 6 86 4 6 27 4 7 14 5 7 81 6 7 40 7 8 90	8 14 1 2 47 1 3 69 1 7 50 2 4 57 2 5 124 3 4 107 3 5 59 3 6 86 4 6 27 4 7 14 5 7 81 6 7 40 5 8 30 7 8 90

※입력3: 탐욕법 길찾기가 실패되는 케이스

- 탐색 횟수 비교 결과

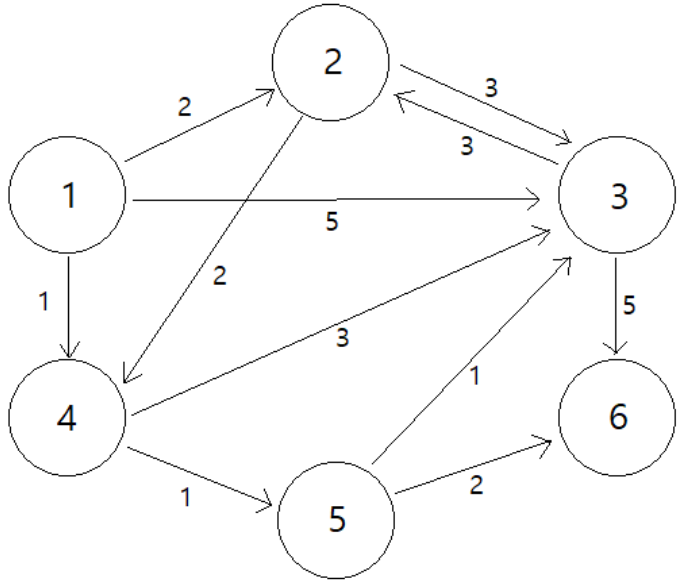
알고리즘	입력 1	입력 2	입력 3	입력 4
전체탐색	12	47	153	176
탐색배제1	10	16	49	39
탐색배제2	7	13	49	28

- 배제된 공간 비율

알고리즘	입력 1	입력 2	입력 3	입력 4
탐색배제1	12.0%	66.0%	69.9%	77.8%
탐색배제2	41.7%	72.3%	69.9%	84.1%

다익스트라(dijkstra) 알고리즘

- 개요
  - 그래프의 한 정점(Vertex)에서 모든 정점까지의 최단거리를 각각 구하는 알고리즘

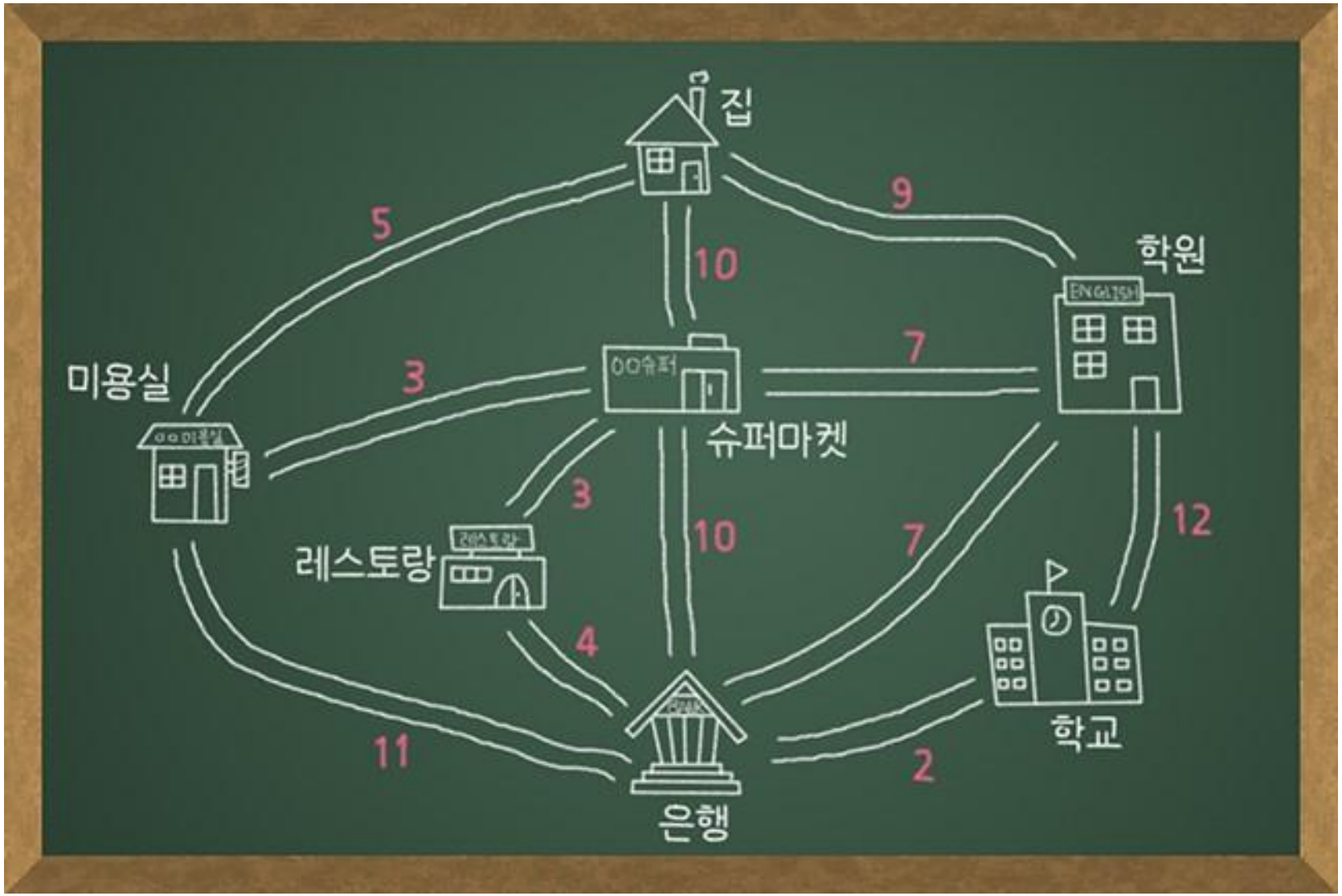


  - 음의 간선이 없을 때 정상 작동
  - 네비게이션소프트웨어의 기본 알고리즘

- EBS 해설 동영상
  - https://youtu.be/tZu4x5825LI

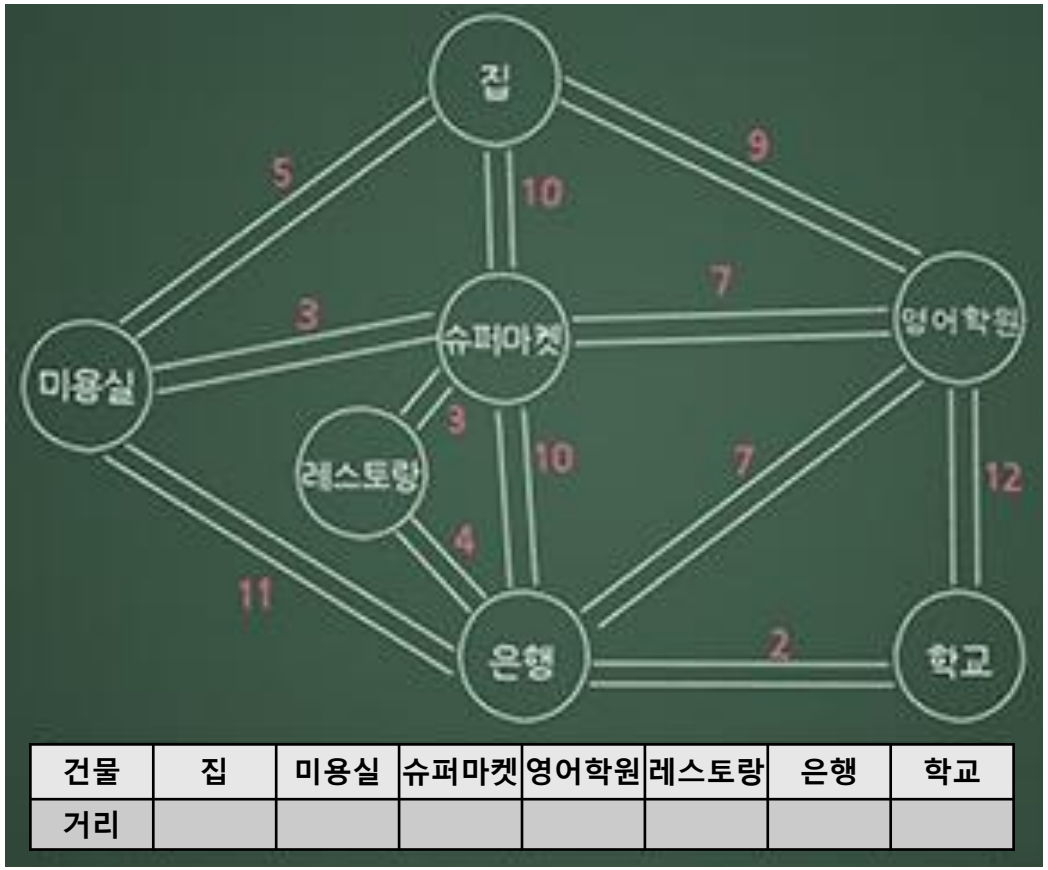
- 특징
  - 에츠허르 다익스트라가 고안한 알고리즘으로, 그가 처음 고안한 알고리즘은 $O(V^2)$의 시간복잡도 였다.
  - 이후 힙 트리등을 이용한 개선된 알고리즘이 나오며 $O((V+E)logV)$ 의 시간복잡도를 가지게 되었다.

