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

정보올림피아드 마스터

< 중급 >

충북교육연구정보원 정보영재교육원 | SW·AI교실 | 정보아카데미 강사

흥덕고등학교 교사 박정진

진도 계획표

차시	7. 23.(토)	7. 24.(일)
1차시 09:00~09:45	다이나믹 프로그래밍의 이해 하향식/상향식 접근 방법	STL 콘테이너(array, vector, deque), STL 반복자, STL 알고리즘
2차시 09:55~10:40	[기출] 계단오르기 전형적인 다이나믹 문제	queue, heap 자료구조, priority_queue [기출]창고(우선순위 큐)
3차시 10:50~11:35	[기출] 직소퍼즐, 약수의 합	그래프 탐색 알고리즘 - 깊이우선탐색(DFS)
4차시 11:45~12:30	트리 [기출]개미는 어디?(이진트리) [기출]스티커소비	그래프 탐색 알고리즘 - 너비우선탐색(BFS)
점심	중식 개별준비 (도시락 및 외부 식당이용)	중식 개별준비 (도시락 및 외부 식당이용)
5차시 13:30~14:15	[기출]꽃밭(수학적사고) LIS알고리즘	[기출]공성전(DFS) [기출]채널 빨리 바꾸기(DFS vs BFS)
6차시 14:25~15:10	반도체 설계 [기출]오르락내리락	DFS 전체탐색과 속도향상을 위한 가지치기 Q.연구활동 가는길
7차시 15:20~16:05	[기출]카드놀이(구현) [기출]고드름(구현)	다익스트라 알고리즘 DFS vs Dijkstra
8차시 16:15~17:00	[기출]선물게임(투포인터)	프로이드 워셜 알고리즘 Q.웜바이러스

Online Judge 프로그램 계정 정보

순번	소속학교	학년	이름	ID	PW
1	샛별초등학교	초6	오준우	iomaster1	io1234!
2	청주교대부설초등학교	초6	이서준	iomaster2	io1234!
3	석성초등학교	초4	송민호	iomaster3	io1234!
4	국원초등학교	초6	인수	iomaster4	io1234!
5	솔밭중학교	중2	정예찬	iomaster5	io1234!
6	양청중학교	중3	신동민	iomaster6	io1234!
7	서현중학교	중2	박승리	iomaster7	io1234!
8	서경중학교	중3	유동우	iomaster8	io1234!

※ OJ 접속주소: https://oj.datahub.pe.kr/

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

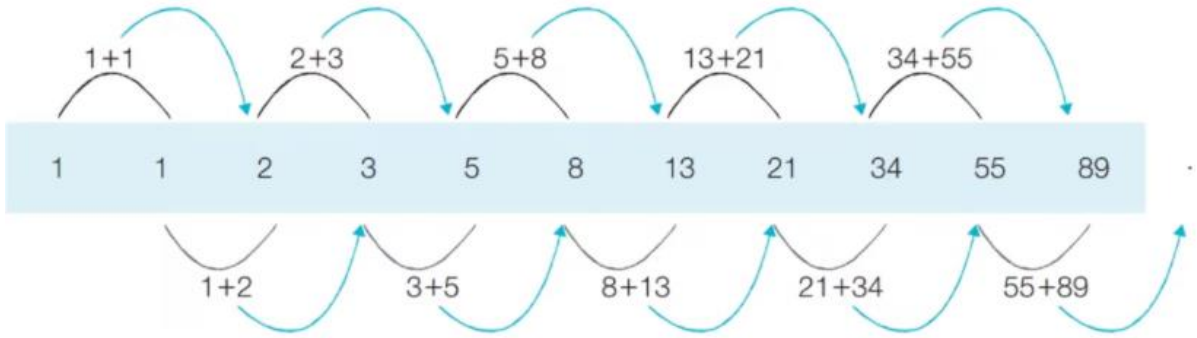
- DP 사용하기
  - DP는 특정한 경우에 사용하는 알고리즘이 아니라 하나의 방법론이므로 다양한 문제해결에 사용가능
  - 진행과정

    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



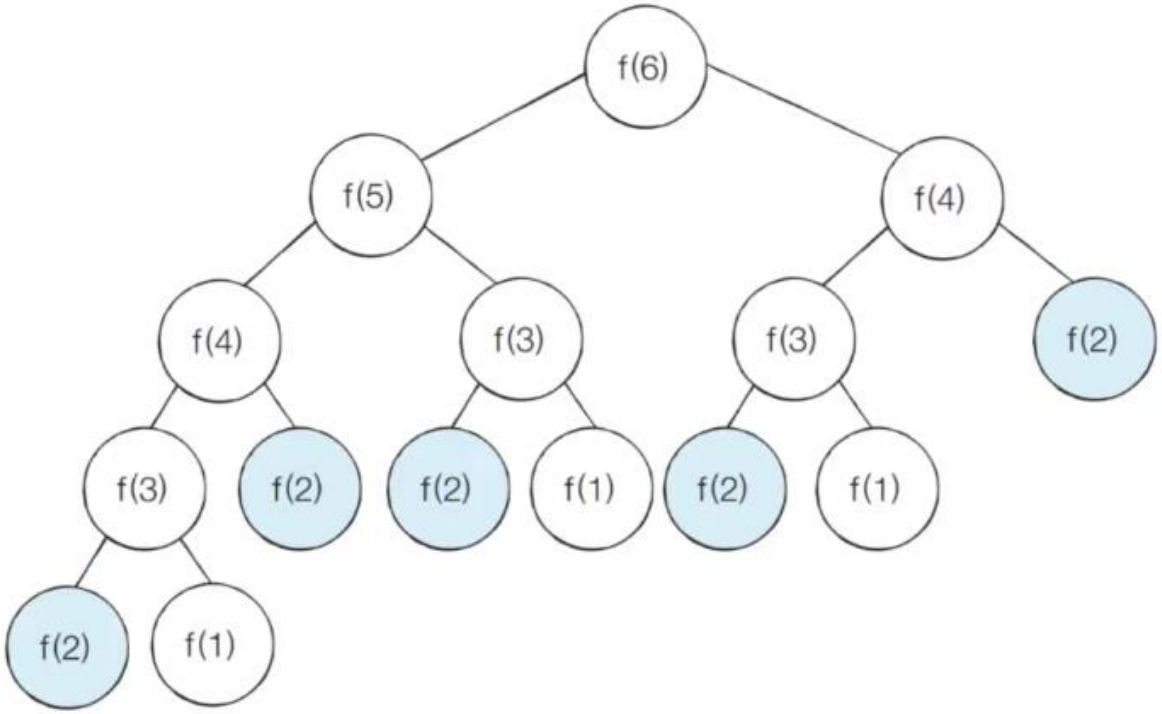
$$\text{Fibo} = \begin{cases} a_1 = 1, \ a_2 = 1 \\ a_n = a_{n-1} + a_{n-2} \quad n \geq 2 \end{cases}$$

- 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)

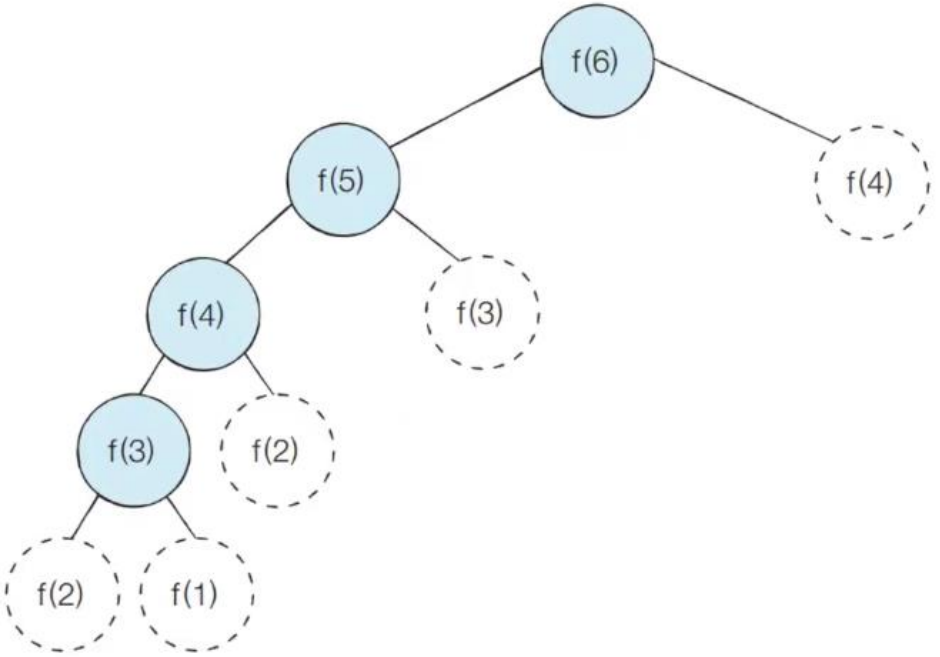
- DP로 피보나치 수열 구현(재귀적)

```
#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)

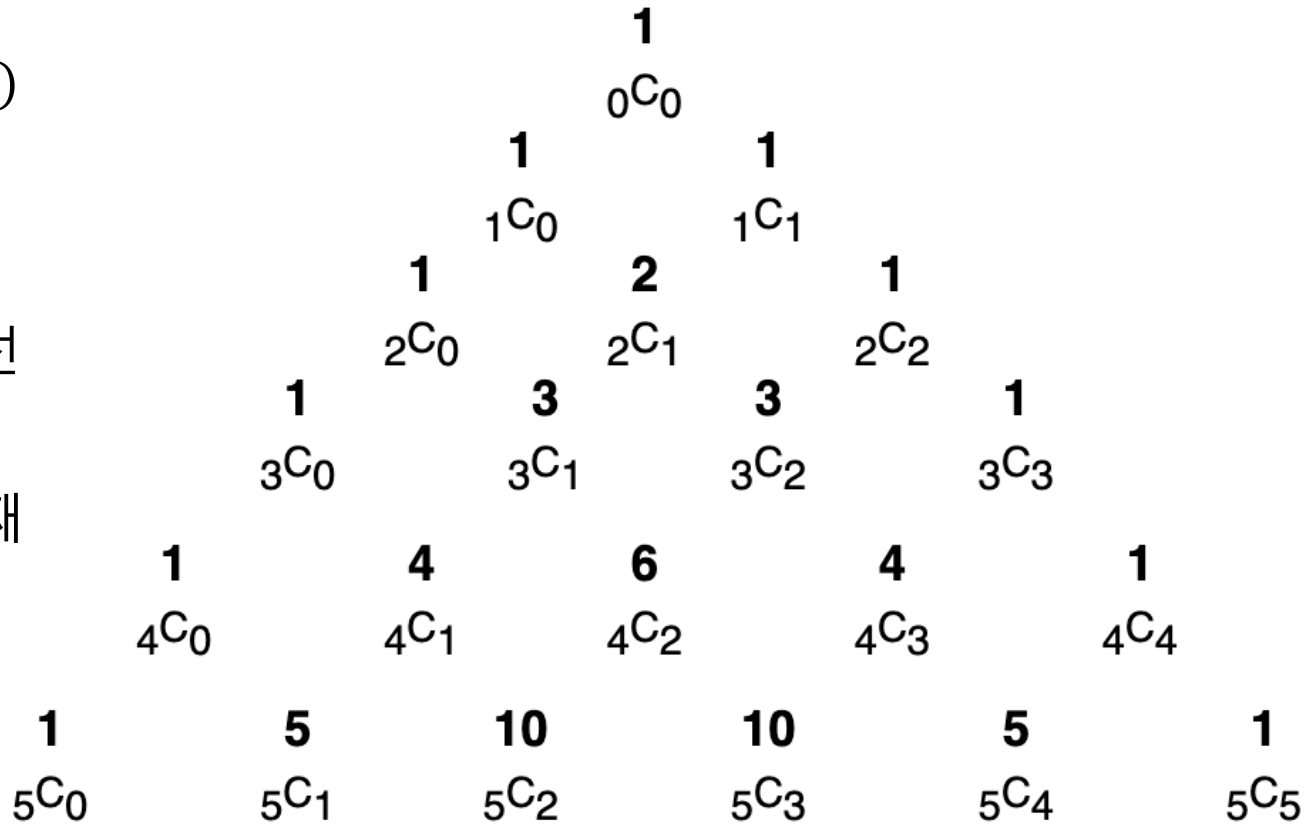
- 하향식 방법이 더 유리한 예시
  - 수학에서 조합 계산

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

    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)

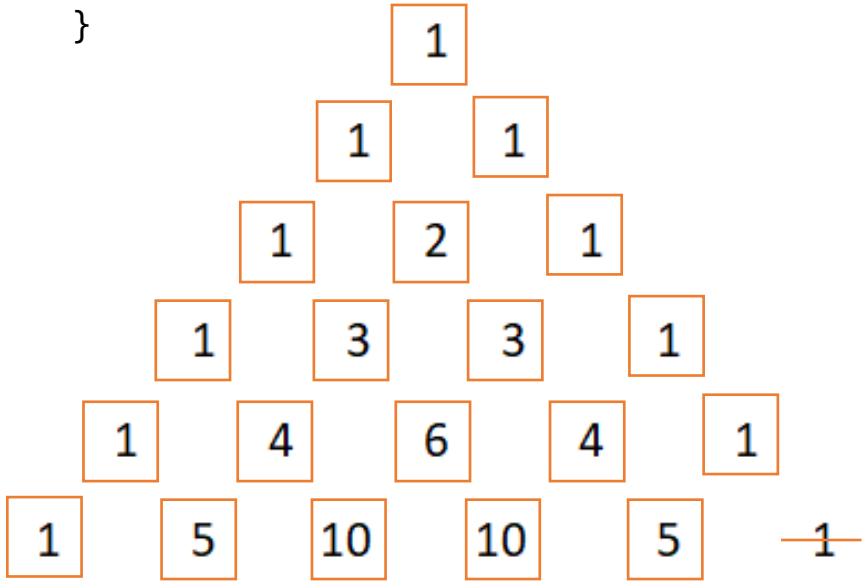
- 파스칼의 삼각형



다이나믹 프로그래밍(DP)

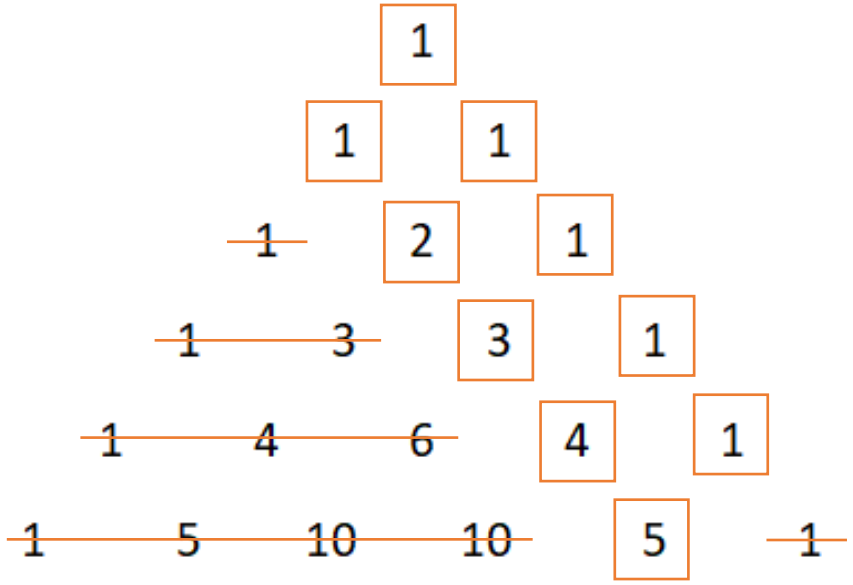
- 상향식 풀이

```
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

```c
// 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;
}
```

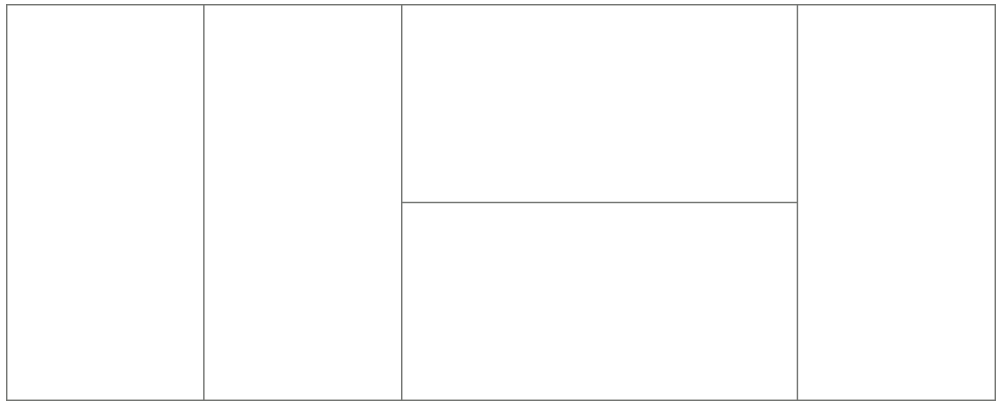
```c
#include <stdio.h>
#define  N  40
int memo[N];

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;
}
```

타일링

- 문제

  2×n 크기의 직사각형을 1×2, 2×1 타일로 채우는 방법의 수를 구하는 프로그램을 작성하시오.

  아래 그림은 2×5 크기의 직사각형을 채운 한 가지 방법의 예이다.



- 입력값

  첫째 줄에 n이 주어진다. (1 <= n <= 1,000)

- 출력값

  첫째 줄에 2×n 크기의 직사각형을 채우는 방법의 수를 10,007로 나눈 나머지를 출력한다.

입력 예	출력 예
3	3

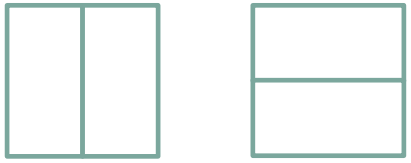
입력 예	출력 예
9	55

타일링

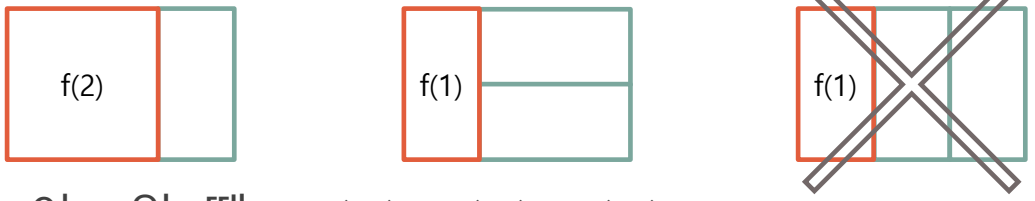
- n이 1일 때, f(1)=1가지



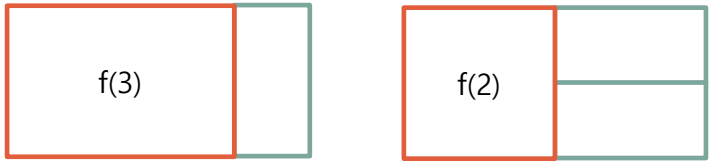
- n이 2일 때, f(2)=2가지



- n이 3일 때, f(3)=f(2)+f(1)



- n이 4일 때, f(4)=f(3)+f(2)



- 풀이 - 재귀함수

```
#include <stdio.h>

int f(int x) {
    if(x == 1)
        return 1;
    if(x == 2)
        return 2;

    return (f(x-1)+f(x-2))%10007;
}

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

```
50
5542

Process returned 0 (0x0)   execution time : 45.680 s
Press any key to continue.
```

타일링

```
1000
1115

Process returned 0 (0x0)   execution time : 0.193 s
Press any key to continue.
```

- 풀이 – DP(하향식)

```
#include <stdio.h>

int d[1001];

int f(int x) {
    if(x == 1)
        return 1;
    if(x == 2)
        return 2;

    if(d[x] != 0) return d[x];
    return d[x] = (f(x-1)+f(x-2)) % 10007;
}

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

```
1000
1115

Process returned 0 (0x0)   execution time : 0.013 s
Press any key to continue.
```

- 풀이 – DP(상향식)

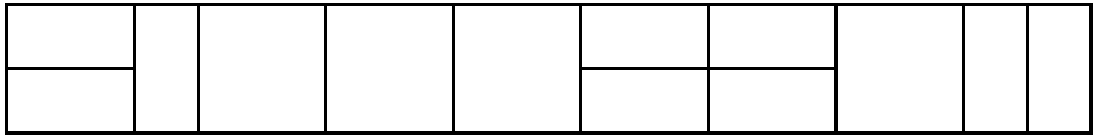
```
#include <stdio.h>
#define MAX  1000

int d[MAX+1];

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

    d[1] = 1;
    d[2] = 2;

    for(int x=3; x<=MAX; x++)
        d[x] = (d[x-1]+d[x-2]) % 10007;

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

타일링 II

- 문제

  2×n 직사각형을 1×2, 2×1과 2×2 타일로 채우는 방법의 수를 구하는 프로그램을 작성하시오.

  아래 그림은 2×17 직사각형을 채운 한가지 예이다.



- 입력값

  첫째 줄에 n이 주어진다. (1 <= n <= 1,000)

- 출력값

  첫째 줄에 2×n 크기의 직사각형을 채우는 방법의 수를 10,007로 나눈 나머지를 출력한다.

입력 예	출력 예
8	171

입력 예	출력 예
12	2731

- 출처: baekjoon

타일링II

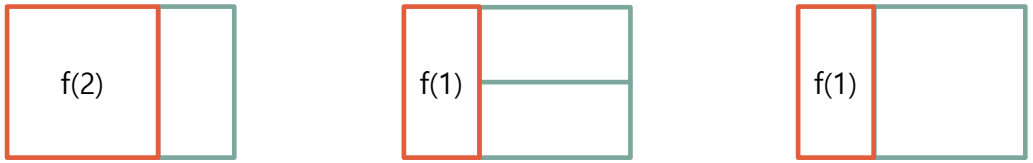
- n이 1일 때, f(1)=1가지



- n이 2일 때, f(2)=3가지



- n이 3일 때, f(3)=



- n이 4일 때, f(4)=



- 풀이 - 재귀함수

```
#include <stdio.h>

int f(int x) {
    if(x == 1)
        return 1;
    if(x == 2)
        return 3;

    return (f(x-1) + 2*f(x-2))%10007;
}

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

타일링II

- 풀이 – DP(하향식)

```
#include <stdio.h>

int d[1001];

int f(int x) {
    if(x == 1)
        return 1;
    if(x == 2)
        return 3;

    if(d[x] != 0) return d[x];
    return d[x] = (f(x-1)+2*f(x-2)) % 10007;
}

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

- 풀이 – DP(상향식)

```
#include <stdio.h>
#define MAX  1000

int d[MAX+1];

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

    d[1] = 1;
    d[2] = 3;

    for(int x=3; x<=MAX; x++)
        d[x] = (d[x-1]+2*d[x-2]) % 10007;

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

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

입력 예 1	출력 예 1
2 15 2 3	5

- 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

  첫째 줄에 경우의 X를 출력한다.

  불가능할 때는 -1을 출력한다.

입력 예 2	출력 예 2
3 4 3 5 7	-1

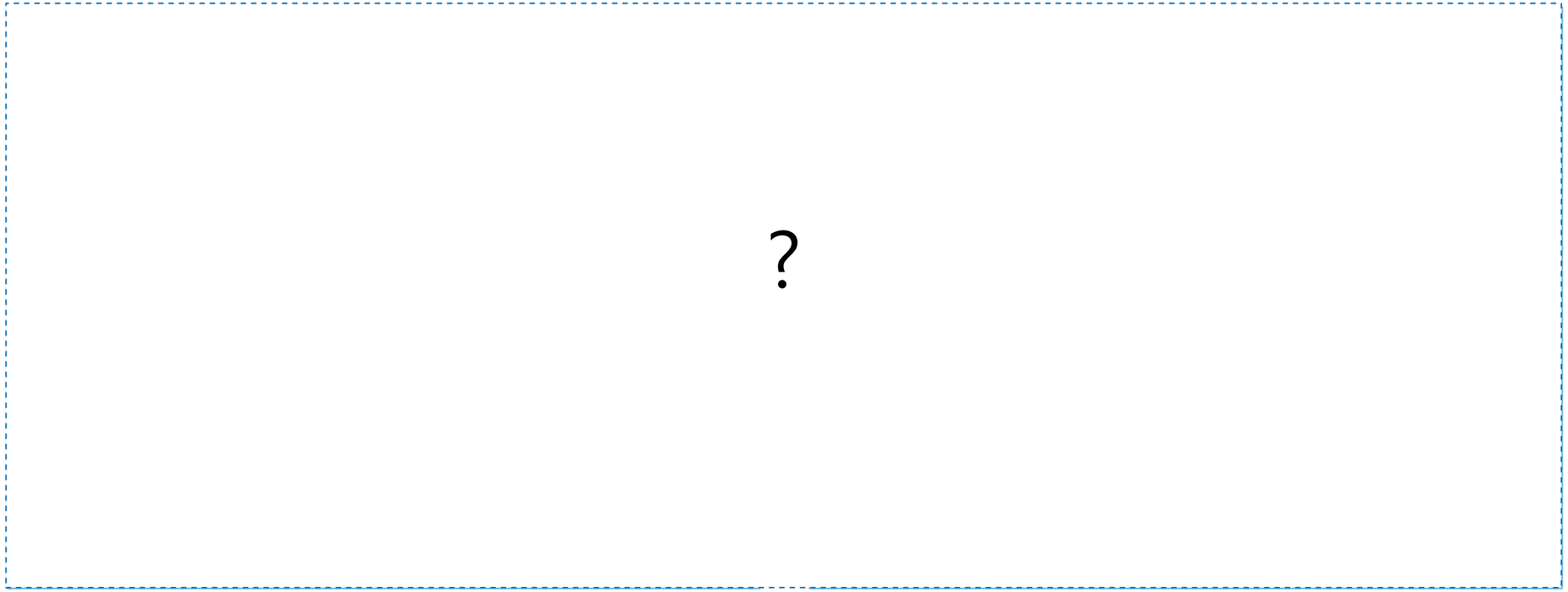
거스름돈II

- 화폐단위가 2원, 3원, 5원 인 경우

  (거스름돈 6원을 만드는 최소 개수)

  = min(6원 만드는 방법, 4원 만드는 방법+1, 3원 만드는 방법+1, 1원 만드는 방법+1)

  $a_6 = \min(a_6, a_4+1, a_3+1, a_1+1)$



거스름돈Ⅱ

- 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, $a_t=10001$

  - 예

    (2원으로 6원을 만드는 방법) $a_6 = \min($6원 만드는 방법, 4원 만드는 방법+1 $) = \min(a_6, a_4+1)$

    (3원으로 6원을 만드는 방법) $a_6 = \min(a_6,$ 3원 만드는 방법+1 $) = \min(a_6, a_3+1)$

    (5원으로 6원을 만드는 방법) $a_6 = \min(a_6,$ 1원 만드는 방법+1 $) = \min(a_6, a_1+1)$

거스름돈II

- 화폐단위가 2원, 3원, 5원 인 경우 문제 해설
  - 초기화

idx	0	1	2	3	4	5	6	7	…
값	0	10001	10001	10001	10001	10001	10001	10001	…

10001은 불가능 하다는 의미
0은 0개로 만들 수 있다.

  - 2원 짜리로 몇 번 만에 만들 수 있는가?

idx	0	1	2	3	4	5	6	7	…
값	0	10001	1	10001	2	10001	3	10001	…

$a_2 = \min(a_2, a_0+1) = \min(10001, 0+1) = 1$

$a_3 = \min(a_3, a_1+1) = \min(10001, 10001+1) = 10001$

$a_4 = \min(a_4, a_2+1) = \min(10001, 1+1) = 2$

$a_5 = \min(a_5, a_3+1) = \min(10001, 10001+1) = 10001$

$a_6 = \min(a_6, a_4+1) = \min(10001, 2+1) = 3$

거스름돈 II

- 화폐단위가 2원, 3원, 5원 인 경우 문제 해설
  - 3원 짜리를 추가로 사용하면 몇 번 만에 만들 수 있는가?

전

idx	0	1	2	3	4	5	6	7	…
값	0	10001	1	10001	2	10001	3	10001	…

후

idx	0	1	2	3	4	5	6	7	…
값	0	10001	1	1	2	2	2	3	…

$a_3 = \min(a_3, a_0+1) = \min(10001, 0+1) = 1$

$a_4 = \min(a_4, a_1+1) = \min(10001, 1+1) = 2$

$a_5 = \min(a_5, a_2+1) = \min(10001, 1+1) = 2$

$a_6 = \min(a_6, a_3+1) = \min(10001, 1+1) = 2$

$a_7 = \min(a_7, a_4+1) = \min(10001, 2+1) = 3$

거스름돈II

- 화폐단위가 2원, 3원, 5원 인 경우

(2원으로 6원을 만드는 방법) a_6 = min(현재 6원 만드는 방법, 4원 만드는 방법+1) = min(a_6, a_4+1)

(3원으로 6원을 만드는 방법) a_6 = min(a_6, 3원 만드는 방법+1) = min(a_6, a_3+1)

(5원으로 6원을 만드는 방법) a_6 = min(a_6, 2원 만드는 방법+1) = min(a_6, a_1+1)

```
#include <stdio.h>  // DP 바텀업 구현
#include <algorithm>
#define  MAX  10000
#define  INF  99999
int d[MAX+1];    // DP 테이블
int c[10];       // 화폐단위 저장 배열
int n, m;

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

    for(int i=0; i<n; i++) { // 각 화폐단위 a[i]에 대하여
        for(int t=c[i]; t<=MAX; t++) {
            if(d[t-c[i]] != INF)
                d[t] = std::min(d[t], d[t-c[i]]+1);
        }
    }
}
```

```
int makeM(int x) {
    return d[x];
}

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

    dp();

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

1로 만들기

- 문제

정수 X가 주어질 때 정수 X에 사용할 수 있는 연산은 다음과 같이 4가지이다.

ⓐ X가 5로 나누어 떨어지면, 5로 나눈다.
ⓑ X가 3로 나누어 떨어지면, 3으로 나눈다.
ⓒ X가 2로 나누어 떨어지면, 2로 나눈다.
ⓓ X에서 1을 뺀다.

정수 X가 주어졌을 때, 연산 4개를 적절히 사용해서 1을 만들려고 한다. 연산을 사용하는 횟수의 최솟값을 출력하시오.

예를 들어 정수가 26이면 다음과 같이 계산해서 3번의 연산이 최솟값이다.

1) 26 – 1 = 25 (ⓓ)
2) 25 / 5 = 5 (ⓐ)
3) 5 / 5 = 1 (ⓐ)

- 입력값

첫째 줄에 정수 X가 주어진다.

(1 <= X <= 30,000)

- 출력값

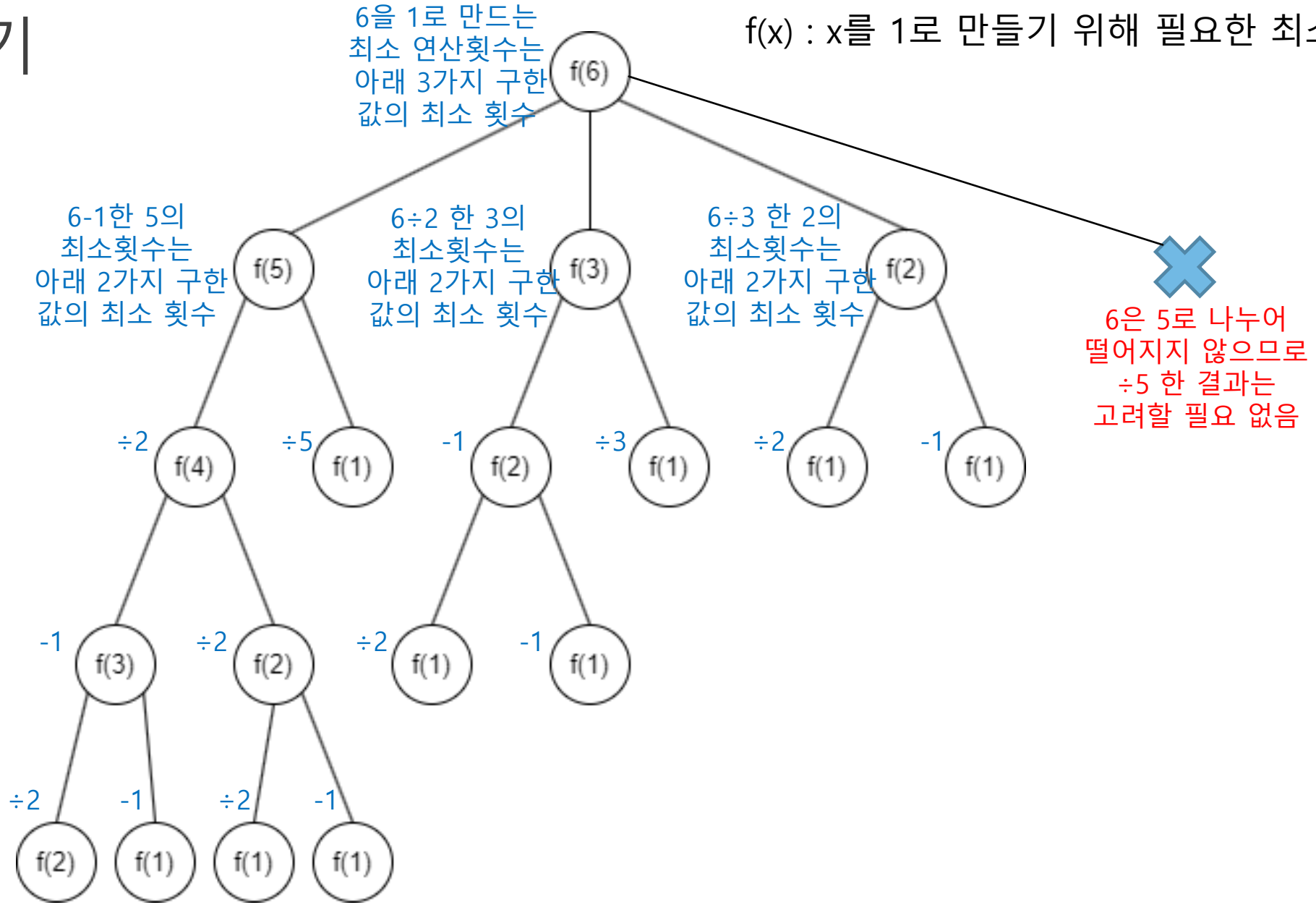
첫째 줄에 연산을 하는 횟수의 최솟값을 출력한다.

입력 예	출력 예
26	3

1로 만들기

- 문제해설
  - X=6일 때,

f(x) : x를 1로 만들기 위해 필요한 최소 횟수



1로 만들기

재귀호출로 구현

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

int make1(int x) {
    if(x == 1)
        return 0;

    int cnt = make1(x-1)+1;  // 현재 수에서 1빼는 경우
    if(x%2 == 0)             // 2로 나누어 떨어지는 경우
        cnt = min(cnt, make1(x/2)+1);
    if(x%3 == 0)             // 3로 나누어 떨어지는 경우
        cnt = min(cnt, make1(x/3)+1);
    if(x%5 == 0)             // 5로 나누어 떨어지는 경우
        cnt = min(cnt, make1(x/5)+1);
    return cnt;
}

int main() {
    int x;
    scanf("%d", &x);
    printf("%d\n", make1(x));
}
```

DP 바텀업 구현

```
#include <stdio.h>
#include <algorithm>
#define  MAX 30000
using namespace std;
int D[MAX+1]; //D[1]은 0인 상태, 1은 1로 만들기 위해 0회 연산 필요

void dp() {
    for(int i=2; i<=MAX; i++)  {
        D[i] = D[i-1] + 1; // 현재의 수에서 1을 빼는 경우
        if(i%2 == 0) // 현재의 수가 2로 나누어 떨어지는 경우
            D[i] = min(D[i], D[i/2]+1);
        if(i%3 == 0) // 현재의 수가 3으로 나누어 떨어지는 경우
            D[i] = min(D[i], D[i/3]+1);
        if(i%5 == 0) // 현재의 수가 5로 나누어 떨어지는 경우
            D[i] = min(D[i], D[i/5]+1);
    }
}
int make1(int x) {
    return D[x];
}
int main() {
    dp();
    int x;
    scanf("%d", &x);
    printf("%d\n", make1(x));
}
```

2023 충북정올 학교예선대회 (초등부)

- 초등부 예선 문항별 난이도 분석

등위	바른 코드	1번 산불을 조심하자!	2번 직소퍼즐 맞추기	3번 개미는 어디있을까?	4번 계단 오르기	5번 채널 빨리 바꾸는방법	합계
1위	4	100	100	100	100	0	400
2위	3	100	100	10	100	0	310
3위	3	100	100	0	100	0	300
4위	3	100	0	100	100	0	300
5위	2	100	50	0	100	0	250
6위	1	100	80	0	0	0	180
7위	1	100	80	0	0	0	180
8위	1	100	80	0	0	0	180
9위	1	100	30	0	0	0	130
10위	1	100	10	10	0	0	120

문제: 직소퍼즐

- 문제

  소타는 최근 재미있게 본 영화 포스터의 퍼즐을 구매하여 퍼즐을 맞추고 있었다. 문득 퍼즐 조각의 총 개수에 따라 만들 수 있는 퍼즐의 종류는 어떻게 달라질지 궁금해지게 되었다. 소타가 맞추던 퍼즐은 900피스인데, 가로는 25개, 세로는 36개로 구성되어있었는데, 가로 18, 세로 50개로 900피스 퍼즐을 만들 수도 있는 것이다. 퍼즐 조각의 개수가 입력되었을 때, 규격에 맞는 퍼즐이 몇 종류가 나올지 출력하는 프로그램을 만들어보자.