다익스트라(dijkstra) 알고리즘



다익스트라(dijkstra) 알고리즘

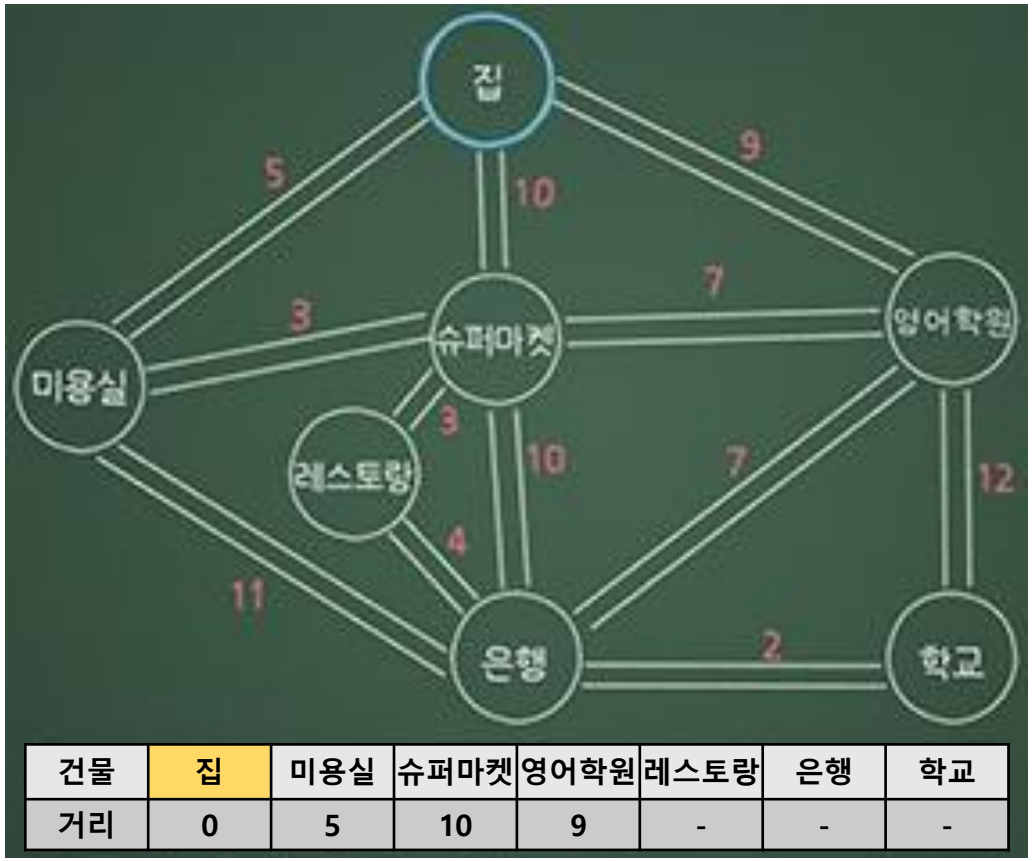
- 과정1



건물	집	미용실	슈퍼마켓	영어학원	레스토랑	은행	학교
거리							

최대거리 테이블 초기화

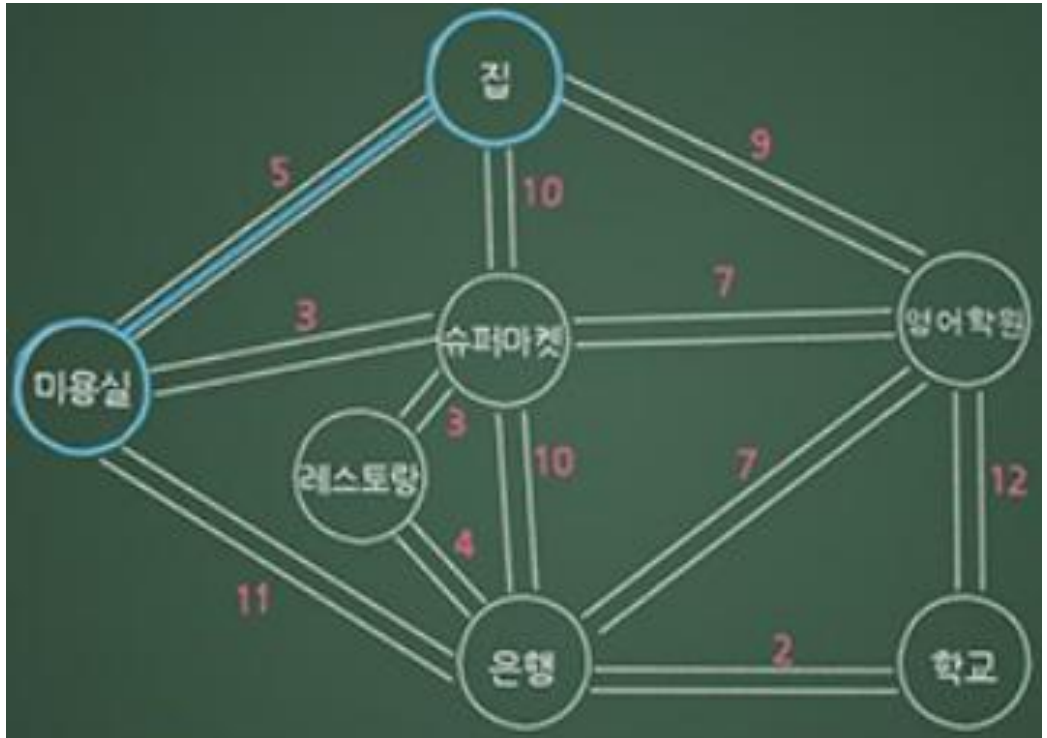
- 과정2



건물	집	미용실	슈퍼마켓	영어학원	레스토랑	은행	학교
거리	0	5	10	9	-	-	-

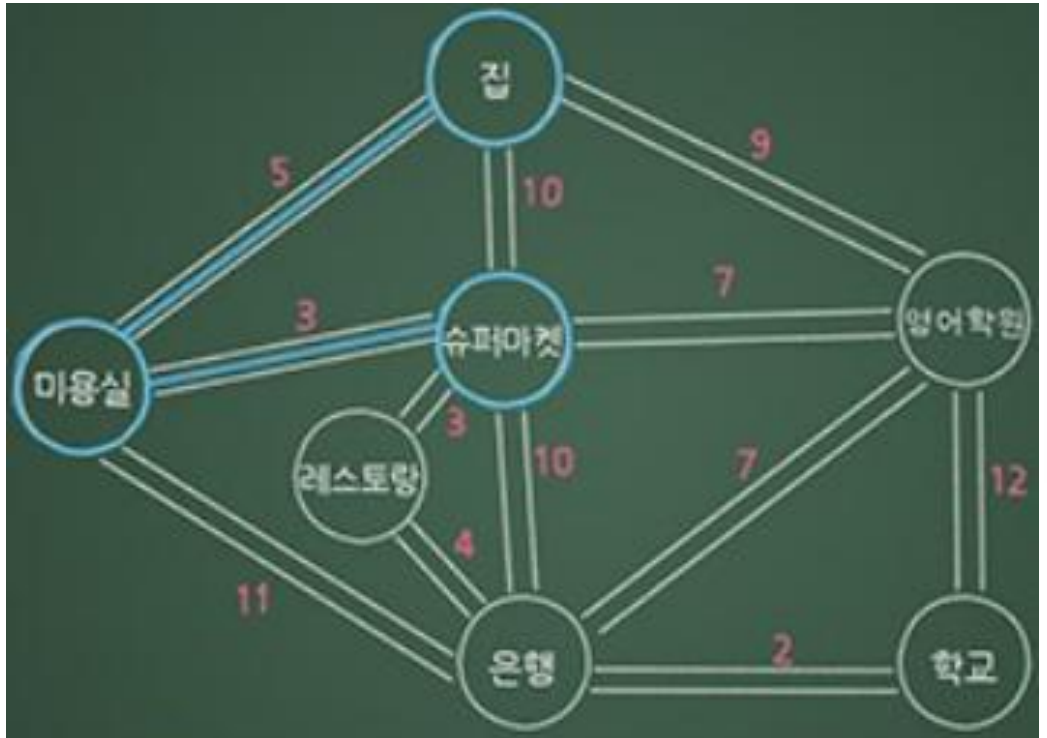
다익스트라(dijkstra) 알고리즘

- 과정3



건물	집	미용실	슈퍼마켓	영어학원	레스토랑	은행	학교
거리	0	5	10→8	9	-	16	-

- 과정4

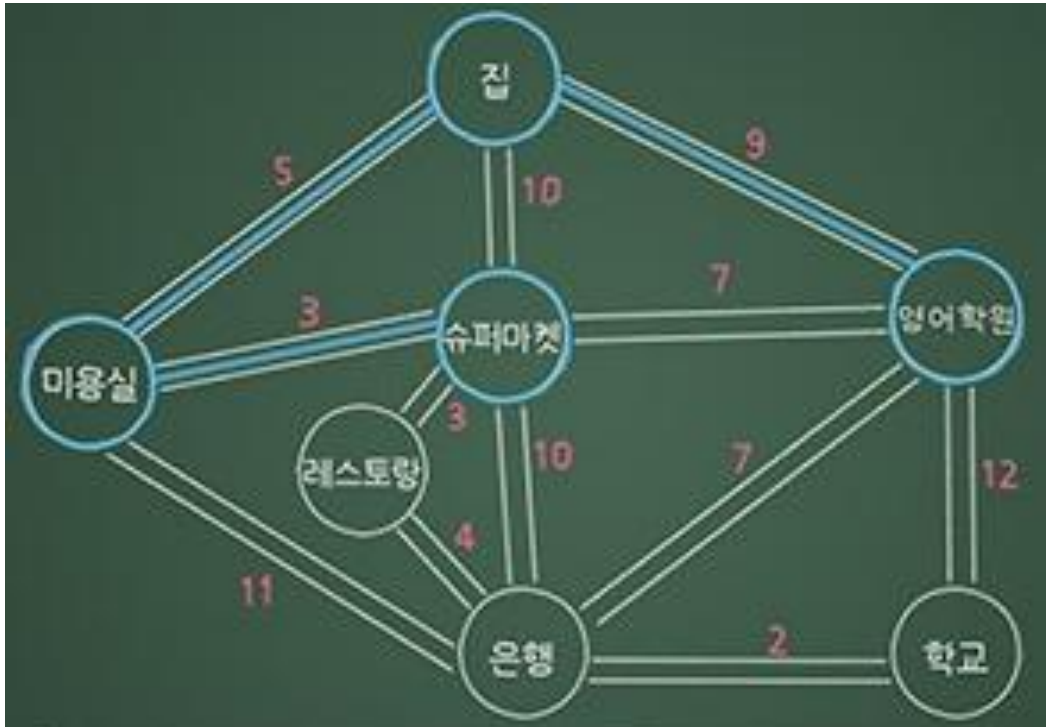


건물	집	미용실	슈퍼마켓	영어학원	레스토랑	은행	학교
거리	0	5	8	9	11	16	-

```
1(0)의
  2(5)
  3(10
  4(9)
2(5)의
  3(8)
  6(16
3(8)의
  4(9)
  5(11
  6(16
4(9)의
  7(21
  6(16
3(10)의
5(11)의
  6(15
6(15)의
  7(17
6(16)의
7(17)의
7(21)의
```

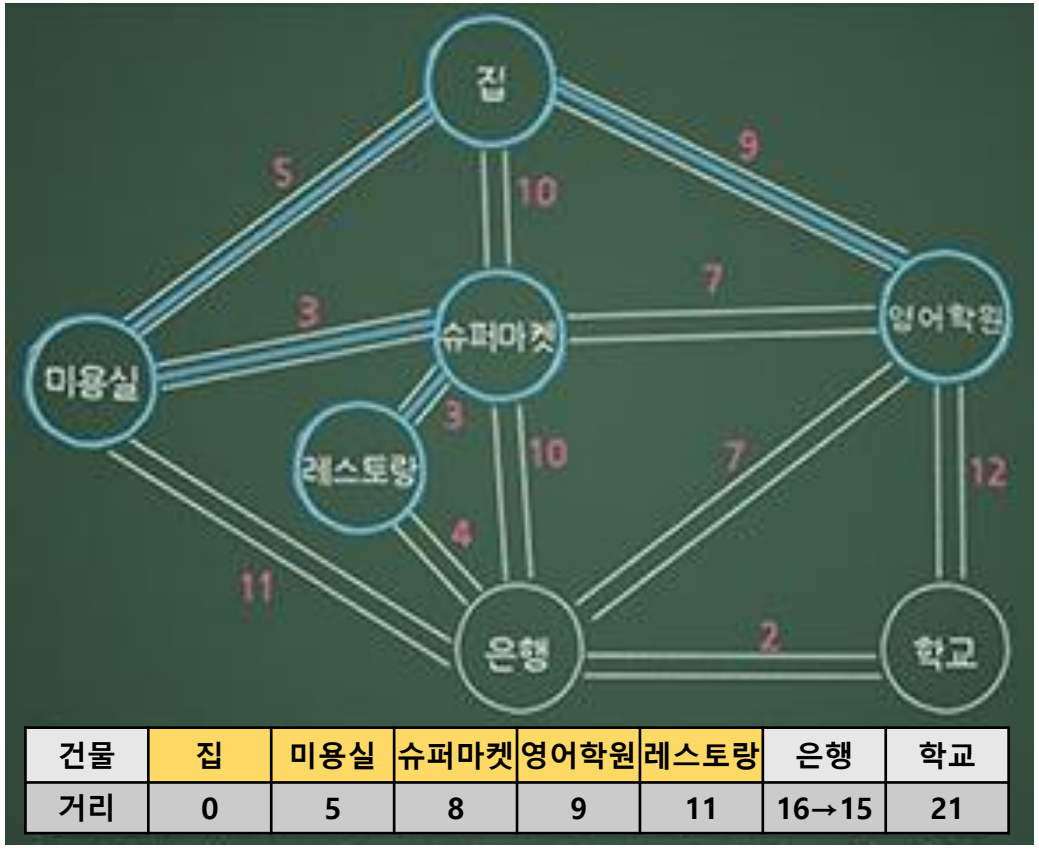
다익스트라(dijkstra) 알고리즘

- 과정5



건물	집	미용실	슈퍼마켓	영어학원	레스토랑	은행	학교
거리	0	5	8	9	11	16	21

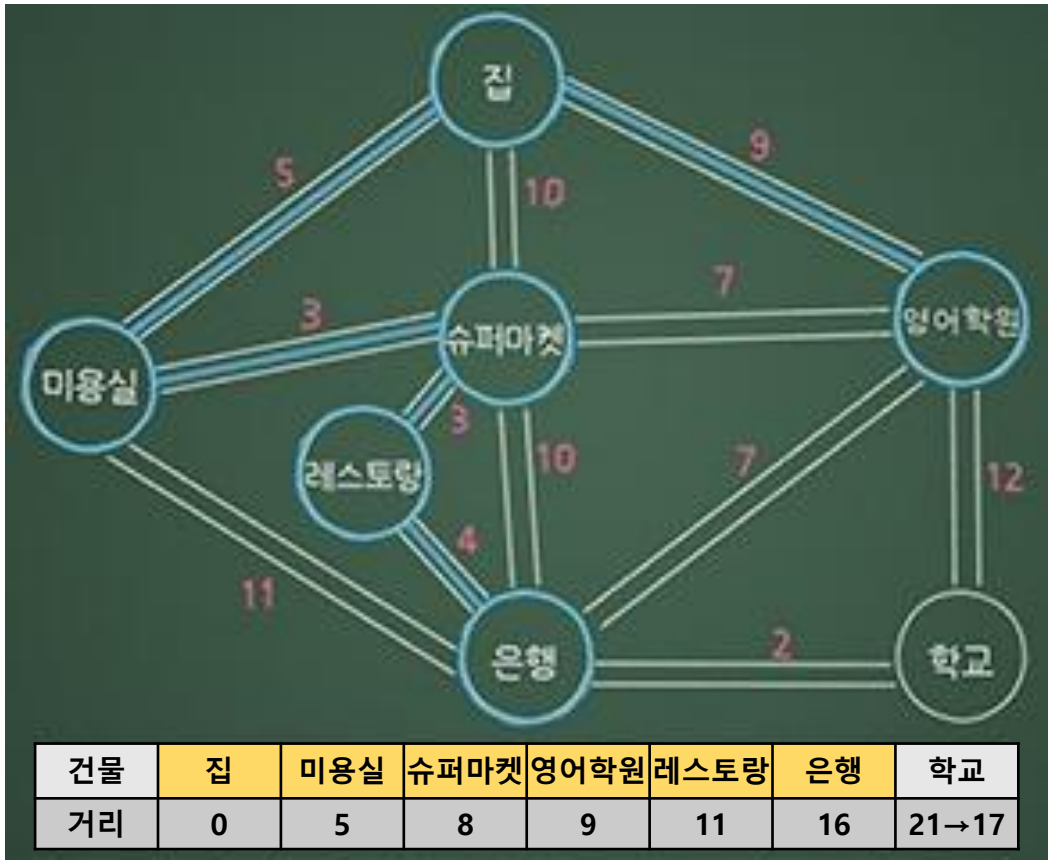
- 과정6



건물	집	미용실	슈퍼마켓	영어학원	레스토랑	은행	학교
거리	0	5	8	9	11	16→15	21

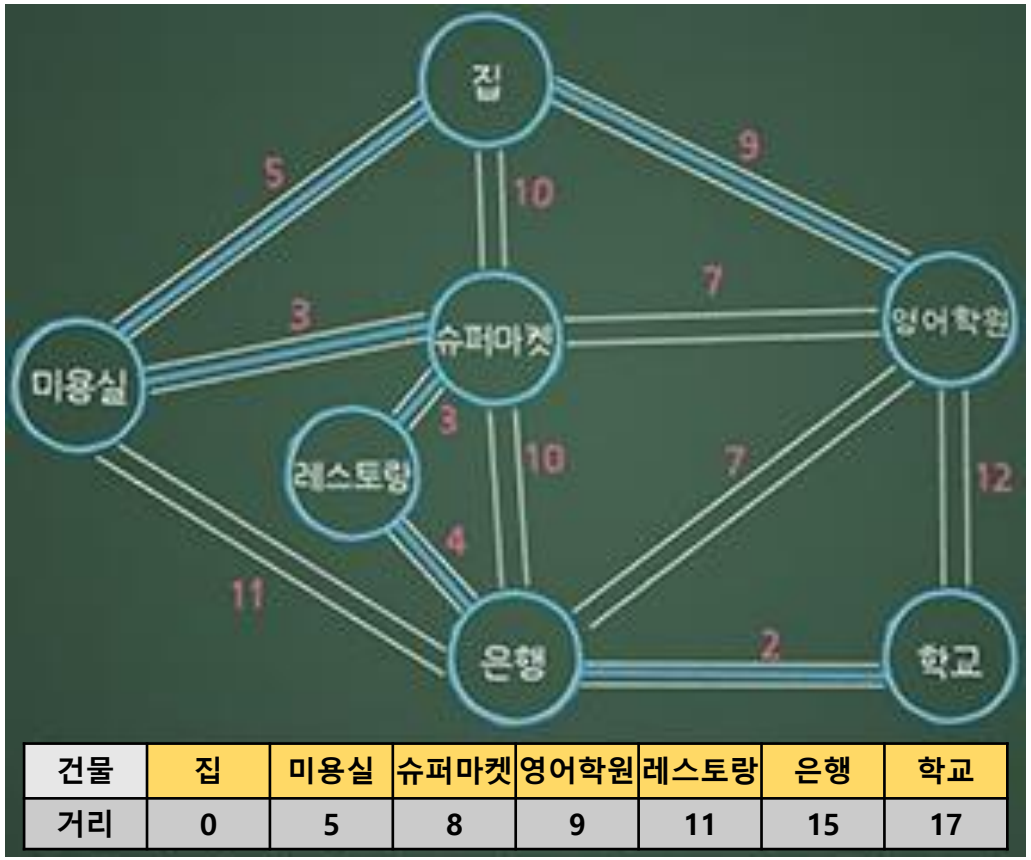
다익스트라(dijkstra) 알고리즘

- 과정7



건물	집	미용실	슈퍼마켓	영어학원	레스토랑	은행	학교
거리	0	5	8	9	11	16	21→17

- 과정8



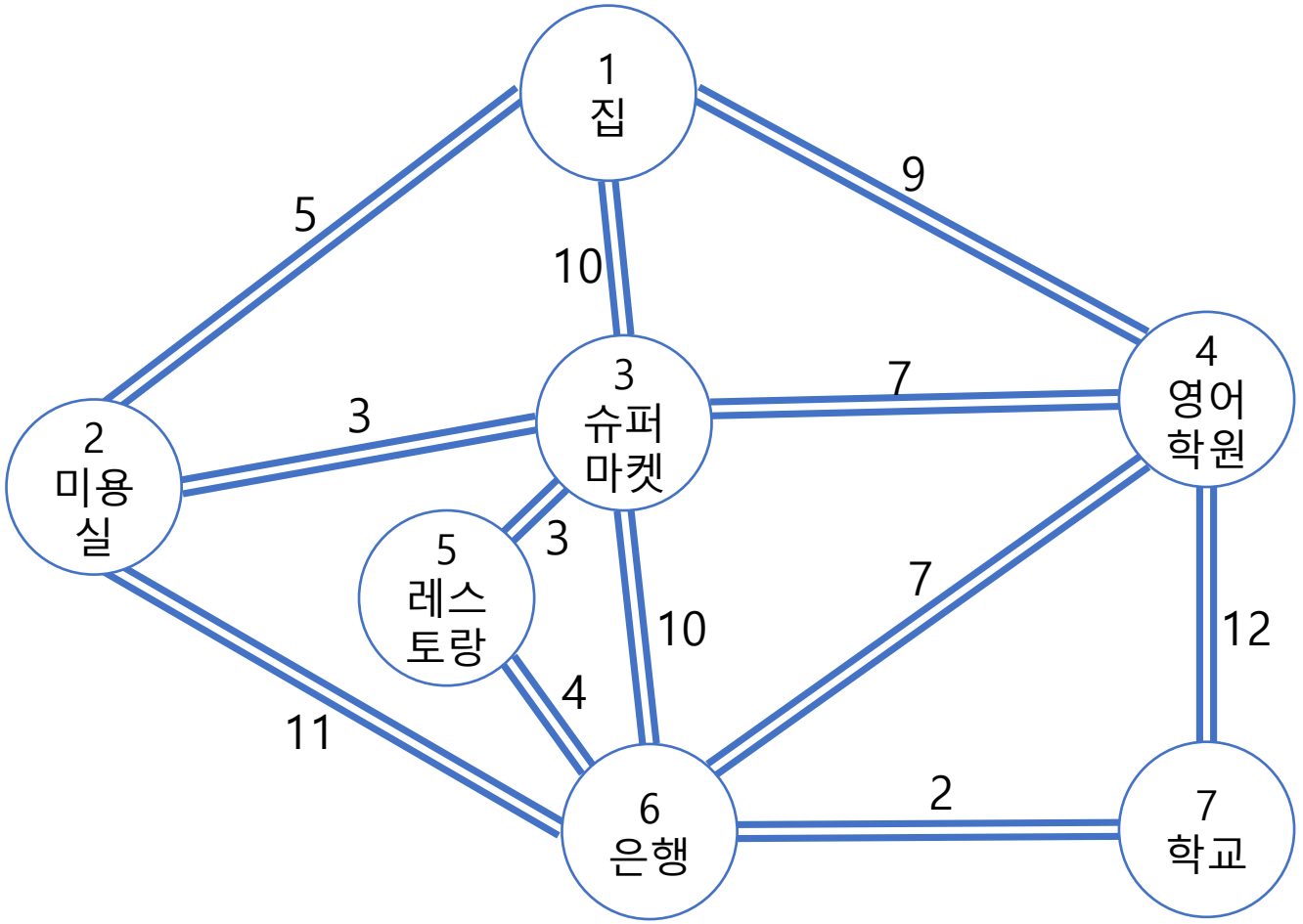
건물	집	미용실	슈퍼마켓	영어학원	레스토랑	은행	학교
거리	0	5	8	9	11	15	17

다익스트라(dijkstra) 알고리즘

- 알고리즘
  1) 출발점으로부터의 최단거리를 저장할 배열 d[v]를 만들고 초기화 한다.
  2) 현재 노드를 나타내는 변수 A에 출발 노드의 번호를 저장한다.
  3) A에 인접한 모든 노드 B에 대해, d[A]+cost(A~B)와 d[B]의 값을 비교
  4) 만약 d[A] + P[A][B]의 값이 더 작다면, 즉 더 짧은 경로라면, d[B]의 값을 이 값으로 갱신한다.
  5) A의 모든 이웃 노드 B에 대해 이 작업을 수행한다.
  6) A의 상태를 "방문 완료"로 바꾼다. 그러면 이제 더 이상 A는 사용하지 않는다.
  7) "미방문" 상태인 모든 노드들 중, 출발점으로부터의 거리가 제일 짧은 노드 하나를 골라서 그 노드를 A에 저장한다.
  8) 도착 노드가 "방문 완료" 상태가 되거나, 혹은 더 이상 미방문 상태의 노드를 선택할 수 없을 때까지, 3~7의 과정을 반복한다.
- 7번 단계에서, 거리가 가장 짧은 노드를 선택 하려면 모든 노드를 순회해야 하므로 시간 복잡도에 결정적인 영향

다익스트라(dijkstra) 알고리즘

- 그래프로 표현



- 데이터 표현

정점수 노드수 시작정점

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

```
vector<noco> graph[100001];
```

V	[0]	[1]	[2]
1	(2,5)	(3,10)	(4,9)
2	(3,3)	(6,11)	
3	(4,7)	(5,3)	(6,10)
4	(7,12)	(6,7)	
5	(6,4)		
6	(7,2)		
7			

개선된 다익스트라 알고리즘 $O((V+E)log(V))$

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

#define INF 1e9 // 무한을 의미하는 값으로 10억을 설정
using namespace std;

struct noco {
    int node;
    int cost;
    //noco(int n, int c) : node(n), cost(c) {}
    bool operator<(noco b) const {
        return cost > b.cost;  // cost기준 오름차순
    }
};
// 위 구조체의 cost멤버는 graph 변수에서는 노드간 거리의 의미로
// 우선순위 큐 pq에서는 계산된 최단거리의 의미로 사용됨

// 노드의 개수(N), 간선의 개수(M), 시작 노드 번호(Start)
// 노드의 개수는 최대 100,000개라고 가정
int n, m, start;
```

```
// 각 노드에 연결되어 있는 노드에 대한 정보를 담는 배열
vector<noco> graph[100001];
// 최단 거리 테이블 dist[a]: a노드까지의 최단거리를 저장
int dist[100001];
// 직전 노드 저장
int from[100001];

void output_dist_ary() { // 최단거리 배열 출력
    putchar('\n');
    for(int i=1; i<=n; i++)
        printf("[%d]%d, ", i, dist[i]);
    printf("\b\b \n");
}
// 우선순위 큐 내용물 출력
void output_pq(priority_queue<noco> pq) {
    while(!pq.empty()) {
        noco nc = pq.top();
        // {노드, (비용)}
        printf("{%d,(%d)} ", nc.node, nc.cost);
        pq.pop();
    }
    putchar('\n');
}
```

건물	집	미용실	슈퍼마켓	영어학원	레스토랑	은행	학교
거리	0	5	8	9	-	16	-

pq는 아직 최단거리 계산이 완료되지 않은 (슈퍼마켓, 8), (영어학원, 9), (은행, 16) 를 갖고 있고 pop() 하면 가장 거리가 짧은 원소를 뽑아주는 역할을 한다.

개선된 다익스트라 알고리즘 $O((V+E)log(V))$

```
int main(void) {
    cin >> n >> m >> start;

    // 모든 간선 정보를 입력받기
    for (int i = 0; i < m; i++) {
        int a, b, c;
        cin >> a >> b >> c;
        // a번 노드에서 b번 노드로 가는 비용이 c라는 의미
        graph[a].push_back({b, c});
        // 양방향 연결일 경우 아래 코드 활성화
        //graph[b].push_back({a, c});
    }

    // 최단 거리 테이블을 모두 무한으로 초기화
    fill(dist, dist + 100001, INF);

    // 다익스트라 알고리즘을 수행
    dijkstra(start);
```

```
    putchar('\n');
    output_shortest_path(n);

    putchar('\n');
    // 모든 노드로 가기 위한 최단 거리를 출력
    for (int i = 1; i <= n; i++) {
        printf("%d 까지 최단거리: ", i);
        // 도달할 수 없는 경우, 무한(INFINITY)이라고 출력
        if (dist[i] == INF) {
            cout << "INFINITY" << '\n';
        }
        // 도달할 수 있는 경우 거리를 출력
        else {
            cout << dist[i] << '\n';
        }
    }
}
```

개선된 다익스트라 알고리즘 $O((V+E)log(V))$

```
void dijkstra(int start) {
    //output_dist_ary(); // 최단거리 테이블 내용 출력
    //output_pq(pq);     // 우선순위 큐 내용 출력
    // start에서 pq.node까지 최단 거리가 pq.cost이다.
    priority_queue<noco> pq;

    // 시작 노드까지의 거리는 0으로 설정하여, 큐에 삽입
    pq.push({start, 0});
    dist[start] = 0;
    while (!pq.empty()) { // 큐가 비어있지 않다면
        // 가장 최단 거리가 짧은 노드에 대한 정보 꺼내기
        int now_node = pq.top().node; // 현재 노드
        // start에서 현재 노드까지 알려진 최단거리
        int now_dist = pq.top().cost;
        pq.pop();

        // 현재노드를 경유하는 인접노드들까지의 최단거리를 계산하려는데
        // 이미 계산해 놓은 dist[now_node]가 now_dist보다 작으면
        // 인접노드들의 최단거리를 계산하는 의미가 없음.
        if (dist[now_node] < now_dist) {
            printf("%d(%d)의 인접노드 계산 건너뜀\n"
                , now_node, now_dist);
            continue;
        }
        printf("%d(%d)의 인접노드 계산\n", now_node, now_dist);
        //현재 노드에 연결된 i번째 노드(graph[now_node][i])에 대하여
        for (int i = 0; i < (int)graph[now_node].size(); i++) {
            // i번째 노드(graph[now_node][i])를 목적지로 설정
            noco tar = graph[now_node][i];
            // 목적지까지 거리 = 현재 노드까지 거리 + 목적지까지 비용
            int tar_dist = now_dist + tar.cost;
            // 현재 노드를 거쳐가는 거리가 다른 방법보다 짧으면,
            if (tar_dist < dist[tar.node]) {
                // target까지의 최단거리 테이블 업데이트
                dist[tar.node] = tar_dist;
                // 이 정보를 pq에 삽입한다. 왜냐하면, tar까지 최단거리가
                // 업데이트 되었으므로 인접노드를 더 짧게 방문할 수 있음.
                pq.push({tar.node, tar_dist});
                // 목적지의 직전 노드가 now_node임을 저장
                from[tar.node] = now_node;

                printf("  %d(%d)노드 업데이트\n", tar.node, tar_dist);
            }
            else
                printf("  %d(%d)노드 업데이트 안함\n",
                        tar.node, dist[tar.node]);
        }
    }
}
```

개선된 다익스트라 알고리즘 $O((V+E)log(V))$

```
void output_shortest_path(int dest) {
    if(dist[dest] == INF) {
        printf("there is no path to %d\n", dest);
        return;
    }

    deque<noco> pathstep;
    int now = dest;

    while(dist[now] > 0) {
        pathstep.push_front({now, dist[now]});
        now = from[now];
    }
    pathstep.push_front({now, dist[now]});

    printf("%d 까지 최단경로:\n", dest);
    for(noco a: pathstep)
        printf("%d(%d) > ", a.node, a.cost);
    printf("\b\b  \n");
}
```

- 실행결과

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

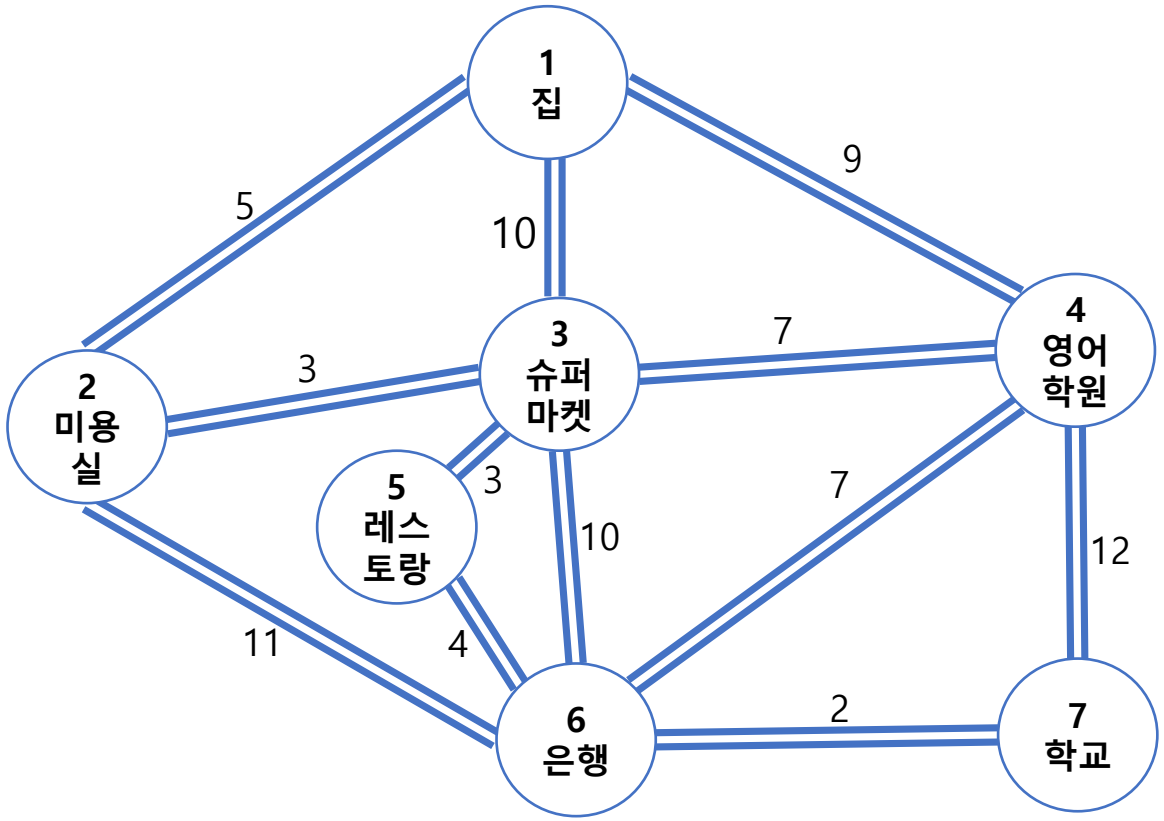
```
1(0)의 인접노드 계산
  2(5)노드 업데이트
  3(10)노드 업데이트
  4(9)노드 업데이트
2(5)의 인접노드 계산
  3(8)노드 업데이트
  6(16)노드 업데이트
3(8)의 인접노드 계산
  4(9)노드 업데이트 안함
  5(11)노드 업데이트
  6(16)노드 업데이트 안함
4(9)의 인접노드 계산
  7(21)노드 업데이트
  6(16)노드 업데이트 안함
3(10)의 인접노드 계산 건너뜀
5(11)의 인접노드 계산
  6(15)노드 업데이트
6(15)의 인접노드 계산
  7(17)노드 업데이트
6(16)의 인접노드 계산 건너뜀
7(17)의 인접노드 계산
7(21)의 인접노드 계산 건너뜀
```