  <퍼즐 규격>

  - 퍼즐은 한 면에 최소 2개 이상의 조각이 있어야 한다.
  - 따라서 퍼즐조각의 개수가 소수(약수가 1과 자기 자신뿐인 자연수)인 경우 가로x세로 형태로 만들 수 없기 때문에 퍼즐 규격에 맞지 않다.
  - 퍼즐은 적당한 비율이 있어야 하는데 가로와 세로 길이가 서로 3배를 초과하지 않아야 한다.(예를 들어 가로가 3개, 세로가 11개로 구성된 퍼즐이라면 세로의 길이가 가로의 길이의 3배를 초과하기 때문에 규격에 맞지 않다.)
  - 가로, 세로의 개수가 다르면 다른 퍼즐로 본다. 즉, 2x3과 3x2는 다른 퍼즐이다.

- 출처: 2023 충북정올 학교예선 초등부 2번 문제

문제: 직소퍼즐

퍼즐 조각이 25개라면, 5x5 사이즈로 1개의 퍼즐밖에 만들 수 없다.

퍼즐 조각이 36개라면, 4x9, 6x6, 9x4로 3개의 퍼즐을 만들 수 있다.

- 입력형식

  퍼즐 조각 개수(N)를 입력받는다.
  (0≤N≤2,147,483,647)

- 출력형식

  생성할 수 있는 퍼즐의 개수를 출력한다.

  규격에 맞지 않을 경우 –1을 출력한다.

- 입력과 출력의 예

입력 예1	출력 예1
25	1

입력 예2	출력 예2
36	3

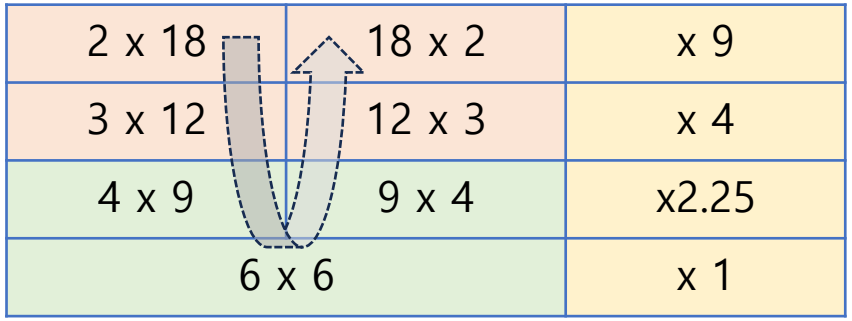
- 출처: 2023 충북정올 학교예선 초등부 2번 문제

풀이: 직소퍼즐

- 시간복잡도 $O(N)$ 풀이

```
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    int a, cnt=0;
    for(a=2; a<n; a++) {
        if(n%a == 0) {
            int b=n/a;
            double r;
            if(a > b) r=(double)a/(double)b;
            else      r=(double)b/(double)a;

            if(r<=3) {
                cnt++;
                //printf("%.2lf: %d %d\n", r, a, b);
            }
        }
    }
    if(cnt==0) printf("-1");
    else       printf("%d", cnt);
}
```

- 고찰
  - 왼쪽과 같은 풀이를 사용한 참가자는 시간초과로 100점을 획득하지 못한다.
  - 더 빠르게 해를 구할 방법은?
- 힌트
  - 퍼즐 조각이 36개 일 때,
  - 가로 x 세로 조합



풀이: 직소퍼즐

- 시간복잡도 $O(logN)$ 풀이

```
#include <stdio.h>

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

    int a, cnt=0;
    for(a=2; a*a<n; a++) {
        if(n%a == 0) {
            int b=n/a;

            double r;
            if(a > b)
                r=(double)a/(double)b;
            else
                r=(double)b/(double)a;
```

- 빈칸을 채워보세요.

```
            if(r<=3) {
                cnt+=2;
                //printf("%.2lf: %d %d\n", r, a, b);
            }
        }
    }
    if(a*a==n)
        cnt++;

    if(cnt==0) printf("-1");
    else       printf("%d", cnt);
}
```

약수의 합

- 문제

  한 정수 n을 입력 받는다.

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

  예를 들어 n이 10이라면,

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

- 입력

  첫 번째 줄에 정수 n이 입력된다.

  (단, 1 <= n <= 10,000,000,000(100억))

- 출력

  n의 약수의 합을 출력한다.

입력 예	출력 예
10	18

탐색공간이 매우 넓기 때문에 일반적인 방법으로는 시간 제한에 걸리게 된다.

약수의 합

- 단순 풀이

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

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

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

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

약수의 합

- 고찰

1) 배제를 위한 수학적 아이디어 1

모든 자연수 n에 대하여 1 과 n 은 항상 n 의 약수이다.

ex) 10의 약수
1, 2, 5, 10

ex) 16의 약수
1, 2, 4, 8, 16

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

- 고찰

2) 배제를 위한 수학적 아이디어 2

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

ex) 10의 약수
1, 2, 5, 10

ex) 16의 약수
1, 2, 4, 8, 16

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

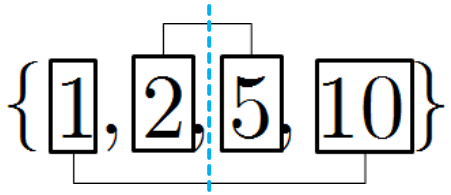
약수의 합

- 고찰

  3) 배제를 위한 수학적 아이디어 3

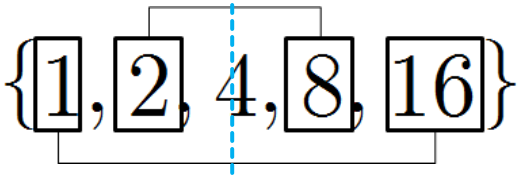
임의의 자연수 n의 약수들 중 두 약수의 곱은, n이 되는 약수a와 약수b가 반드시 존재한다. 단, n이 완전제곱수 일 경우에는 약수 a와 약수 b가 같을 수 있다.

반쪽만 찾으면 됨



10의 약수간 관계

반쪽만 찾으면 됨



16의 약수간 관계

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

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

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

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

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

약수의 합

- 고찰

  3) 배제를 위한 수학적 아이디어 3 구현

  ex) 100의 약수 모두 구하기

  ① $1 \sim \sqrt{100} = 10$ 까지 조사

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

  ② 약수 집합a { 1, 2, 5, 10}으로부터

  약수 집합b {100, 50, 20, 10}유도

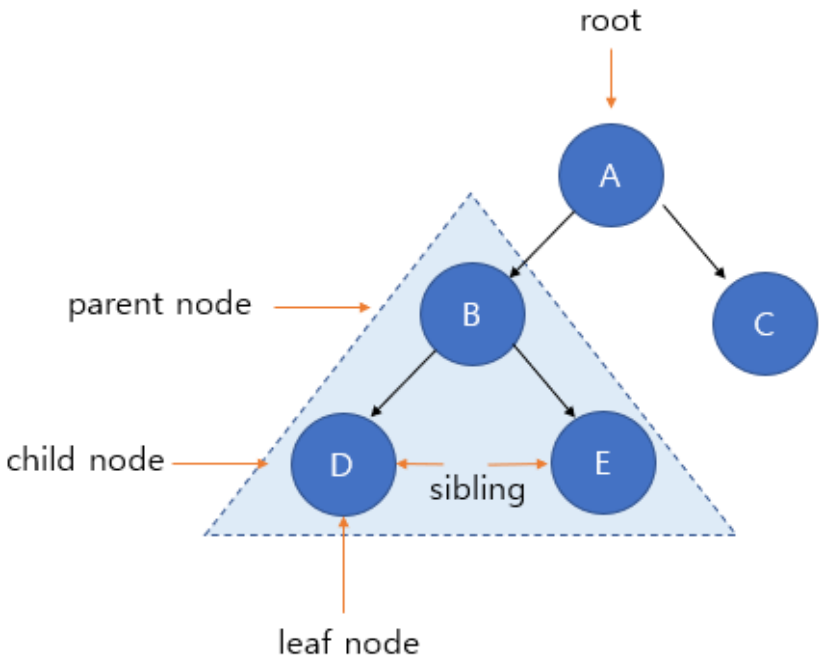
  ③ 약수 합집합 {1,2,5,10,20,50,100}

- 수학적 배제 적용

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









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

트리

- 개념
  - 트리는 계층적 관계를 표현하는 비선형 자료구조
- 예시

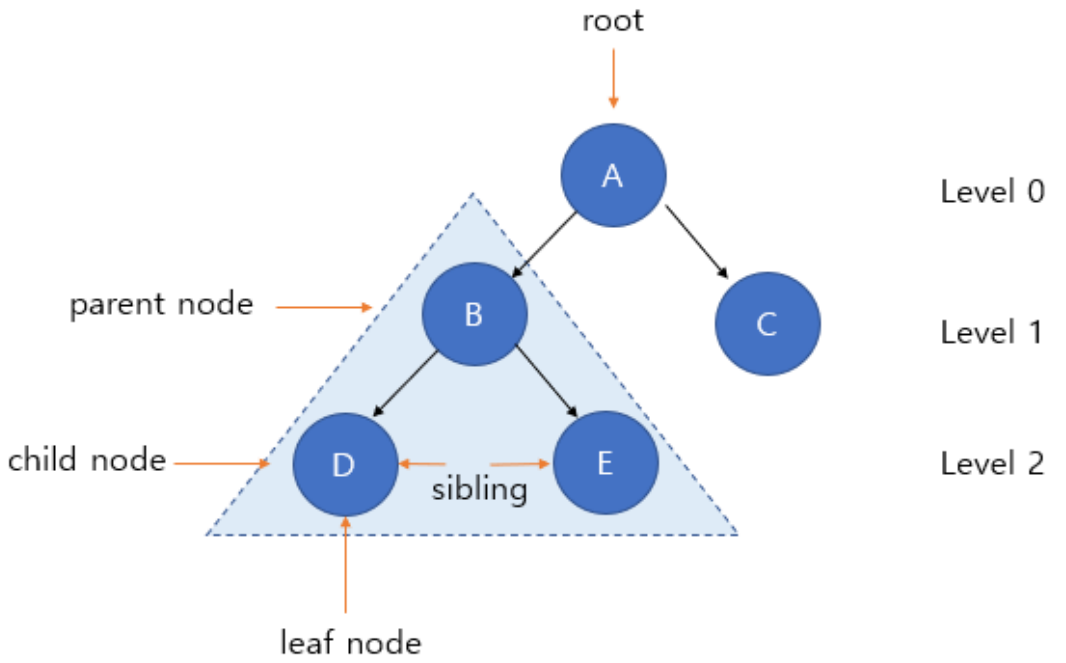


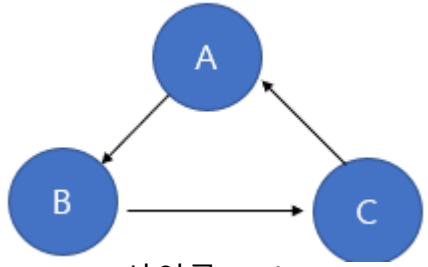
- 용어
  - 루트 노드(root node): 부모가 없는 노드로 트리는 단 하나의 루트 노드를 가진다. (ex : A- 루트노드)
  - 단말 노드(leaf node): 자식이 없는 노드로 terminal 노드라고도 부른다. (ex : C, D, E – 단말 노드)
  - 내부 노드(internal node): 단말 노드가 아닌 노드(ex : A, B – 내부 노드)
  - 간선(edge): 노드를 연결하는 선
  - 형제(sibling): 같은 부모 노드를 갖는 노드들 (ex : D-E, B-C : 형제)
  - 노드의 깊이(depth): 루트 노드에서 어떤 노드에 도달하기 위해 거쳐야 하는 간선의 수(ex: D의 depth : 2)
  - 노드의 레벨(level): 트리의 특정 깊이를 가지는 노드의 집합 (ex : level 1- {B, C})
  - 노드의 차수(degree): 자식 노드의 개수 (ex : B의 degree - 2, C의 degree - 0)
  - 트리의 차수(degree of tree): 트리의 최대 차수 (ex : 이 트리의 차수는 2)

트리의 특징

- 특징
  - 트리는 하나의 루트 노드를 갖는다.
  - 루트 노드는 0개 이상의 자식 노드를 갖는다.
  - 자식 노드 또한 0개 이상의 자식 노드를 갖는다.
  - 노드(node)들과 노드들을 연결하는 간선(edge)들로 구성되어 있다.
  - 사이클이 존재할 수 없다.
  - N개의 노드를 갖는 트리는 항상 N-1 개의 간선(edge)을 갖는다.
  - 모든 자식 노드는 한 개의 부모 노드만을 갖는다.

- 트리 예시

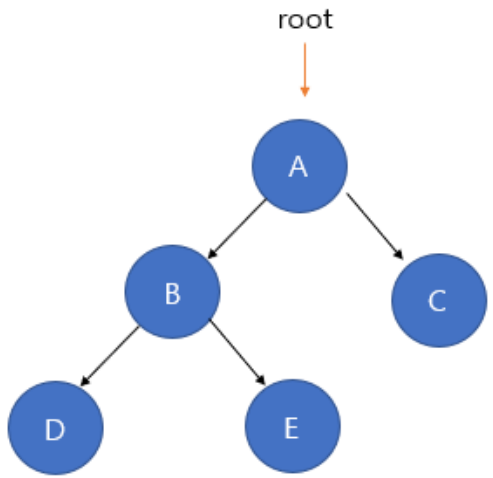




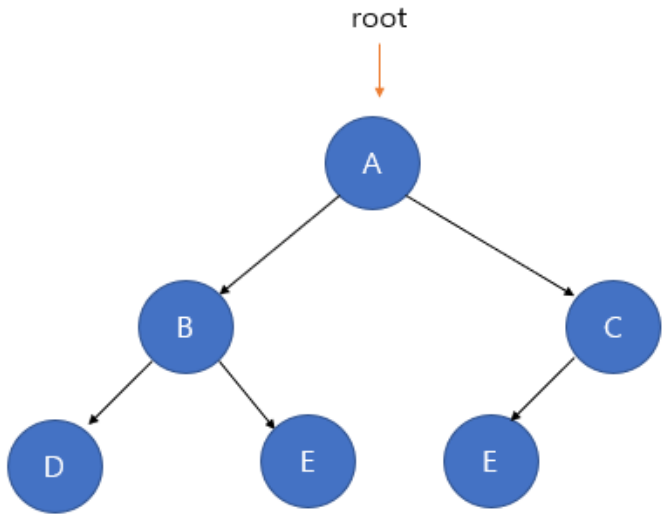
사이클(cycle)

트리의 종류

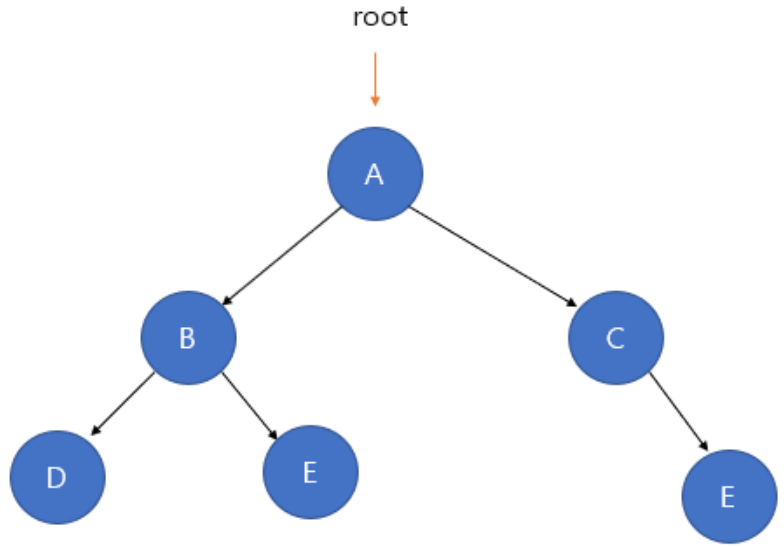
- 이진트리
  - 이진트리는 각 노드가 최대 두 개의 자식을 갖는 트리를 뜻한다. 즉, 각 노드는 자식이 없거나 한 개 이거나 두 개 만을 갖는 것이다.



- 완전 이진트리
  - 완전 이진트리는 마지막 레벨을 제외 하고 모든 레벨이 완전히 채워져 있다.
  - 마지막 레벨은 꽉 차 있지 않아도 되지만, 노드가 왼쪽에서 오른쪽으로 채워져야 한다.



완전 이진 트리



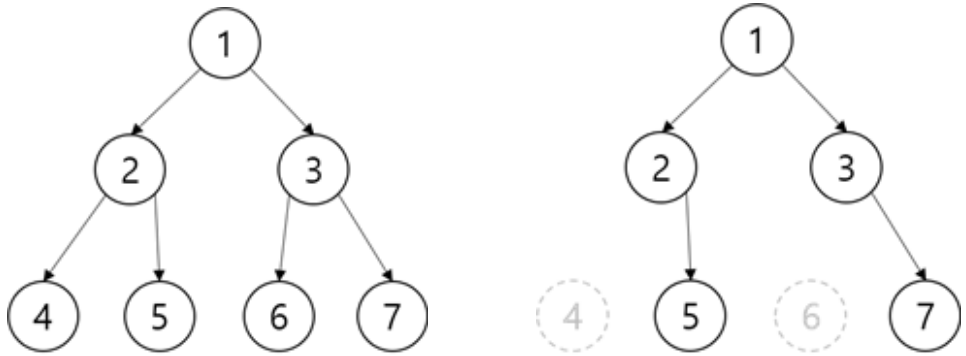
완전 이진 트리 아님

문제: 개미는 어디 있을까?

- 문제

단테는 개미집에 관한 다큐멘터리를 보다가 개미가 개미집에서 어느 곳에 있는지 궁금해졌다. 개미집과 유사한 이진 트리 구조를 활용하기로 하였다. 개미집은 수 많은 방들로 구성되어 있으며 동그라미로 방을 나타내기로 했다. 개미집은 깊이가 증가할수록 방이 많아지는 구조인데, 한 방에서는 바로 아래로 최대 2개의 방을 만들 수 있다.

아래 그림은 깊이가 3인 개미집이고 왼쪽 개미집은 깊이가 3인 개미집들 중에서 최대 방 개수를 갖는 개미집이다. 오른쪽에는 깊이가 3인 다른 개미집이다.



방의 번호는 1번부터 시작하고 최대로 뻗어져 있는 개미집 모양을 기준으로 방 번호가 정해진다. 즉, 방 번호는 1씩 증가하지 않을 수 있다.

개미집의 상태를 나타내기 위해 최대로 뻗어져있는 개미집의 모양을 기준으로 방이 있으면 1, 없으면 0으로 표현한다. 예를 들어, 깊이가 3인 개미집은 총 7개의 방 상태로 표현할 수 있는데, 왼쪽 개미집의 경우 1 1 1 1 1 1 1 로 표현할 수 있고, 오른쪽 개미집의 경우 1 1 1 0 1 0 1 으로 표현할 수 있다.(4, 6번 방은 없다.)

개미는 개미집의 최상단 방부터 시작하여 왼쪽 또는 오른쪽으로 이동할 수 있다.(왼쪽 = 0, 오른쪽 = 1) 예를 들어 1 0이 입력되면 오른쪽 방으로 이동한 뒤 왼쪽 방으로 이동을 한다. 개미가 가야하는 방향에 한 번이라도 방이 없다면 개미는 이동하지 않으며 마지막에 있던 방에 가만히 있으며 그 이후 입력된 이동은 모두 무시한다.

- 출처: 2023 충북정올 학교예선 초등부 3번 문제

문제: 개미는 어디 있을까?

개미집의 상태와 개미의 이동방향 정보를 입력받아 개미의 최종 위치를 출력하는 프로그램을 작성하시오.

입력형식

첫 번째 줄에 개미집의 깊이(d)가 자연수로 입력된다.(1≤d≤6)

두 번째 줄에 개미집의 각 방의 상태가 공백으로 분리되어 입력된다.(0 : 없음, 1 : 있음)

세 번째 줄에 개미의 이동 방향이 공백으로 분리되어 입력된다.(0 : 왼쪽, 1 : 오른쪽)

출력형식

- 개미가 마지막에 위치하고 있는 방의 번호를 출력한다.

입력과 출력의 예

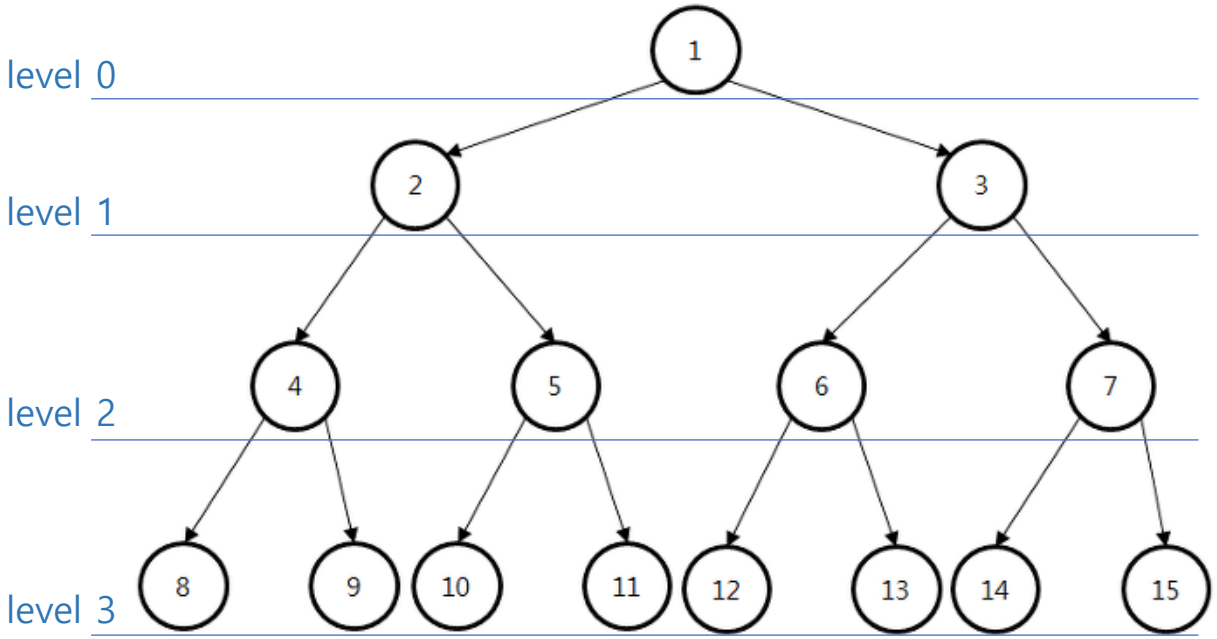
입력 예1	출력 예1
3 1 1 1 1 1 1 1 0 1	5

입력 예2	출력 예2
4 1 1 1 0 1 1 1 0 0 1 0 1 1 1 1 0 0 1	2

- 출처: 2023 충북정올 학교예선 초등부 3번 문제

풀이: 개미는 어디 있을까?

- 포화 이진 트리
  - 모든 노드가 꽉 차 있는 상태의 트리
  - 노드의 개수 = $2^{h+1} - 1$



- 배열을 이용한 이진트리 구현
  - 인덱스 계산
    - 루트 노드 인덱스 = 1
    - 왼쪽 자식 노드 인덱스
      = 부모 노드 인덱스*2
    - 오른쪽 자식 노드 인덱스
      = 부모 노드 인덱스*2 + 1

(ex) 5번 노드의
✓왼쪽자식: 5*2 = 10
✓오른쪽자식: 5*2+1 = 11

풀이: 개미는 어디 있을까?

- 모범답안

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

enum direction {
    LEFT, RIGHT
};

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

    int n = pow(2, d)-1;
    bool rooms[n];

    for(int i=1; i<=n; i++)
        scanf("%d", &rooms[i]);
```

```
    int pos=1;
    int dir;
    for(int i=0; i<d-1; i++) {
        scanf("%d", &dir);
        if(dir==LEFT) {
            if(rooms[pos*2]) pos = pos*2;
            else break;
        }
        else if(dir==RIGHT) {
            if(rooms[pos*2+1]) pos = pos*2+1;
            else break;
        }
    }
    printf("%d\n", pos);
    return 0;
}
```

2023 충북정올 본선(도)대회 (초등부)

- 초등부 본선 문항별 난이도 분석

등위	바른 코드	1번 제37회정보 올림피아드	2번 1을 보는 시간	3번 스티커 소비	4번 꽃밭	5번 오르락 내리락	합계
1위	3	100	100	100	0	0	300
2위	3	100	100	0	0	100	300
3위	2	100	100	20	0	0	220
4위	2	100	100	0	0	0	200
5위	2	100	100	0	0	0	200
6위	2	100	100	0	0	0	200
7위	1	100	20	20	0	0	140
8위	1	100	10	0	0	0	110
9위	1	100	0	10	0	0	110
10위	1	100	10	0	0	0	110

문제: 스티커 소비

문제

충북이는 여러 개의 스티커를 가지고 있다. 가지고 있는 스티커에는 1번부터 n번까지 숫자로 구분되어 있다. 이 때, 충북이는 a번부터 b번까지의 스티커를 매일 소비하고자 한다. 만약 전날 짝수개의 스티커가 남아 있다면 반으로 줄고, 홀수개의 스티커가 남아 있다면 a번부터 b번까지 남은 홀수개 스티커 중 가장 적은 양보다 1개가 더 줄어든다. (혼돈스럼)

예를 들어, 2번부터 4번까지의 스티커 수가 10, 5, 7이라고 가정해보자.

	2번	3번	4번
1일	10	5	7
2일	5	4	4
3일	4	2	2
4일	2	1	1
5일	1	0	0
6일	0		

따라서, 2번~4번 스티커가 선택되었을 때 각각의 스티커는 6일, 5일, 5일만에 스티커를 모두 소비할 수 있다.

입력형식

- 첫 번째 줄에 충북이가 가지고 있는 스티커의 종류 n이 입력된다.(1 ≤ n ≤ 100)
- 두 번째 줄에 번호 순서대로 가지고 있는 스티커의 개수 k가 입력된다.(0 ≤ k ≤ 1000)
- 세 번째 줄에 소비하고자 하는 스티커의 시작 번호 a와 끝 번호 b가 입력된다.(1 ≤ a ≤ b ≤ n)

출력형식

- a번부터 b번까지 번호 순서대로 각 번호마다 모든 스티커가 소요되는 일 수를 출력한다.

입출력 예시데이터

입력 예	출력 예
5 7 10 5 7 9 2 4	6 5 5
3 0 2 3 1 1	1

- 출처: 2023 충북정올 본선(도)대회 초등부 3번 문제

풀이: 스티커 소비

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

typedef struct {
    int no;
    int cv;     // current value
    int days;
} stiker;

stiker sa[100]; // sticker array

int main() {
    int sn;
    scanf("%d", &sn);
    for(int i=0; i<sn; i++) {
        sa[i].no = i+1;
        scanf("%d", &sa[i].cv);
        sa[i].days=1;
    }

    int a, b;
    scanf("%d %d", &a, &b);
    a--, b--;   // zoro-base 로 변경
```

전날 짝수개의 스티커가 남아 있다면 반으로 줄고,
홀수개의 스티커가 남아 있다면 a번부터 b번까지 남은
홀수개 스티커 중 가장 적은 양보다 1개가 더 줄어든다

```
    while(true) {
        int odd_min = 99999999;
        for(int i=a; i<=b; i++) {
            if(sa[i].cv>0 && sa[i].cv%2==1) {
                odd_min = min(sa[i].cv, odd_min);
            }
        }
        for(int i=a; i<=b; i++) {
            if(sa[i].cv > 0) {
                if(sa[i].cv % 2 == 0) sa[i].cv /= 2;
                else sa[i].cv = odd_min-1;
                sa[i].days++;
            }
        }
        int cv_sum=0;
        // [a, b] 범위의 남은 스티커 모두 합산
        for(int i=a; i<=b; i++)
            cv_sum += sa[i].cv;

        // [a, b] 범위의 스티커가 모두 소진되었으면
        if(cv_sum == 0) {
            for(int i=a; i<=b; i++) {
                printf("%d ", sa[i].days);
            }
            break;
        }
    }
}
```

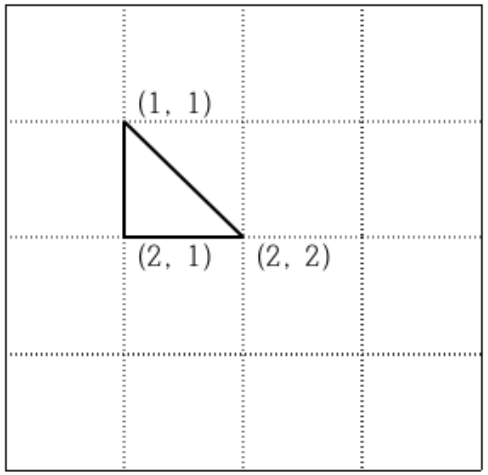
문제: 꽃밭

- 문제

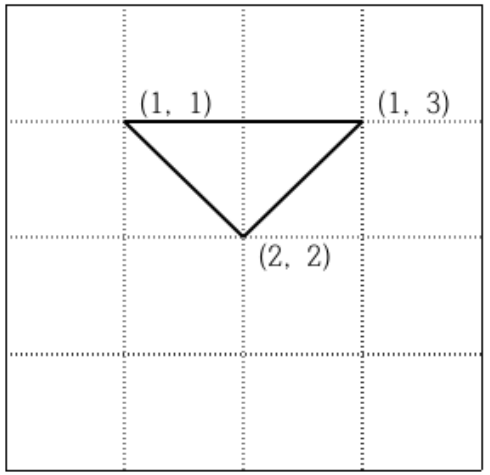
청주 문의면 일대에서 열리는 영춘제를 위해 직사각형 모양의 땅에 삼각형 모양으로 꽃밭을 만들고자 한다. 땅에는 구간을 나누기 위해 일정한 간격으로 말뚝이 심어져 있고, 각 말뚝 위치는 좌표로 표현한다. 꽃밭은 말뚝 세 개를 일직선으로 이어서 만든다. 단, 말뚝과 말뚝을 이을 때는 중간에 다른 말뚝이 있으면 꽃밭을 만들 수 없다.

예를 들어, [예시1]과 같은 경우에는 (1, 1), (2, 1), (2, 2)를 이어서 꽃밭을 만들 수 있다.

[예시2]와 같은 경우에는 (1, 1)과 (1, 3)을 잇는 중간에 (1, 2)에 말뚝이 있기 때문에 꽃밭을 만들 수 없다.



[예시1]



[예시2]

땅의 크기인 N, M과 입력받을 말뚝의 개수(T), 이후 T개의 줄에 말뚝의 좌표(x, y)가 주어지면, 각 말뚝에서 만들 수 있는 삼각형 모양의 땅의 경우의 수를 10,007로 나눈 나머지를 T개의 줄에 걸쳐 출력하는 프로그램을 작성해보자.

- 출처: 2023 충북정올 본선(도)대회 초등부 5번 문제

문제: 꽃밭

- 입력형식

  첫 번째 줄에 땅의 크기를 나타내는 행(N), 열(M)이 공백으로 구분되어 입력된다.

  (2 ≤ N, M ≤ 40)

  두 번째 줄에 입력받을 점들 말뚝의 개수(T)가 입력된다. (1 ≤ T ≤ 50)

  세 번째 줄부터 이후 T개의 줄에 말뚝의 좌표 (x, y)가 입력된다. (1 x ≤ N, 1 ≤ y ≤ M)

- 출력형식
  - 말뚝(x, y)에서 만들 수 있는 꽃밭의 수를 10,007로 나눈 나머지를 T줄에 걸쳐 출력한다.