```
1 까지 최단거리: 0
2 까지 최단거리: 5
3 까지 최단거리: 8
4 까지 최단거리: 9
5 까지 최단거리: 11
6 까지 최단거리: 15
7 까지 최단거리: 17
```

```
7 까지 최단경로:
1(0) > 2(5) > 3(8) > 5(11) > 6(15) > 7(17)
```

다익스트라(dijkstra) 알고리즘 작동과정

- 그래프로 표현




```
[1]0, [2]1000000000, [3]1000000000, [4]1000000000, [5]1000000000, [
{1,(0)}
1(0)의 인접노드 계산
  2(5)노드 업데이트
  3(10)노드 업데이트
  4(9)노드 업데이트
```

```
[1]0, [2]5, [3]10, [4]9, [5]1000000000, [6]1000000000, [7]100000000
{2,(5)} {4,(9)} {3,(10)}
2(5)의 인접노드 계산
  3(8)노드 업데이트
  6(16)노드 업데이트
```

```
[1]0, [2]5, [3]8, [4]9, [5]1000000000, [6]16, [7]1000000000
{3,(8)} {4,(9)} {3,(10)} {6,(16)}
3(8)의 인접노드 계산
  4(9)노드 업데이트 안함
  5(11)노드 업데이트
  6(16)노드 업데이트 안함
```

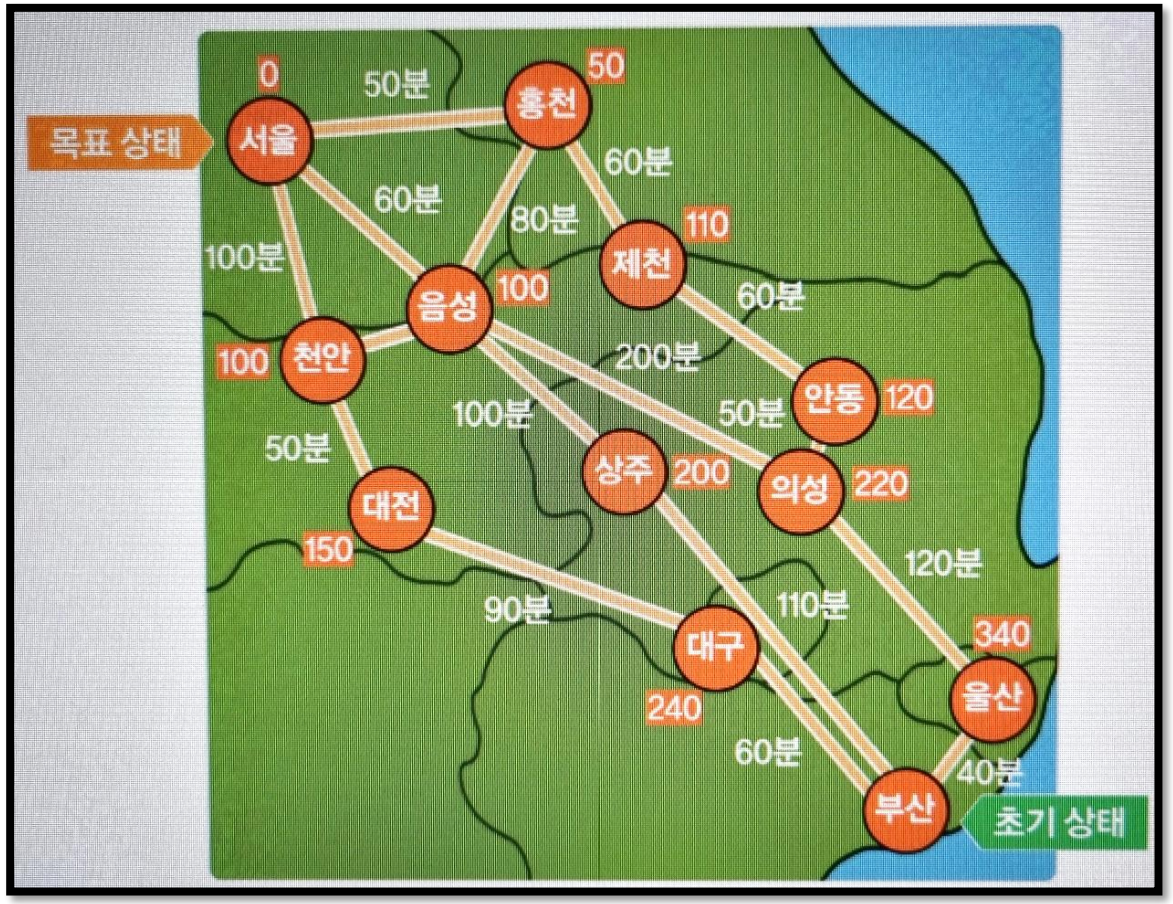
```
[1]0, [2]5, [3]8, [4]9, [5]11, [6]16, [7]1000000000
{4,(9)} {3,(10)} {5,(11)} {6,(16)}
4(9)의 인접노드 계산
  7(21)노드 업데이트
  6(16)노드 업데이트 안함
```

```
[1]0, [2]5, [3]8, [4]9, [5]11, [6]16, [7]21
{3,(10)} {5,(11)} {6,(16)} {7,(21)}
  3(10)노드 건너뜀
```

```
[1]0, [2]5, [3]8, [4]9, [5]11, [6]16, [7]21
{5,(11)} {6,(16)} {7,(21)}
5(11)의 인접노드 계산
  6(15)노드 업데이트
```

다익스트라(dijkstra) 알고리즘의 적용

- 부산에서 서울까지 최단거리를 안내하는 내비게이터를 만드시오.



- 최단 경로는?
- 최단 경로 이동에 소요되는 시간은?

- 시뮬레이터
  - https://gifted.datahub.pe.kr/src/visualizer/dijkstra_shortest_path_search.html

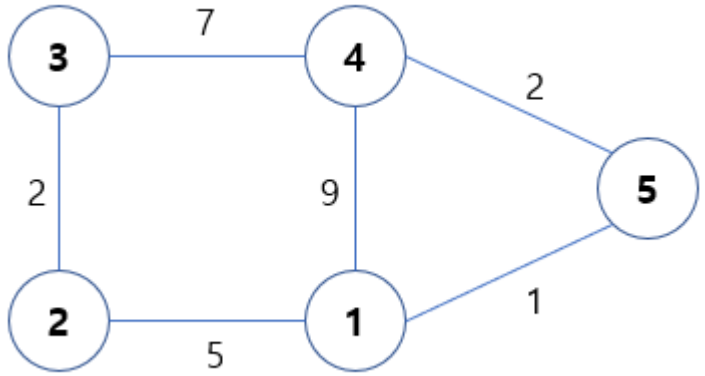
플로이드-워셜(Floyd-Warshall) 알고리즘

- 특징
  - 다익스트라가 하나의 정점에서 다른 모든 정점까지의 최단 거리를 구하는 알고리즘이었다면,
  - 플로이드-워셜은 모든 정점에서 다른 모든 정점까지의 최단 경로를 모두 구하는 알고리즘이다.
  - 플로이드-워셜 알고리즘은 다익스트라와는 달리 음의 간선도 사용가능
  - 정점이 N개일 때 모든 정점에 대하여 $O(N^2)$의 연산이 필요하므로 총 시간 복잡도는 $O(N^3)$이 된다

- 알고리즘 과정
  - 모든 노드 간의 최단거리를 구해야 하므로 2차원 인접 행렬을 구성
  - 알고리즘은 여러 라운드로 구성
  - 라운드마다 각 경로에서 새로운 중간 노드로 사용할 수 있는 노드를 선택하고, 더 짧은 길이를 선택하여 줄이는 과정을 반복
  - $D_{ab} = \min(D_{ab}, D_{ak} + D_{kb})$
    의미: a에서 b로 가는 거리가 a에서k를 경유하여 b로 가는 길이 더 짧다면 최단거리를 업데이트!

플로이드-워셜(Floyd-Warshall) 알고리즘

- 초기 그래프

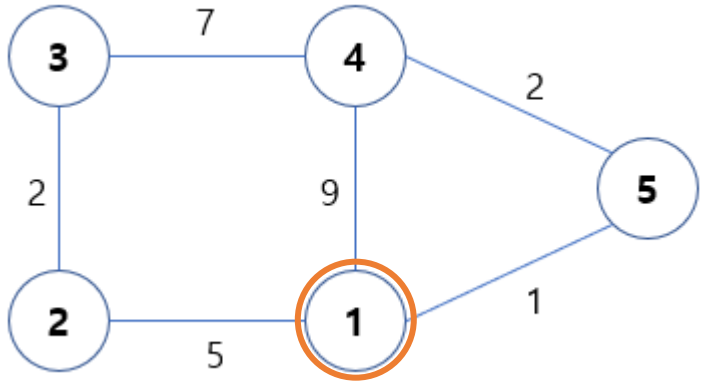


- 초기 인접행렬

f\t	1	2	3	4	5
1	0	5	INF	9	1
2	5	0	2	INF	INF
3	INF	2	0	7	INF
4	9	INF	7	0	2
5	1	INF	INF	2	0

플로이드-워셜(Floyd-Warshall) 알고리즘

- Round1: 1번 노드를 중간 노드로 설정



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

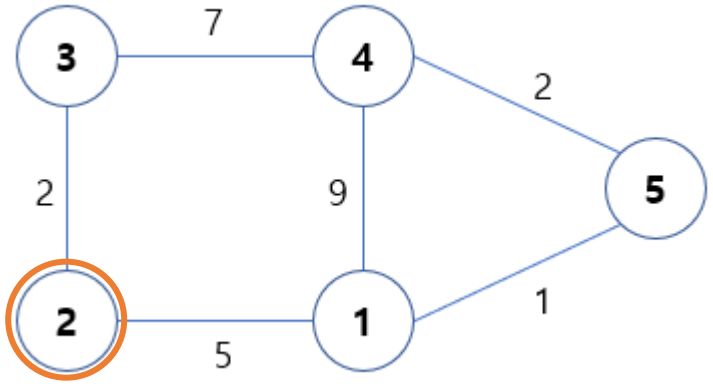
- Round1

f\t	1	2	3	4	5
1	0	5	INF	9	1
2	5	0	2	INF	INF
3	INF	2	0	7	INF
4	9	INF	7	0	2
5	1	INF	INF	2	0

f\t	1	2	3	4	5
1	0	5	INF	9	1
2	5	0	2	14	6
3	INF	2	0	7	INF
4	9	14	7	0	2
5	1	6	INF	2	0

플로이드-워셜(Floyd-Warshall) 알고리즘

- Round2: 2번 노드를 중간 노드로 설정



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

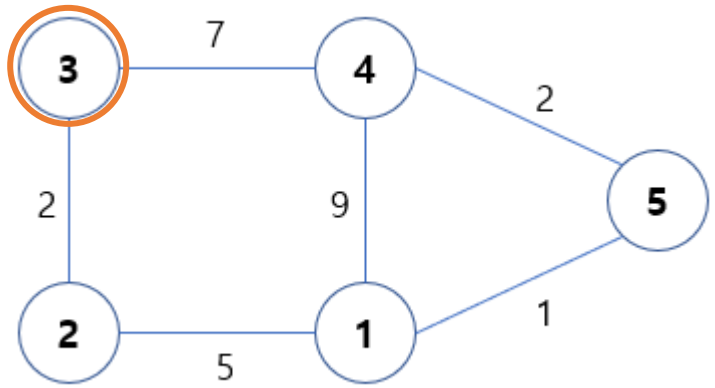
- Round2

f\t	1	2	3	4	5
1	0	5	INF	9	1
2	5	0	2	14	6
3	INF	2	0	7	INF
4	9	14	7	0	2
5	1	6	INF	2	0

f\t	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	14	6
3	7	2	0	7	8
4	9	14	7	0	2
5	1	6	8	2	0

플로이드-워셜(Floyd-Warshall) 알고리즘

- Round3: 3번 노드를 중간 노드로 설정



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

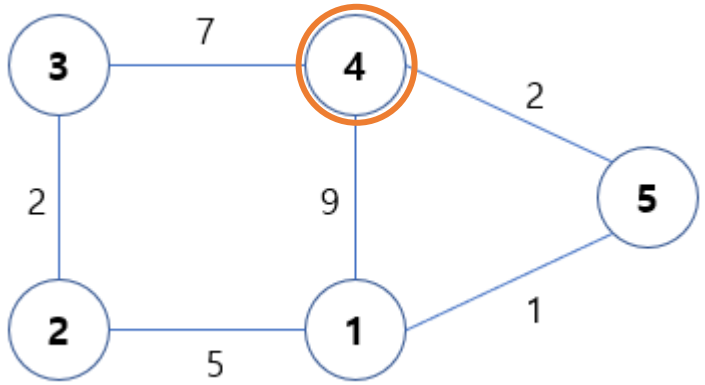
- Round3

f\t	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	14	6
3	7	2	0	7	8
4	9	14	7	0	2
5	1	6	8	2	0

f\t	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	9	6
3	7	2	0	7	8
4	9	9	7	0	2
5	1	6	8	2	0

플로이드-워셜(Floyd-Warshall) 알고리즘

- Round4: 4번 노드를 중간 노드로 설정



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

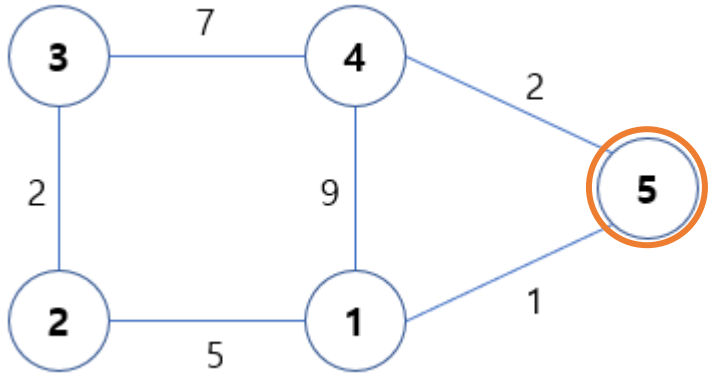
- Round4

f\t	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	9	6
3	7	2	0	7	8
4	9	9	7	0	2
5	1	6	8	2	0

f\t	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	9	6
3	7	2	0	7	8
4	9	9	7	0	2
5	1	6	8	2	0

플로이드-워셜(Floyd-Warshall) 알고리즘

- Round5: 5번 노드를 중간 노드로 설정



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

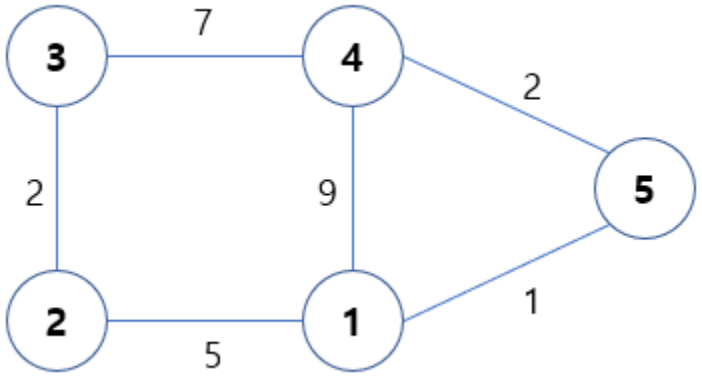
- Round5

f\t	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	9	6
3	7	2	0	7	8
4	9	9	7	0	2
5	1	6	8	2	0

f\t	1	2	3	4	5
1	0	5	7	3	1
2	5	0	2	9	6
3	7	2	0	7	8
4	3	9	7	0	2
5	1	6	8	2	0

플로이드-워셜(Floyd-Warshall) 알고리즘 구현

- 입력 데이터




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

- 출력

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

- 소스 코드