- 입력과 출력의 예

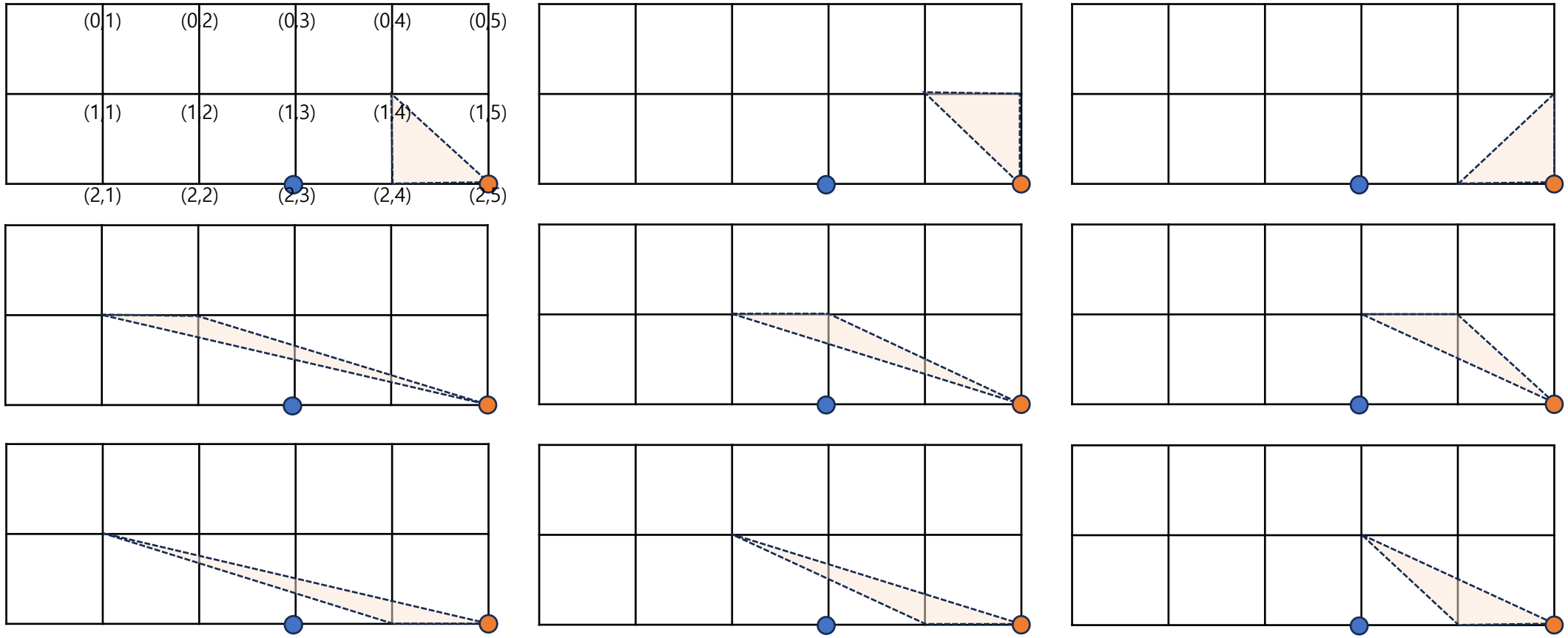
입력 예1	출력 예1
2 2 3 1 1 1 2 2 1	3 3 3

입력 예2	출력 예2
2 5 2 2 3 2 5	14 9

- 출처: 2023 충북정올 본선(도)대회 초등부 5번 문제

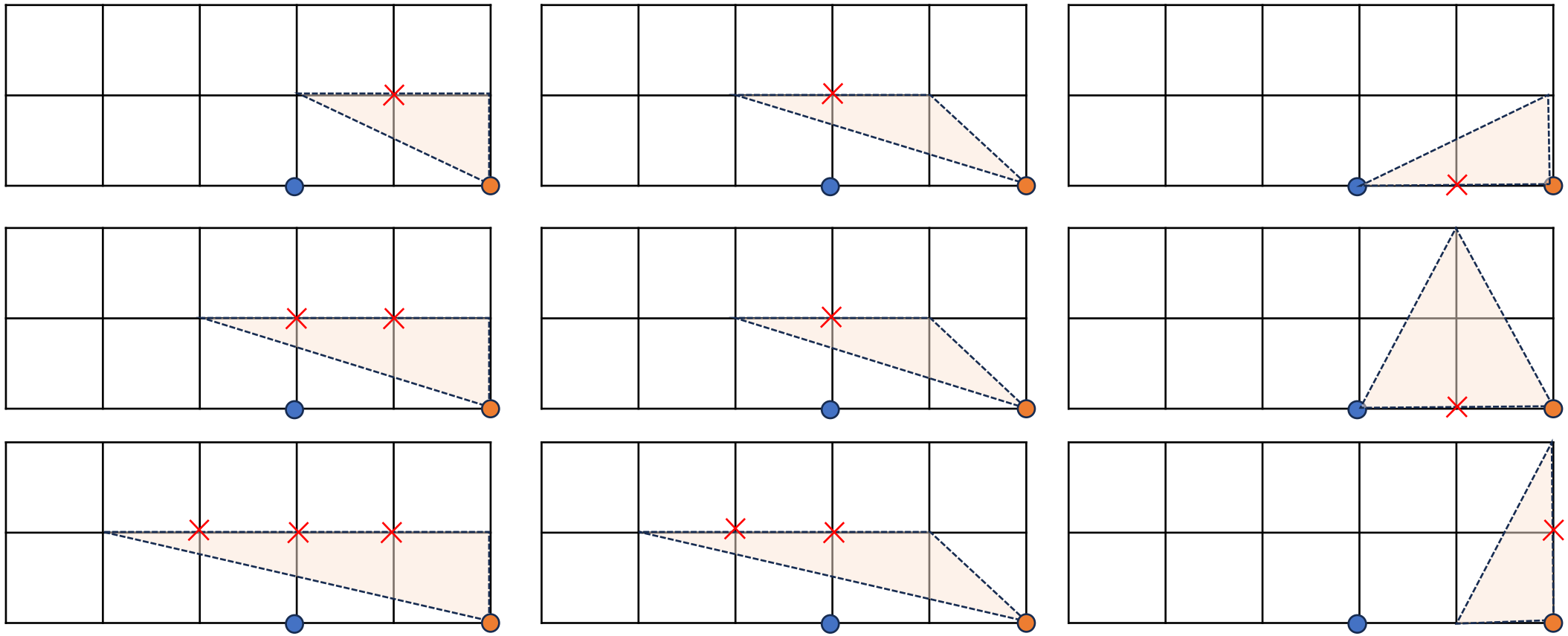
풀이: 꽃밭

- 고찰
  - 2행 5열 땅에 (2,3) (2,5) 좌표가 입력되었을 시 만들 수 있는 꽃밭 종류



풀이: 꽃밭

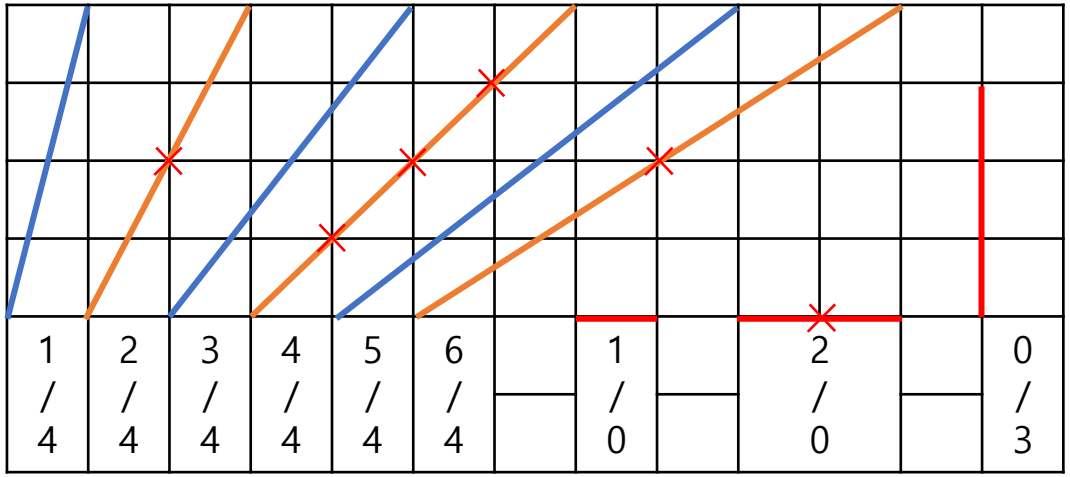
- 고찰
  - 2행 5열 땅에 (2,3) (2,5) 좌표가 입력되었을 시 만들 수 없는 꽃밭 종류



풀이: 꽃밭

- 고찰
  - 좌표 평면상 삼각형 성립 조건
    - 세 점의 좌표가 모두 다르다.
    - 두 선분의 길이의 합이 나머지 보다 크다.
  - 이 문제에서는 세 점의 좌표 중 하나는 문제에서 지정이고 나머지 두 점은 n행 m열 범위내에서 프로그래머가 직접 정하므로 위 성립조건과는 무관.
  - 그리려는 삼각형의 세 변에 말뚝이 걸치는지 여부만 중요.

  - 꽃밭을 그리는 것이 가능한 선분 기울기 조건



  - 기울기가 기약분수 형태(분모와 분자의 공약수가 1뿐)가 아니면 중간에 말뚝과 겹친다. → 공약수가 1뿐인 수: 서로소
  - dx, dy 둘 중 하나가 0이면 나머지는 반드시 1이어야 한다.

최대공약수 구하기 <유클리드 호제법>

① 수가 크면 계산이 복잡

```
2 | 2304  1440
2 | 1152   720
2 |  576   360
2 |  288   180
2 |  144    90
3 |   72    45
   24    15
    8     5
```

2 × 2 × 2 × 2 × 2 × 3 × 3 = 288

② 약수 찾기가 어려운 경우

```
31 | 403  155
     13    5
```

a b 몫(s), 나머지(r)

① 큰수를, 작은수로 나눈다

② 나누는 수를, 나머지로 계속 나눈다

나머지가 0이 나오면,

나누는 수가 최대공약수

⑥ 1512, 1008 최대공약수

	a		b		r
①	1512	÷	1008	= 1 …	504
②	1008	÷	504	= 2 …	0

<유클리드 호제법>
a와 b의 최대공약수는
b과 r의 최대공약수와 같다.

a b r

① 2304 ÷ 1440 = 1 … 864

② 1440 ÷ 864 = 1 … 576

864 ÷ 576 = 1 … 288

576 ÷ 288 = 2 … 0

① 403 ÷ 155 = 2 … 93

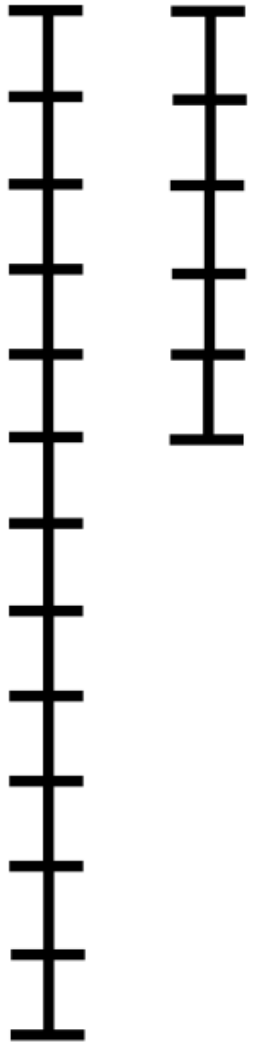
② 155 ÷ 93 = 1 … 62

93 ÷ 62 = 1 … 31

62 ÷ 31 = 2 … 0

최대공약수 구하기 <유클리드 호제법>

- 그림으로 이해하기



- 숫자로 이해하기
  - 78696과 19332의 최대공약수를 구하면,

A		B		r
78696	=	19332×4	+	1368
19332	=	1368×14	+	180
1368	=	180×7	+	108
180	=	108×1	+	72
108	=	72×1	+	36
72	=	36×2	+	0

풀이: 꽃밭 (첫번째 시도)

B04_flower_garden1.cpp

```
#include <iostream>
#include <string.h>
using namespace std;

struct coordi {
    int x, y;
} cord[50];   // 입력받은 좌표 저장

int n, m, t;
int ans = 0;

bool coprime(int dx, int dy) {
    //기약분수란 분모와 분자의 공약수가 1뿐인 분수
    int a = abs(dx);
    int b = abs(dy);
    // 입력받은 기울기 팩터 dx, dy로
    while(b != 0) {  // 유클리드 호제법 적용
        int r = a % b;
        a = b;
        b = r;
    }
    //기약분수 이면 true 리턴
    return (a==1)? true : false;
}
```

a, b 둘 중 하나가 0이면, 나머지 수가 최대공약수로 계산됨.

```
int main() {
    scanf("%d %d", &n, &m);
    scanf("%d", &t);
    for(int i=0; i<t; i++)
        scanf("%d %d", &cord[i].x, &cord[i].y);

    // 삼각형 a,b,c가 꽃밭이 될 수 있는지 확인한다.
    for(int i=0; i<t; i++) {
        int ax = cord[i].x; //a점을 입력된 좌표로 지정
        int ay = cord[i].y;
        printf("a:(%d %d)\n", ax, ay);
        ans = 0;

        //지도의 모든 가능한 지점을 b점으로 하여 탐색
        for(int bx=1; bx<=n; bx++) {
            for(int by=1; by<=m; by++) {
                //선분 ab의 기울기가 기약분수 이면,
                if(coprime(bx-ax, by-ay)) {
                    //현재 b점으로
                    printf("  b:(%d %d)\n", bx, by);
                    //a점과 b점으로 삼각형 만들기를 시도함.
                    make_trangle(ax, ay, bx, by);
                }
            }
        }
        printf("%d\n", ans);
    }
}
```

```
2 5
2
2 3
2 5
a:(2 3)
  b:(1 1)
  b:(1 2)
  b:(1 3)
  b:(1 4)
  b:(1 5)
  b:(2 2)
  b:(2 4)
0
a:(2 5)
  b:(1 1)
  b:(1 2)
  b:(1 3)
  b:(1 4)
  b:(1 5)
  b:(2 4)
0
```

풀이: 꽃밭 (최종답)

```
#include <iostream>
#include <string.h>
using namespace std;

struct coordi {
    int x, y;
} cord[50];   // 입력받은 좌표 저장
bool used[50][50]; //(x,y)점이 사용되었는지 저장

int n, m, t;
int ans = 0;

bool coprime(int dx, int dy) {
    //기약분수란 분모와 분자의 공약수가 1뿐인 분수
    int a = abs(dx);
    int b = abs(dy);
    // 입력받은 기울기 팩터 dx, dy로
    while(b != 0) {  // 유클리드 호제법 적용
        int r = a % b;
        a = b;
        b = r;
    }
    //기약분수 이면 true 리턴
    return (a==1)? true : false;
}
```

a, b 둘 중 하나가 0이면, 나머지 수가 최대공약수로 계산됨.

```
void make_trangle(int ax, int ay, int bx, int by) {
    //지도의 모든 가능한 지점을 c점으로 하여 탐색
    for(int cx=1; cx<=n; cx++) {
        for(int cy=1; cy<=m; cy++) {
            // (cx, cy)가 b점으로 사용된적이 없으면,
            if(!used[cx][cy])
                // 선분 ac, bc의 기울기가 기약분수이면,
                if(coprime(cx-ax, cy-ay) &&
                   coprime(cx-bx, cy-by)) {
                    printf("    c:(%d %d)\n", cx, cy);
                    ans++; //꽃밭 삼각형 개수 증가
                }
            }
    }
    ans = ans % 10007;
}

int main() {
    scanf("%d %d", &n, &m);
    scanf("%d", &t);
    for(int i=0; i<t; i++)
        scanf("%d %d", &cord[i].x, &cord[i].y);
```

```
    // 삼각형 a,b,c가 꽃밭이 될 수 있는지 확인한다.
    for(int i=0; i<t; i++) {
        int ax = cord[i].x; //a점을 입력된 좌표로 지정
        int ay = cord[i].y;
        printf("a:(%d %d)\n", ax, ay);
        ans = 0;
        memset(used, false, sizeof(used));

        //지도의 모든 가능한 지점을 b점으로 하여 탐색
        for(int bx=1; bx<=n; bx++) {
            for(int by=1; by<=m; by++) {
                //선분 ab의 기울기가 기약분수 이면,
                if(coprime(bx-ax, by-ay)) {
                    //현재 b점으로
                    printf("  b:(%d %d)\n", bx, by);
                    //bx, by가 사용되었음을 마킹
                    used[bx][by] = true;
                    //a점과 b점으로 삼각형 만들기를 시도함.
                    make_trangle(ax, ay, bx, by);
                }
            }
        }
        printf("%d\n", ans);
    }
}
```

- 데이터 입력 및 실행 결과
  - 입력

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

(2,3)에서

```
a:(2 3)
  b:(1 1)
    c:(1 2)
    c:(2 2)
    c:(2 4)
  b:(1 2)
    c:(1 3)
    c:(2 2)
    c:(2 4)
  b:(1 3)
    c:(1 4)
    c:(2 2)
    c:(2 4)
  b:(1 4)
    c:(1 5)
    c:(2 2)
    c:(2 4)
  b:(1 5)
    c:(2 2)
    c:(2 4)
  b:(2 2)
  b:(2 4)
14
```

(2,5)에서

```
a:(2 5)
  b:(1 1)
    c:(1 2)
    c:(2 4)
  b:(1 2)
    c:(1 3)
    c:(2 4)
  b:(1 3)
    c:(1 4)
    c:(2 4)
  b:(1 4)
    c:(1 5)
    c:(2 4)
  b:(1 5)
    c:(2 4)
  b:(2 4)
9
```

최장 증가 부분 수열(LIS) 알고리즘

- 최장 증가 부분 수열이란?
  - **LIS, Longest Increasing Subsequence**
  - 원소가 n개인 배열의 일부 원소를 골라내서 만든 부분 수열 중, 각 원소가 이전 원소보다 크다는 조건을 만족하고, 그 길이가 최대인 부분 수열을 최장 증가 부분 수열이라 한다.
- 알고리즘
  - 속도가 느린 $O(n^2)$ 알고리즘과
  - 개선된 $O(nlogn)$ 알고리즘이 있다.

- 예시

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

  - 증가 부분 수열

    `{ 2, 5 }, { 2, 7, 8 } …`
  - 최장 증가 부분 수열(LIS)

    `{ 2, 5, 7, 8 }`

최장 증가 부분 수열(LIS) 알고리즘

- 첫번째 알고리즘 $O(n^2)$
  - len[k]: arr[k]를 마지막 값으로 가지는 가장 긴 증가 부분 수열의 길이
  - arr[k]가 어떤 증가부분수열의 마지막 값이 되기 위해서는 arr[k]가 추가되기 전 증가부분수열의 마지막 값이 arr[k]보다 작은 값이여야 한다.
  - 따라서 arr[k]를 마지막 값으로 가지는 '가장 긴' 증가부분수열의 길이는 arr[k]가 추가될 수 있는 증가부분수열 중 가장 긴 수열의 길이에 1을 더한 값이 된다.

[k]

idx	0	1	2	3	4	5	6	7
arr	6	2	5	1	7	4	8	3
len	1	1	2	1	3			

[i]

```
for(int k=0; k<N; k++) {
    len[k]=1; // 증가부분수열의 길이는 1부터 시작
    for(int i=0; i<k; i++) {
        if(arr[i] < arr[k])  {
            len[k] = max(len[k], len[i]+1);
        }
    }
}
```

최장 증가 부분 수열(LIS) 알고리즘

- 첫번째 알고리즘 $O(n^2)$

idx	0	1	2	3	4	5	6	7
arr	6	2	5	1	7	4	8	3
len	1	1	2	1				

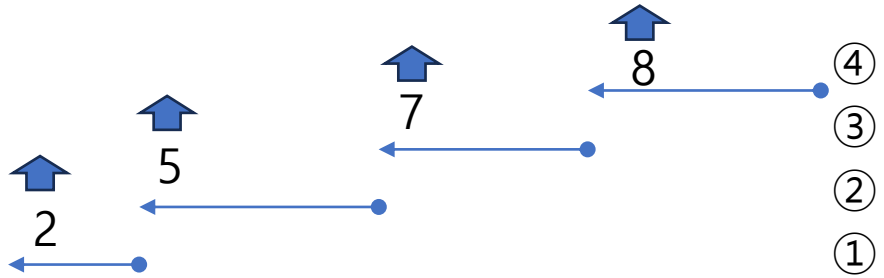
[i] ← → [k]

결국 k앞쪽의 len[]에서 가장 큰 수 찾아서 +1 한다는 뜻으로 아래와 같이 코딩해도 동일 결과

```
for(int k=0; k<N; k++) {
    int max=0;
    for(int i=0; i<k; i++) {  //k 이전 인덱스에서
        if(arr[i] < arr[k]) { //증가조건 만족하는
            if(max < len[i])  //길이 최대값을 업데이트
                max = len[i];
        }
    }
    len[k] = max+1; //arr[k]가 추가되어 늘어나는 길이
}
```

- LIS 수열 구하기

idx	0	1	2	3	4	5	6	7
arr	6	2	5	1	7	4	8	3
len	1	1	2	1	3	2	4	2



- arr배열을 역순으로 탐색하며 최대 길이부터 시작하여,
  - 오른쪽부터 len이 4인 arr원소 출력,
  - len이 3인 arr원소 출력,
  - len이 2인 arr원소 출력,
  - len이 1인 arr원소 출력,

LIS1.cpp

최장 증가 부분 수열(LIS) 알고리즘

- 첫번째 알고리즘 $O(n^2)$ 소스코드

```
#include <iostream>
#define N 8
using namespace std;

int main() {
    int arr[N] = { 6,2,5,1,7,4,8,3 };
    int len[N];

    for(int i=0; i<N; i++)
        printf("%3d", arr[i]);
    cout << endl;

    // LIS 알고리즘 수행
    for(int k=0; k<N; k++) {
        len[k]=1;
        for(int i=0; i<k; i++) {
            if(arr[i] < arr[k])  {
                len[k] = max(len[k], len[i]+1);
            }
        }
    }
    for(int i=0; i<N; i++)
        printf("%3d", len[i]);
    cout << endl;

    int m = len[0];
    for(int i=1; i<N; i++) {  //LIS 길이 구하기
        if(m < len[i])
            m = len[i];
    }
    printf("LIS length: %d\n", m);

    int lis[m];  // LIS를 저장하는 배열
    cout << "LIS: ";
    int s = m;  //찾으려는 수
    for(int i=N-1; i>0 && s>0; i--) {
        if(s == len[i]) {
            lis[s-1] = arr[i];
            s--;
        }
    }

    for(int i=0; i<m; i++)  // LIS 출력
        printf("%3d", lis[i]);
}
```

```
  6  2  5  1  7  4  8  3
  1  1  2  1  3  2  4  2
LIS length: 4
LIS:   2  5  7  8
```

최장 감소 부분 수열(LDS) 알고리즘

- 소스코드

```
#include <iostream>
#define N 8
using namespace std;

int main() {
    int arr[N] = { 6,2,5,1,7,4,8,3 };
    int len[N];

    for(int i=0; i<N; i++)
        printf("%3d", arr[i]);
    cout << endl;

    for(int k=0; k<N; k++) {
        len[k]=1;
        for(int i=0; i<k; i++) {
            if(arr[i] > arr[k]) {
                len[k] = max(len[k], len[i]+1);
            }
        }
    }
    for(int i=0; i<N; i++)
        printf("%3d", len[i]);
    cout << endl;

    return 0;
}
```

- 부등호만 뒤집으면 최장 감소 부분 수열 알고리즘으로 변경됨.
- 실행결과

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

최장 증가 부분 수열(LIS) 알고리즘

- 개선된 $O(nlogn)$ 알고리즘
  - 기존 알고리즘은 배열 내부의 모든 k에 대하여, 그 이전의 원소들을 모두 탐색하기 때문에 $O(n^2)$ 의 시간 복잡도를 갖는다.
  - 따라서 n이 10,000보다 커지게 되면 제한 시간 내 해결이 어려울 수 있다.
  - 이에 대한 해결책은 이전의 원소들을 탐색하는 과정($O(n)$) 을, $O(logn)$ 시간복잡도를 갖는 lower_bound()를 이용하여 시간을 줄여주는 것이다.

- 개선 아이디어
  - LIS를 만들기 위해서는 만드는 과정에서 LIS의 마지막 원소가 가능한 작을수록 더 긴 LIS를 생성할 수 있다는 것이다.
  - 그러므로 원소가 들어올 때, 만약 현재 생성된 LIS의 마지막 원소보다 작은 경우, LIS에 들어갈 위치를 lower_bound()로 찾은 후 원소를 대체한다.

최장 증가 부분 수열(LIS) 알고리즘

- 개선된 알고리즘 $O(nlogn)$

```
/* 입력데이터
8
6 2 5 1 7 4 8 3
*/

#include <iostream>
#include <algorithm>
#define  MAX  1000000
using namespace std;

int arr[MAX];
int len[MAX];
```

- 최장 길이만 탐색

```
int main(void) {
    int n;
    cin >> n;
    int in;     // input number
    int idx = 0;
    for (int i = 0; i < n; i++) {
        cin >> in;
        arr[i] = in;
        if (idx == 0)  // 최초로 입력되는 숫자이면,
            len[idx++] = in;
        else {
            if (len[idx-1] < in)   // in이 마지막 원소보다 크면,
                len[idx++] = in;   // 마지막에 in 추가
            else
                len[lower_bound(len, len + idx, in) - len] = in;
        }
    }
    cout << idx;
}
```

최장 증가 부분 수열(LIS) 알고리즘

- 개선된 알고리즘 $O(nlogn)$

```
/* 입력데이터
8
6 2 5 1 7 4 8 3
*/
#include <iostream>
#include <algorithm>
#define  MAX  1000000
using namespace std;

int arr[MAX];
int len[MAX];
int idx[MAX];
int ans[MAX];

int main(void) {
    int n;
    cin >> n;
    int in;     // input number
    int ix = 0; // index
```

- 최장 부분수열까지 출력

```
    for (int i = 0; i < n; i++) {
        cin >> in;
        arr[i] = in;
        if (ix == 0) {  // 최초로 입력되는 숫자이면,
            len[ix++] = in;
            idx[i] = 0;
        }
        else {
            if (len[ix-1] < in) { // in이 마지막 원소보다 크면,
                idx[i] = ix;
                len[ix++] = in;   // 마지막에 in 추가
            }
            else {
                idx[i] = lower_bound(len, len + ix, in) - len;
                len[lower_bound(len, len + ix, in) - len] = in;
            }
        }
    }
    cout << ix << endl;
```

최장 증가 부분 수열(LIS) 알고리즘

- 개선된 알고리즘 $O(nlogn)$

```
    // idx 배열 출력
    for(int i=0; i<n; i++)
        cout << idx[i] << ' ';
    cout << endl;

    // 부분 증가수열 ans에 삽입
    int t = 0;
    for (int i = n-1; i >= 0; i--) {
        if (ix == idx[i]+1) {
            ans[t++] = arr[i];
            ix--;
        }
    }
    // 출력
    for (int i = t-1; i >=0; i--)
        cout << ans[i] << ' ';
}
```

- 입력과 출력

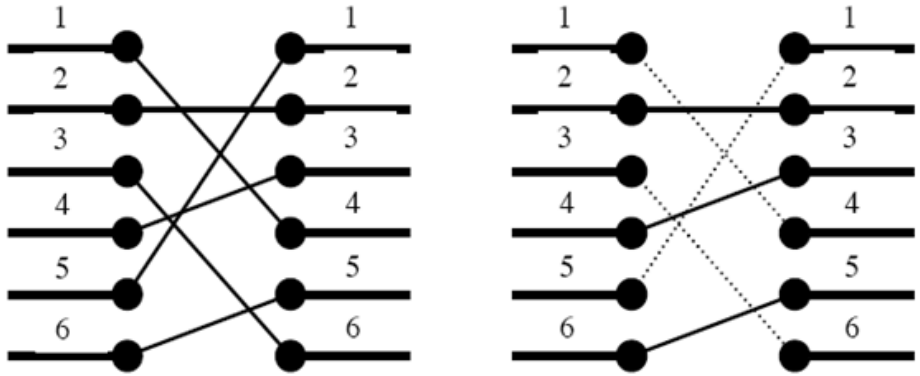


반도체 설계

- 문제

반도체를 설계할 때 n개의 포트를 다른 n개의 포트와 연결해야 할 때가 있다.

예를 들어 오른쪽 그림이 n개의 포트와 다른 n개의 포트를 어떻게 연결해야 하는지를 나타낸다. 하지만 이와 같이 연결을 할 경우에는 연결선이 서로 꼬이기 때문에 이와 같이 연결할 수 없다. n개의 포트가 다른 n개의 포트와 어떻게 연결되어야 하는지가 주어졌을 때, 연결선이 서로 꼬이지(겹치지, 교차하지) 않도록 하면서 최대 몇 개까지 연결할 수 있는지를 알아내는 프로그램을 작성하시오.



- 출처: BOJ https://www.acmicpc.net/problem/2352 추가문제: https://www.acmicpc.net/problem/1365

문제: 오르락 내리락

- 문제

영동 포도축제에서 많은 관광객들의 참여를 위해 오르락 내리락이라는 게임을 기획하고자 한다. 이 게임의 규칙은 다음과 같다.

1) 여러 사람들이 한 줄로 선다.
2) 이 중 일부를 뽑아 오르락 내리락 순서를 만족하는 줄을 만든다.

오르락 내리락 순서란, 모든 사람들이 처음에 서 있던 순서를 지키며 한 줄로 서고, 사람들의 키가 점점 커지다가 다시 작아지는 순서를 의미한다.

예를 들어, 5명의 사람이 서 있고 각각의 키가 {120, 180, 160, 190, 130}이라고 할 때, {120, 180, 160, 130}의 사람들을 뽑으면 규칙에 맞지만, {120, 180, 160, 190}은 규칙에 맞지 않는다.

또한, 오르락 내리락 순서가 된다면 서 있는 모든 사람을 뽑아도 되고, 가장 키가 큰 사람이 맨 앞 혹은 맨 뒤에 서도 된다.

사람의 수(N)과 N명의 사람들의 키(A_i)를 입력하면, 오르락 내리락 순서를 만족하는 가장 긴 줄의 길이를 출력하는 프로그램을 작성해보자.

- 출처: 2023 충북정올 도본선 초등 5번

문제: 오르락 내리락

- 입력형식

  첫 번째 줄에 사람의 수(N)가 입력된다.

  (단, N은 정수이며, 1 <= N <= 1,000의 범위를 갖는다.)

  두 번째 줄에 N명의 키(A_i)가 공백으로 구분되어 입력된다.

  (단, A_i는 정수이며, 100 <= A_i <= 200의 범위를 갖는다.)

- 출력형식

  N명의 사람들로 만들 수 있는 가장 긴 오르락 내리락 순서의 길이를 출력한다.

- 입력과 출력의 예

입력 예1	출력 예1
5 120 180 160 190 130	4

입력 예2	출력 예2
3 120 120 120	1

- 고찰
  - LIS 알고리즘 사용
  - LDS 알고리즘도 사용??
  - N이 1000이하이므로 $O(N^2)$알고리즘을 사용하여도 무방

- 출처: 2023 충북정올 도본선 초등 5번

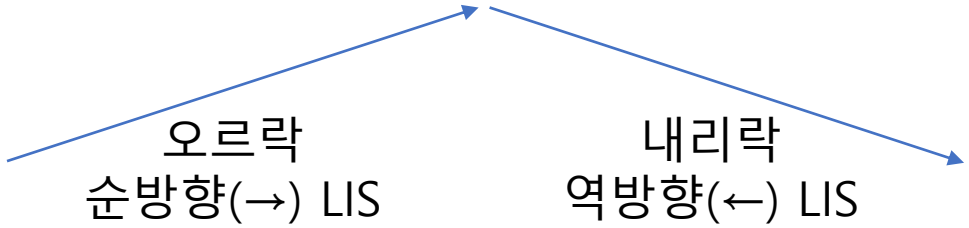
풀이: 오르락 내리락

- 고찰
  - 오르락 최대 부분 수열
    - LIS 알고리즘으로 알아낼 수 있음.
  - 내리락 최대 부분 수열
    - LDS 알고리즘으로 알아낼 수 있음.
    - 그러나,...

7							
	465	80	494	934	164	631	756
LIS 길이	1	1	2	3	2	3	4
LDS 길이	1	2	1	1	2	2	2

7							
	465	80	494	934	164	631	756
LIS 길이	1	1	2	3	2	3	4
	2	1	2	2	1	1	1

- 순방향/역방향 LIS



- 오르락이 종료되고 내리락이 시작되는 지점을 부분수열 길이의 합이 최대가 되는 지점으로 선정

7							
	465	80	494	934	164	631	756
오르락 길이	1	1	2	3	2	3	4
내리락 길이	2	1	2	2	1	1	1

예)934는 오르락길이 3, 내리락길이 2인 지점

그러나, 934는 오르락에도 사용되고 내리락에도 사용되므로 -1 해야 함.

- 출처: 2023 충북정올 도본선 초등 5번

```
// 오르락 내리락 reverse 안하는 버전 (속도는 빠르지면 코드가 덜 길어..)
#include <stdio.h>
#include <algorithm>
#define  MAX  1000
using namespace std;
enum  dir { RIGHT, LEFT };

int height[MAX];
int up_cnt[MAX];
int dn_cnt[MAX];

int fill_LIS_cnt_ary(dir d, int *arr, int *len, int N) {
    if(d == RIGHT) { // 오르락 탐색
        for(int k=0; k<N; k++) {
            len[k]=1;
            for(int i=0; i<k; i++) {
                if(arr[i] < arr[k])
                    len[k] = max(len[k], len[i]+1);
            }
        }
    }
    else if(d==LEFT) { // 내리락 탐색
        for(int k=N-1; k>=0; k--) {
            len[k]=1;
            for(int i=N-1; i>k; i--) {
                if(arr[i] < arr[k])
                    len[k] = max(len[k], len[i]+1);
            }
        }
    }
    return *max_element(len, len+N);
}
```

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

    for(int i=0; i<n; i++)
        scanf("%d", &height[i]);

    fill_LIS_cnt_ary(RIGHT, height, up_cnt, n);
    fill_LIS_cnt_ary(LEFT,  height, dn_cnt, n);

    for(int i=0; i<n; i++)
        printf("%3d ", up_cnt[i]);
    putchar('\n');
    for(int i=0; i<n; i++)
        printf("%3d ", dn_cnt[i]);
    putchar('\n');

    int max = 0;
    for(int i=0; i<n; i++) {
        int sum = up_cnt[i]+dn_cnt[i];
        if(sum > max)
            max = sum;
    }
    printf("%d", max-1);
}
```

```
// 테스트 데이터
5
120 180 160 190 130
```

```
5
120 180 160 190 130
  1   2   2   3   2
  1   3   2   2   1
4
```

2023 충북정올 학교예선대회 (중등부)

- 중등부 예선 문항별 난이도 분석

등위	바른 코드	1번 우영우의 인사말	2번 암산 게임	3번 공성전	4번 카드 놀이	5번 창고	합계
1위	5	100	100	100	100	100	500
2위	4	100	100	100	100	0	400
3위	3	100	100	100	30	0	330
4위	3	100	100	100	0	0	300
5위	2	100	100	90	40	0	330
6위	2	100	100	50	60	0	310
7위	2	100	100	70	30	0	300
8위	2	100	100	80	10	0	290
9위	2	100	100	50	40	0	290
10위	2	100	100	80	0	0	280

문제: 카드놀이

- 문제

  충북이는 1부터 N까지의 정수가 각각 하나씩 적힌 N장의 카드와, 숫자가 적히지 않은 카드 하나를 가지고 있다. 충북이는 이들 N+1장의 카드 중, K개의 카드를 임의로 선택하여 일렬로 펼쳐놓았다.

  이렇게 펼쳐진 K개의 카드에서, 충북이는 가능한 가장 긴 연속 정수열을 만들려고 한다.

  연속 정수열이란, [1, 2, 3, 4]나 [7, 8, 9]처럼 숫자가 하나씩 늘어나는 정수열을 말한다.

  충북이는 모든 카드를 선택하지 않을 수도 있으며, 카드의 순서를 바꿀 수도 있다.

  또한, K개의 카드 중 숫자가 적히지 않은 카드가 있다면, 해당 카드에 1부터 N까지의 정수 중 하나를 적어서 해당 카드를 원하는 숫자 카드처럼 사용할 수도 있다.

  충북이는 주어진 카드들에 최선의 전략을 사용하여, 충북이가 만든 연속 정수열의 길이를 최대화하려고 한다.

  카드의 정보 N, K와 K장의 카드가 주어질 때, 충북이가 만들 수 있는 연속 정수열의 최대 길이를 구해보자.

- 출처: 2023 충북정올 학교예선 중등부 4번 문제

문제: 카드놀이

- 입력형식

  첫 줄에 정수 N과 K가 공백을 사이에 두고 주어진다. N, K는 각각 숫자가 적힌 카드의 수와 펼쳐진 카드의 수를 의미한다.

  다음 K개의 줄 중 i번째 줄에는 정수 ai가 주어진다. 이는 펼쳐진 카드 중 i번째 카드에 쓰인 숫자를 의미한다. 숫자가 적히지 않은 카드는 0으로 주어진다.

  - $1 \le N \le 100{,}000$
  - $1 \le K \le N$
  - $0 \le a_i \le N$

- 출력형식

  충북이가 만들 수 있는 연속 정수열의 최대 길이를 출력한다.

- 입력과 출력의 예

입력 예1	출력 예1
6 4 3 1 6 5	2

입력 예2	출력 예2
6 4 3 1 0 5	3

- 출처: 2023 충북정올 학교예선 중등부 4번 문제

풀이: 카드놀이

검색범위

idx	0	1	2	3	4	5	6
cards	1	1	0	1	0	1	1
block		b0		b1		b2	

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

struct block {
    int st;
    int en;
    int len;
};

int main() {
    int n, k;
    cin >> n >> k;
    int  arr[k];
    bool cards[n+2] = {false, };
    block blocks[k] = {0, };

    for(int i=0; i<k; i++) {
        scanf("%d", &arr[i]);
        cards[arr[i]] = true;
    }
    bool wildcard = (cards[0]==true);

    int b_cnt=0;
    int len = 0;
    for(int i=1; i<=n+1; i++) {
        if(cards[i]==true) len++;
        else {  // 수 연속이 종료되면 block 정보 저장
            if(len>0) {
                blocks[b_cnt].len = len;
                blocks[b_cnt].en = i-1;
                blocks[b_cnt].st = blocks[b_cnt].en - len +1;
                b_cnt++;
                len = 0;
            }
        }
    }

    int res=1;
    for(int c=0; c<b_cnt; c++) {
        printf("[%d] %2d~%2d (%2d)\n", c, blocks[c].st, blocks[c].en, blocks[c].len);
        res = max(res, blocks[c].len + wildcard); //와이드카드를 이용하면 +1
        // 와이드카드가 있고, c가 1일 때 부터,
        if(wildcard && c!= 0 && blocks[c-1].en+2 == blocks[c].st)
            res = max(res, blocks[c-1].len + blocks[c].len +1);
    }
    cout << res;
}
```

```
6 4
3
1
6
5
[0]  1~ 1 ( 1)
[1]  3~ 3 ( 1)
[2]  5~ 6 ( 2)
```

문제: 창고

우선순위 큐부터 배우고 올 것.

- 문제

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

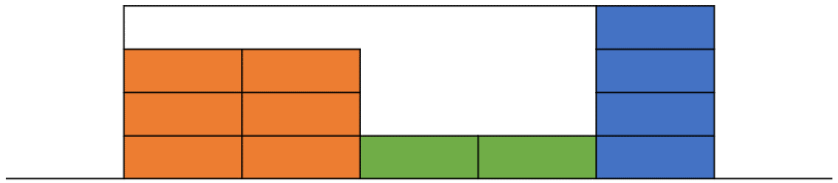
각 상자에는 1~N번까지 번호가 붙여져 있으며, (Wi 너비 * Hi 높이)인 직사각형 모양을 하고 있다.