```
#include <iostream>
#define INF 1e9 // 무한을 의미하는 값으로 10억을 설정
#define MAX 101
using namespace std;

// 노드의 개수(N), 간선의 개수(M)
// 노드의 개수는 최대 101개라고 가정
int n, m;
// 2차원 배열(그래프 표현)를 만들기
int graph[MAX][MAX];

int main(void) {
    cin >> n >> m;

    // 최단 거리 테이블을 모두 무한으로 초기화
    for (int i = 0; i < MAX; i++) {
        fill(graph[i], graph[i] + MAX, INF);
    }
```

플로이드-워셜(Floyd-Warshall) 알고리즘 구현

```
// 자기 자신에서 자기 자신으로 가는 비용은 0으로 초기화
for (int a = 1; a <= n; a++) {
    for (int b = 1; b <= n; b++) {
        if (a == b) graph[a][b] = 0;
    }
}

// 각 간선에 대한 정보를 입력 받아, 그 값으로 초기화
for (int i = 0; i < m; i++) {
    // a에서 b로 가는 비용은 c라고 설정
    int a, b, c;
    cin >> a >> b >> c;
    graph[a][b] = c;
    graph[b][a] = c; // 단방향 간선인 경우 삭제
}
```

f\t	1	2	3	4	5
1	0	5	INF	9	1
2	5	0	2	INF	INF
3	INF	2	0	7	INF
4	9	INF	7	0	2
5	1	INF	INF	2	0

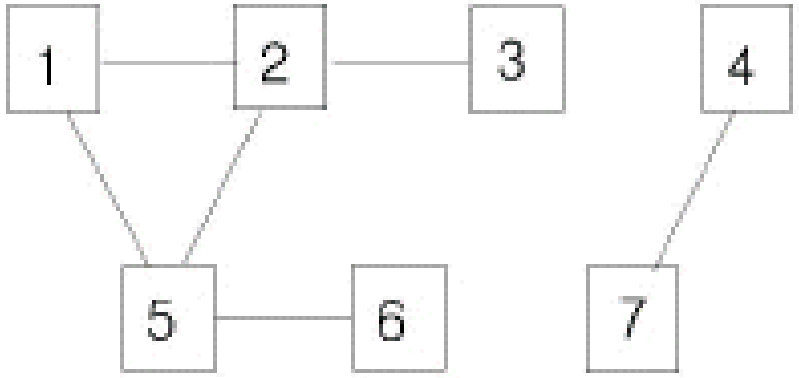
```
    // 플로이드 워셜 알고리즘을 수행
    for (int r = 1; r <= n; r++)  //라운드
        for (int a = 1; a <= n; a++) //출발지
            for (int b = 1; b <= n; b++) { //목적지
                graph[a][b] = min(graph[a][b],
                    graph[a][r] + graph[r][b]);
            }

    // 수행된 결과를 출력
    for (int a = 1; a <= n; a++) {
        for (int b = 1; b <= n; b++) {
            // 도달할 수 없는 경우, 무한(INF)이라고 출력
            if (graph[a][b] == INF)
                cout << "INF" << ' ';
            // 도달할 수 있는 경우 거리를 출력
            else
                printf("%3d ", graph[a][b]);
        }
        cout << '\n';
    }
}
```

웜 바이러스

- 문제

신종 바이러스인 웜 바이러스는 네트워크를 통해 전파된다. 한 컴퓨터가 웜 바이러스에 걸리면 그 컴퓨터와 네트워크 상에서 연결되어 있는 모든 컴퓨터는 웜 바이러스에 걸리게 된다.



< 그림 1 >

예를 들어 7대의 컴퓨터가 <그림 1>과 같이 네트워크 상에서 연결되어 있다고 하자. 1번 컴퓨터가 웜 바이러스에 걸리면 웜 바이러스는 2번과 5번 컴퓨터를 거쳐 3번과 6번 컴퓨터까지 전파되어 2, 3, 5, 6 네 대의 컴퓨터는 웜 바이러스에 걸리게 된다. 하지만 4번과 7번 컴퓨터는 1번 컴퓨터와 네트워크상에서 연결되어 있지 않기 때문에 영향을 받지 않는다.

어느 날 1번 컴퓨터가 웜 바이러스에 걸렸다. 컴퓨터의 수와 네트워크 상에서 서로 연결되어 있는 정보가 주어질 때, 1번 컴퓨터를 통해 웜 바이러스에 걸리게 되는 컴퓨터의 수를 출력하는 프로그램을 작성하시오.

- 출처: 한국정보올림피아드(2004 지역본선 초등부 3번)

웜 바이러스

- 입력

  첫째 줄에는 컴퓨터의 수가 주어진다. 컴퓨터의 수는 100 이하인 양의 정수이고 각 컴퓨터에는 1번 부터 차례대로 번호가 매겨진다. 둘째 줄에는 네트워크 상에서 직접 연결되어 있는 컴퓨터 쌍의 수가 주어진다. 이어서 그 수만큼 한 줄에 한 쌍씩 네트워크 상에서 직접 연결되어 있는 컴퓨터의 번호 쌍이 주어진다.

- 출력

  1번 컴퓨터가 웜 바이러스에 걸렸을 때, 1번 컴퓨터를 통해 웜 바이러스에 걸리게 되는 컴퓨터의 수를 첫째 줄에 출력한다.

- 입력과 출력의 예

입력 예	출력 예
7 6 1 2 2 3 1 5 5 2 5 6 4 7	4

- 출처: 한국정보올림피아드(2004 지역본선 초등부 3번)

웜 바이러스 풀이

```
#include <iostream>
#define INF 1e9 // 무한을 의미하는 값으로 10억을 설정
using namespace std;

// 노드의 개수(N), 간선의 개수(M)
// 노드의 개수는 최대 101개라고 가정
int n, m;
// 2차원 배열(그래프 표현)를 만들기
int graph[101][101];

int main(void) {
    cin >> n >> m;

    // 최단 거리 테이블을 모두 무한으로 초기화
    for (int i = 0; i < 101; i++) {
        fill(graph[i], graph[i] + 101, INF);
    }

    // 자기 자신에서 자기 자신으로 가는 비용은 0으로 초기화
    for (int a = 1; a <= n; a++) {
        for (int b = 1; b <= n; b++) {
            if (a == b) graph[a][b] = 0;
        }
    }
    // 각 간선에 대한 정보를 입력 받아, 그 값으로 초기화
    for (int i = 0; i < m; i++) {
        // A에서 B로 가는 거리 1로 설정
        int a, b;
        cin >> a >> b;
        graph[a][b] = 1;  // a에서 b로
        graph[b][a] = 1;  // b에서 a로 (양방향 간선)
    }

    // 플로이드 워셜 알고리즘을 수행
    for (int k = 1; k <= n; k++)
        for (int a = 1; a <= n; a++)
            for (int b = 1; b <= n; b++)
                graph[a][b] =
                  min(graph[a][b], graph[a][k] + graph[k][b]);

    int count=0;
    for(int i = 1; i <= n; i++) {
        if(graph[1][i] != INF) //INF 아니면, 연결되어 있다는 뜻
            count++;
    }
    // 1번 자신은 제외
    cout << count-1 << endl;
}
```

DP (동적계획법)

(Dynamic Programming)

다이나믹 프로그래밍(DP)

- DP란?
  - 큰 문제를 작은 문제로 나누어 푸는 문제를 일컫는 말
  - 한 번 계산한 문제는 다시 계산하지 않도록 하는 알고리즘
- DP의 사용조건
  - 최적 부분 구조(Optimal Substructure)

    큰 문제를 작은 문제로 나눌 수 있고, 작은 문제의 답을 모아 큰 문제를 해결할 수 있는 경우를 의미
  - 중복되는 부분 문제(Overlapping Subproblem)

    동일한 작은 문제를 반복적으로 해결해야 하는 경우

- DP 사용하기
  - DP는 특정한 경우에 사용하는 알고리즘이 아니라 하나의 방법론이므로 다양한 문제해결에 사용가능
  - 진행과정

    1) DP로 풀 수 있는 문제인지 확인한다.

    2) 문제의 변수 파악

    3) 변수 간 관계식 만들기(점화식)

    4) 메모하기(memoization or tabulation)

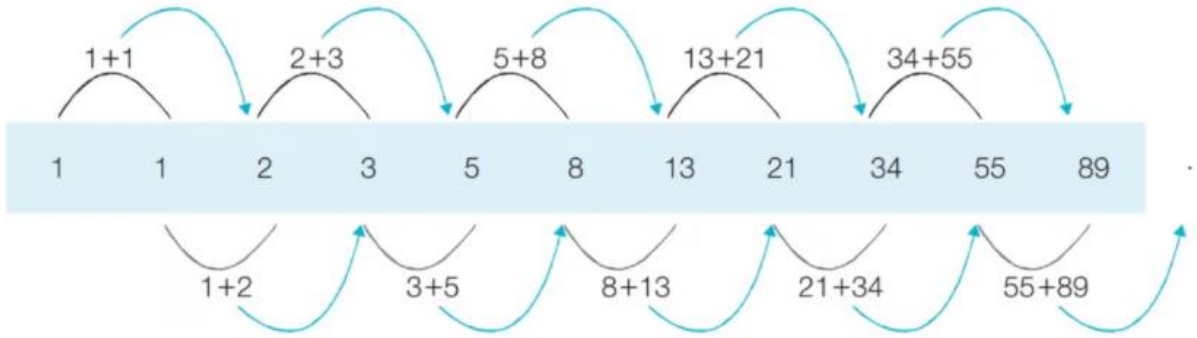
    5) 기저 상태 파악하기

    6) 구현하기

다이나믹 프로그래밍(DP)

- 피보나치 수열

  피보나치 수열이란 이전 두 항의 합을 현재의 항으로 설정하는 특징을 가진 수열



$$\text{Fibo} = \begin{cases} a_1 = 1, \ a_2 = 1 \\ a_n = a_{n-1} + a_{n-2} \quad n \geq 2 \end{cases}$$

- 재귀함수를 통한 피보나치 수열 구현

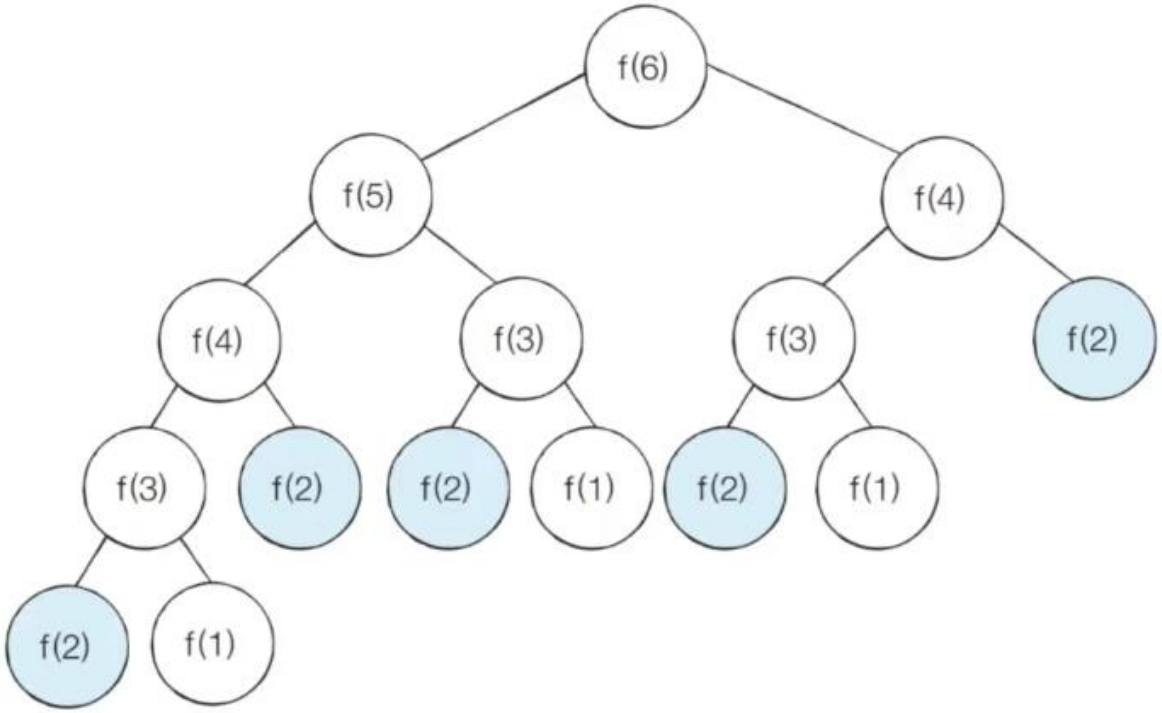
```
#include <stdio.h>

int fibo(int x) {
   if(x==1 || x==2)
      return 1;
   else
      return fibo(x-1) + fibo(x-2);
}

int main() {
    printf("%d", fibo(6));
}
```

다이나믹 프로그래밍(DP)

- 재귀함수로 구현했을 때 문제점



- n이 커지면 커질수록 수행시간이 기하급수적으로 늘어난다.
- f(6)을 계산할 때에 그림과 같이 f(2)가 여러 번 호출되는 것을 확인할 수 있다.
- 즉, 같은 연산을 여러 번 수행한다는 뜻이고 이를 '중복되는 부분 문제'라고 하며 이럴 때 DP가 필요하다.
- 피보나치 수열의 시간 복잡도는 $O(n^2)$ 이다. 예를 들어 f(30)을 계산하기 위해 약 10억 번의 연산을 수행해야 한다.

다이나믹 프로그래밍(DP)

- DP로 피보나치 수열 계산하기
  - DP는 항상 사용할 수 없기 때문에 DP의 사용 조건을 만족하는지 확인 필요
  - DP의 사용조건
    1) 큰 문제를 작은 문제로 나눌 수 있다.
    2) 작은 문제에서 구한 정답은 그것을 포함하는 큰 문제에서도 동일하다.
      - f(30)을 구하기 위해 필요한 f(10) 값이,
      - f(20)을 구하기 위해 필요한 f(10) 값과 동일할 때,

- 메모이제이션(Memoization) 이란?
  - DP를 구현하는 방법 중 한 종류
  - 한 번 구한 결과를 메모리 공간에 메모해 두고 같은 식을 호출하면 메모한 결과를 그대로 가져오는 기법
  - 값을 기록해 놓는다는 점에서 캐싱(Caching)이라고도 한다.

다이나믹 프로그래밍(DP)

- DP로 피보나치 수열 구현(재귀적)

```
#include <stdio.h>
// 한번 계산한 결과를 메모이제이션 하기 위한 배열
unsigned long long D[100];

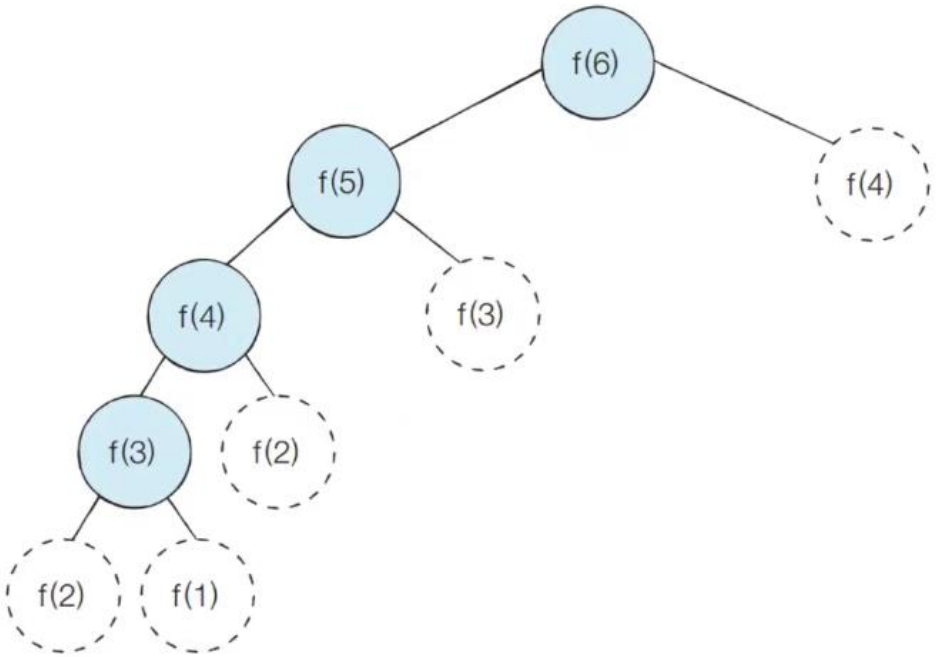
unsigned long long fibo(int x) {
    printf("f(%d) ", x);
    if(x == 1 || x == 2) return 1;
    // 이미 계산한 적 있는 문제라면 그대로 반환
    if(D[x] != 0) return D[x];
    // 아직 계산하지 않은 문제라면 계산
    D[x] = fibo(x-1) + fibo(x-2);
    return D[x];
}

int main() {
    printf("\n%llu", fibo(6));
    return 0;
}
```

- 출력 결과

```
f(6) f(5) f(4) f(3) f(2) f(1) f(2) f(3) f(4)
```

- 호출되는 순서



다이나믹 프로그래밍(DP)

- DP 안한 재귀호출

```
#include <stdio.h>

unsigned long long fibo(int x) {
   if(x==1 || x==2)
      return 1;
   else
      return fibo(x-1) + fibo(x-2);
}

int main() {
    printf("%llu", fibo(50));
}
```

```
12586269025
Process returned 0 (0x0)   execution time : 20.853 s
Press any key to continue.
```

- DP 한 재귀호출

```
#include <stdio.h>
unsigned long long D[100];

unsigned long long fibo(int x) {
   if(x <= 2) return 1;

   if(D[x] != 0) return D[x];

   D[x] = fibo(x-1) + fibo(x-2);
   return D[x];
}

int main() {
    printf("%llu", fibo(50));
}
```

```
12586269025
Process returned 0 (0x0)   execution time : 0.033 s
Press any key to continue.
```

다이나믹 프로그래밍(DP)

✓ DP 탑다운(Top-Down) vs 바텀업(Bottom-Up)

- 탑다운(Top-Down)
  - 하향식 (메모이제이션 or 메모 전략) 이라고도 함
  - 큰 문제를 해결하기 위해 작은 문제를 호출하는 방식
  - 여전히 재귀호출을 사용함
  - 점화식을 이해하기 쉬운 장점
- 바텀업(Bottom-Up)
  - 상향식 이라고도 함
  - 가장 작은 문제들부터 답을 구해가며 전체 문제의 답을 찾는 방식
  - 모든 중간 답을 다 찾아냄
  - 재귀 호출을 하지 않기 때문에 시간과 메모리 사용량을 줄일 수 있는 장점

하향식을 사용하든 상향식을 사용하든 계산과 값의 흐름은 언제나 상향이다.

다이나믹 프로그래밍(DP)

- 피보나치 수열 DP 탑다운 구현

```
#include <stdio.h>
unsigned long long D[100];

unsigned long long fibo(int x) {
   if(x==1 || x==2) return 1;

   // 이미 계산한 적 있으면 그대로 반환
   if(D[x] != 0) return D[x];

   // 아직 계산하지 않은 문제라면 계산
   D[x] = fibo(x-1) + fibo(x-2);
   return D[x];
}

int main() {
    printf("%llu", fibo(50));
}

// 하향식 방법이 더 좋은가?
```

- 피보나치 수열 DP 바텀업 구현

```
#include <stdio.h>
#define MAX 100
unsigned long long D[MAX+1];

void dp() {
    D[1] = D[2] = 1;
    for(int i=3; i<=MAX; i++)
        D[i] = D[i-1] + D[i-2];
}

unsigned long long fibo(int x) {
    return D[x];
}

int main() {
    dp();
    printf("%llu", fibo(50));
}
// 상향식 방법이 더 좋은가?
```

다이나믹 프로그래밍(DP)

- 상향식 다이나믹 프로그래밍이 좋지 않은 경우
  - 대부분의 경우 하향식으로 문제를 푸는 것보다 상향식으로 문제를 푸는 것이 좋다. 하향식은 재귀호출로 인해 발생하는 부하 때문에 속도가 더 느리기 때문이다.
  - 하지만 경우에 따라서 하향식 풀이법을 선택해야 할 수도 있다.
  - 하향식 접근 방법은 모든 하위 문제를 풀지 않고 전체 문제의 해답을 얻는데 필요한 하위 문제만을 푼다.
  - 상향식 다이나믹 프로그래밍에서는 전체 문제의 풀이에 도달하기 전 모든 하위 문제에 대해서 계산을 수행한다.
  - 따라서 상향식 방법은 드물게 실제 필요한 것보다 훨씬 더 많은 하위 문제를 풀어야 하는 경우가 있다.
  - 따라서 이를 살펴야 한다.

다이나믹 프로그래밍(DP)

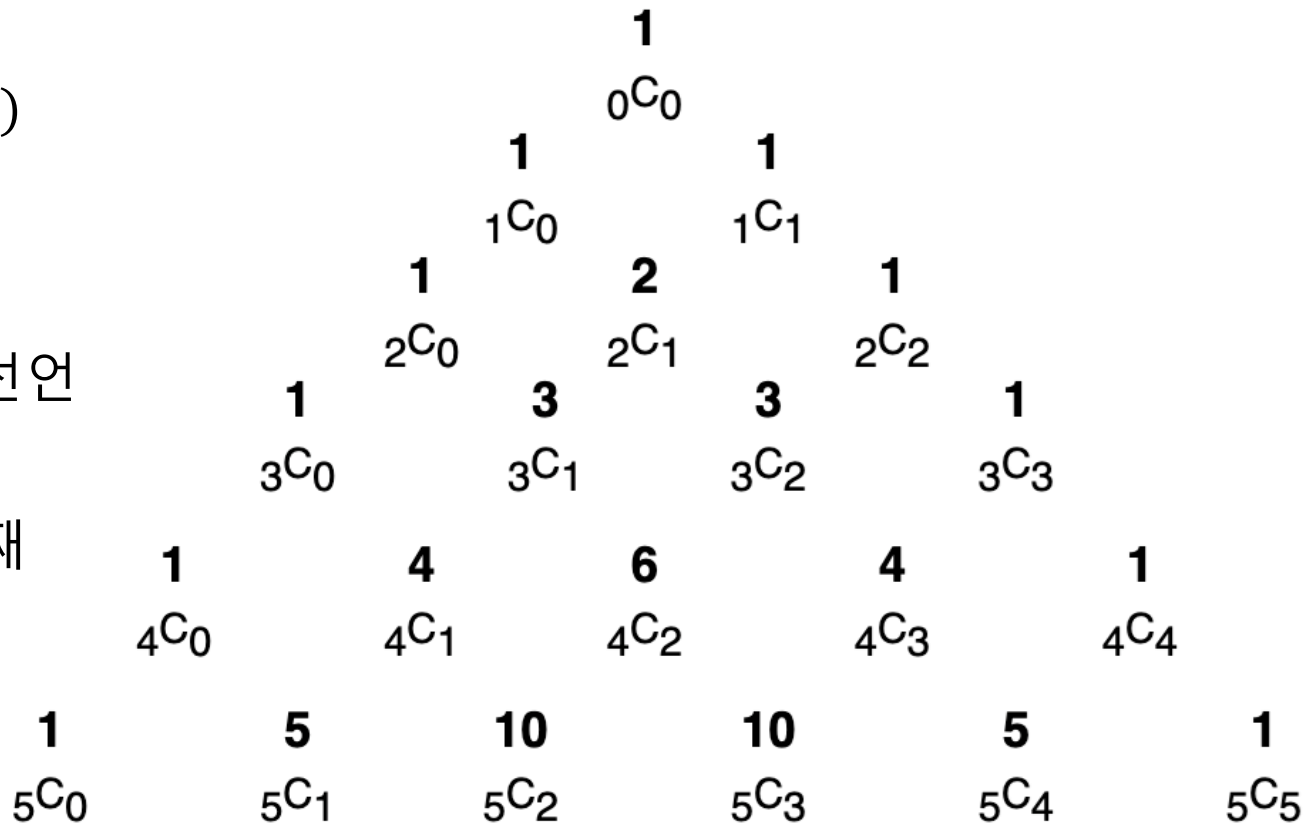
- 하향식 방법이 더 유리한 예시
  - 수학에서 조합 계산

    $C(n,m) = C(n-1,\ m) + C(n-1, m-1)$

    ex) $C(5,4) = \mathrm{C}(4,4) + \mathrm{C}(4,3)$

  - 상향식에서는 D[n+1][m+1] 배열을 선언하고,
  - 파스칼의 삼각형을 구성한 다음 n번째 행의 m번째 값을 반환 (zero-base index)

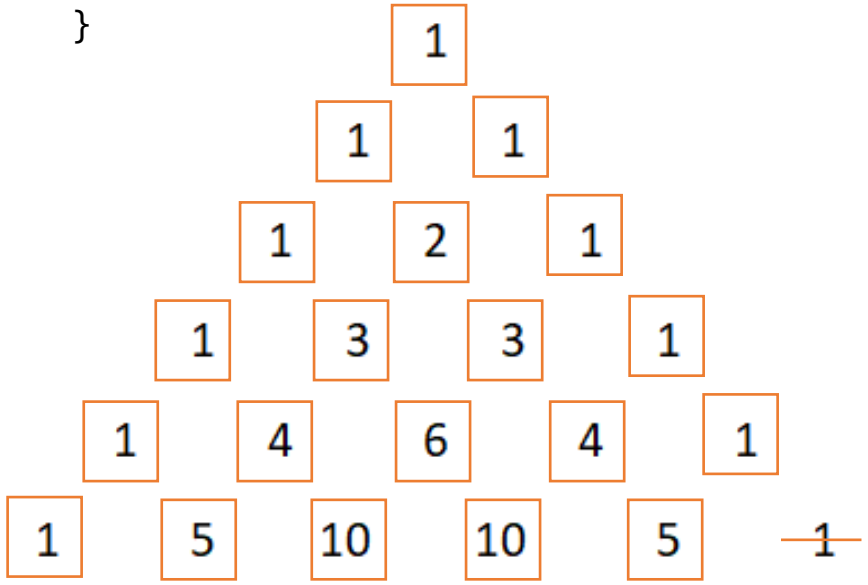
- 파스칼의 삼각형



다이나믹 프로그래밍(DP)

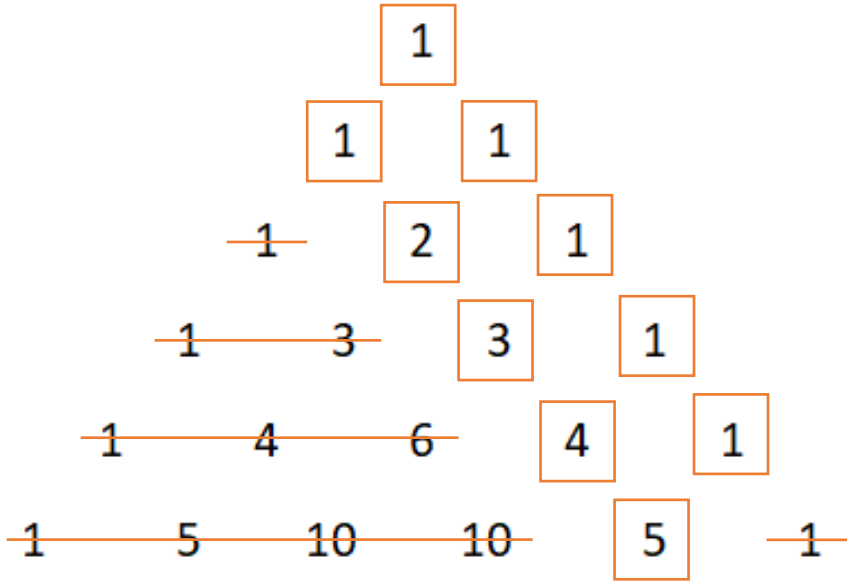
- 상향식 풀이

```
for(int a=0; a<=n; a++) {
    for(int b=0; b<=m; b++) {
        if(a==0 || b==0 || a==b)
            D[a][b] = 1;
        else
            D[a][b] = D[a-1][b] + D[a-1][b-1];
    }
}
```



- 하향식 풀이

```
int combi(int n,  int m) {
    if(n==0 || m==0 || n==m)
        return 1;
    else
        return combi(n-1, m)+combi(n-1, m-1);
}
```



다이나믹 프로그래밍(DP)

- 파스칼 삼각형(하향식)

```
#include <iostream>

int n, m;
int D[50][50];

void output_pascal_trangle() {
    putchar('\n');
    for(int a=0; a<=n; a++) {
        for(int b=0; b<=m; b++) {
            if(b<=a) {
                printf("%d ", D[a][b]);
            }
        }
        putchar('\n');
    }
}
```

```
int combi(int n,  int m) {
    if(n==0 || m==0 || n==m)
        return D[n][m] = 1;
    else
        return D[n][m] = combi(n-1, m)+combi(n-1, m-1);
}

int main() {
    scanf("%d %d", &n, &m);
    printf("%d\n", combi(n, m));

    output_pascal_trangle();
    return 0;
}
```

```
5 4
5

0
1 1
0 2 1
0 0 3 1
0 0 0 4 1
0 0 0 0 5
```

다이나믹 프로그래밍(DP)

- 파스칼 삼각형(상향식)

```
#include <iostream>
int n, m;
int D[50][50];

void make_pascal_triangle() {
    putchar('\n');
    for(int a=0; a<=n; a++) {
        for(int b=0; b<=m; b++) {
            if(b<=a) {
                if(a==0 || b==0 || a==b)
                    D[a][b] = 1;
                else
                    D[a][b] = D[a-1][b] + D[a-1][b-1];
                printf("%d ", D[a][b]);
            }
        }
        putchar('\n');
    }
    putchar('\n');
}
```

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

    printf("%d\n", D[n][m]);
    return 0;
}
```

```
5 4

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5

5
```

문제: 계단오르기

5명이 해결한 문제

- 문제

  1층에서 2층으로 올라가는 계단을 생각해 보자 여러분은 계단을 어떻게 올라가는가? 안전하게 한 칸, 한 칸씩 오르는가? 아니면 성큼성큼 두 칸씩 오르는가? 아니면 한 칸 또는 두 칸 마음 내키는 대로...? 아마도 수많은 방법이 있을 것이다.