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 보관해야하는 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

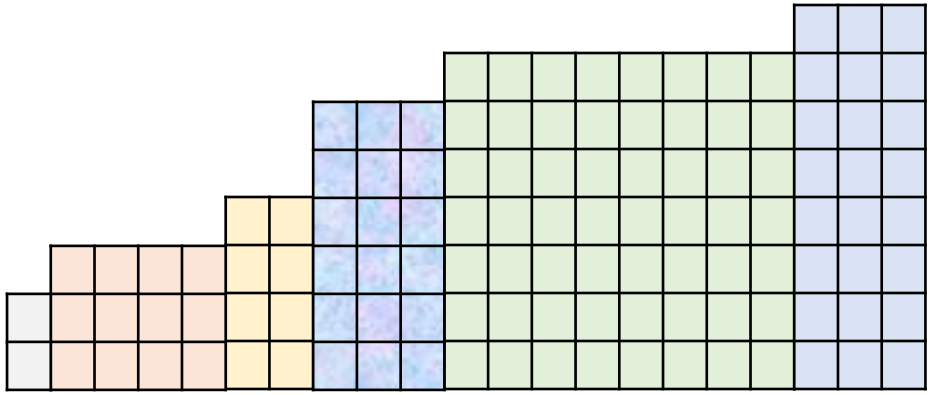
입력 예1	출력 예1
4 3 2 3 2 1 1 4 4 5	20

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 둘 다 작아야 유리하다.
  - 일단 높이를 기준을 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 k-1 개 선택한 상태에서
(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
```

2023 충북정올 본선(도)대회 (중등부)

- 중등부 본선 문항별 난이도 분석

등위	바른 코드	1번 나의인생 최고점수	2번 나온번호가 잘나와	3번 치명적 바이러스	4번 고드름	5번 선물 게임	합계
1위	5	100	100	100	100	100	500
2위	4	100	100	100	100	30	430
3위	4	100	100	100	100	10	410
4위	4	100	100	100	100	0	400
5위	3	100	100	100	90	30	420
6위	3	100	100	100	30	10	340
7위	3	100	100	100	30	0	330
8위	3	100	100	100	20	0	320
9위	3	100	100	100	20	0	330
10위	3	100	100	100	0	10	310

문제: 고드름

- 문제

  충북이는 겨울방학을 맞아 할머니 댁에 찾아갔다. 겨울철 할머니 댁에는 처마에 긴 고드름이 자라는 특징이 있는데, 이번에도 N개의 고드름이 처마에 일렬로 맺혀 있었다.

  각 고드름은 왼쪽에서 오른쪽으로 하나씩 늘어서 있으며, 왼쪽에서 i번째 고드름은 현재 길이 ai를 가지고 있다. 이 1번부터 N번까지의 고드름은 시간에 따라 길이가 성장하는데, 성장 규칙은 다음과 같다.

  1) i번째 고드름은 i-1번째 고드름과 i+1번째 고드름보다 길이가 긴 경우에만 1시간마다 길이가 1씩 늘어난다. (단, 1번 고드름은 2번보다 긴 경우에, N번 고드름은 N-1번 고드름보다 길 경우에 길이가 늘어난다)
  2) 모든 고드름은 길이가 M이 되는 순간 부러져서 길이가 0이 된다. 처음 처마에 있는 모든 고드름은 인접한 고드름들과 길이가 모두 다르다.

  충북이는 충분한 시간이 흘러, 처마의 모든 고드름이 부러질 수 있는지 알고 싶다. 그리고 만약 모든 고드름이 부러질 수 있다면 얼마의 시간이 걸리는지 알고 싶어, 이를 여러분에게 부탁했다.

- 출처: 2023 충북정올 도본선 중등부 4번

문제: 고드름

Q

- 입력형식

  첫 줄에 정수 N, M이 공백을 사이에 두고 주어진다. N은 고드름의 수, M은 고드름이 부러지는 길이를 의미한다.

  다음 N개의 줄 중 i번째 줄에는 정수 ai가 주어진다. 이는 왼쪽에서 i번째 고드름의 길이가 a_i 임을 의미한다.

  - $1 \le N \le 100{,}000$
  - $2 \le M \le 50{,}000$
  - $1 \le a_i < M$

- 출력형식

  모든 고드름이 부러지는데 걸리는 시간을 출력한다.

  만약, 모든 고드름이 부러지는 것이 불가능하다면 –1을 출력한다.

- 입력과 출력의 예

입력 예1	출력 예1
4 7 1 3 2 5	10

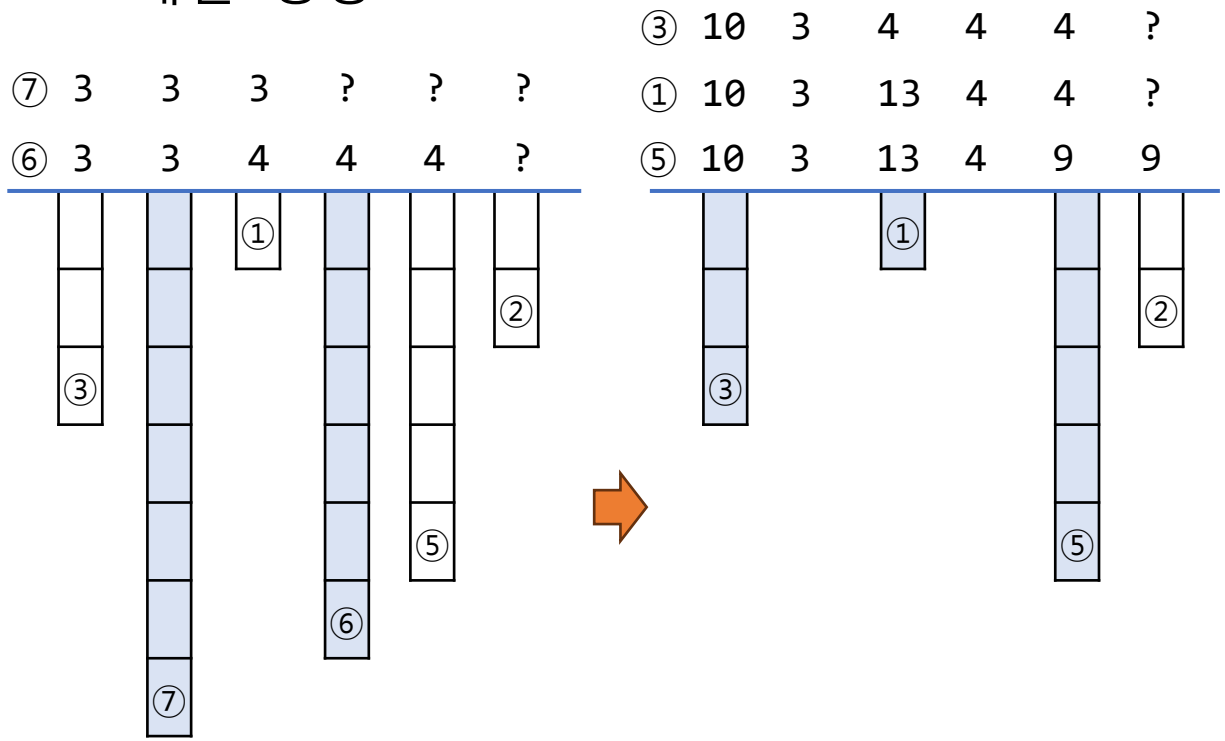
입력 예2	출력 예2
6 10 3 7 1 6 5 2	17

- 출처: 2023 충북정올 도본선 중등부 4번

풀이: 고드름

- 고찰
  - 성장하는 고드름은 양옆보다 길다.
  - 성장하는 고드름이 한계까지 자라서 부러지기 전에는 다른 고드름은 성장하지 않는다.
  - 성장하는 고드름이 부러지는 시간은 (한계길이-현재길이)이다.
  - 성장하는 고드름 양옆은 위 시간만큼 성장이 지연된다.
  - 따라서 고드름이 부러지는 시간은 (지연시간 + 한계까지 자라는 시간)이 된다.

- 부러지는 시간계산
  - 각 고드름이 부러지는 시간을 저장하는 배열 생성



풀이: 고드름

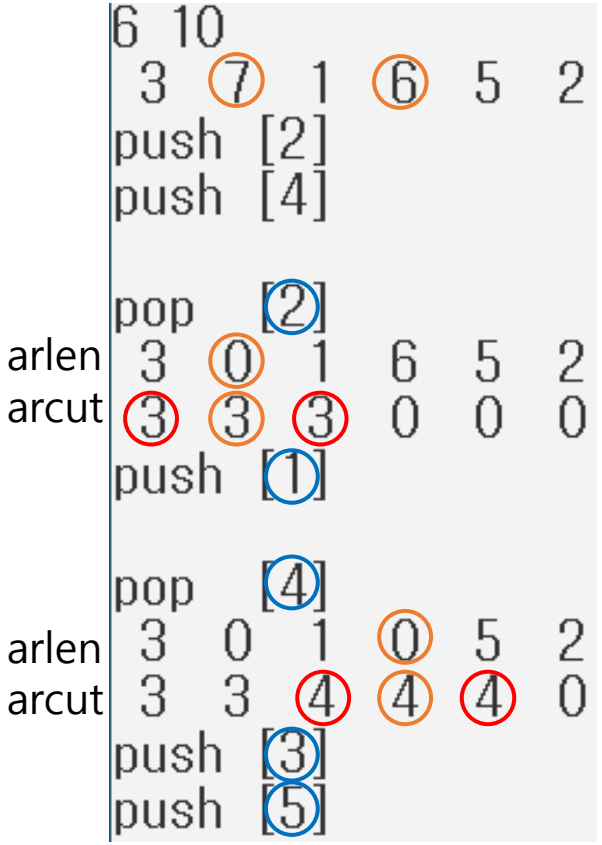
```
#include <iostream>
#include <queue>
using namespace std;

int main(){
    int n, m;
    scanf("%d %d", &n, &m);
    int arlen[n+1]={0, }; // 고드름 길이
    int arcut[n+1]={0, }; // 부러지는데 걸리는 시간
    queue <int> que;

    for(int i=1; i<=n; i++)
        scanf("%d", &arlen[i]);
    for(int i=1; i<=n; i++) { // 양옆 보다 크면 큐에 넣는다.
        if(arlen[i] > arlen[i-1] && arlen[i] > arlen[i+1]) {
            que.push(i);
            printf("push [%d]\n", i);
        }
    }
    putchar('\n');
```

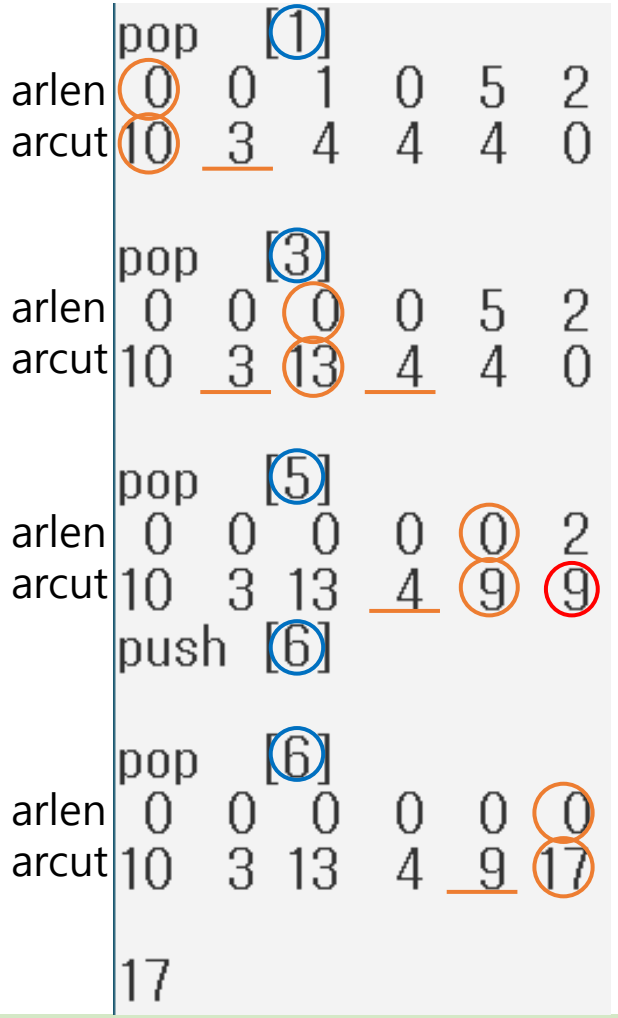
```
    int hour = 0;
    while(!que.empty()) { //큐가 빌때까지 반복
        int a = que.front(); que.pop();
        printf("pop  [%d]\n", a);

        //부러지는데 걸리는시간=지연시간+한계까지 자라는 시간
        arcut[a] += (m - arlen[a]);
        arlen[a] = 0;  // 부러짐 처리

        // 아직 부러지지 않은 고드름이면,
        if(arlen[a-1] > 0) //왼쪽 지연시간 업데이트
            arcut[a-1] = max(arcut[a-1], arcut[a]);
        if(arlen[a+1] > 0) //오른쪽 지연시간 업데이트
            arcut[a+1] = max(arcut[a+1], arcut[a]);

        for(int i=1; i<=n; i++)
            printf("%2d ", arlen[i]);
        putchar('\n');
        for(int i=1; i<=n; i++)
            printf("%2d ", arcut[i]);
        putchar('\n');
```

풀이: 고드름

```
        hour = max(hour, arcut[a]);

        // 부러진 a번째 고드름 왼쪽이 성장조건 만족하면,
        if(arlen[a-1] > 0 && arlen[a-2] < arlen[a-1]) {
            que.push(a-1);
            printf("push [%d]\n", a-1);
        }
        // 부러진 a번째 고드름 오른쪽이 성장조건 만족하면,
        if(arlen[a+1] > 0 && arlen[a+1] > arlen[a+2]) {
            que.push(a+1);
            printf("push [%d]\n", a+1);
        }
        putchar('\n');
    }
    printf("%d\n", hour);

    return 0;
}
```

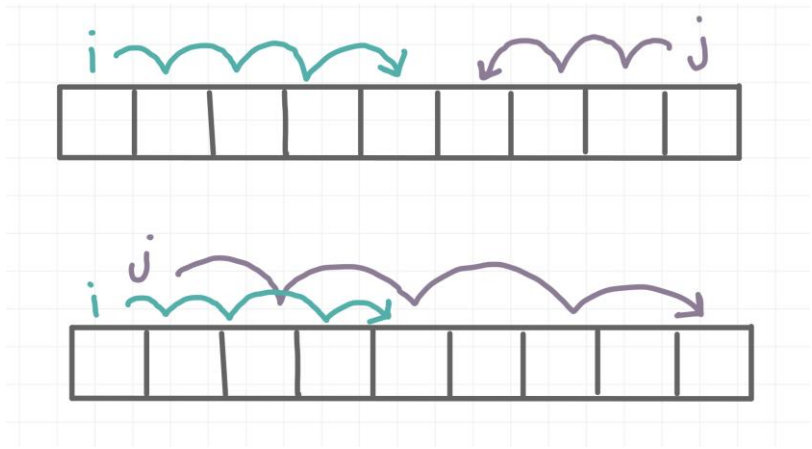
- 실행결과





투 포인터(Two Pointers) 테크닉

- 개념
  - 2개의 포인터를 활용하여 문자열이나 배열에서 원하는 값을 얻어내는 방법이다.
  - 기본 탐색 방식(다중 반복)을 사용하면 시간 초과가 발생하는 경우 사용을 고려해볼만 하다.
  - 메모리와 시간 효율성을 높일 수 있다.
  - $O(N3)$, $O(N2)$ 알고리즘을 $O(N)$ 알고리즘으로 바꿀 수 있다.



투 포인터(Two Pointers) 테크닉

- 유형1 <범위 탐색>
  - 10개의 자연수의 부분합이 5가 되는 경우의 수를 구하시오.

입력 예1	출력 예1
10 5 1 2 3 4 2 5 3 1 1 2	3

  - 고전적 풀이법

    3중 for 문을 이용하여 시작점, 끝점, 두 지점의 합을 구하는 방식은 시작복잡도 $O(N^3)$으로 너무 느림.

```
// 다중반복문을 이용한 해결
#include <iostream>
using namespace std;

int main() {
    int n, t;
    scanf("%d %d", &n, &t);
    int arr[n+1] = {0, };
    for(int i=1; i<=n; i++)
        scanf("%d", &arr[i]);

    int cnt=0;
    for(int a=1; a<=n; a++) {          // a: 시작점
        for(int b=1; b<=n; b++) {      // b: 끝점
            int sum=0;
            for(int c=a; c<=b; c++) { // a ~ b까지의 합 계산
                sum += arr[c];
            }
            if(sum == t) {
                cnt++;
                printf("[%d, %d]\n",a,b);
            }
        }
    }
    cout << cnt;
}
```

```
10 5
1 2 3 4 2 5 3 1 1 2
[2, 3]
[6, 6]
[7, 9]
3
```

투 포인터(Two Pointers) 테크닉

- 유형1 <범위 탐색>

  빠른 테크닉

  - start, end라는 두 개의 포인터를 사용
  - start는 부분배열의 앞 쪽을 가리키는 인덱스, end는 부분배열의 뒤쪽을 가리키는 인덱스
  - 맨 처음에 두 포인터는 0에서 시작하며 항상 start<=end를 만족해야 함.
  - 그리고 매 순간마다 부분합 배열의 합과 구해야하는 값을 비교하여 포인터를 이동

  ① 부분합 배열의 합 >= 구해야하는 값
  start를 오른쪽으로 한 칸 이동하여 부분합 배열의 크기를 감소시킵니다.

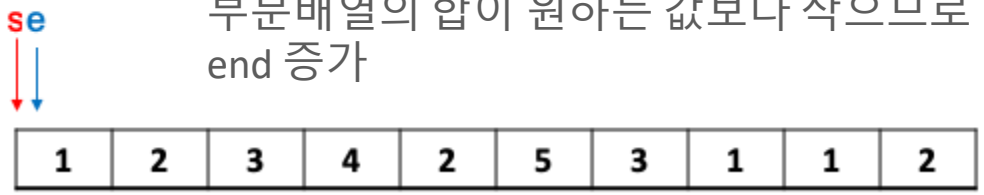
  ② 부분합 배열의 합 < 구해야하는 값
  end를 오른쪽으로 한 칸 이동하여 부분합 배열의 크기를 증가시킵니다.

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

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

    int cnt=0, sum=0;
    int s=0, e=0;

    while(e <= n) {
        if(sum >= t)
            sum -= arr[s++];
        else if(sum < t)
            sum += arr[e++];

        if(sum == t) {
            cnt++;
        }
    }
    printf("%d", cnt);
}
```

투 포인터(Two Pointers) 테크닉

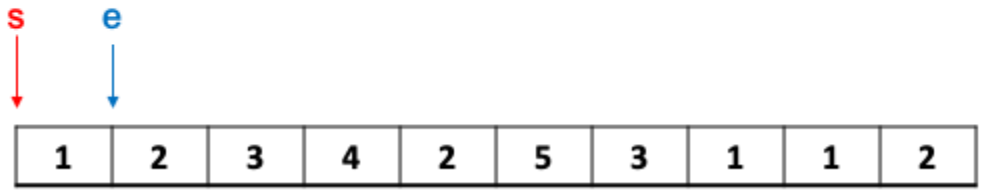
- 유형1 <범위 탐색>

부분배열의 합 : 0

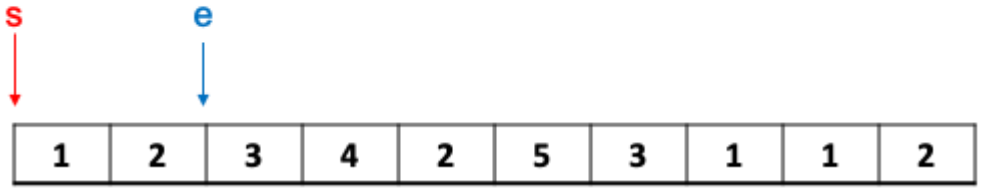


부분배열의 합이 원하는 값보다 작으므로
end 증가

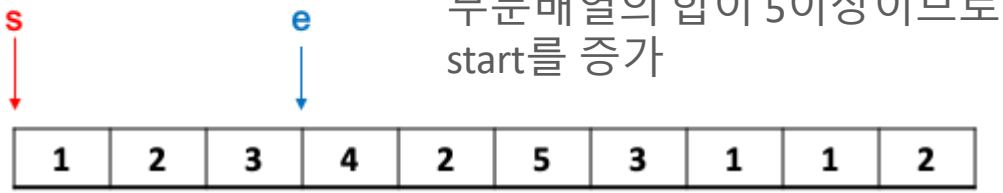
부분배열의 합 : 1



부분배열의 합 : 3

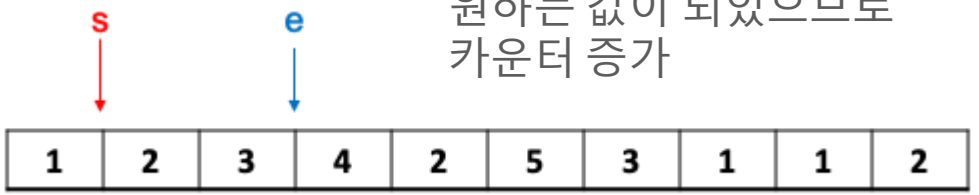


부분배열의 합 : 6



부분배열의 합이 5이상이므로
start를 증가

부분배열의 합 : 5



원하는 값이 되었으므로
카운터 증가

투 포인터(Two Pointers) 테크닉

- 유형2 <합 구하기>
  - 작은 수부터 큰 수로 정렬된 배열 arr와 숫자 target가 주어진다.
  - 이 배열 내에서 숫자 두 수의 합이 target인 그 두 수의 인덱스 (0-base)를 출력하시오.

입력 예1	출력 예1
7 20 2 4 6 9 16 20 22	1 4

  - 2중 반복문을 사용하면 time out 되도록 문제가 설계됨.

```
// 다중 반복문 사용하여 느린 방법
#include <iostream>
using namespace std;

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

    int sum=0;
    int a=0, b=n-1;
    for(int a=0; a<n; a++) {
        for(int b=0; b<n; b++) {
            if(a!=b && a<b && arr[a]+arr[b] == t) {
                printf("%d %d\n", a+1, b+1);
                return 0;
            }
        }
    }
}
```

```
7 20
2 4 6 9 16 20 22
2 5
```

투 포인터(Two Pointers) 테크닉

- 유형2 <합 구하기>
  - 투 포인터 활용

idx	0	1	2	3	4	6	7
arr	2	4	6	9	16	20	22
	s						e

- 초기값 s=0, e=n-1

idx	0	1	2	3	4	6	7
arr	2	4	6	9	16	20	22
	s				e		

- arr[s]+arr[e] < t(20) 이면, e가 이동

idx	0	1	2	3	4	6	7
arr	2	4	6	9	16	20	22
		s			e		

- arr[s]+arr[e] > t(20) 이면, s가 이동

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

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

    int sum=0;
    int s=0, e=n-1;
    while(s < e) { // 두 포인터가 만날때까지
        if(arr[s] + arr[e] < t)
            s++;
        else if(arr[s] + arr[e] > t)
            e--;
        else  //  arr[s]+arr[e] == t
            break;
    }
    if(s < e) printf("%d %d", s+ 1, e+1); // 탐색 성공
    else printf("-1");  // 탐색 실패
}
```

```
7 20
2 4 6 9 16 20 22
2 5
```

```
7 21
2 4 6 9 16 20 22
-1
```

문제: 선물 게임

- 문제

  충북이는 놀이공원에서 놀던 도중, 게임 이벤트 중인 가게를 발견했다. 해당 이벤트는 참가한 모든 사람 중, 1등에게 고드미 인형을 준다고 한다.

  고드미 인형을 받고 싶은 충북이는 위 이벤트에 참가하기로 한다.

  이벤트 규칙은 다음과 같다.

  1) N개의 칸이 있는 점수판이 있으며, i번째 칸에는 점수 Pi가 적혀있다.
  2) 한 사람당 점수판에 최대 4번의 공을 던질 수 있으며, 맞춘 칸에 해당하는 점수 합이 해당 참가자의 총점이 된다. 이때, 공은 무조건 하나의 점수 칸에만 맞는다. (단, 반드시 4개를 모두 던질 필요는 없으며, 공을 하나도 던지지 않아도 된다. 또한, 같은 칸을 여러 번 맞추는 것도 허용된다).
  3) 게임이 끝난 후 총점이 점수 상한선 M을 초과할 경우, 해당 참가자의 총점은 0점으로 취급한다.

  충북이는 야구 신동이기 때문에, 이벤트에서 원하는 칸만 맞추는 것이 가능하다. 그래서 주어진 4번의 기회에서 최적의 칸들을 맞추다가, 해당 점수판에서 얻을 수 있는 M 이하의 최대 점수를 얻고 나면 멈추어 최종 1등을 하기로 계획한다.

  이를 위해, 주어진 점수판에서 충북이가 이벤트에서 얻을 수 있는 M 이하의 최대 점수를 구하고자 한다.

- 출처: 2023 충북정올 도본선 중등부 5번

문제: 선물 게임

- 입력형식

  첫 줄에 정수 N과 M이 공백을 사이에 두고 주어진다. N은 점수판 칸의 개수, M은 충북이가 초과하면 안되는 점수 상한선을 의미한다.

  다음 N개의 줄 중 i번째 줄에는 정수 Pi가 주어진다. Pi는 i번째 점수 칸에 적힌 점수를 의미한다.

  - $1 \le N \le 500$
  - $1 \le M \le 200{,}000{,}000$
  - $1 \le P_i \le 100{,}000{,}000$

- 출력형식
  - 충북이가 이벤트에서 얻을 수 있는 M 이하의 최대 점수를 출력한다.

- 입력과 출력의 예

입력 예1	출력 예1
4 50 9 21 20 5	50

입력 예2	출력 예2
3 21 6 14 2	20

- 출처: 2023 충북정올 도본선 중등부 5번

풀이1: 선물 게임

- 다중 반복문을 활용한 풀이
  - timeout에 빠진다.

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

int main()
{
    int n, m;
    scanf("%d %d", &n, &m);
    int p[n+2] = {0, };
    for(int i=1; i<=n; i++)
        scanf("%d", &p[i]);

    // 입력된 점수 목록을 내림차순 정렬하여,
    // 큰 수를 이용하여 먼저 합계를 만들도록 한다.
    sort(p, p+n, greater<int>());
```

```
    int res = 0;
    for(int a=1; a<=n; a++) {
        if(p[a]>m)
            continue;
        for(int b=0; b<=n; b++) {
            if(p[a]+p[b]>m)
                continue;
            for(int c=0; c<=n; c++) {
                if(p[a]+p[b]+p[c]>m)
                    continue;
                for(int d=0; d<=n; d++) {
                    int score = p[a]+p[b]+p[c]+p[d];
                    if(score == m) {
                        cout << res;
                        return 0;
                    }
                    else if(score < m) {
                        res = max(res, score);
                    }
                }
            }
        }
    }
    cout << res;
    return 0;
}
```

- 출처: 2023 충북정올 도본선 중등부 5번

풀이2: 선물 게임

- Two Pointers 활용
  - 투 포인터 '합 구하기 유형'에 해당
  - 이 유형은 합을 구할 때, 2중 반복문을 단일 반복문으로 바꾸는 테크닉임
  - 지금 문제는 4중 반복 문제이므로 우선 2중 반복 문제로 바꾸는 것이 선행되어야 함.
  - 2중 반복으로 만들어지는 데이터 생성

	0	6	14	2
0	0	6	14	2
6	6	12	20	8
14	14	20	28	16
2	2	8	16	4

	0	6	14	2
0	0	2	2	4
6	6	6	8	8
14	12	14	14	16
2	16	20	20	28

S 0	2	2	4
6	6	8	8
12	14	14	16
16	20	e 20	28

res: 20 (종료)

만약 합13을 찾는 것이었다면,

S 0	2	2	4
6	6	8	8
e 12	14	14	16
16	20	20	28

0	S 2	2	4
6	6	8	8
e 12	14	14	16
16	20	20	28

0	S 2	2	4
6	6	8 e	8
12	14	14	16
16	20	20	28

0	2	2	4
S 6	6	8 e	8
12	14	14	16
16	20	20	28

- 출처: 2023 충북정올 도본선 중등부 5번

풀이2: 선물 게임

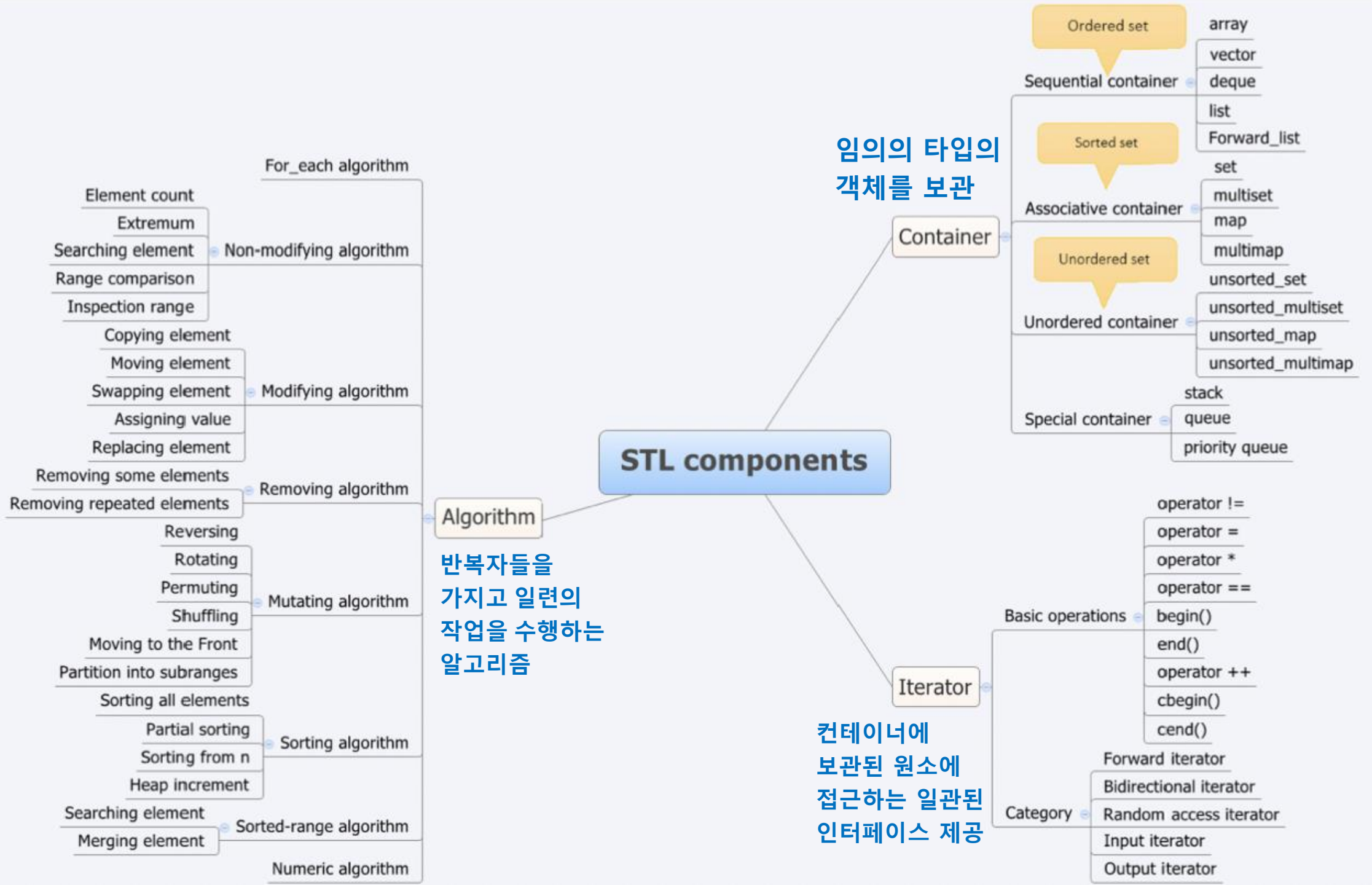
- Two Pointers 활용

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

int arp[501];     //점수 목록이 저장되는 배열
int ars[501*501]; //두 번 던졌을 때 점수를 저장하는 배열

int main() {
    int n, m;
    scanf("%d %d", &n, &m);
    //공을 반드시 4번 던져야 하는 것이 아니므로,
    n++; //공을 안 던진 경우(0점)도 저장하기 위해 1더함
    for(int i=1; i<n; i++)  // 0번 인덱스에는 0점 저장
        scanf("%d", &arp[i]);

    for(int a=0; a<n; a++) //두번 던졌을때 점수조합 계산
        for(int b=0; b<n; b++)
            ars[a*n+b] = arp[a]+arp[b];
    sort(ars, ars+n*n);  //오름차순 정렬

    int res = 0;
    int s=0, e=n*n-1;
    while(s < e) {    //투 포인터 테크닉
        int sum = ars[s]+ars[e];

        if(sum < m) { //m보다 작을 땐 점수도 기록
            s++;
            res = max(res, sum);
        }
        else if(sum > m) {
            e--;
        }
        else {  //목표숫자(m)와 일치하면 중단
            res=sum;
            break;
        }
    }
    cout << res;
    return 0;
}
```

- 출처: 2023 충북정올 도본선 중등부 5번

STL

(Standard Template Library)


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

1) 컨테이너

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

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

1) 컨테이너 (container)

- 컨테이너별 성능비교

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

2) 반복자 (interator)

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

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

3) 알고리즘

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

종류	설명	대표 함수

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

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

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

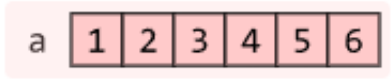
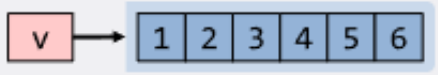
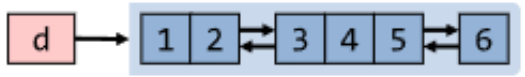
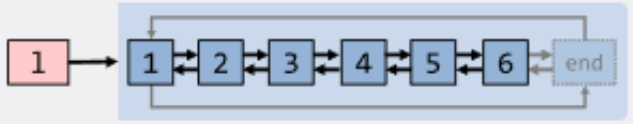
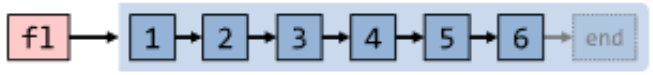
수치 알고리즘	STL에 직접 속하지 않고 C++라이브러리로 분류되는 알고리즘으로 수치적 해석을 위해 사용	#include <numeric>
		accumulate()

STL Container

Standard Template Library

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

Standard Sequence Containers Overview

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

array 컨테이너

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

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

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

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

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

반복자 무효화

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

array 컨테이너

▪ 대입연산자 (oprator=)

배열의 각각의 원소들에 대해 대입 연산자를 호출합니다. 쉽게 말해

C/C++ | 확대 | 축소

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

b = a;  // b 에 {1,2,3} 이 들어간다
```

가 됩니다. 반면에 C 배열의 경우

C/C++ | 확대 | 축소

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

b = arr;  // <-- 불가능!!
```

▪ at, operator[]

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

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

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

vector container

- vector의 특징
  - 크기를 바꿀 수 있는 순차 컨테이너
  - 가변길이 배열이라고 생각하면 쉬움
  - 특정 위치의 원소에 빠르게 접근 가능
  - 벡터에 원소가 삽입되고 삭제됨에 따라 자동으로 크기 조절됨
  - #include <vector> 필요
  - 원소의 추가/삭제

    벡터의 뒤쪽 끝에서만 삽입, 삭제 가능

    삽입: push_back(x)

    삭제: push_pop();

- 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};          // 벡터 v 선언 및 초기화
    v.push_back(6);                     // 맨 마지막에 6 삽입
    v.pop_back();                       // 맨 마지막 원소 제거
    v[1] = 3;                           // 1번 원소 3으로 교체

    printf("%d\n", v[2]);               // 2번 인덱스 원소 출력
    for(int x: v) printf("%d ", x);     // 벡터의 모든 원소를 순회하면서 출력
    v.reserve(8);                       // 벡터에 할당된 메모리 8칸으로 조정
    v.resize(5, 0);                     // 벡터의 크기 5로 조절하고 빈 공간 0으로 채움
    printf("\n%d\n", v.capacity());     // 벡터가 차지하는 공간 출력
    printf("%d\n", v.size());           // 벡터의 크기 출력
}
```

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} };

    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

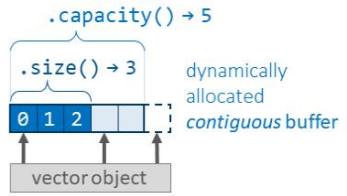
Construct A New Vector Object

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

C++17 value type deducable from argument type

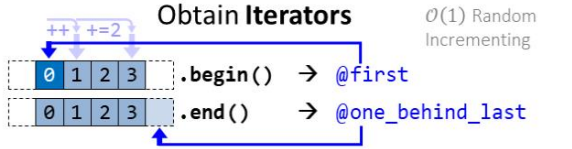
```
vector w {7,4,2};   // vector<int>
```

Typical Memory Layout

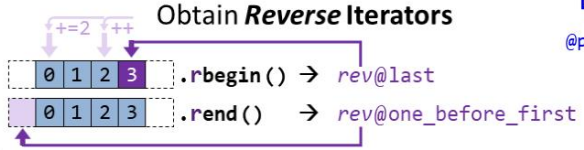


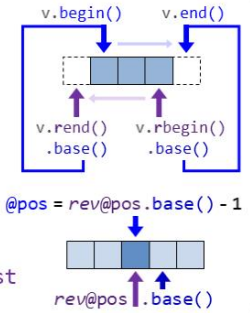
Obtain Iterators

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



Obtain *Reverse* Iterators



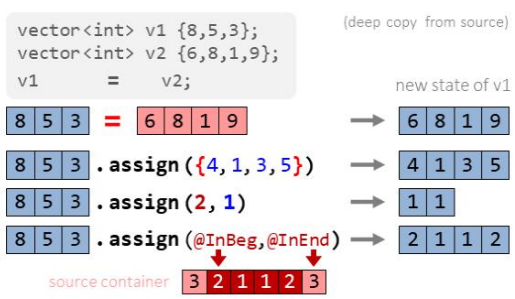


Assign New Content To An Existing Vector

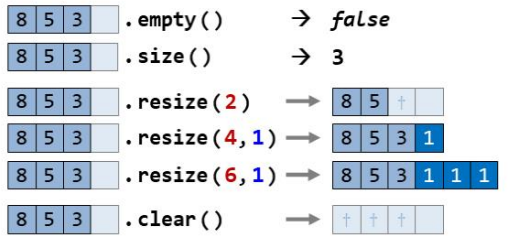
```
vector<int> v1 {8,5,3};
vector<int> v2 {6,8,1,9};
v1     =    v2;
```

(deep copy from source)

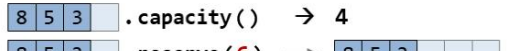
new state of v1



Query/Change Size (= Number of Elements)



Query/Grow Capactiy (= Memory Buffer Size)




```
2 8 5 3 .data() → pointer_to_first
```

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

Append Elements

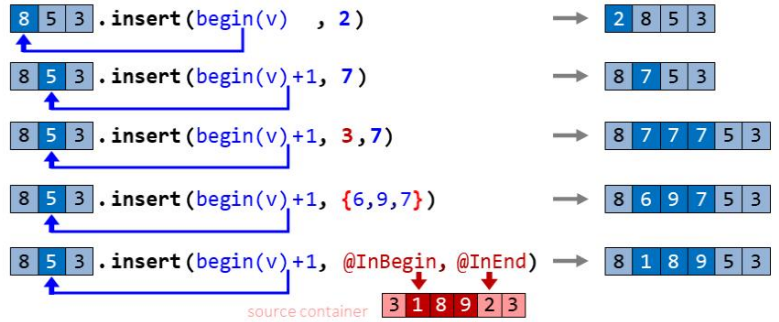
$\mathcal{O}(1)$ Amortized Complexity

```
8 5 3 .push_back(7) → 8 5 3 7
```

Insert Elements at Arbitrary Positions

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

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



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
```

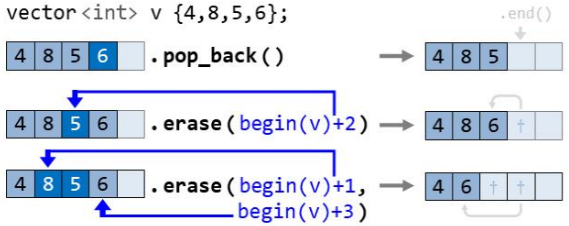
Out of Bounds Access

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

Erase Elements

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

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



Shrink The Capacity

(might be inefficient)

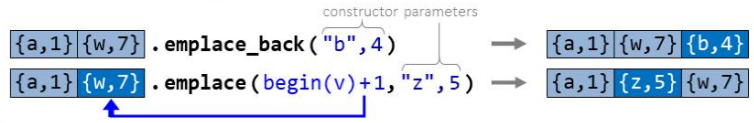
Erasing, resizing or clearing will not shrink the capacity!

```
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

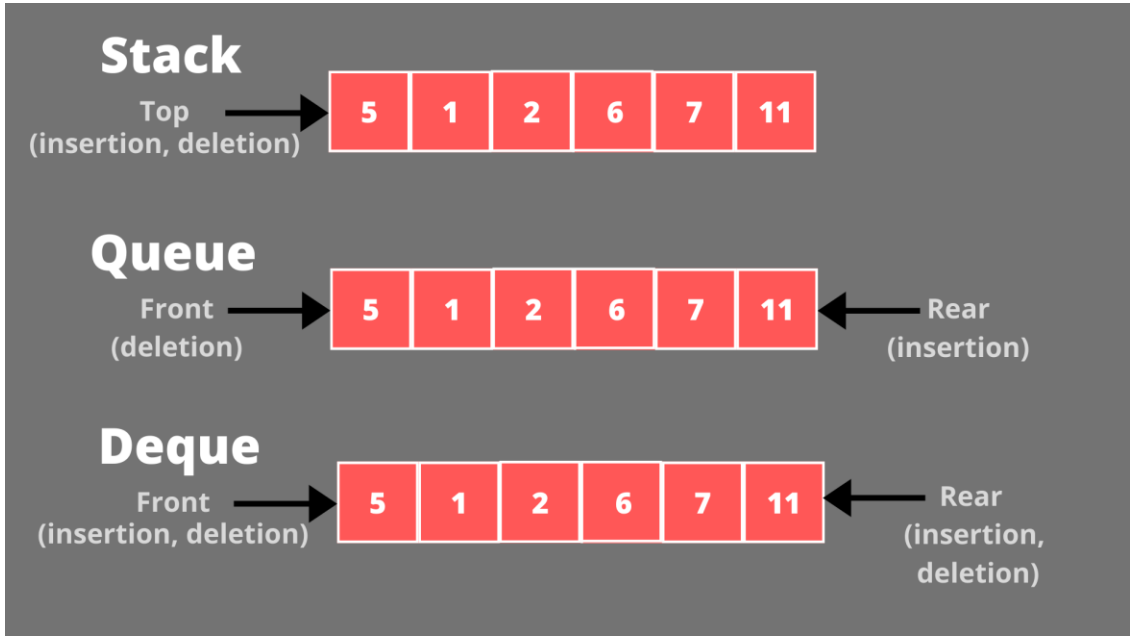
```
vector<pair<string,int>> v {{"a",1},{"w",7}};
```



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

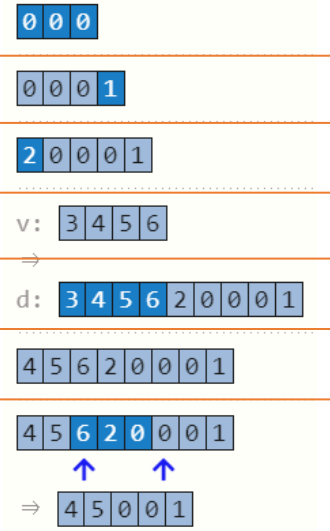
- 수정자(Modifier) 메소드
  - assign : 데크에 원소를 집어넣는다.
  - push_back : 데크 끝에 원소를 집어 넣는다.
  - push_front : 데크 맨 앞에 원소를 집어 넣는다.
  - pop_back : 마지막 원소를 제거한다.
  - pop_front : 첫번째 원소를 제거한다.
  - insert : 데크 중간에 원소를 추가
  - erase : 원소를 제거한다.
  - swap : 다른 데크와 원소와 교환한다.
  - clear : 원소를 모두 제거한다.

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 = dq.begin();
  cout << "[ ";
  for ( ; 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} → 2 9 1 8 5 4

deque<int> d2 (begin(d1)+3, end(d1)) → 8 5 4

deque<int> d3 (5,3) → 3 3 3 3 3

deque<int> deep_copy_of_d1 (d1) → 2 9 1 8 5 4

C++17 value type deducable from argument type

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

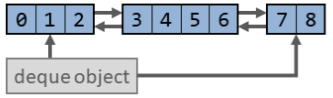
Typical Memory Layout

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

0 1 2 → 3 4 5 6 → 7 8

deque object



Obtain Iterators

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

++ +=2

0 1 2 3 .begin() → @first

0 1 2 3 .end() → @one_behind_last

Obtain *Reverse* Iterators

+=2 ++

0 1 2 3 .rbegin() → rev@last

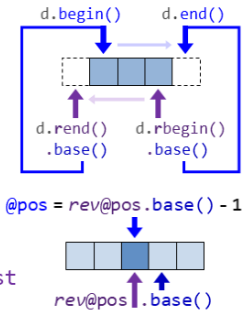
0 1 2 3 .rend() → rev@one_before_first

d.begin() d.end()

d.rend() .base() d.rbegin() .base()

@pos = rev@pos.base() - 1

rev@pos .base()



Assign New Content To An Existing Deque

(deep copy from source)

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

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}};

constructor parameters

{a,1} {w,7} .emplace_back ("b",4) → {a,1} {w,7} {b,4}

{a,1} {w,7} .emplace_front("c",6) → {c,6} {a,1} {w,7}

{a,1} {w,7} .emplace(begin(d)+1,"z",5) → {a,1} {z,5} {w,7}

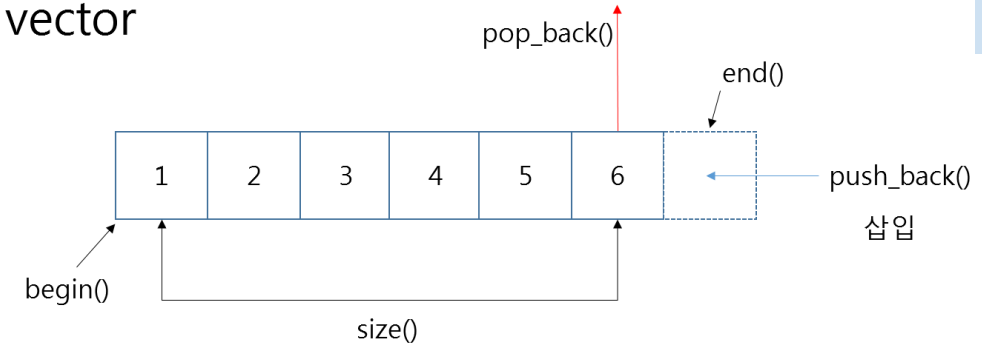
v1.1

STL iterator

Standard Template Library

1. vector에 적용 예시

STL 반복자 (iterator)



- 원소의 삽입과 삭제 (반복자의 활용)

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

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

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

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

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

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

STL algorithm

Standard Template Library

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

STL 알고리즘 **sort** (primitive type)

- 기본 데이터 타입 sort 예시

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

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

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

```
 vector<int> vec(begin(ary), end(ary
    cout << "initial vector sequence
    print_elements(vec.begin(), vec.

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

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

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

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

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

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

STL 알고리즘 **sort** (custom type)

- 사용자 정의 타입 sort 예시

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

struct noco {
    int node;
    int cost;
};

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

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

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

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

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

STL 알고리즘 **sort** (custom type) cont.

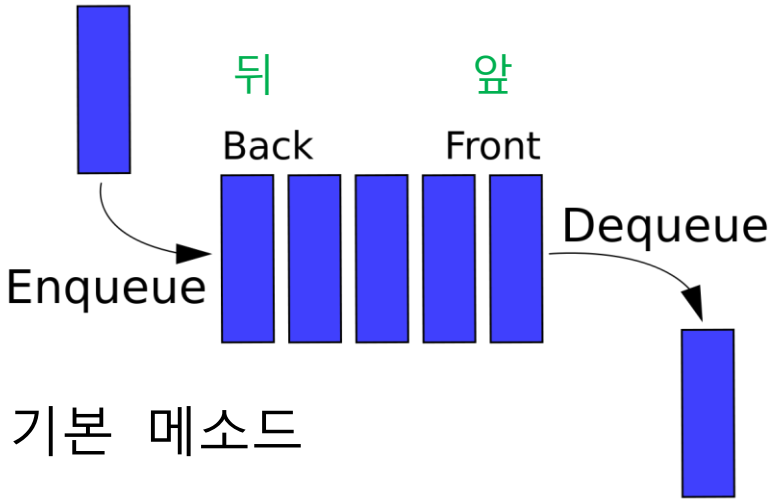
- 사용자 정의 타입 sort 예시

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

struct noco {
    int node;
    int cost;

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

```
  :
  :

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

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

queue container

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



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

- 선언과 사용 예

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

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

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

```
1
2
10
20
```

queue container

- Queue의 내용물을 확인하려면?

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

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

Q	1	3	5	7	9

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

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

    output_Q(q);
}
```

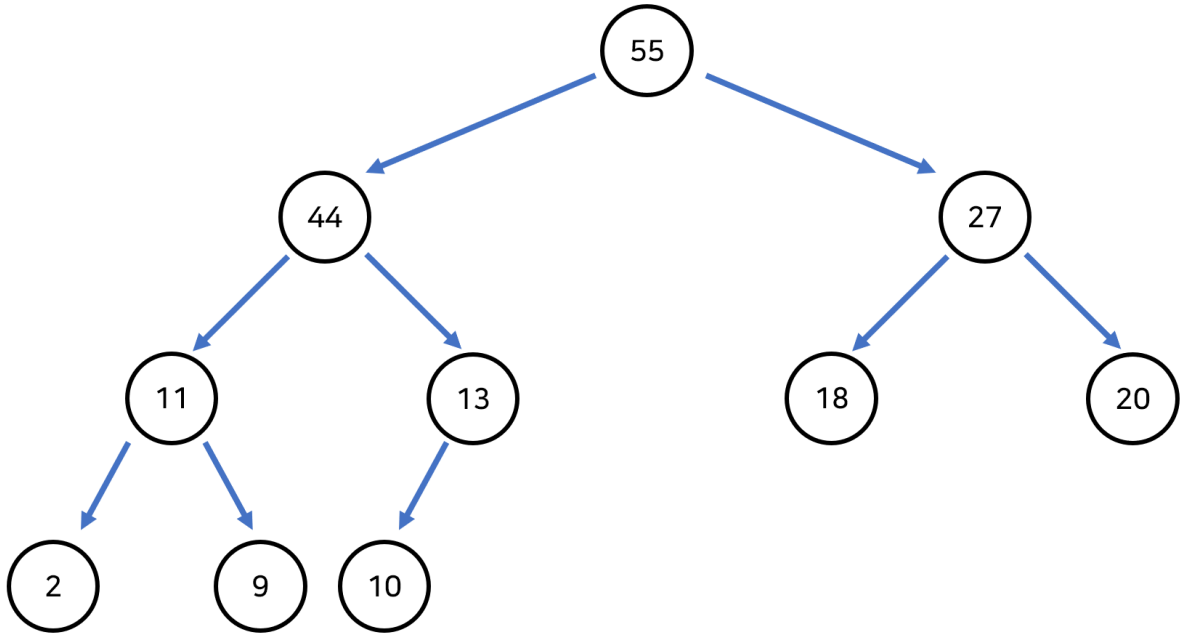
q	1	3	5	7	9

힙(heap)

- 정의
  - 영단어 힙(heap)은 ' 무엇인가를 차곡차곡 쌓아올린 더미 ' 라는 뜻
  - 힙은 완전 이진 트리로 구현된 자료구조
  - 가장 크거나 작은 값을 찾아내는 연산을 빠르게 하기위해 고안된 자료구조
- 힙의 종류
  - 최대힙(Max Heap) : 부모 노드의 값이 무조건 자식 노드의 값보다 큰 힙
  - 최소힙(Min Heap) :부모 노드의 값이 무조건 자식 노드의 값보다 작은 힙

힙(heap)

- 최대 힙의 예

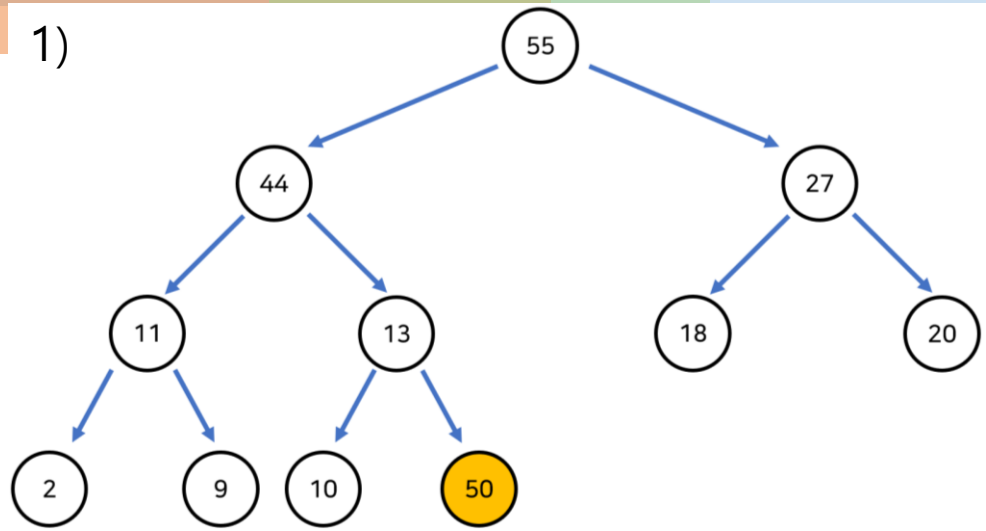


- Root의 값이 가장 크다.
- 또한 모든 서브트리에 대해서도 같다.

- 자료의 추가

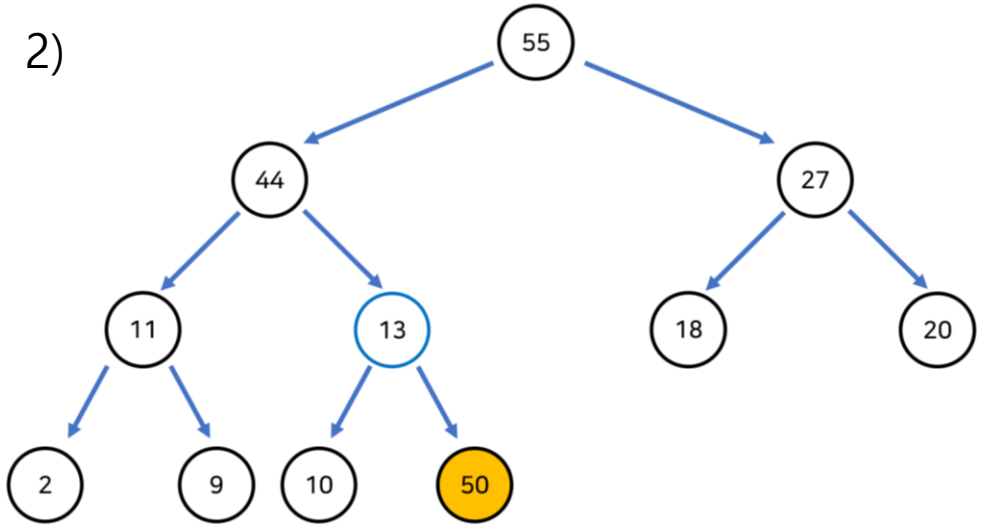
1) 새로운 노드를 트리의 맨 뒤에 추가한다. (완전 이진 트리의 형태를 깨면 안됨)
2) 추가된 노드와 부모 노드를 비교하여 자식 노드가 크다면 서로의 위치를 바꾼다.
3) 2번 작업을 부모 노드가 더 클때 까지 반복한다.

1)



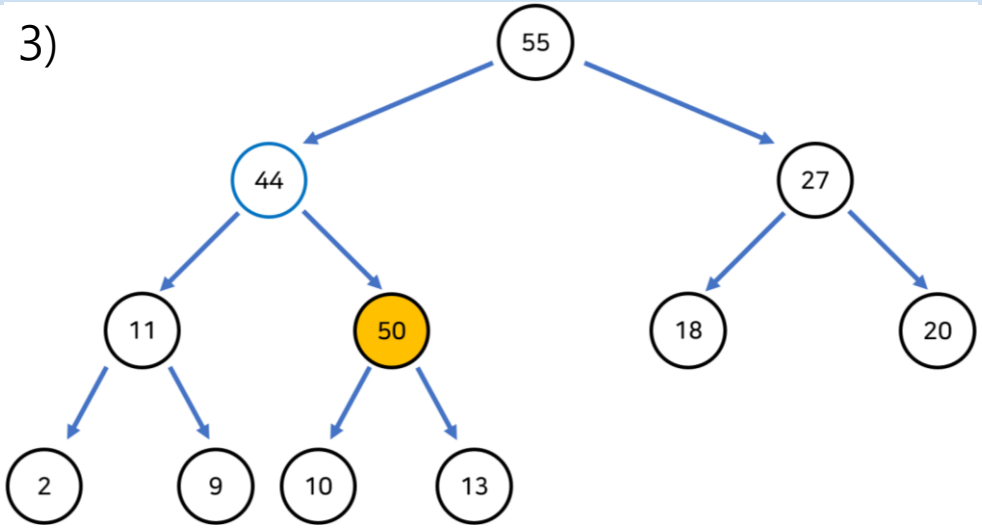
MaxHeap에 50의 값을 가진 노드를 추가하려는 상황이다. 그러면 이 노드는 트리의 맨 뒤인 13의 rightChild로 들어가게 된다.

2)



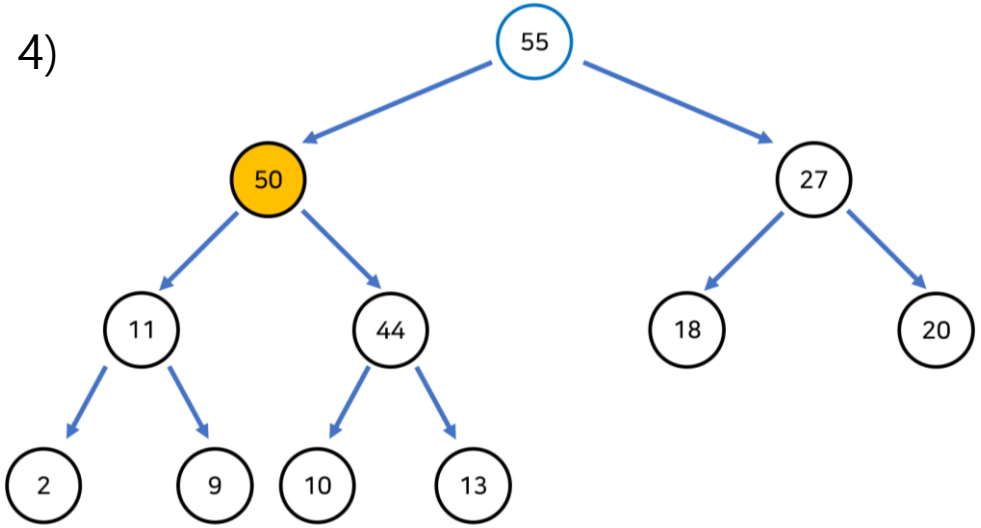
50의 부모노드인 13과 비교한다.
50이 크므로 둘의 자리를 바꿔준다.

3)



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

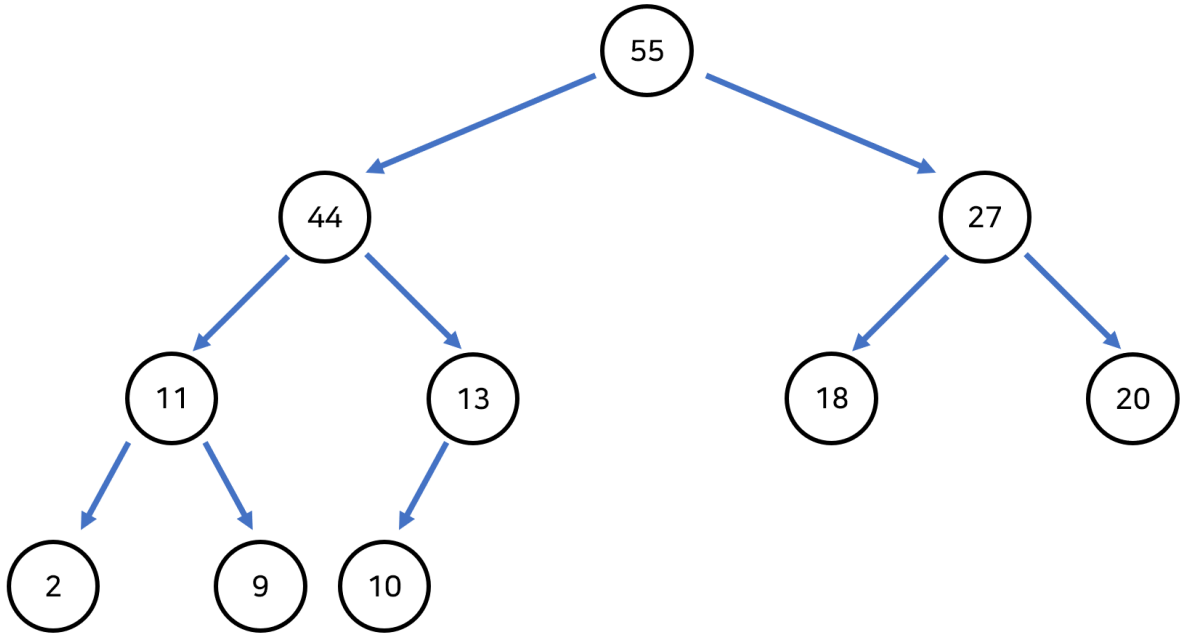
4)



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

힙(heap)

- 자료의 삭제

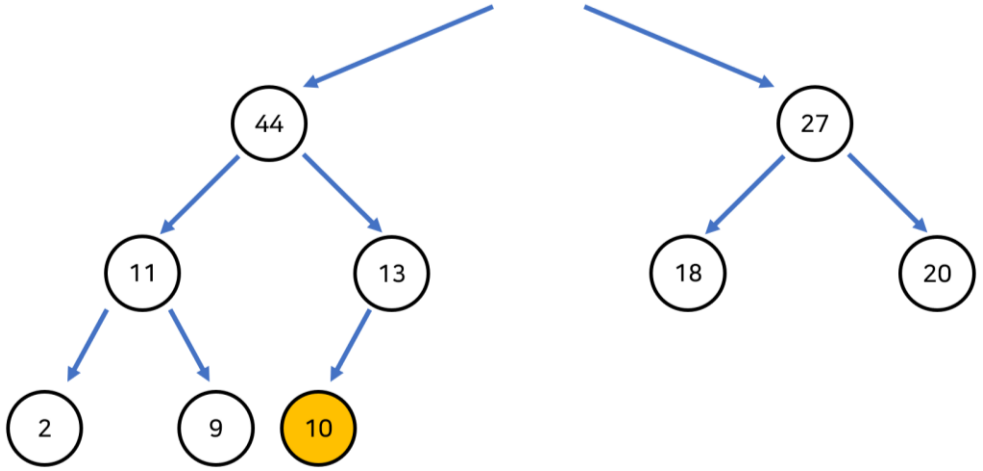


- 자료가 삭제되는 경우는 맨 위에 있는 Root노드가 빠지는 경우밖에 없다.
- 그렇게되면 다시 힙의 형태를 갖추어야...

- 삭제 알고리즘

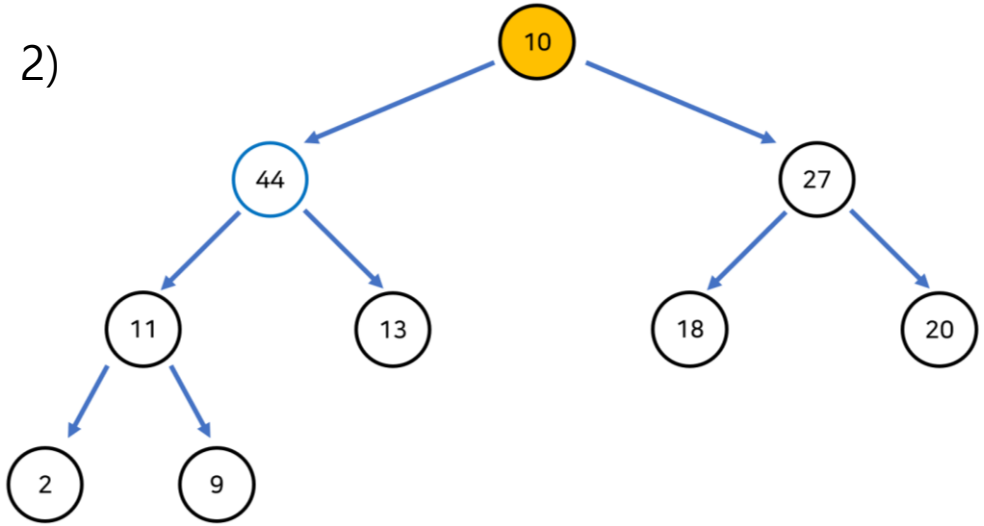
1) 맨 뒤에있는 노드를 Root자리로 옮긴다.
2) 자식 노드중 값이 더 큰 노드와 비교하여 자식 노드가 값이 더 크다면 위치를 바꾼다.
3) 2번의 작업을 자식노드보다 자신이 클때까지 반복한다.

1)



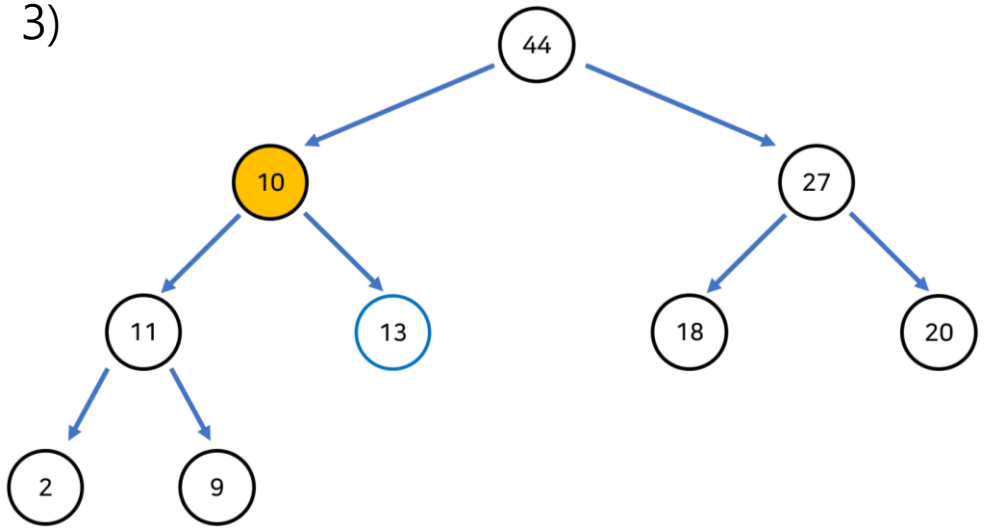
Root노드가 빠진 상황이다. 이렇게 되면 맨뒤에 있던 노드인 10을 Root로 올린다.

2)



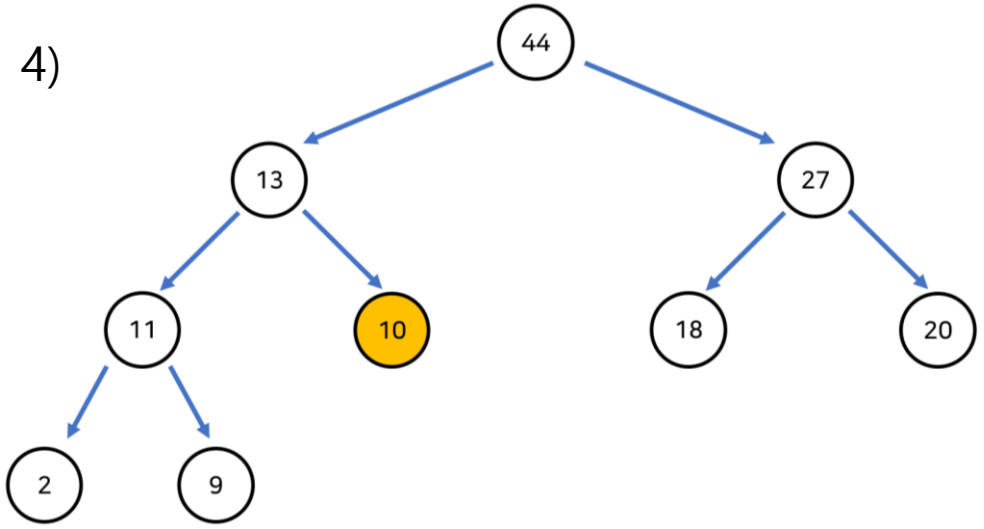
Root로 올리고 자식들과 비교를 시작하게 되는데, 자식 둘 중에 44가 더 크므로 44와 비교를 한다. 그런데 44가 10보다 크므로 44와 자리를 바꾼다.

3)



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

4)



그 결과 이렇게 힙의 구조를 다시 유지할수 있게 되었다.

priority_queue container

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

priority_queue container

- 우선순위 큐 테스트

```
#include <iostream>                    90 49 45 30 22 14 10
#include <queue>
using namespace std;                   max heap 으로 작동함.

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 << ' ‘;
   }
}
```

- min heap(최소값 우선)으로 하려면?

```
#include <iostream>                    10 14 22 30 45 49 90
#include <queue>
using namespace std;                   min heap 으로 작동함.

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

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

priority_queue container

우선순위 큐를 배웠으니 중등부 5번 창고를 풀러 가보자!