  초등학생인 충북이는 아직 다리가 짧아 한 걸음에 계단을 최대 3개까지 오를 수 있다.

  충북이가 n개의 계단을 오르는 모든 방법의 수를 계산하는 프로그램을 작성하시오.

  예를 들어 2개의 계단으로 구성되어 있다면,

  ① 한 칸, 한 칸

  ② 두 칸

  위와 같이 두 가지 방법이 존재하고,

  예를 들어 3개의 계단으로 구성되어 있다면,

  ① 한 칸, 한 칸, 한 칸

  ② 한 칸, 두 칸

  ③ 두 칸, 한 칸

  ④ 세 칸

  위와 같이 네 가지 방법이 존재한다.

- 입력형식

  첫 번째 줄에 계단의 수 자연수 N이 입력된다.

  (1 ≤ N ≤ 36)

- 출력형식

  첫 번째 줄에 계단을 오르는 방법의 수를 자연수로 출력한다.

입력 예	출력 예
3	4

- 출처: 2023 충북정올 학교예선 초등부 4번

풀이: 계단오르기

- 고찰

계단 수	오르는 방법		방법 개수	
(1)	①	①	1	1
(2)	(1)+① ②	①+① ②	1 1	2
(3)	(1)+② (2)+① ③	①+② ①+①+①, ②+① ③	1 2 1	4
(4)	(1)+③ (2)+② (3)+①	①+③ ①+①+②, ②+② ①+②+①, ①+①+①+①, ②+①+①, ③+①	1 2 4	7
(5)	(2)+③ (3)+② (4)+①	①+①+③, ②+③ 생략 생략	2 4 7	13

- 점화식 표현

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

풀이: 계단오르기

```
// 풀이 1: 재귀 호출
// 계단의 수가 33이 넘어가면 제한시간 1초 내에 해결 불가능
#include <stdio.h>

int methods(int f) {
    if(f == 1)
        return 1;
    else if(f == 2)
        return 2;
    else if(f == 3)
        return 4;
    else
        return methods(f-3)+methods(f-2)+methods(f-1);
}

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

```
#include <stdio.h>
#define  N  40
int memo[N];

int methods(int f) {
    if(memo[f] != 0)
        return memo[f];

    if(f == 1)      memo[1]=1;
    else if(f == 2) memo[2]=2;
    else if(f == 3) memo[3]=4;
    else   memo[f] = methods(f-3) + methods(f-2) + methods(f-1);

    return memo[f];
}

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

거스름돈II

- 문제

  N가지 종류의 화폐가 있다. 이 화폐들을 최소한으로 이용해서 거스름돈 M원을 만들려고 한다. 이 때 각 화폐는 몇 개라도 사용할 수 있으며, 사용한 화폐의 구성은 같지만 순서만 다른 것은 같은 경우로 구분한다.

  예를 들어 500원, 100원, 50원, 10원 단위의 화폐가 있을 때, 거스름돈 1270원을 만들려면 500원 2개, 100원 2개, 50원 1개, 10원 2개 총 7개를 사용하는 것이 가장 최소한의 화폐 개수이다.

입력 예	출력 예
4 1270 500 100 50 10	7
4 35 1 2 7 10	5

- 입력값

  첫째 줄에 M, N이 주어진다.

  (1 <= N <= 100, 1 <= M <= 10,000)

  둘째 줄에는 각 화폐의 가치가 공백으로 구분되어 N개 주어진다. 화폐의 가치는 10,000보다 작거나 같은 자연수 이다.

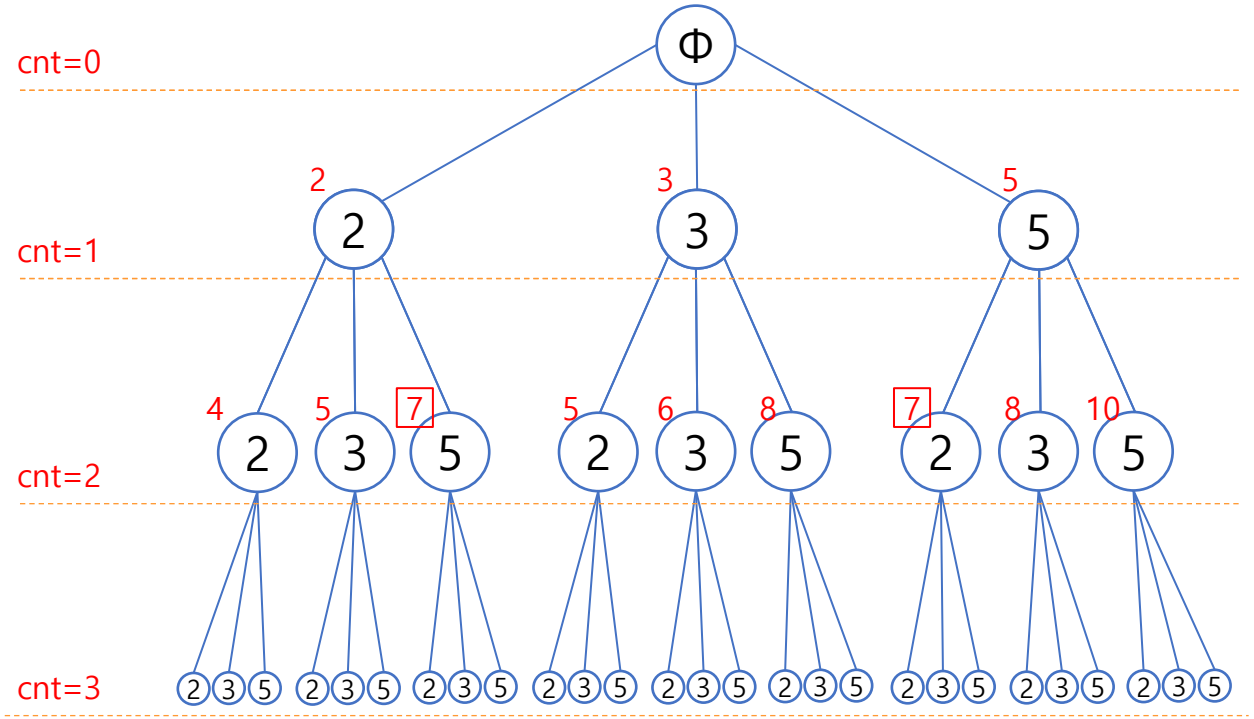
- 출력값

  M원을 만들기 위해 필요한 최소 화폐 개수를 출력한다. 불가능할 때는 -1을 출력한다.

입력 예	출력 예
3 4 3 5 7	-1

거스름돈II – DFS 풀이

- DFS 탐색
  - 화폐 단위: 2원, 3원, 5원 일 때,



- dfs 소스코드

```
//cnt: 사용한 화폐 수, won: 현재 금액
void dfs(int cnt, int won) {
    //더 많은 화폐를 써야 한다면 서브트리 탐색 중단
    if(cnt > sol) return;
    //만들어낸 금액이 목표보다 크면 서브 트리 탐색 중단
    if(won > m)   return;

    if(won == m) {
        if(cnt < sol) {
            sol = cnt;
        }
        return;
    }

    for(int i=0; i<n; i++) {
        dfs(cnt+1, won+c[i]);
    }
}
```

거스름돈II - DFS를 이용한 풀이 기본

dfs_make_chang.cpp

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

int c[10]; // 화폐단위 저장 배열
int n, m;
int sol = INT_MAX;
int methods = 0;  // 몇 가지 방법이 존재하는가?

void dfs(int cnt, int won);

int main() {
    scanf("%d %d", &n, &m);
    for(int i=0; i<n; i++)
        scanf("%d", &c[i]);

    dfs(0, 0);  // 동전0개, 0원에서 출발
    if(sol != INT_MAX)
        printf("%d\n", sol);
    else
        printf("-1\n");
}
```

```
3 7
2 3 5
2
```

```
//cnt: 사용한 화폐수, won: 현재 금액
void dfs(int cnt, int won) {
    //만들어낸 금액이 목표보다 크면 서브 트리 탐색 중단
    if(won > m)   return;
    //지금까지 알아낸 방법보다 더 많은 화폐를 써야 한다면 중단
    if(cnt > sol) return;

    if(won == m) {
        if(cnt < sol) {
            sol = cnt;
        }
        return;
    }

    for(int i=0; i<n; i++) {
        dfs(cnt+1, won+c[i]);
    }
}


// M원을 만들기 위한 최소 화폐 조합이 몇 가지 있는지 알아 내려면?
// 해당 화폐 조합을 추적하려면?
```

거스름돈II - DFS를 이용한 풀이 업그레이드

dfs_make_chang_methods.cpp

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

int c[10]; // 화폐단위 저장 배열
int n, m;
int sol = INT_MAX;
int methods = 0;  // 몇 가지 방법이 존재하는가?
vector<int> v;
void dfs(int cnt, int won);

void output() {
    for(int x : v)
        printf("%3d,", x);
    printf("\b (%d)\n", sol);
}

int main() {
    scanf("%d %d", &n, &m);
    for(int i=0; i<n; i++)
        scanf("%d", &c[i]);

    dfs(0, 0);  // 동전0개, 0원에서 출발
    if(sol != INT_MAX)
        printf("%d\n%d\n", sol, methods);
    else
        printf("-1\n");
}
```

```
//cnt: 사용한 화폐수, won: 현재 금액
void dfs(int cnt, int won) {
    //만들어낸 금액이 목표보다 크면 서브 트리 탐색 중단
    if(won > m)   return;
    //지금까지 알아낸 방법보다 더 많은 화폐를 써야 한다면 중단
    if(cnt > sol) return;

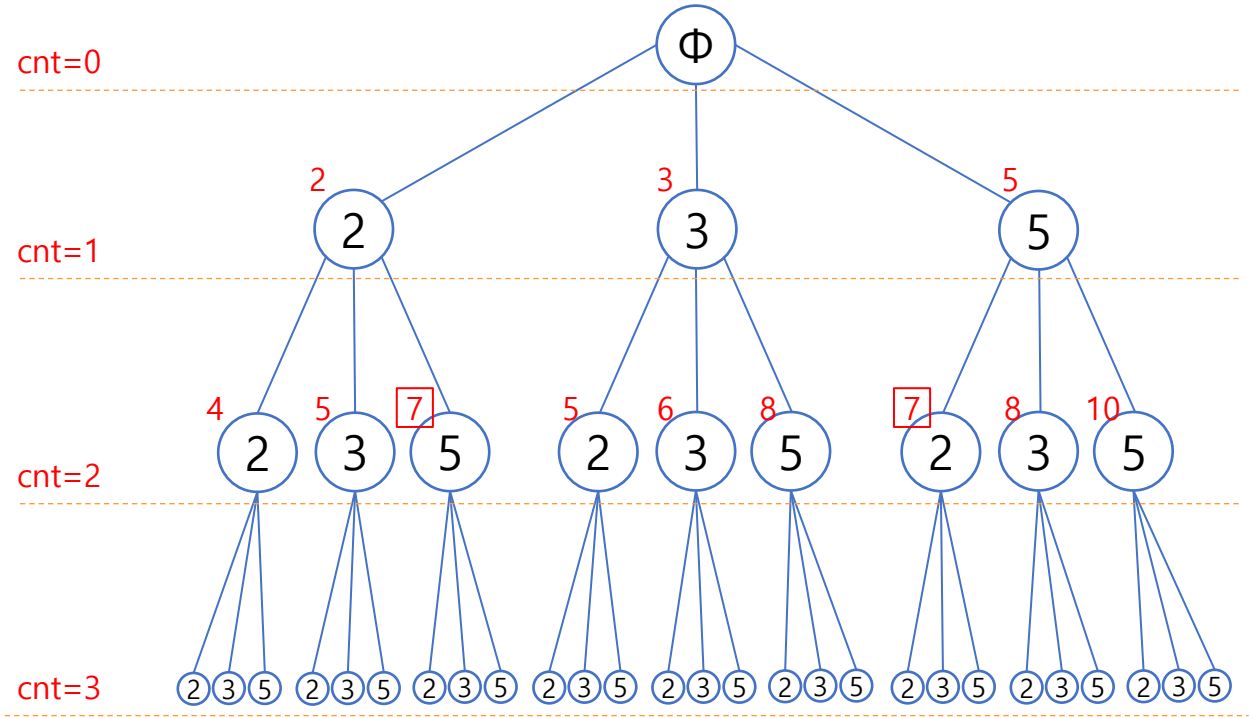
    if(won == m) {
        if(cnt < sol) {
            sol = cnt;
            methods = 1;
            output();
        }
        else if(cnt == sol) {
            methods++;
            output();
        }
        return;
    }

    for(int i=0; i<n; i++) {
        v.push_back(c[i]);
        dfs(cnt+1, won+c[i]);
        v.pop_back();
    }
}
```

```
3 7
2 3 5
  2,  2,  3 (3)
  2,  3,  2 (3)
  2,  5 (2)
  5,  2 (2)
2
```

거스름돈II – BFS 풀이

- BFS 탐색
  - 화폐 단위: 2원, 5원, 7원 일 때,




```
typedef struct {
    int cnt;
    int won;
} node;

void bfs() {
    queue <node> Q;
    Q.push({0, 0});  // 0개, 0원에서 시작

    while(! Q.empty()) {
        node v = Q.front();  //큐의 첫 원소
        Q.pop();             //뽑아내기

        if(v.won == m) { // 목표 금액에 도달하면
            sol = v.cnt;
            break;
        }

        for(int i=0; i<n; i++) {
            // 목표 금액 m 이하 일때만 서브노트 탐색
            if(v.won+c[i] <= m)
                Q.push({v.cnt+1, v.won+c[i]});
        }
    }
}
```