- operator< 오버라이딩 방법

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

using namespace std;

struct noco {
    int node;
    int cost;

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

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

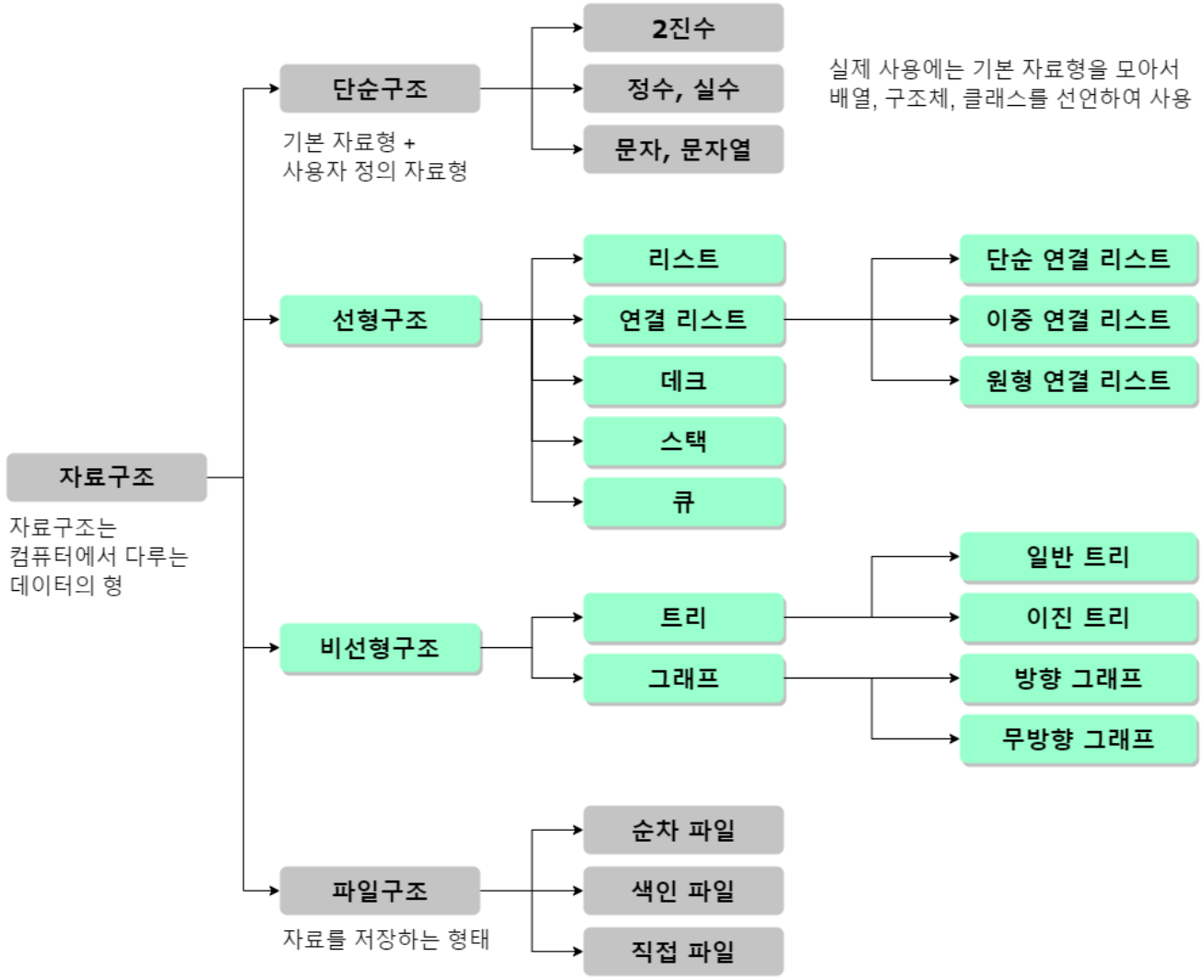
```
int main() {
    priority_queue <noco> pq;

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

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

자료구조

선형구조와 비선형구조

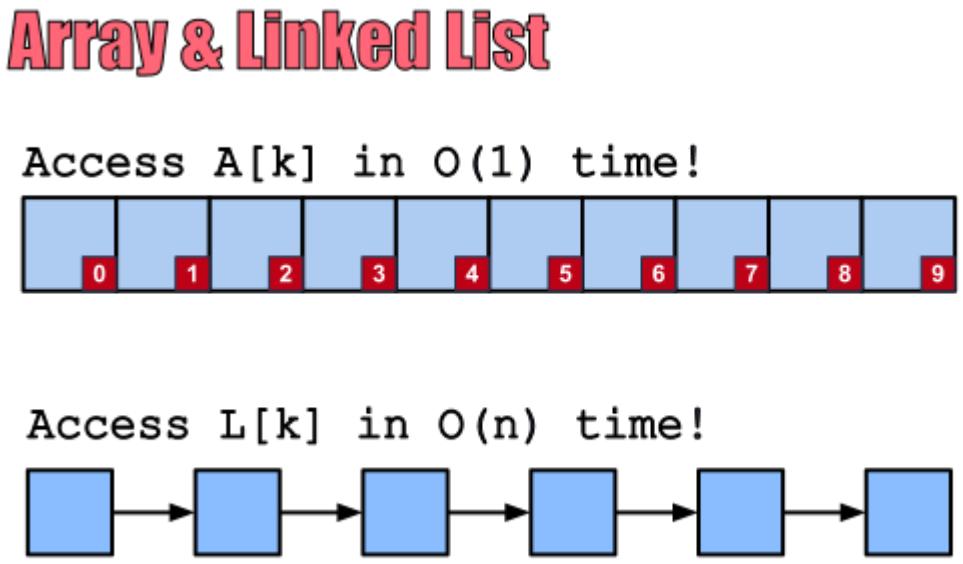
자료구조

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

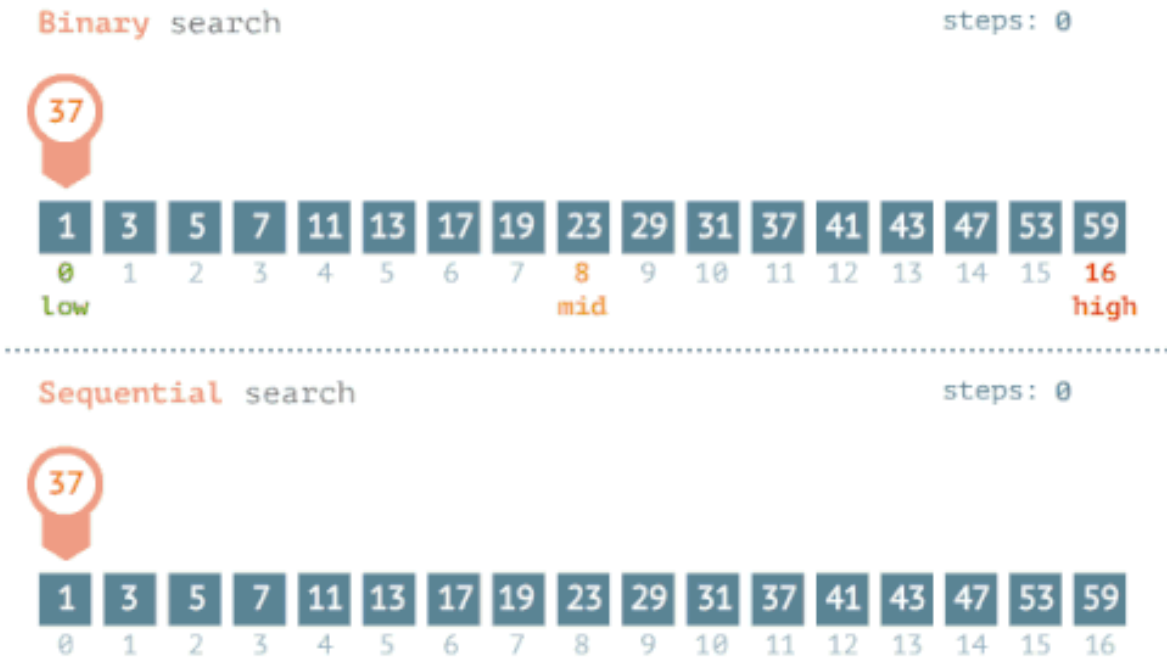


선형구조

- 선형구조란?
  - 자료를 구성하는 데이터를 순차적으로 나열시킨 형태를 의미

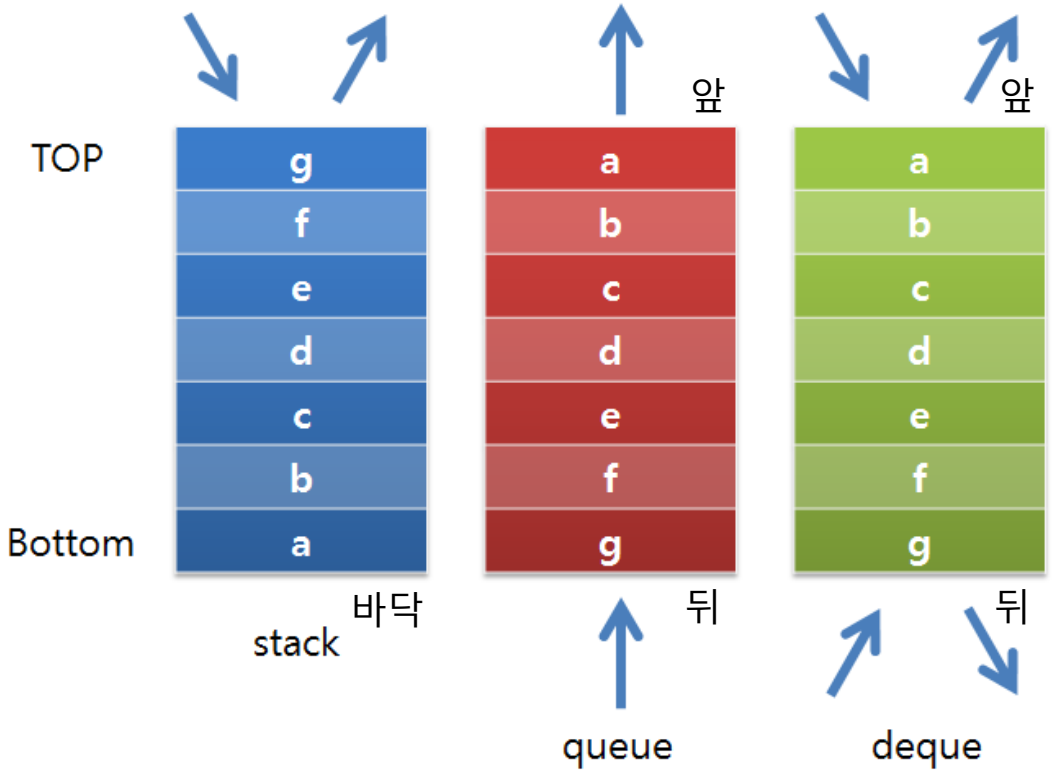


- 탐색법
  - 순차탐색
  - 이분탐색



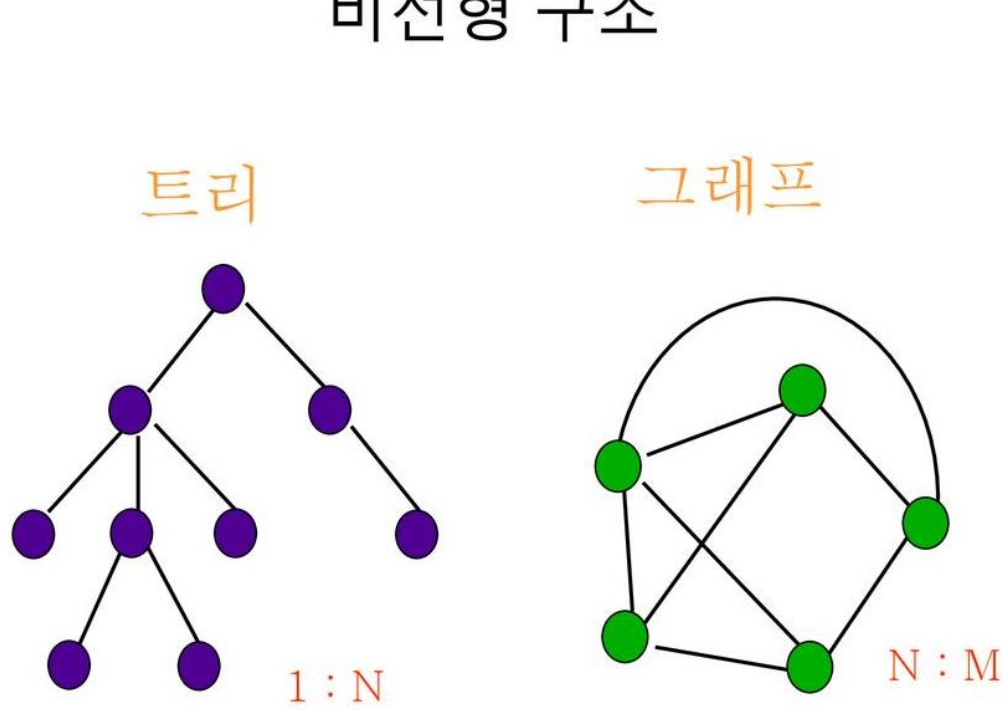
선형구조

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



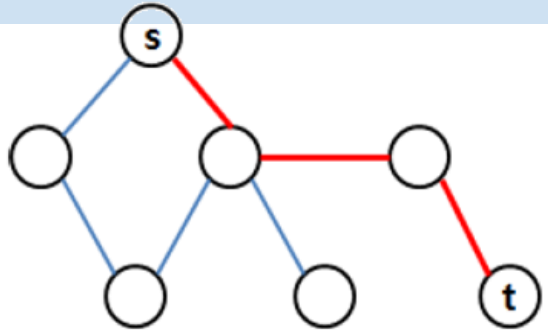
비선형구조

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

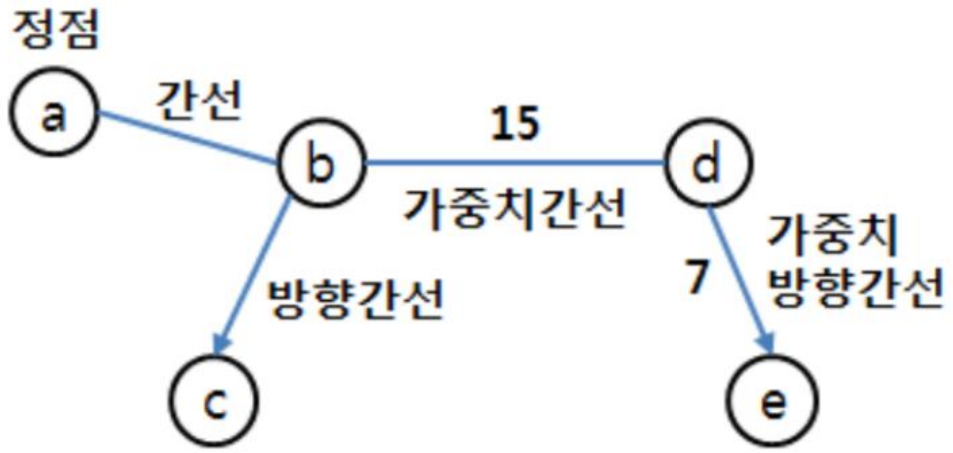


- 그래프 중에 회로가 없는 그래프를 트리라고 한다.

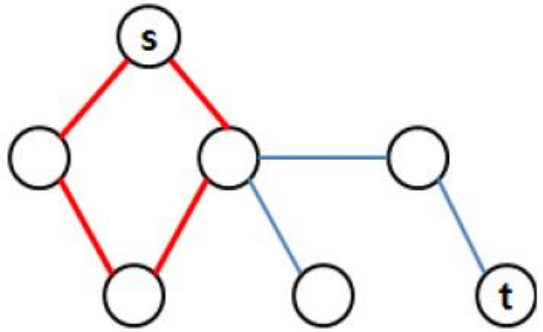
비선형구조 - 그래프



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

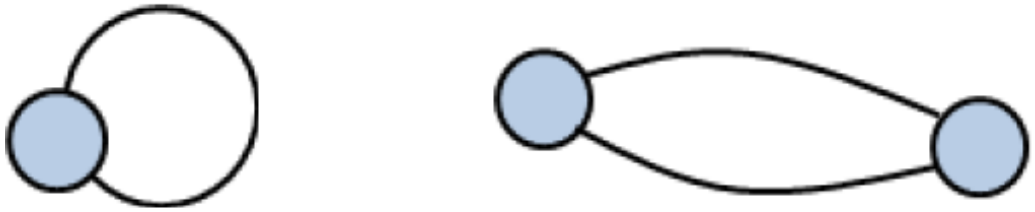


- 경로(path)
  - 임의의 정점 s에서 임의의 정점 t로 이동할 때, s에서 t로 이동하는데 사용한 정점들을 연결하고 있는 간선들의 순서로된 집합
- 회로(cycle)
  - 그래프에서 임의의 정점 s에서 같은 정점 s로의 경로들



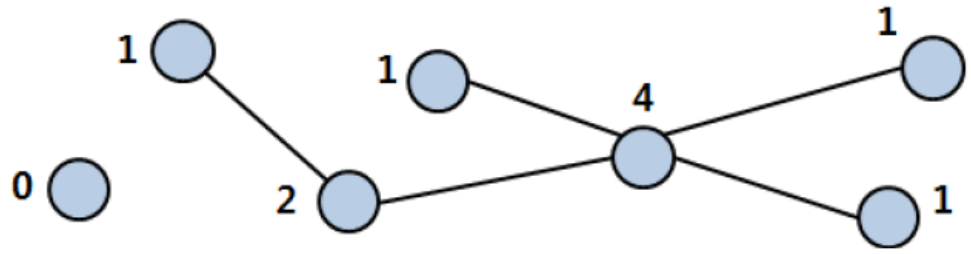
비선형구조 - 그래프

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



왼쪽은 자기간선 오른쪽은 다중간선을 나타낸다.

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



각 정점에서의 차수

비선형구조 - 그래프

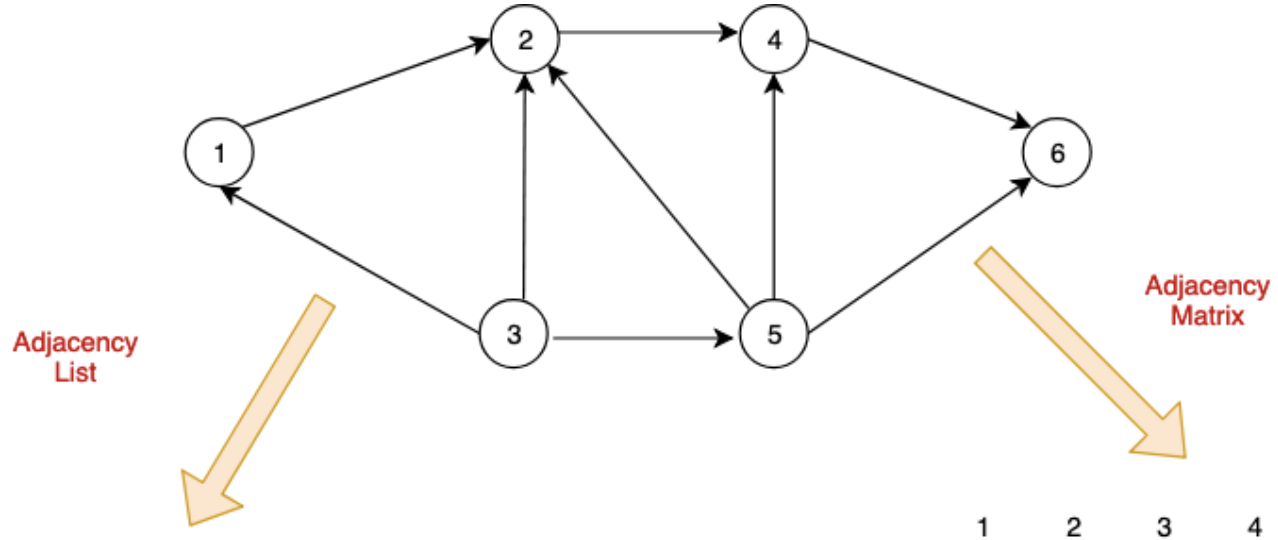
- 그래프의 자료 구현
  - 인접리스트

    (adjacency list)

  - 인접행렬

    (adjacency matrix)

  - 기타방법 …



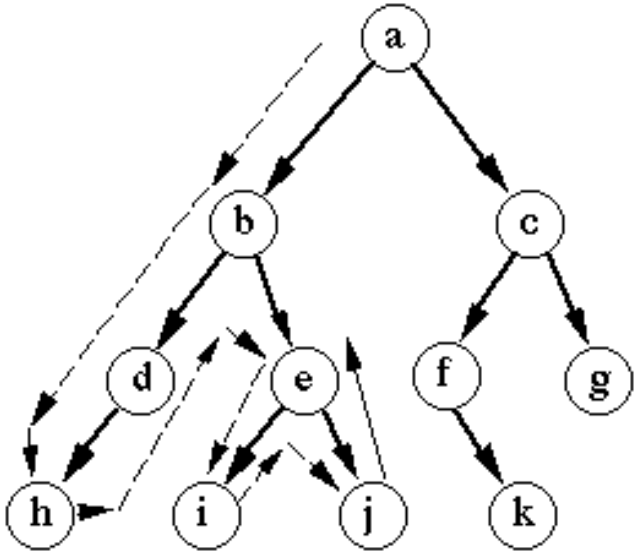
	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	0	1	0	0
3	1	1	0	0	1	0
4	0	0	0	0	0	1
5	0	1	0	1	0	1
6	0	0	0	0	0	0

비선형 탐색

비선형구조의 전체탐색(DFS vs BFS)

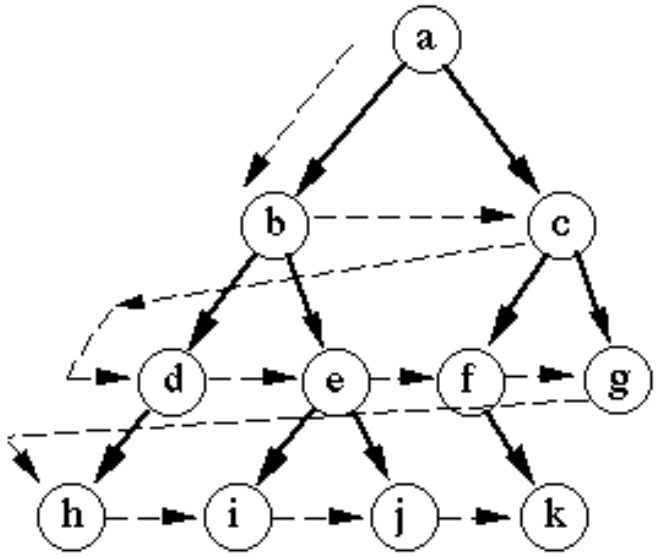
탐색

- 깊이우선탐색(DFS)



Depth-first search

- 너비우선탐색(BFS)



Breadth-first search

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

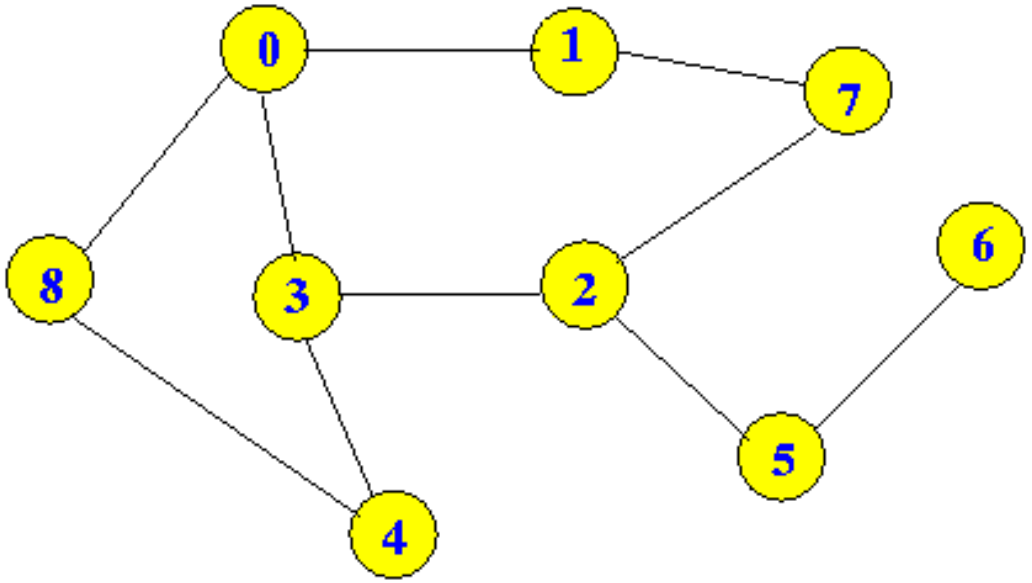
- 너비우선탐색(BFS)
  - 최대한 넓게 이동한 다음, 더 이상 갈 수 없을 때 아래로 이동
  - 루트 노드(혹은 다른 임의의 노드)에서 시작해서 인접한 노드를 먼저 탐색하는 방법
  - 시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회 방법

깊이우선탐색(DFS)

- 그래프의 순회

  트리와 달리 그래프는 사이클이 존재함

  방문정보를 유지하여 재방문을 막아야 함



- 깊이우선탐색 알고리즘

  : go deep(before going wide)

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

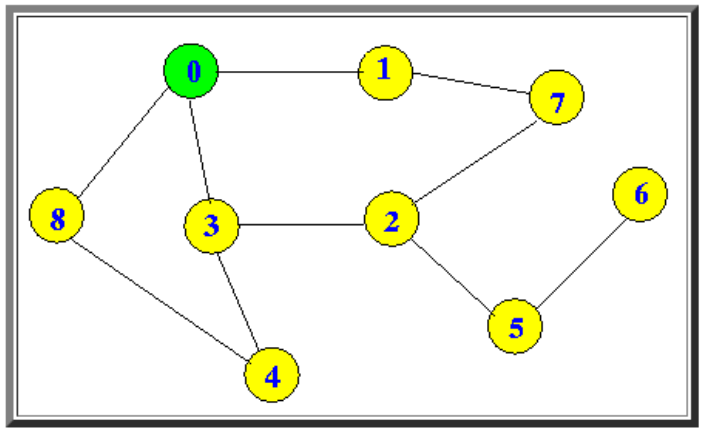
1) 정점 k를 처리하고 방문한 것으로 표시
2) 정점 k와 연결된 모든 정점에 대하여 방문한적이 없으면 그 정점에서 dfs, 완료되면 되돌아오기(백트랙)

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

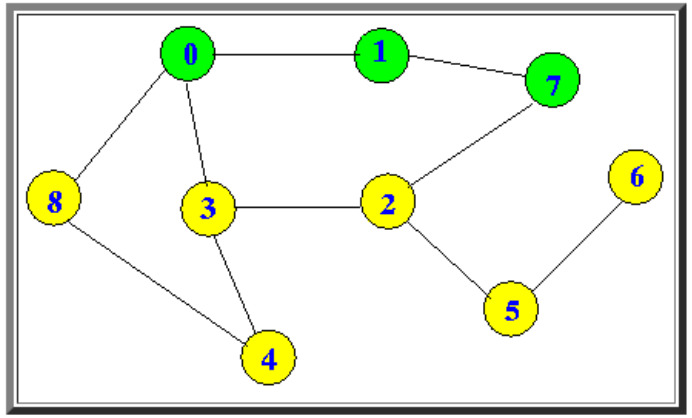
깊이우선탐색(DFS)

- 탐색순서

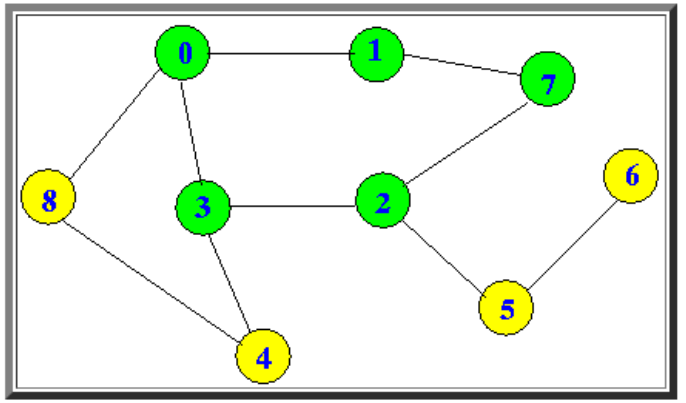
① dfs(0):



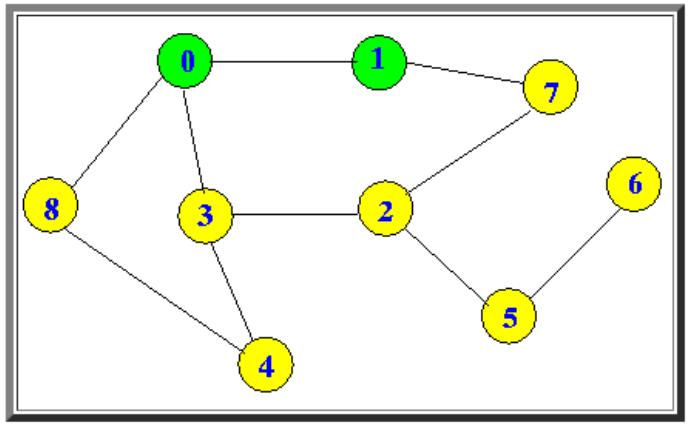
- ~~dfs(1) → dfs(0)~~ (because node 0 is "visited"); dfs(1) → dfs(7)



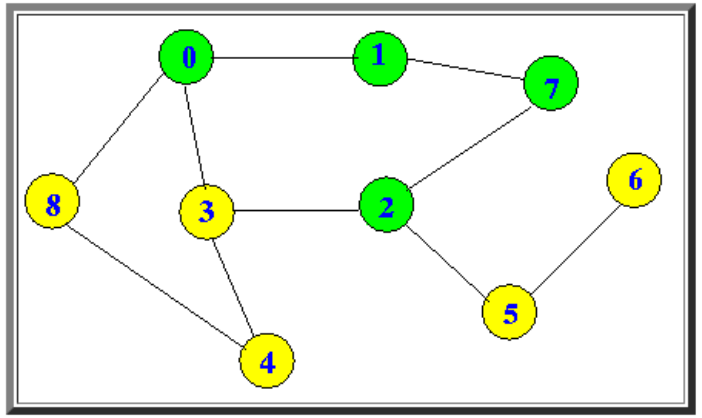
- dfs(2) → dfs(3)



② dfs(0) → dfs(1)



- ~~dfs(7) → dfs(1)~~; dfs(7) → dfs(2)



- ~~dfs(3) → dfs(0)~~; ~~dfs(3) → dfs(2)~~; dfs(3) → dfs(4)

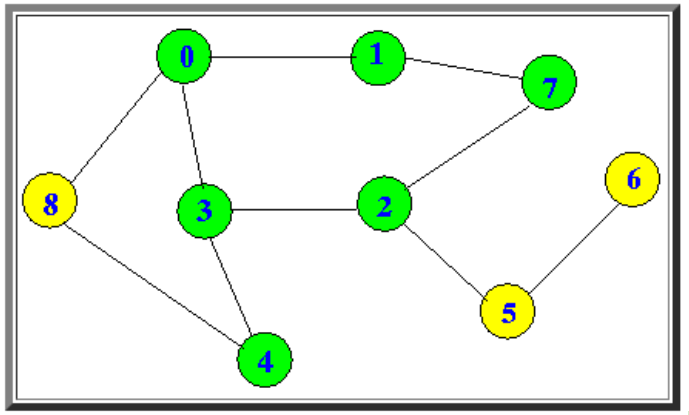
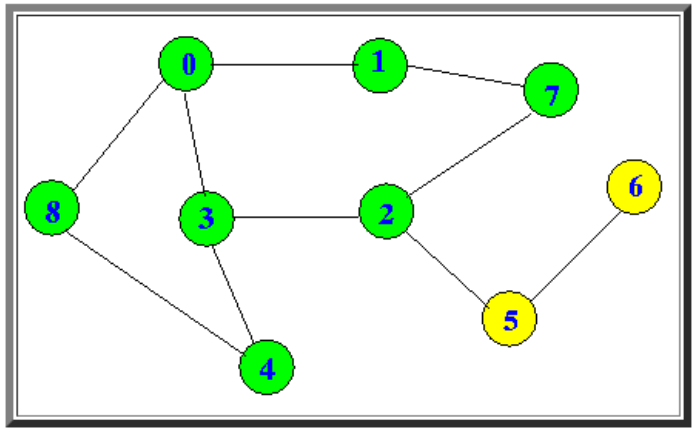


image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

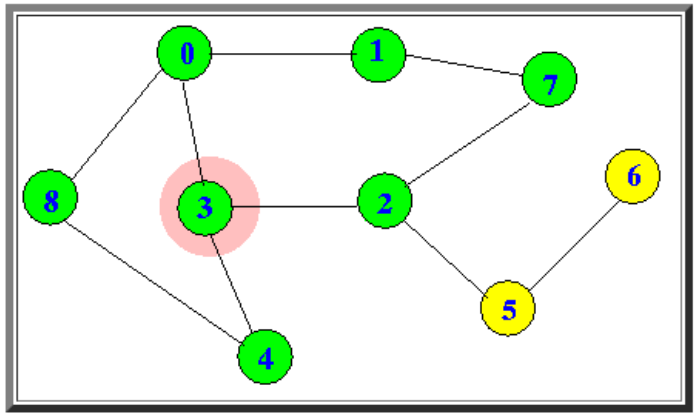
깊이우선탐색(DFS)

- 탐색순서

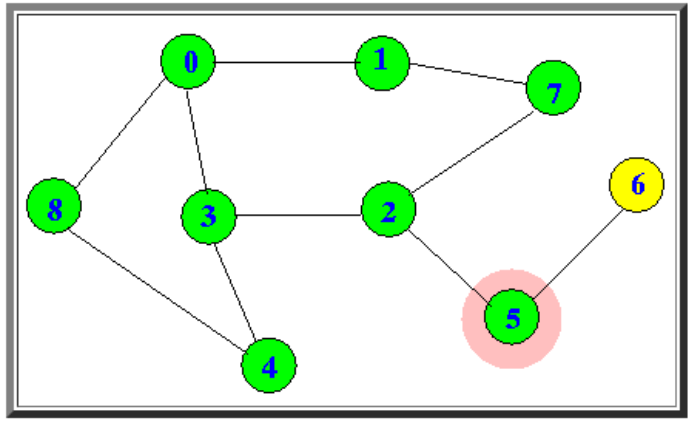
- ~~dfs(4) → dfs(3)~~; dfs(4) → dfs(8)



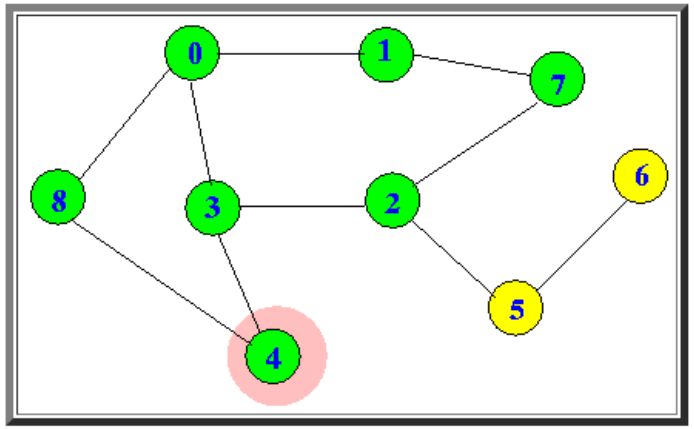
- return to dfs(3)



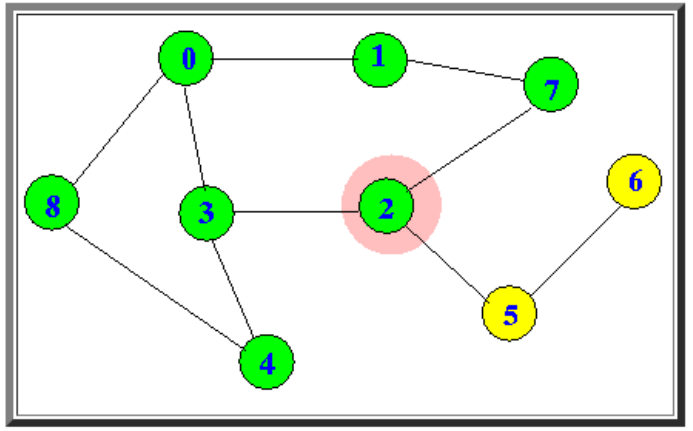
- ~~dfs(2) → dfs(3)~~; dfs(2) → dfs(5)



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



- return to dfs(2)



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

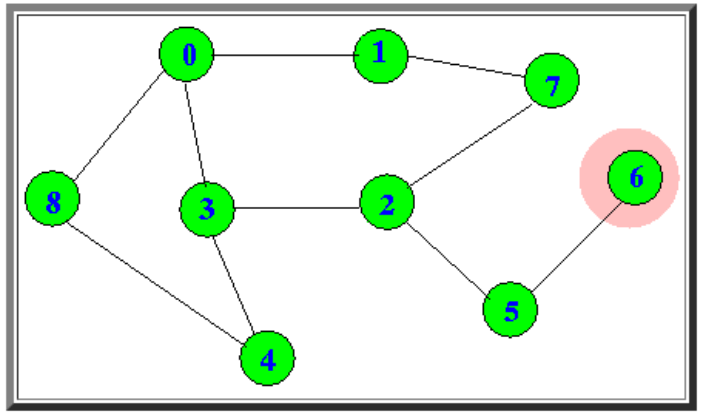
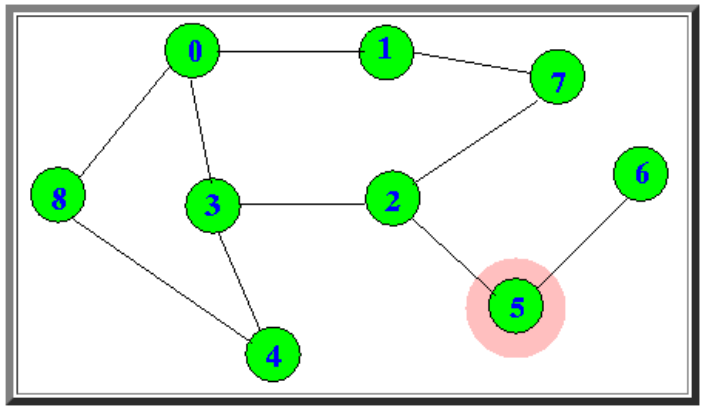


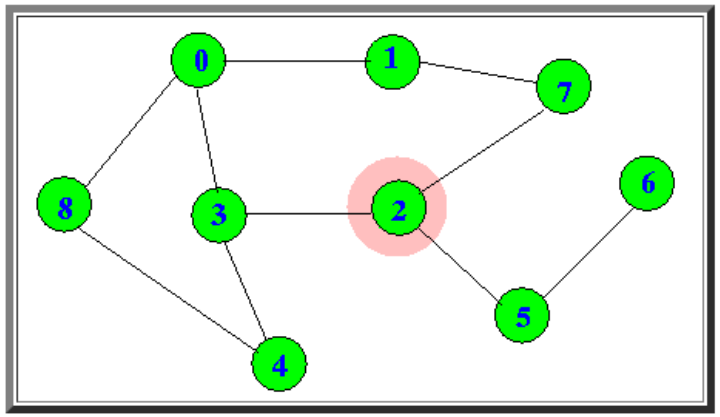
image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

깊이우선탐색(DFS)

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



  - return to dfs(2)



  - return to dfs(7)

  - return to dfs(1)

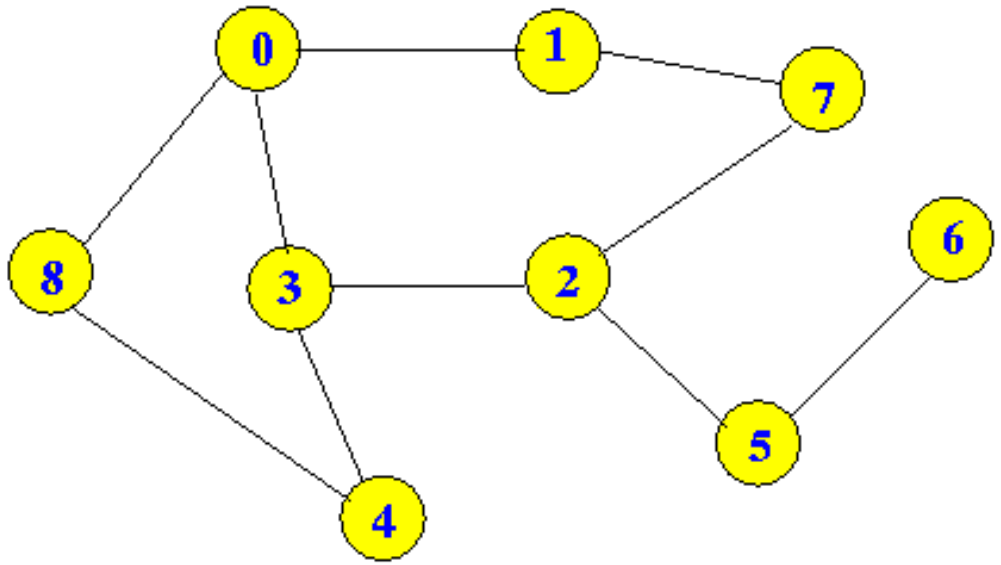
  - return to dfs(0)

    DONE

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

깊이우선탐색(DFS)

- vector를 인접리스트로 활용

그래프	데이터
	8 10 0 1 0 3 0 8 1 7 2 3 2 5 3 4 2 7 4 8 5 6


```
int n, m;
vector<vector<int>> G;
vector<bool> visited;

void input_G() {
   scanf("%d %d", &n, &m);
   // 정점이 0부터 시작하므로 n+1
   G.resize(n+1); // 공간확보
   visited.assign(n+1, 0);

   for(int i=0; i<m; i++) {
      int a, b;
      scanf("%d %d", &a, &b);
      // 양방향 연결이므로 a→b, a←b
      G[a].push_back(b);
      G[b].push_back(a);
   }
}

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

```
 idx  [0][1][2]
G[0]:  1  3  8
G[1]:  0  7
G[2]:  3  5  7
G[3]:  0  2  4
G[4]:  3  8
G[5]:  2  6
G[6]:  5
G[7]:  1  2
G[8]:  0  4

visited[0]: 0
visited[1]: 0
visited[2]: 0
visited[3]: 0
visited[4]: 0
visited[5]: 0
visited[6]: 0
visited[7]: 0
visited[8]: 0
```

깊이우선탐색(DFS)

- DFS 알고리즘 구현(인접리스트)
  - DFS 알고리즘

  def dfs(k):

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

dfs함수가 종료되면 호출 위치로 알아서 되돌아오므로 딱히 백트랙을 구현할 필요는 없음.

```
int n, m;
vector<vector<int>> G;
vector<bool> visited;

// 정점 k에서 dfs
void dfs(int k) {
  // 정점 k를 방문하였음을 출력
  printf("dfs(%d) started.\n", k);
  // 나중에 다시오지 않기 위해 방문을 표시하고,
  ?

  // 정점 k와 연결된 모든 정점에 대하여
  for (int i=0; i<G[k].size(); i++) {
    // k와 연결된 i번째 정점에 방문한 적 없으면,
    if (?) {
      // 그 정점에서 다시 dfs 시작
      ?
      // dfs 완료하고 돌아왔다고 메시지 출력
      printf("return to dfs(%d).\n", k);
    }
  }
  return; //연결된 모든 정점을 방문완료하여 되돌아가기
}
```

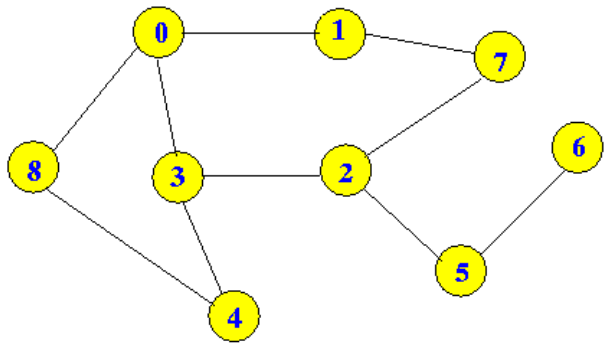
```
 idx  [0][1][2]
G[0]:  1  3  8
G[1]:  0  7
G[2]:  3  5  7
G[3]:  0  2  4
G[4]:  3  8
G[5]:  2  6
G[6]:  5
G[7]:  1  2
G[8]:  0  4

visited[0]: 0
visited[1]: 0
visited[2]: 0
visited[3]: 0
visited[4]: 0
visited[5]: 0
visited[6]: 0
visited[7]: 0
visited[8]: 0
```

깊이우선탐색(DFS) 전체 소스코드

dfs_full_src_quiz.cpp

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

void input_G() {
  scanf("%d %d", &n, &m);
  G.resize(n+1);  // 정점이 0부터 시작하므로 n+1
  visited.assign(n+1, 0);
  for(int i=0; i<m; i++) {
    int a, b;
    scanf("%d %d", &a, &b);
    G[a].push_back(b);
    G[b].push_back(a);
  }
}
void output_G() {
  printf("\n");
  for(int a=0; a<G.size(); a++) {
    printf("%2d:", a);
    for(int i : G[a])
      printf("%3d", i);
    printf("\n");
  }
  printf("\n");
}
```

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


  // 정점 k와 연결된 모든 정점에 대하여
  for (int i=0; i<          ; i++) {
    // 정점k의 i번째 정점에 방문한 적이 없으면,
    if (            ) {
      // 그 정점에서 다시 dfs 시작

      // dfs 완료하고 돌아왔다고 메시지 출력
      printf("return to dfs(%d).\n", k);
    }
  }
  return;
}

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




```
 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

dfs(0) started.
dfs(1) started.
dfs(7) started.
dfs(2) started.
dfs(3) started.
dfs(4) started.
dfs(8) started.
return to dfs(4).
return to dfs(3).
return to dfs(2).
dfs(5) started.
dfs(6) started.
return to dfs(5).
return to dfs(2).
return to dfs(7).
return to dfs(1).
return to dfs(0).
```

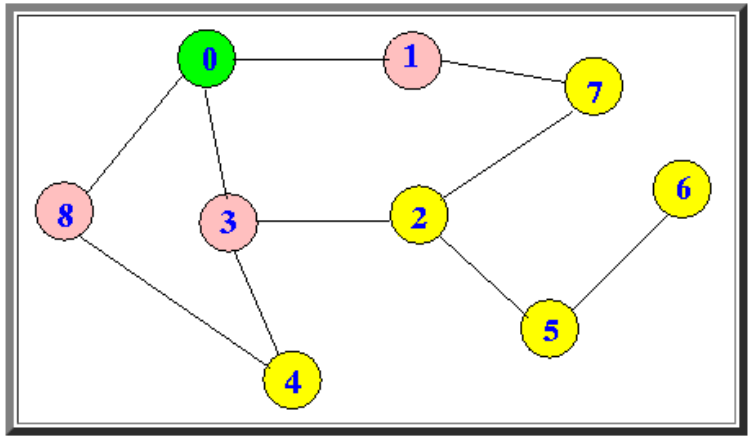
너비우선탐색(BFS)

- 너비우선탐색 알고리즘

  : Breadth First Search

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

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



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

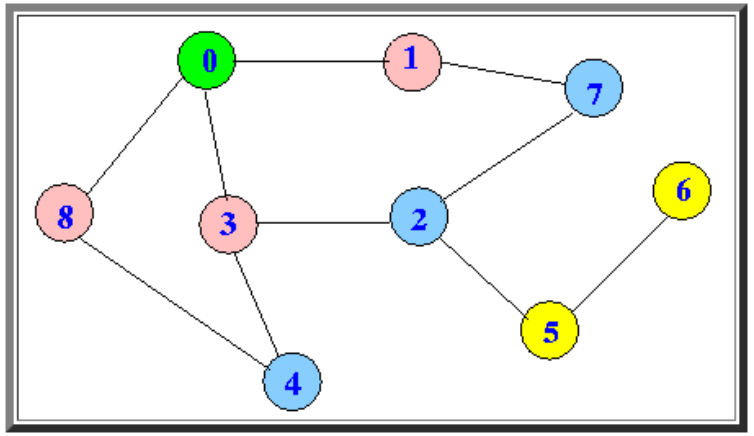
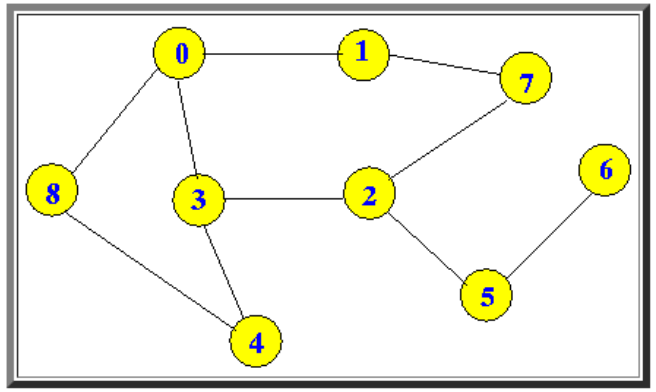


image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

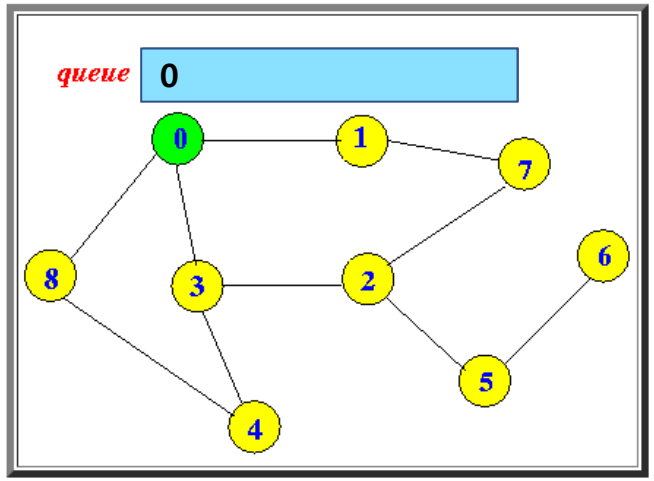
1. 정점 k를 처리하고 큐에 삽입, k를 방문한 것으로 표시
2. 큐가 빌 때까지 다음을 반복:
   1) 큐에서 첫 번째 항목 삭제
   2) 삭제된 항목과 이웃하는 모든 정점에 대하여 방문하지 않은 정점이라면,
      ① 그 정점을 처리하고 큐에 삽입
      ② 그 정점에 방문을 표시

너비우선탐색(BFS)

① Graph:

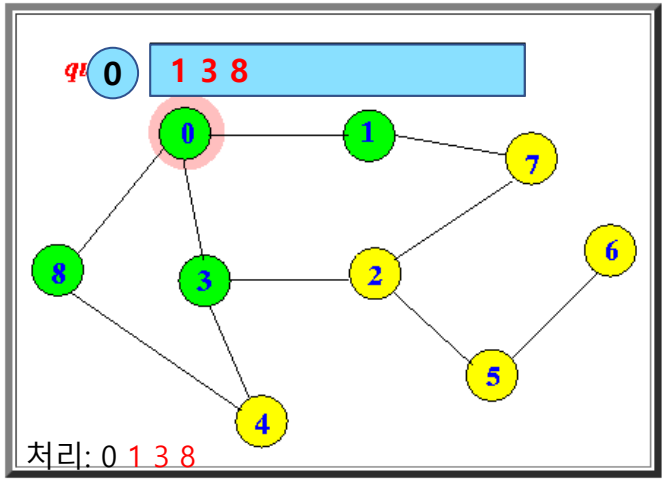


② Initial state:

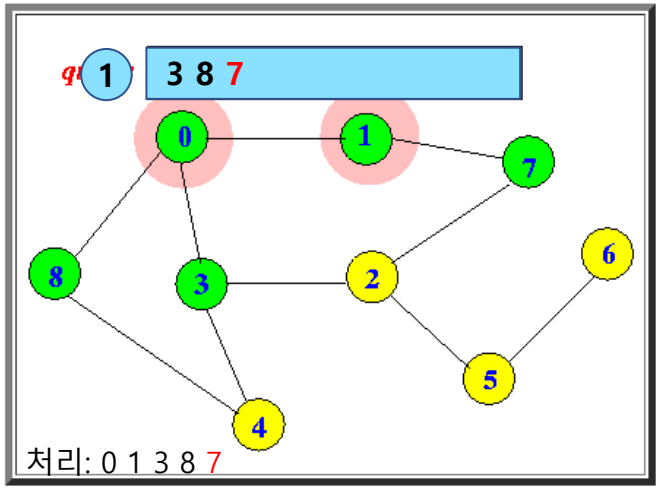


처리: 0

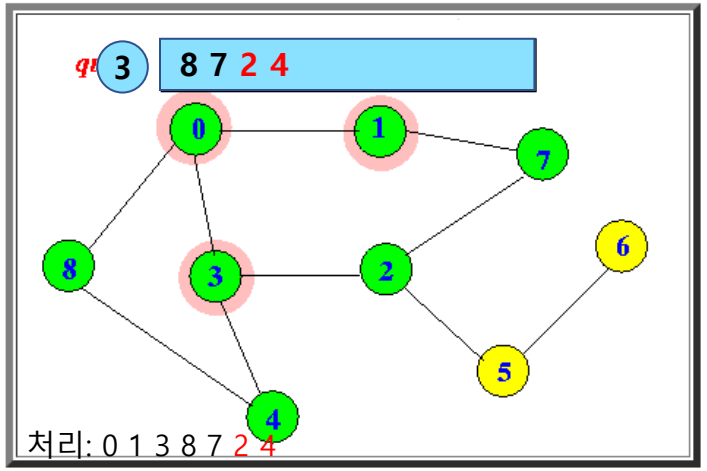
- State *after* processing 0



- State *after* processing 1



- State *after* processing 3



- State *after* processing 8

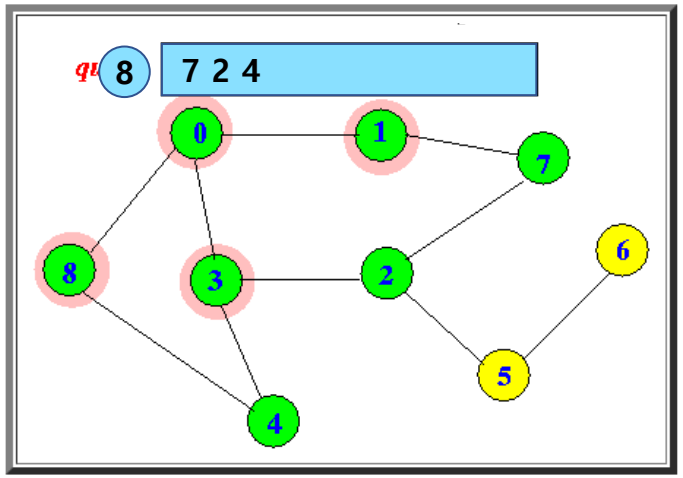
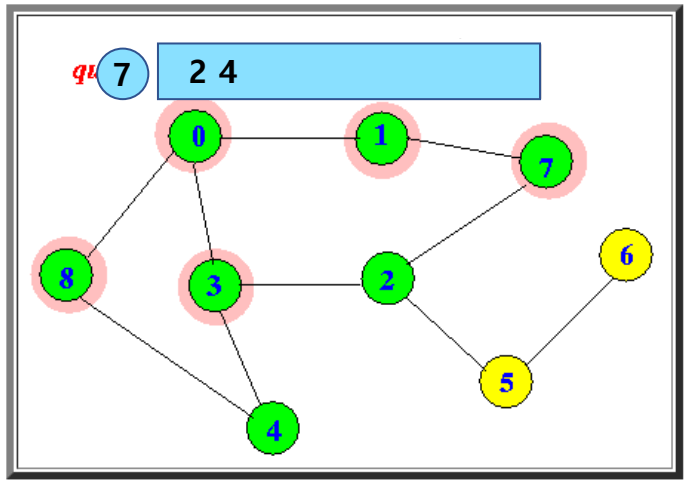


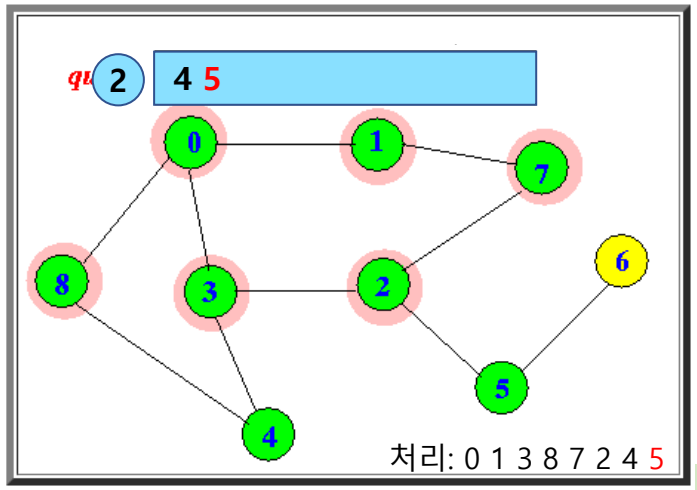
image source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/bfs.html

너비우선탐색(BFS)

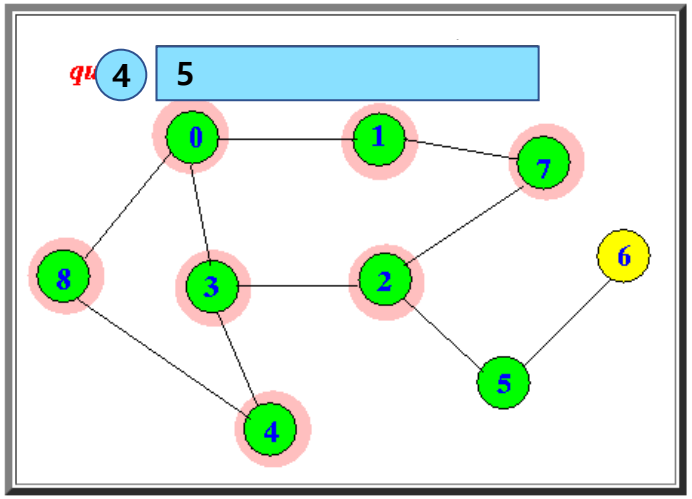
① State *after* processing 7



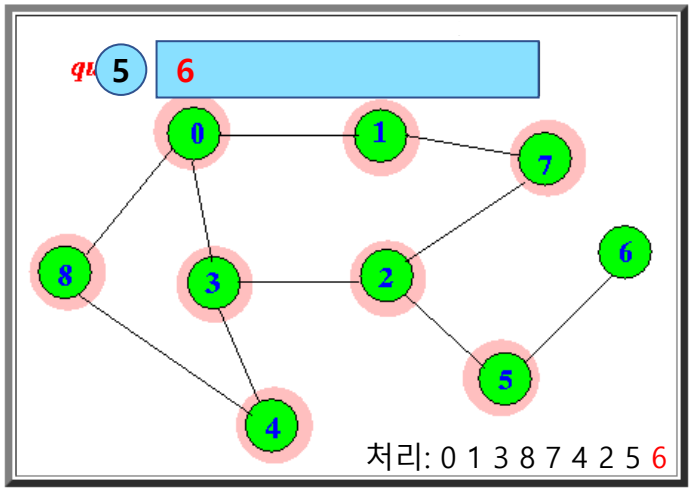
② State *after* processing 2



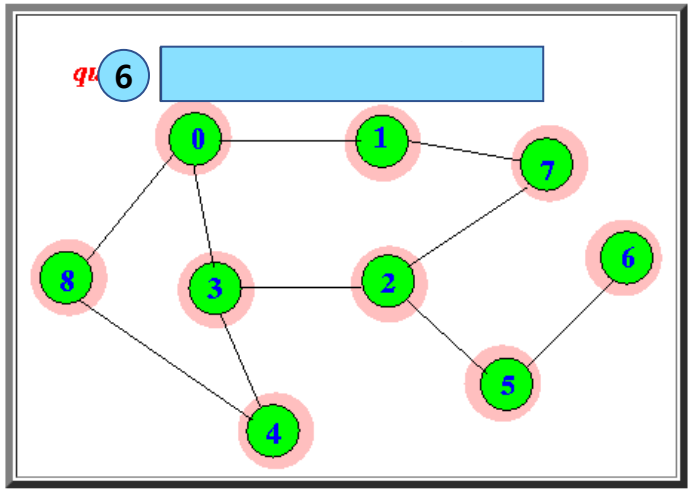
▪ State *after* processing 4



▪ State *after* processing 5



▪ State *after* processing 6



▪ DONE

(The queue has become empty)

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

너비우선탐

- 너비우선탐색 알고리즘

  : Breadth First Search

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

```
int n, m;
vector<vector<int>> G;
vector<bool> visited;

// 정점 k에서 bfs 시작
void bfs(int k) {
  queue<int> Q;   // 방금 방문한 이웃의 정점을 넣어 놓는 큐

  printf("bfs: (%d)\n", k);
  Q.push(k);          // 시작 정점을 Q에 삽입
  visited[k]=1;       // 시작 정점 방문했다고 표시

  while(     ?     ) {    // 큐에 내용물 있으면 계속반복
    int cur=Q.front();    // 큐의 첫번째 원소
    Q.pop();              // 빼냄(삭제)
    printf("%d deleted.\nbfs: ", cur);

    // 방금 전에 삭제한 정점과 이웃하는 모든 정점에 대하여
    for(int i=0; i<G[cur].size(); i++) {
      // 방문한 적이 없으면
      if(          ?          ) {
        printf("(%d) ", G[cur][i]);
            ?
            ?
      }
    }
    printf("\n");
  }
}
```

idx	[0]	[1]	[2]
G[0]:	1	3	8
G[1]:	0	7	
G[2]:	3	5	7
G[3]:	0	2	4
G[4]:	3	8	
G[5]:	2	6	
G[6]:	5		
G[7]:	1	2	
G[8]:	0	4	

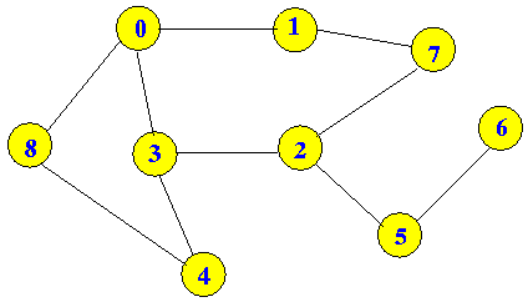
너비우선탐색(BFS) 전체 소스코드

bfs_full_src_quiz.cpp

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

int n, m;
vector<vector<int>> G;
vector<bool> visited;

// 현재 큐의 모습을 출력
void output_Q(queue<int> Q) {
  printf("Q:[");
  while(!Q.empty()) {
    int cur=Q.front();
    printf("%3d", cur);
    Q.pop();
  }
  printf("], ");
}
```

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

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

  printf("bfs: (%d)\n", k);
  Q.push(k);          // 시작 정점을 Q에 삽입
  visited[k]=1;       // 시작 정점 방문했다고 표시

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

    int cur=                // 큐의 첫번째 원소
    Q.    ();               // 빼냄(삭제)
    printf("%d deleted.\nbfs: ", cur);

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




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

void input_G() {
  scanf("%d %d", &n, &m);
  // 정점이 0부터 시작하므로 n+1
  G.resize(n+1);
  visited.assign(n+1, 0);

  for(int i=0; i<m; i++) {
    int a, b;
    scanf("%d %d", &a, &b);
    G[a].push_back(b);
    G[b].push_back(a);
  }
}

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

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

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

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.

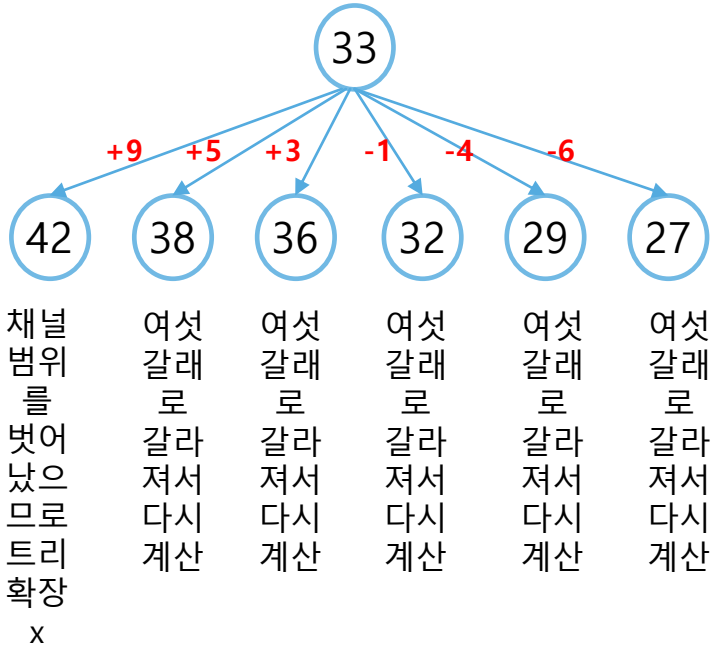
  두 번째 줄에 최소 버튼 조작 횟수로 이동하는 방법의 가짓수를 자연수로 출력한다.

- 입출력의 예

입력 예	출력 예
31 39	2 4
33 40	3 5

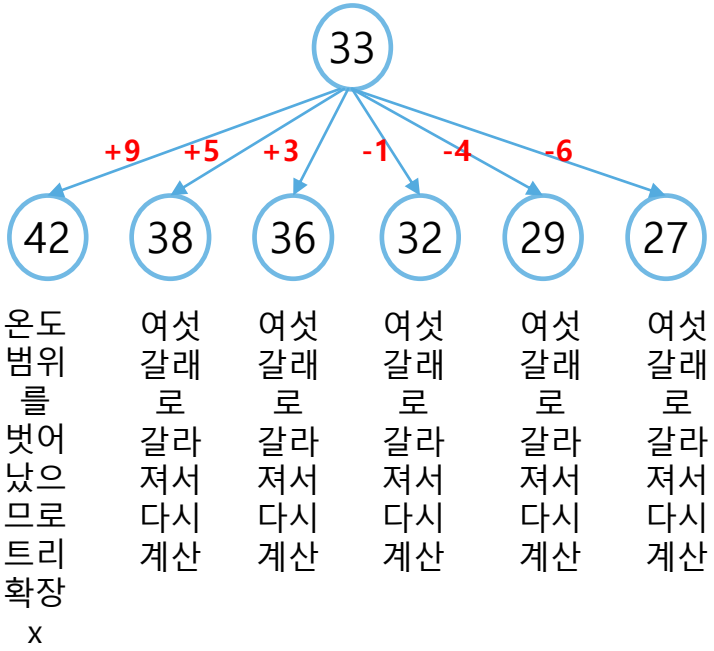
리모콘2

- 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

remocon_dfs_try1.cpp

```
// 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;
//리모컨 최다 조작 횟수를 최대 탐색 깊이로 설정
int res = MAX;
vector<int> v;

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

```
void f(int ch, int cnt) {
    if(cnt > MAX) {     // 최악 횟수를 넘어가면 그만,
        return;
    }

    if(ch < CH_LOW)  return;
    if(ch > CH_HIGH) return;

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

    f(ch+9, cnt+1);
    f(ch+5, cnt+1);
    f(ch+3, cnt+1);
    f(ch-6, cnt+1);
    f(ch-4, cnt+1);
    f(ch-1, cnt+1);
}
```

remocon_dfs_try2.cpp

DFS를 이용한 방법(확장

```
// 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;
//리모컨 최다 조작 횟수를 최대 탐색 깊이로 설정
int res = MAX;
vector<int> v;

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

```
void f(int ch, int cnt) {
    if(cnt > MAX) {     // 최악 횟수를 넘어가면 그만,
        return;
    }

    if(ch < CH_LOW)  return;
    if(ch > CH_HIGH) return;

    if(ch == b) {
        if(cnt <= res) { // 지금까지 찾아낸 횟수보다 작은 방식이면
            res = cnt;  // 결과에 현재 횟수를 기록
            // 답 찾아서 나가기 직전에,
            for(int x : v)  // 버튼 누른 순서 모두 출력
                printf("%3d,", x);
            printf("\b (%d)\n", res);  // 누른 횟수 출력
        }
        return;
    }

    v.push_back( 9); f(ch+9, cnt+1); v.pop_back();
    v.push_back( 5); f(ch+5, cnt+1); v.pop_back();
    v.push_back( 3); f(ch+3, cnt+1); v.pop_back();
    v.push_back(-6); f(ch-6, cnt+1); v.pop_back();
    v.push_back(-4); f(ch-4, cnt+1); v.pop_back();
    v.push_back(-1); f(ch-1, cnt+1); v.pop_back();
}
```

BFS를 이용한 방법

remocon_bfs_try1.cpp

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

#define  INT_MAX  0x7fffffff
#define  CH_LOW    1
#define  CH_HIGH  40

using namespace std;

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

int main(void)  {           // BFS를 이용한 풀이
    int min_cnt=INT_MAX;    // 최소 버튼조작 횟수

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

    queue<node> Q;
    Q.push({a, 0});

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

        if(n.ch < CH_LOW)  continue;
        if(n.ch > CH_HIGH) continue;

        if(n.ch == b) {  // 목표 채널에 도달하면
            min_cnt = n.cnt;
            break;
        }

        Q.push({n.ch+9, n.cnt+1});
        Q.push({n.ch+5, n.cnt+1});
        Q.push({n.ch+3, n.cnt+1});
        Q.push({n.ch-1, n.cnt+1});
        Q.push({n.ch-4, n.cnt+1});
        Q.push({n.ch-6, n.cnt+1});
    }

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

```cpp
#include <stdio.h>
#include <vector>
#include <queue>

#define  INT_MAX  0x7fffffff
#define  CH_LOW    1
#define  CH_HIGH  40

using namespace std;

typedef struct node {
    int ch;     // channel
    int cnt;    // count
    vector<int> btns;   // buttons
};

// 버튼조작 히스토리 btns에 btn을 추가한 벡터를 반환
vector<int> add_btn(vector<int>& btns, int btn) {
    vector<int> vt;
    vt = btns;
    vt.push_back(btn);
    return vt;
}

int main(void)  {               // BFS를 이용한 풀이
    int min_cnt=INT_MAX;        // 최소 버튼조작 횟수
    int methods=0;              // 목표채널에 도달하는 방법의 수

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

    queue<node> Q;
    Q.push({a, 0, vector<int>()});

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

        // 지금 계산 중인 조작횟수가 최소 조작횟수를 넘기면 스톱
        if(n.cnt > min_cnt)
            break;

        if(n.ch < CH_LOW)  continue;
        if(n.ch > CH_HIGH) continue;

        if(n.ch == b) {  // 목표 채널에 도달하면
            min_cnt = n.cnt;
            methods++;
            /*printf("[%d]: ", min_cnt);

            for(int x: n.btns)
                printf("%3d", x);
            puts("");*/
        }

        Q.push({n.ch+9, n.cnt+1, add_btn(n.btns, 9)});
        Q.push({n.ch+5, n.cnt+1, add_btn(n.btns, 5)});
        Q.push({n.ch+3, n.cnt+1, add_btn(n.btns, 3)});
        Q.push({n.ch-1, n.cnt+1, add_btn(n.btns,-1)});
        Q.push({n.ch-4, n.cnt+1, add_btn(n.btns,-4)});
        Q.push({n.ch-6, n.cnt+1, add_btn(n.btns,-6)});
    }

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

문제: 공성전

- 문제

  흑우나라는 좋은 관계를 유지하기 위해 강대국인 올림피아드나라에 매년 공물을 바치고 있었다. 그런데 갑자기 앞으로는 매년 보내던 공물의 2배를 바치라는 것이었다. 이러한 요구를 받아들일 수 없다고 생각한 흑우나라는 올림피아드나라의 성들을 공격하기로 했다.

  올림피아드 나라의 성들은 높은 성벽 때문에 일반적인 공성전으로는 이길 가능성이 매우 낮았다. 그리하여 특별부대가 야간에 몰래 성벽을 넘어 성문을 열어주면 전 병력이 공격하는 작전을 세웠다.

  성벽을 넘기 위해서는 성벽 높이 H ($1 \le H \le 1000$) 이상의 사다리가 필요하다. 흑우나라는 그렇게 긴 사다리가 없으므로 조립식 사다리 N ($1 \le N \le 20$)개를 준비하였는데 각 사다리의 길이는 Li ($1 \le L_i \le 1{,}000$)의 길이를 가진다.

  주어진 성벽 높이보다 조립식 사다리를 연결한 길이가 짧으면 성벽을 넘을 수 없으나 연결한 사다리의 높이가 성벽 높이 이상일 때는 성벽을 넘을 수 있다. 성벽의 높이와 조립한 사다리 길이 차이가 최소가 되는 길이가 얼마인지 구하시오.

  단, 모든 경우에서 조립한 사다리의 길이는 성벽 높이 이상으로 성벽을 무조건 넘을 수 있다고 가정한다.

- 출처: 2023 충북정올 학교예선 중등 3번 문제

문제: 공성전

- 입력형식

  첫 번째 줄에는 자연수 N, H가 주어진다. ($1 \le N \le 20, 1 \le H \le 1000$)

  다음 N개의 줄에는 L_i ($1 \le L_i \le 1{,}000$) 가 각 줄마다 주어진다.

- 출력형식

  성벽의 높이와 조립한 사다리 길이 차이가 최소가 되는 길이를 자연수 형태로 출력하시오.

  단, 모든 경우에서 조립한 사다리의 길이는 성벽 높이 이상으로 성벽을 무조건 넘을 수 있다고 가정한다.

- 입력과 출력의 예

입력 예1	출력 예1
3 3 1 2 3	0

입력 예2	출력 예2
5 5 999 999 999 999 999	994

- 출처: 2023 충북정올 학교예선 중등 3번 문제

문제: 공성전

siege_dfs_try1_c.cpp

```
// 100점 나오는 답안이지만 이렇게 짜면 안됨,

#include <iostream>
using namespace std;

int ladder[20];
int n, h;
int ans = 1e9;

void solve(int v, int len);

int main() {
    cin >> n >> h;

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

    solve(0, 0);
    cout << ans-h;
    return 0;
}
```

```
void solve(int v, int len) {
    len += ladder[v];
    if(len == h) {  // 사다리 조합이 성벽 높이와 일치
        ans = len;
        return;     // 더 이상 좋은 답이 없으므로 탐색 중단
    }
    else if(len > h) { // 사다리 조합이 성벽을 넘어섬
        ans = min(ans, len);  // 이 답이 최상의 답이면 기록
        if(v<n-1) {
            // 이미 현재 길이가 성벽높이를 넘어가므로 지금 사다리를
            // 포함하는 탐색은 의미 없음, 지금 사다리 빼고 탐색
            solve(v+1, len-ladder[v]);
        }
    }
    else { //사다리 조합이 아직 성벽높이에 도달 못함
        if(v<n-1) {
            //지금 사다리를 포함하여 다른 조합으로 탐색
            solve(v+1, len); // 만약 이 호출에서 일치 답을 찾는다면?
            //지금 사다리 빼고 다른 조합으로 탐색
            solve(v+1, len-ladder[v]);
        }
    }
}
```

문제: 공성전