거스름돈II - BFS를 이용한 풀이 기본

bfs_make_chang.cpp

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

typedef struct {
    int cnt;
    int won;
}node;

int c[10];          // 화폐단위 저장 배열
int n, m;
int sol = INT_MAX; // 답(최소 화폐 개수)
void bfs();

int main() {
    scanf("%d %d", &n, &m);
    for(int i=0; i<n; i++)
        scanf("%d", &c[i]);

    bfs();

    if(sol != INT_MAX)
        printf("%d\n", sol);
    else
        printf("-1\n");
}
```

```
void bfs() {
    queue <node> Q;
    Q.push({0, 0});  // 0개, 0원에서 시작

    while(! Q.empty()) {
        node v = Q.front();
        Q.pop();

        if(v.won == m) { // 목표 금액에 도달하면
            sol = v.cnt;
            break;
        }

        for(int i=0; i<n; i++) {
            // 목표 금액 m이하일 때만 서브노드 탐색
            if(v.won+c[i] <= m) {
                Q.push({v.cnt+1, v.won+c[i]});
            }
        }
    }
}

// M원을 만들기 위한 최소 화폐 조합이 몇 가지 있는지 알아 내려면?
// 해당 화폐 조합을 추적하려면?
```

```
3 7
2 3 5
2
```

거스름돈II - BFS를 이용한 풀이 업그레이드

bfs_make_chang_methods.cpp

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

typedef struct {
    int cnt;
    int won;
    vector<int> cv;
}node;

int c[10];          // 화폐단위 저장 배열
int n, m;
int sol = INT_MAX; // 답(최소 화폐 개수)
int methods = 0;   // 몇 가지 방법이 존재하는가?
void bfs();

int main() {
    scanf("%d %d", &n, &m);
    for(int i=0; i<n; i++)
        scanf("%d", &c[i]);

    bfs();

    if(sol != INT_MAX)
        printf("%d\n%d\n", sol, methods);
    else
        printf("-1\n");
}
```

```
void bfs() {
    queue <node> Q;
    Q.push({0, 0, vector<int>()}); // 0개, 0원에서 시작

    while(! Q.empty()) {
        node v = Q.front();
        Q.pop();

        // 찾아낸 답보다 크면 더 이상 탐색할 필요 없음
        if(v.cnt > sol) break;

        if(v.won == m) { // 목표 금액에 도달하면
            sol = v.cnt;
            methods++;
            //break;  //답 한 개 찾았다고 바로 나가면 안됨.
            printf("[%d]: ", sol);
            for(int c: v.cv)
                printf("%3d, ", c);
            printf("\b \n");
        }

        for(int i=0; i<n; i++) {
            // 목표 금액 m이하일 때만 서브노드 탐색
            if(v.won+c[i] <= m) {
                vector<int> cv(begin(v.cv), end(v.cv));
                cv.push_back(c[i]);
                Q.push({v.cnt+1, v.won+c[i], cv});
            }
        }
    }
}
```

```
3 7
2
3
5
[2]:   2,   5,
[2]:   5,   2,
2 ←sol
2 ← methods
```

거스름돈II – DP를 이용한 풀이

- 화폐단위가 2원, 3원, 5원 인 경우 문제 해설

  화폐의 단위: k, 금액 t를 만들 수 있는 최소한의 화폐 개수: a_t

  a_{t-k}는 금액 (t-k)를 만들 수 있는 최소한의 화폐 개수

  - 점화식

    a_{t-k}를 만드는 방법이 존재하는 경우, $a_t=\min(a_t, a_{t-k}+1)$

    a_{t-k}를 만드는 방법이 존재하지 않는 경우, $a_t=99999$

  - 예

    (2원으로 7원을 만드는 방법) $a_7 = \min($7원 만드는 방법, 5원 만드는 방법+1 $) = \min(a_7, a_5+1)$

    (3원으로 7원을 만드는 방법) $a_7 = \min(a_7,$ 4원 만드는 방법+1 $) = \min(a_7, a_4+1)$

    (5원으로 7원을 만드는 방법) $a_7 = \min(a_7,$ 2원 만드는 방법+1 $) = \min(a_7, a_2+1)$

거스름돈II - 바텀업 DP 구현

- 화폐단위가 2원, 3원, 5원 인 경우 문제 해설
  - 초기화

99999은 불가능 하다는 의미
0은 0개로 만들 수 있다.

idx	0	1	2	3	4	5	6	7		…
값	0	99999	99999	99999	99999	99999	99999	99999		…

(1) 2원 짜리로 몇 번 만에 만들 수 있는가?

idx	0	1	2	3	4	5	6	7	8	…
값	0	99999	1	99999	2	99999	3	99999	4	…

$a_2 = \min(a_2, a_0+1) = \min(99999, 0+1) = 1$

$a_3 = \min(a_3, a_1+1) = \min(99999, 99999+1) = 99999$

$a_4 = \min(a_4, a_2+1) = \min(99999, 1+1) = 2$

$a_5 = \min(a_5, a_3+1) = \min(99999, 99999+1) = 99999$

$a_6 = \min(a_6, a_4+1) = \min(99999, 2+1) = 3$

거스름돈II - 바텀업 DP 구현

- 화폐단위가 2원, 3원, 5원 인 경우 문제 해설

(2) 3원 짜리를 추가로 사용하면 몇 번 만에 만들 수 있는가?

전

idx	0	1	2	3	4	5	6	7	8	…
값	0	99999	1	99999	2	99999	3	99999	4	…

후

idx	0	1	2	3	4	5	6	7	8	…
값	0	99999	1	1	2	2	2	3	3	…

$a_3 = \min(a_3, a_0+1) = \min(99999, 0+1) = 1$

$a_4 = \min(a_4, a_1+1) = \min(2, 99999+1) = 2$

$a_5 = \min(a_5, a_2+1) = \min(99999, 1+1) = 2$

$a_6 = \min(a_6, a_3+1) = \min(3, 1+1) = 2$

$a_7 = \min(a_7, a_4+1) = \min(99999, 2+1) = 3$

$a_8 = \min(a_8, a_5+1) = \min(4, 2+1) = 3$

거스름돈II - 바텀업 DP 구현

- 화폐단위가 2원, 3원, 5원 인 경우 문제 해설

(3) 5원 짜리를 추가로 사용하면 몇 번 만에 만들 수 있는가?

전

idx	0	1	2	3	4	5	6	7	8	…
값	0	99999	1	1	2	2	2	3	3	…

후

idx	0	1	2	3	4	5	6	7	8	…
값	0	99999	1	1	2	1	2	2	2	…

$a_5 = \min(a_5, a_0+1) = \min(2, 0+1) = 1$

$a_6 = \min(a_6, a_1+1) = \min(2, 99999+1) = 2$

$a_7 = \min(a_7, a_2+1) = \min(3, 1+1) = 2$

$a_8 = \min(a_8, a_3+1) = \min(3, 1+1) = 2$

: :

거스름돈II - 바텀업 DP 구현

dp_make_chang.cpp

```
#include <iostream>  // DP 바텀업 구현
#include <algorithm>
#define  MAX  10000
#define  INF  999
using namespace std;

int d[MAX+1];   // DP 테이블
int c[10];      // 화폐단위 저장 배열
int N, M;

void output(int idx) {
    printf("\n(%2d)\n", c[idx]);
    for(int x=0; x<=M; x++)
        printf(" [%3d]", x);
    puts("");
    for(int x=0; x<=M; x++)
        printf("  %3d,", d[x]);
    puts("");
}
```

```
void dp() {
    d[0] = 0;
    for(int i=1; i<=M; i++)
        d[i]=INF;

    for(int i=0; i<N; i++) { // 각 화폐단위 a[i]에 대하여
        for(int t=c[i]; t<=M; t++) {
            d[t] = min(d[t], d[t-c[i]]+1);
        }
        output(i);
    }
}

int main() {
    scanf("%d %d", &N, &M);
    for(int i=0; i<N; i++)
        scanf("%d", &c[i]);

    dp();

    int min_count = d[M];
    if(min_count != INF)
        printf("%d\n", min_count);
    else
        printf("-1\n");
}
```

거스름돈II - 바텀업 DP 구현

```
3 7
2 3 5

( 2)  2원짜리 사용하였을 때, 최소 개수 계산
 [  0] [  1] [  2] [  3] [  4] [  5] [  6] [  7]
    0,  999,    1,  999,    2,  999,    3,  999,

( 3)  3원짜리 사용하였을 때, 최소 개수 계산
 [  0] [  1] [  2] [  3] [  4] [  5] [  6] [  7]
    0,  999,    1,    1,    2,    2,    2,    3,

( 5)  5원짜리 사용하였을 때, 최소 개수 계산
 [  0] [  1] [  2] [  3] [  4] [  5] [  6] [  7]
    0,  999,    1,    1,    2,    1,    2,    2,
2

Process returned 0 (0x0)   execution time : 4.703 s
Press any key to continue.
```

거스름돈을 만드는 방법

- 문제

  N가지 종류의 화폐가 있다. 이 화폐들을 최소한으로 이용해서 거스름돈 M원을 만들려고 한다. 이 때 각 화폐는 몇 개라도 사용할 수 있으며, 사용한 화폐의 구성은 같지만 순서만 다른 것은 같은 경우로 구분한다.

  합이 정확히 M원이 되도록 만드는 방법의 개수를 구하는 프로그램을 작성하시오. (동전 순서는 무시)

- 입력값

  첫째 줄에 M, N이 주어진다.

  (1 <= N <= 100, 1 <= M <= 10,000)

  이후의 N개의 줄에는 각 화폐의 가치가 주어진다. 화폐의 가치는 10,000보다 작거나 같은 자연수 이다.

- 출력값

  합이 정확히 M원이 되도록 만드는 방법의 개수를 출력한다.

입력 예	출력 예
3 5 1 2 5	4

거스름돈을 만드는 방법

- 손으로 시뮬레이션
  - c원 동전 사용하여 v원을 만드는 방법 = 기존 방법 + (v-c원 만드는 방법에 c원 동전을 추가)
  - dp[v] = dp[v] + dp[v-c]

예) 동전 `{1, 2, 5}`, 목표 `M=5`

초기: `dp = [1, 0, 0, 0, 0, 0]` // 인덱스 = 금액

1. 동전 1 사용:

- v=1..5
  - v=1: dp[1]+=dp[0]=1 → [1,1,0,0,0,0]
  - v=2: dp[2]+=dp[1]=1 → [1,1,1,0,0,0]
  - v=3: dp[3]+=dp[2]=1 → [1,1,1,1,0,0]
  - v=4: dp[4]+=dp[3]=1 → [1,1,1,1,1,0]
  - v=5: dp[5]+=dp[4]=1 → [1,1,1,1,1,1]
  - 의미: 1원만으로 만들 수 있는 모든 금액은 "딱 1가지 방법" (1을 여러 개)

2. 동전 2 사용:

- v=2..5
  - v=2: dp[2]+=dp[0]=1 → [1,1,2,1,1,1]
  - v=3: dp[3]+=dp[1]=1 → [1,1,2,2,1,1]
  - v=4: dp[4]+=dp[2]=2 → [1,1,2,2,3,1]
  - v=5: dp[5]+=dp[3]=2 → [1,1,2,2,3,3]
  - 의미: 2원을 추가로 쓸 수 있는 경우를 누적(순서는 무시)

3. 동전 5 사용:

- v=5..5
  - v=5: dp[5]+=dp[0]=1 → [1,1,2,2,3,4]

결과: `dp[5] = 4`

실제 조합(순서무시):

- 1+1+1+1+1
- 1+1+1+2
- 1+2+2
- 5

거스름돈을 만드는 방법 DP 구현

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

int main() {
    int N, M;
    if (!(cin >> N >> M))
        return 0;

    vector<int> coin(N);
    for (int i = 0; i < N; ++i)
        cin >> coin[i];

    // 1) 같은 액면 중복 제거(중복되면 같은 조합을 여러 번 세게 됨)
    sort(coin.begin(), coin.end());
    coin.erase(unique(coin.begin(), coin.end()), coin.end());

    // 2) dp[v] = 금액 v를 만드는 "순서-무시" 방법 수
    vector<int> dp(M + 1, 0);

    // 3) 기저값: 0원을 만드는 방법 "아무 동전도 사용하지 않는" 1가지
    dp[0] = 1;

    // 4) 동전 바깥 루프(조합만 카운트하기 위해 순서 고정)
    for (int c : coin) {
        if (c > M) continue; // 동전이 M보다 크면 계산 스킵

        // 5) 오름차순 금액 순회(무한 사용 허용)
        for (int v = c; v <= M; ++v) {
            // v를 만들 때 마지막으로 c를 하나 더 붙이는 경우의 수를 더한다.
            // 그 수는 "v - c를 만드는 모두의 경우 수"와 동일.
            dp[v] += dp[v - c];
        }
    }

    // 6) 정답
    cout << dp[M] << "\n";

    return 0;
}
```

거스름돈을 만드는 방법 DP 구현

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

int main() {
    int N, M;
    if (!(cin >> N >> M))
        return 0;

    vector<int> coin(N);
    for (int i = 0; i < N; ++i)
        cin >> coin[i];

    // 반드는 방법을 저장하는 벡터
    vector<vector<vector<int>>> methods(M+1);
    methods[0].push_back({}); // 0원 = 빈 조합 1개

    // 1) 같은 액면 중복 제거(중복되면 같은 조합을 여러 번 세게 됨)
    sort(coin.begin(), coin.end());
    coin.erase(unique(coin.begin(), coin.end()), coin.end());

    // 2) dp[v] = 금액 v를 만드는 "순서-무시" 방법 수
    vector<int> dp(M + 1, 0);

    // 3) 0원을 만드는 방법 "아무 동전도 사용하지 않는" 1가지
    dp[0] = 1;

    // 4) 동전 바깥 루프(조합만 카운트하기 위해 순서 고정)
    for (int c : coin) {
        if (c > M) continue; // 동전이 M보다 더 크면 계산 건너뜀

        // 5) 오름차순 금액 순회(무한 사용 허용)
        for (int v = c; v <= M; v++) { // c원부터 M원까지
            // v를 만들 때 마지막으로 c를 하나 더 붙이는 경우의 수를 더한다.
            // 그 수는 "v - c를 만드는 모두의 경우 수"와 동일.
            dp[v] += dp[v - c];

            for(vector<int> b : methods[v-c]) {
                b.push_back(c);
                methods[v].push_back(b);
            }
        }
    }

    // 6) 정답
    cout << dp[M] << "\n";

    for(vector<int> v: methods[M]) {
        for(int a: v)
            printf("%d,", a);
        printf("\b \n");
    }
    return 0;
}
```

질의 응답