```
// 이것이 정확한 답안

#include <iostream>
using namespace std;

int ladder[20];
int n, h;
int ans = 1e9;

bool solve(int v, int len);

int main() {
    cin >> n >> h;

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

    solve(0, 0);
    cout << ans-h;
    return 0;
}
```

```
bool solve(int v, int len) { // 정확히 원하는 답을 찾으면 참을 리턴
    len += ladder[v];
    if(len == h) {  // 사다리 조합이 성벽 높이와 일치
        ans = len;
        return true;     // 더 이상 좋은 답이 없으므로 탐색 중단
    }
    else if(len > h) { // 사다리 조합이 성벽을 넘어섬
        ans = min(ans, len);  // 이 답이 최상의 답이면 기록
        if(v<n-1) {
            // 이미 현재 길이가 성벽높이를 넘어가므로 지금 사다리를
            // 포함하는 탐색은 의미 없음, 지금 사다리 빼고 탐색
            return solve(v+1, len-ladder[v]);
        }
    }
    else { //사다리 조합이 아직 성벽높이에 도달 못함
        if(v<n-1) {
            bool success = solve(v+1, len);
            if(success) return true;  // 찾았으면 true반환
            else { // 아직 못찾았으면 다른 방법으로 시도
                return solve(v+1, len-ladder[v]);
            }
        }
    }
    return false;
}
```

풀이: 공성전 (탐색과정 살펴보기)

siege_dfs_try3_c.cpp

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

int ladder[20];
int n, h;
int ans = 1e9;
deque<int> seq;
bool solve(int v, int len);

int main() {
    cin >> n >> h;

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

    solve(0, 0);
    cout << ans-h;
    return 0;
}

void output_combi(deque<int>& seq, int len)
{
    printf("combi: ");
    for(int a : seq)
        printf("%d-", a);
    printf("\b (%d)\n", len);
}
```

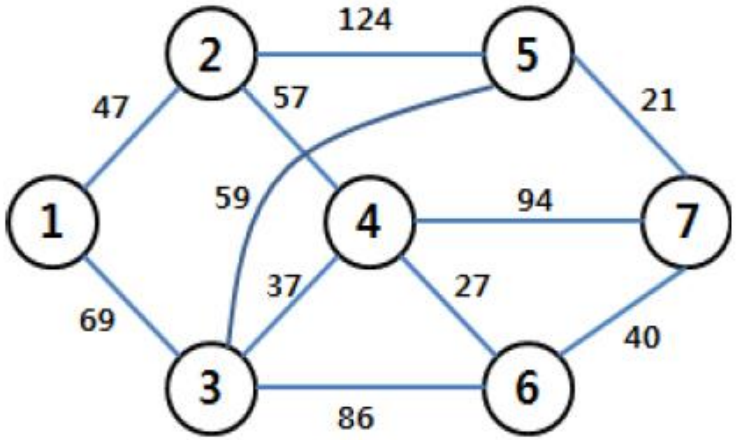
```
bool solve(int v, int len) {
    len += ladder[v];
    seq.push_back(ladder[v]);
    output_combi(seq, len);

    if(len == h) {  // 사다리 조합이 성벽 높이와 일치
        ans = len;
        return true;     // 더이상 좋은 답이 없으므로 탐색 중단
    }
    else if(len > h) { //사다리 조합이 성벽을 넘어섬
        ans = min(ans, len);  // 이 답이 최상의 답이면 기록
        if(v<n-1) {
            seq.pop_back();
            return solve(v+1, len-ladder[v]);
        }
    }
    else { //사다리 조합이 아직 성벽높이에 도달 못함
        if(v<n-1) {
            bool r = solve(v+1, len);
            if(r==false) {
                seq.pop_back();
                return solve(v+1, len-ladder[v]);
            }
            else return r;
        }
    }
    seq.pop_back();
    return false;
}
```

```
4 11
1 2 5 6
combi: 1 (1)
combi: 1-2 (3)
combi: 1-2-5 (8)
combi: 1-2-5-6 (14)
combi: 1-2-6 (9)
combi: 1-5 (6)
combi: 1-5-6 (12)
combi: 1-6 (7)
combi: 2 (2)
combi: 2-5 (7)
combi: 2-5-6 (13)
combi: 2-6 (8)
combi: 5 (5)
combi: 5-6 (11)
0
```

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 <iostream>
#include <deque>
#define MAX_INT 0x7fffffff
#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 chk[MAX_V+1];  // 방문여부를 저장하는 배열
int sol=MAX_INT;
deque<int> path;   // 방문 정점 순서를 기록하는 데크
void solve(int V, int W);

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

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

}

int main() {
  input();
  solve(1, 0);  // 정점 1, 거리 0에서 시작
  printf("\nmin: %d\n", sol==MAX_INT ? -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 solve(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 <iostream>
#include <deque>
#define MAX_INT 0x7fffffff
#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=MAX_INT;
deque<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 i=0; i<path.size(); i++)
      printf("%d-", path[i]);
    printf("\b : [%d]\n", W);
}
```

```
// 정점 V에서 목적지까지 가는 거리를 계산해라,
// 현재까지 이동거리는 W이다.
void solve(int V, int W) {
  path.push_back(V);
  chk[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(!chk[i] && G[V][i])  {
      solve(i, G[V][i]+W);
      path.pop_back();
      chk[i] = 0;
    }
  }
}

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

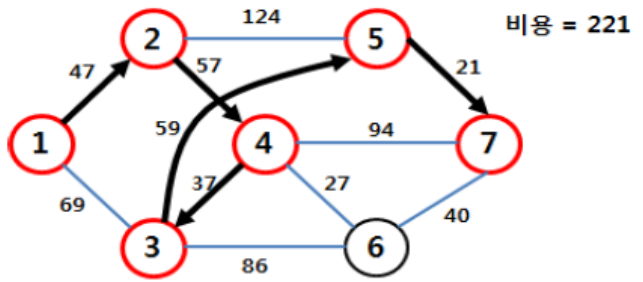
```
1-2-4-3-5-7 : [221]
1-2-4-3-6-7 : [267]
1-2-4-6-3-5-7 : [297]
1-2-4-6-7 : [171]
1-2-4-7 : [198]
1-2-5-3-4-6-7 : [334]
1-2-5-3-4-7 : [361]
1-2-5-3-6-4-7 : [437]
1-2-5-3-6-7 : [356]
1-2-5-7 : [192]
1-3-4-2-5-7 : [308]
1-3-4-6-7 : [173]
1-3-4-7 : [200]
1-3-5-2-4-6-7 : [376]
1-3-5-2-4-7 : [403]
1-3-5-7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

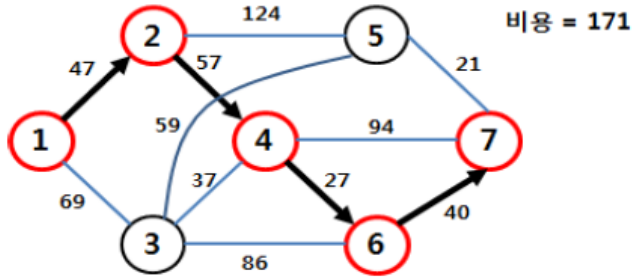
    현재 알아낸 최소 거리 >
    지금까지 구한 경로의 거리

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



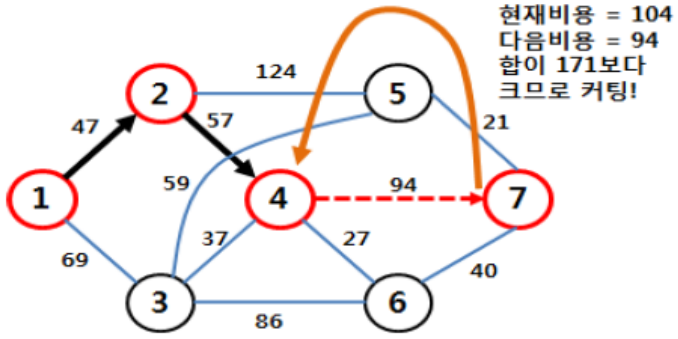
처음으로 찾게 되는 경로, 이 경로의 길이는 221이다.

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 가는 길

- 탐색의 배제
  - 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

```
// 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;
    }
  }
}
```

```
// 탐색배제 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 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

- 탐색 횟수 비교 결과

알고리즘	입력 1	입력 2	입력 3	입력 4
전체탐색	12	47	153	176
탐색배제	10	16	49	39

- 배제된 공간 비율

알고리즘	입력 1	입력 2	입력 3	입력 4
탐색배제	12.0%	66.0%	69.9%	77.8%

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.

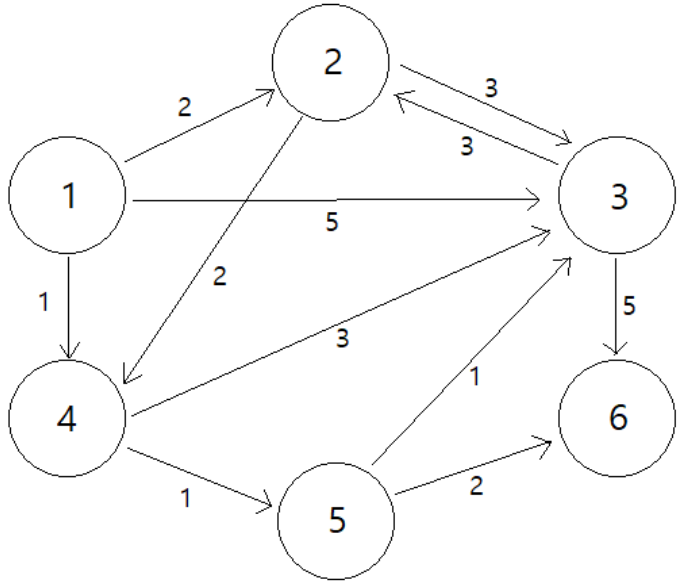
- 입력과 출력의 예

입력 예2	출력 예2
3 4 sooo ooxo ooog	4

- 출처: 2023 전국상업경진대회 비즈니스프로그래밍 부문 시도예선 8번

다익스트라(dijkstra) 알고리즘

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



  - 음의 간선이 없을 때 정상 작동
  - 네비게이션소프트웨어의 기본 알고리즘

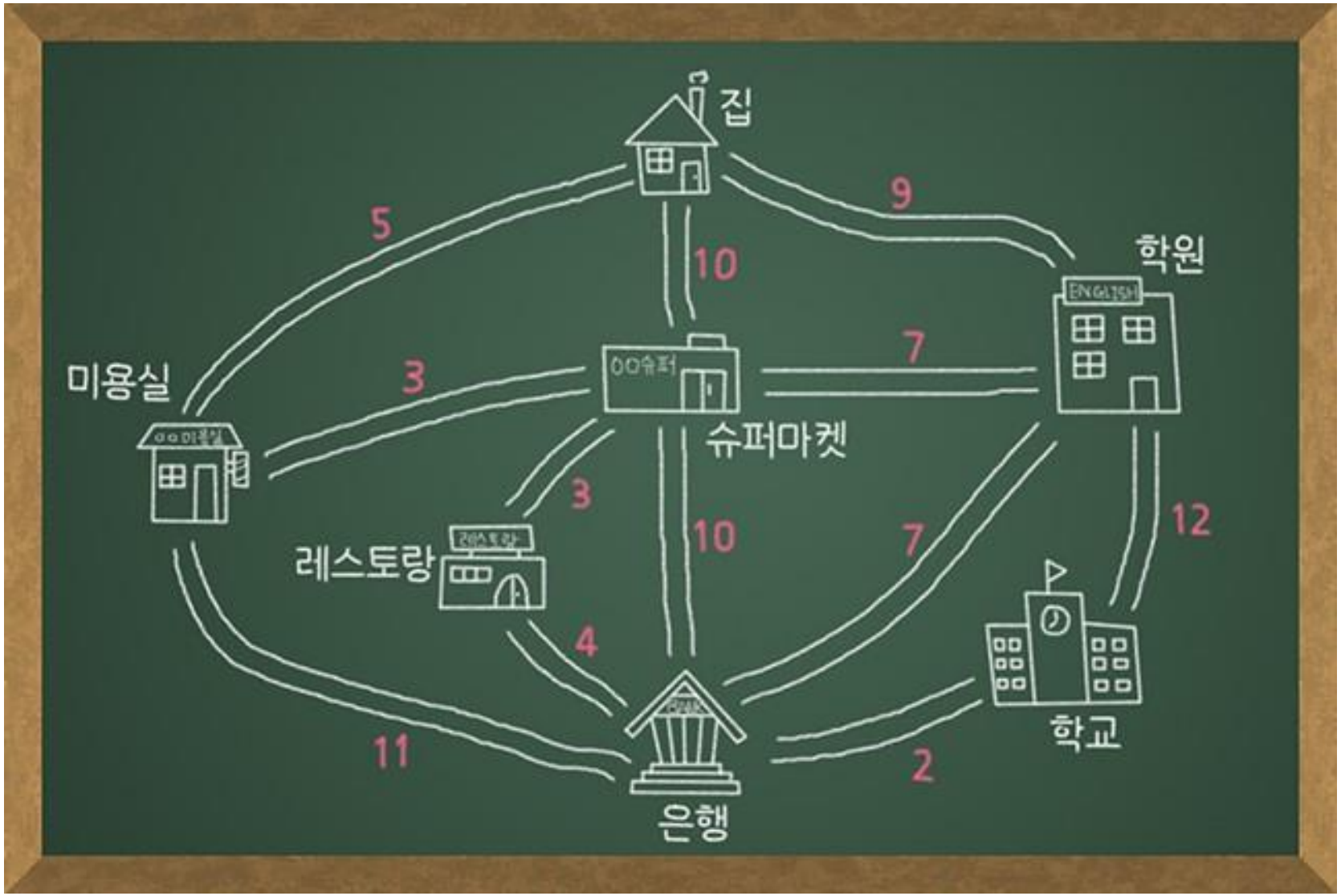
- EBS 해설 동영상
  - https://youtu.be/tZu4x5825LI

- 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

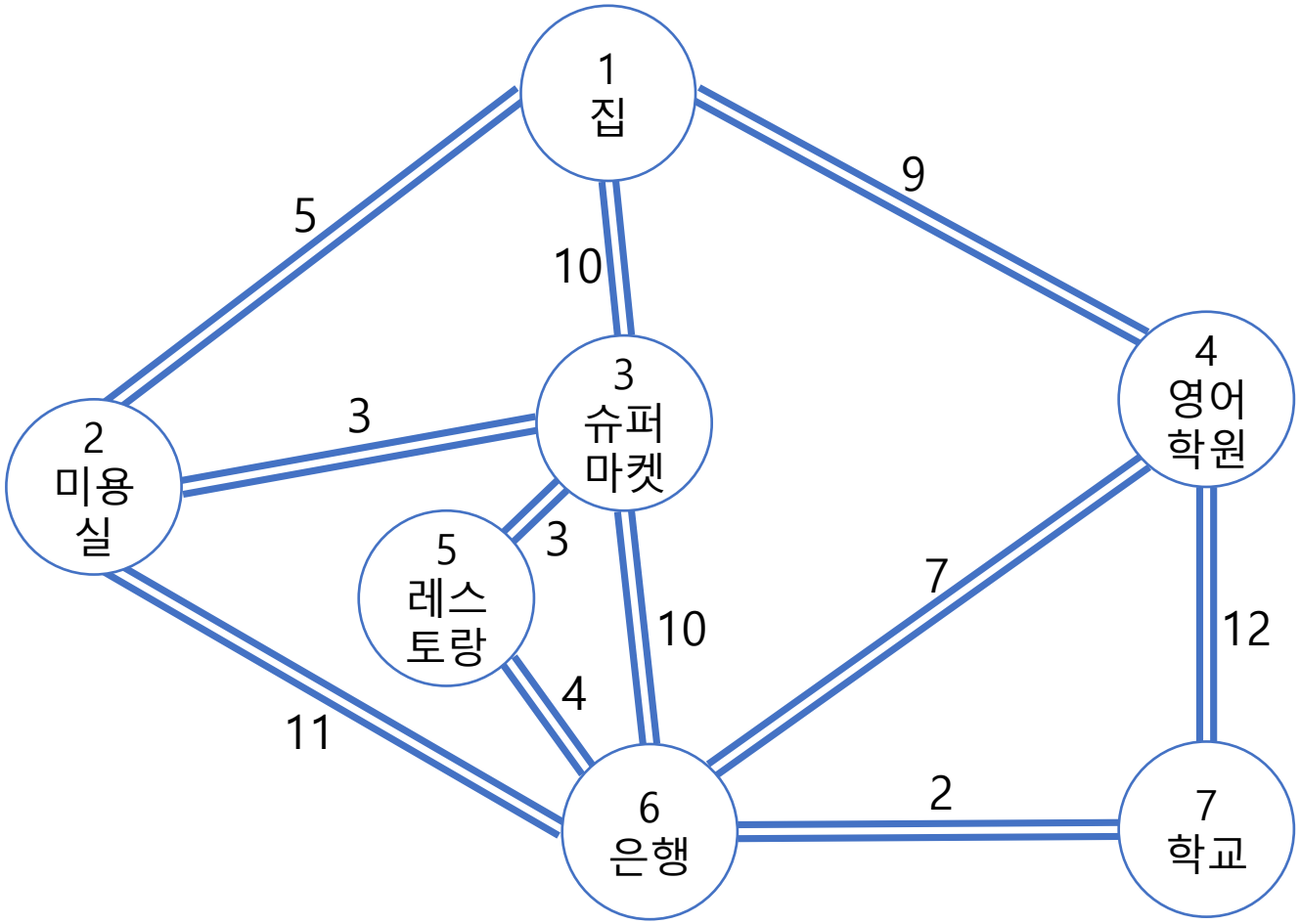
- 알고리즘
  1) 출발점으로부터의 최단거리를 저장할 배열 d[v]를 만들고 초기화 한다.
  2) 현재 노드를 나타내는 변수 A에 출발 노드의 번호를 저장한다.
  3) A에 인접한 모든 노드 B에 대해, d[A]+cost(A~B)와 d[B]의 값을 비교
  4) 만약 d[A] + P[A][B]의 값이 더 작다면, 즉 더 짧은 경로라면, d[B]의 값을 이 값으로 갱신한다.
  5) A의 모든 이웃 노드 B에 대해 이 작업을 수행한다.
  6) A의 상태를 "방문 완료"로 바꾼다. 그러면 이제 더 이상 A는 사용하지 않는다.
  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



다익스트라(dijkstra) 알고리즘

- 그래프로 표현



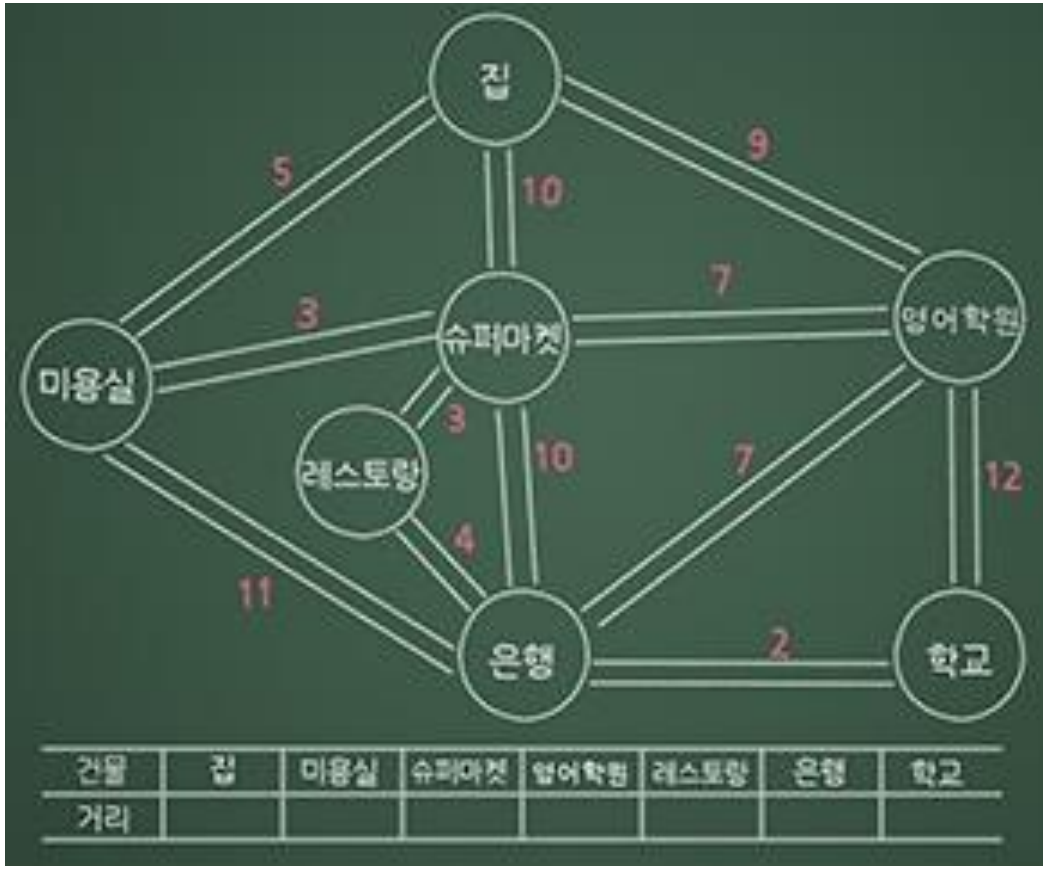
- 데이터 표현

정점수 노드수 시작정점

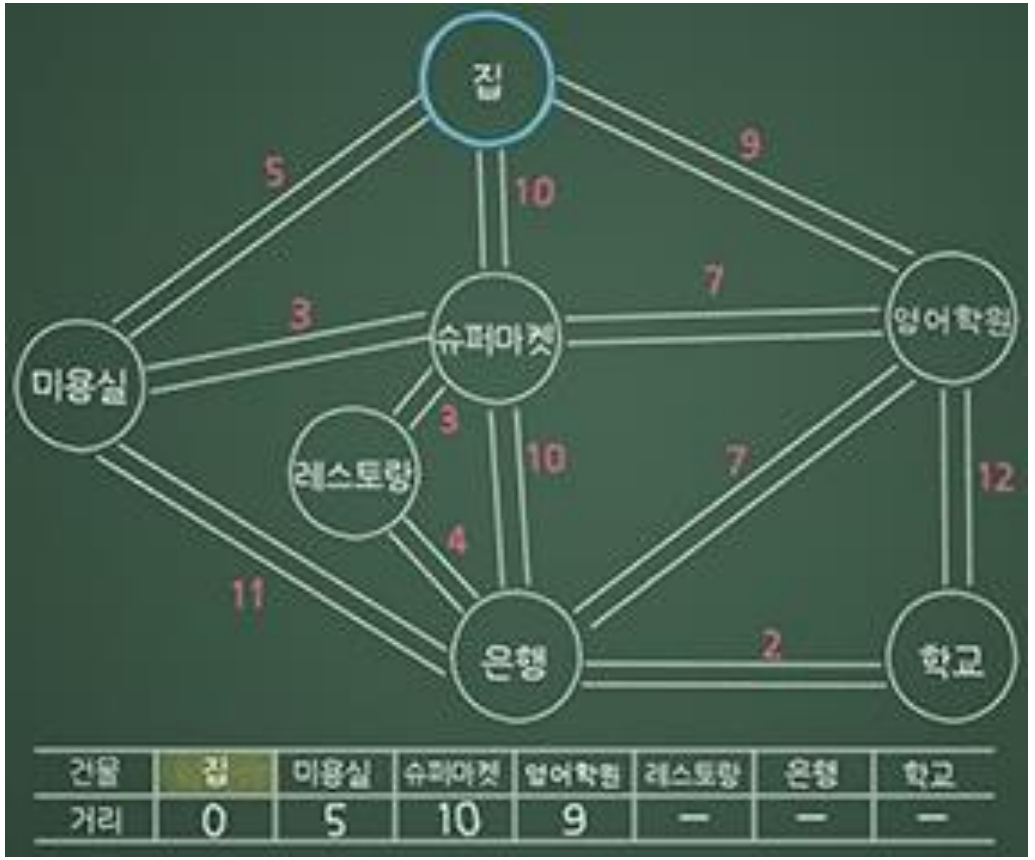
```
7 12 1
1 2 5
1 3 10
1 4 9
2 3 3
2 6 11
3 4 7
3 5 3
3 6 10
4 7 12
4 6 7
5 6 4
6 7 2
```

다익스트라(dijkstra) 알고리즘

- 과정1



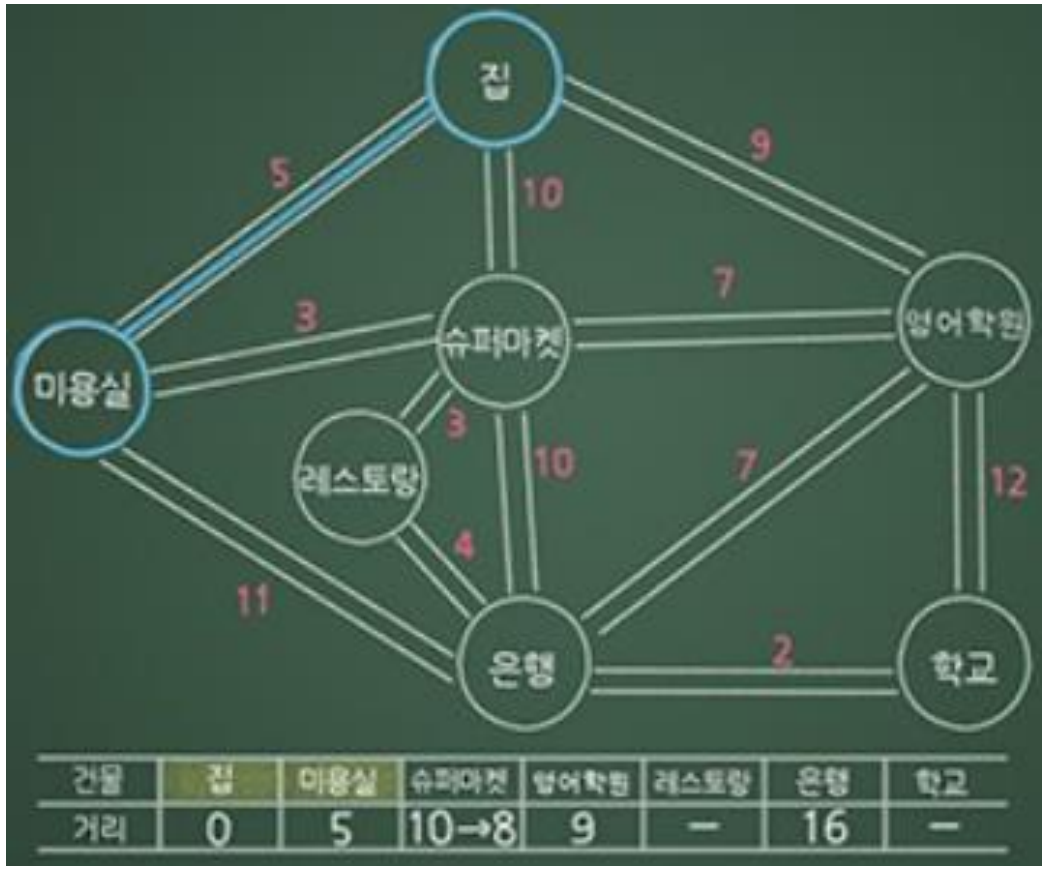
- 과정2



최대거리 테이블 초기화

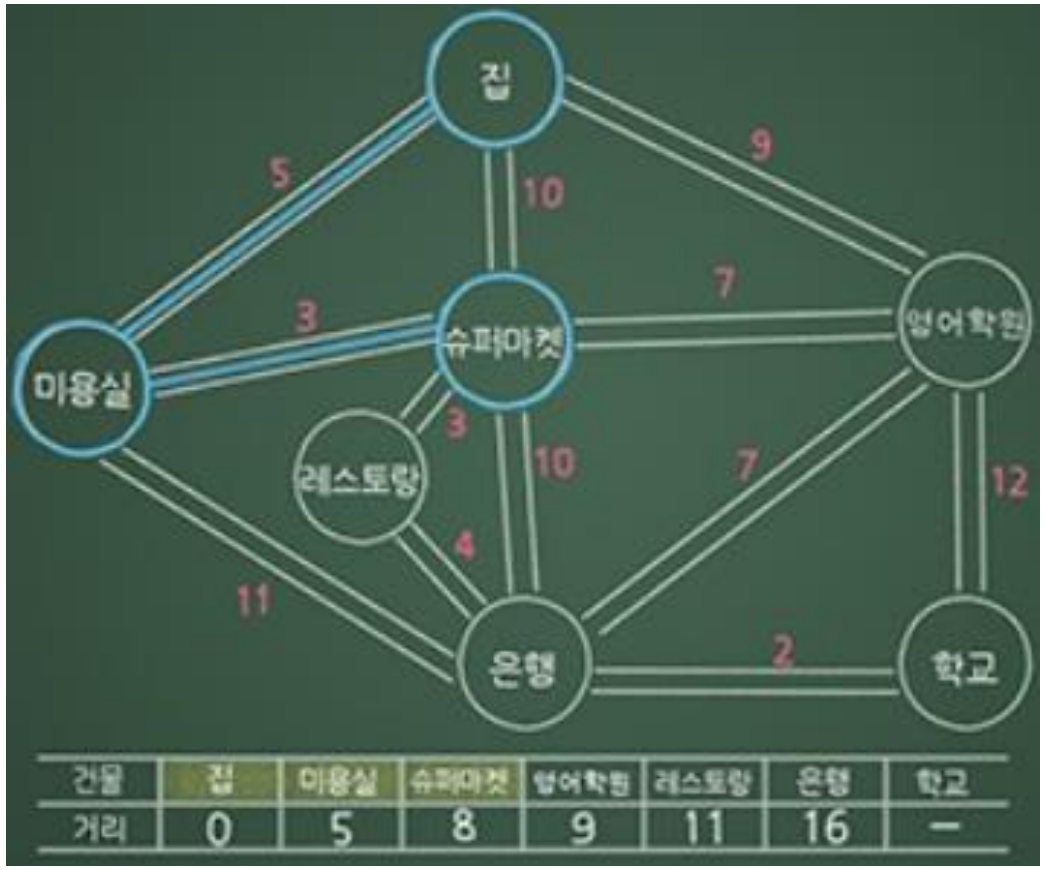
다익스트라(dijkstra) 알고리즘

- 과정3



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

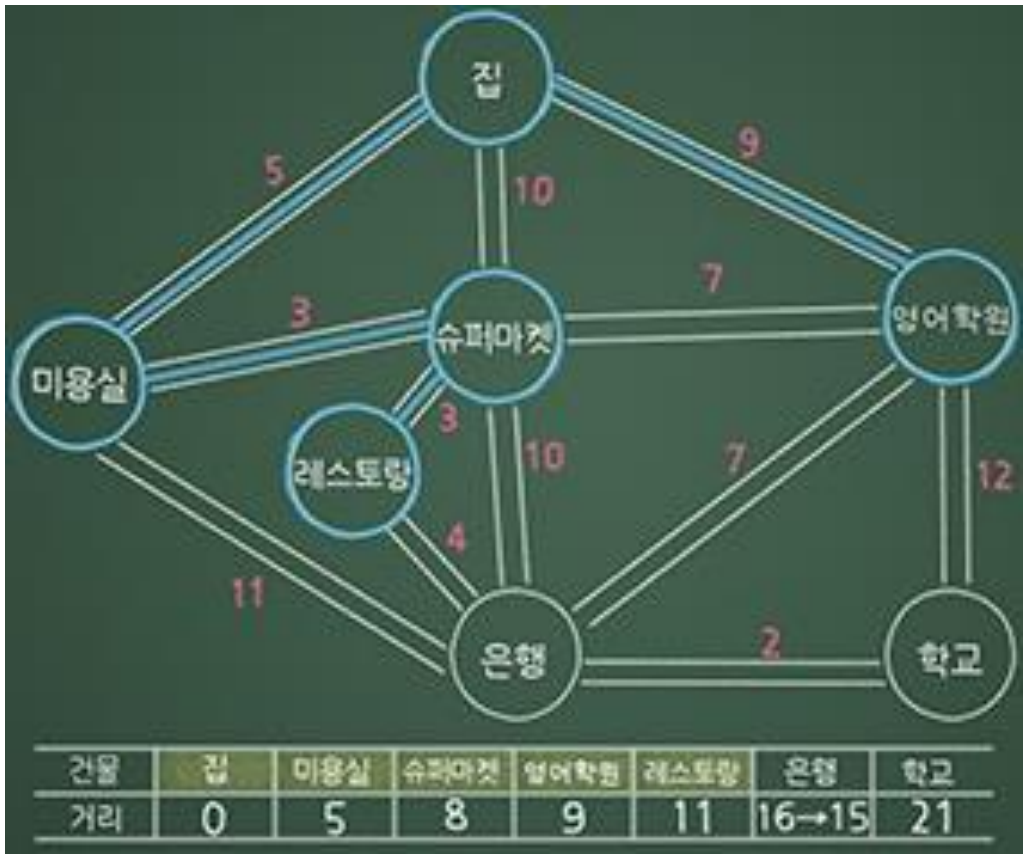
- 과정4



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

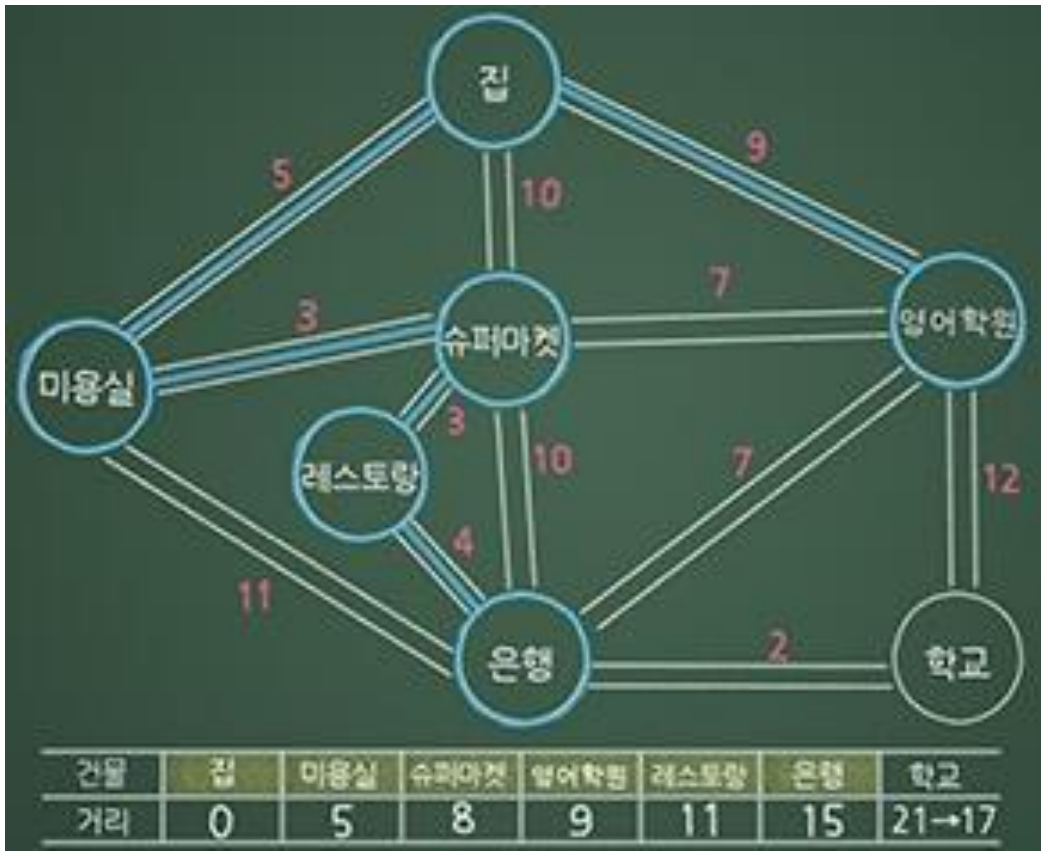
다익스트라(dijkstra) 알고리즘

- 과정5



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

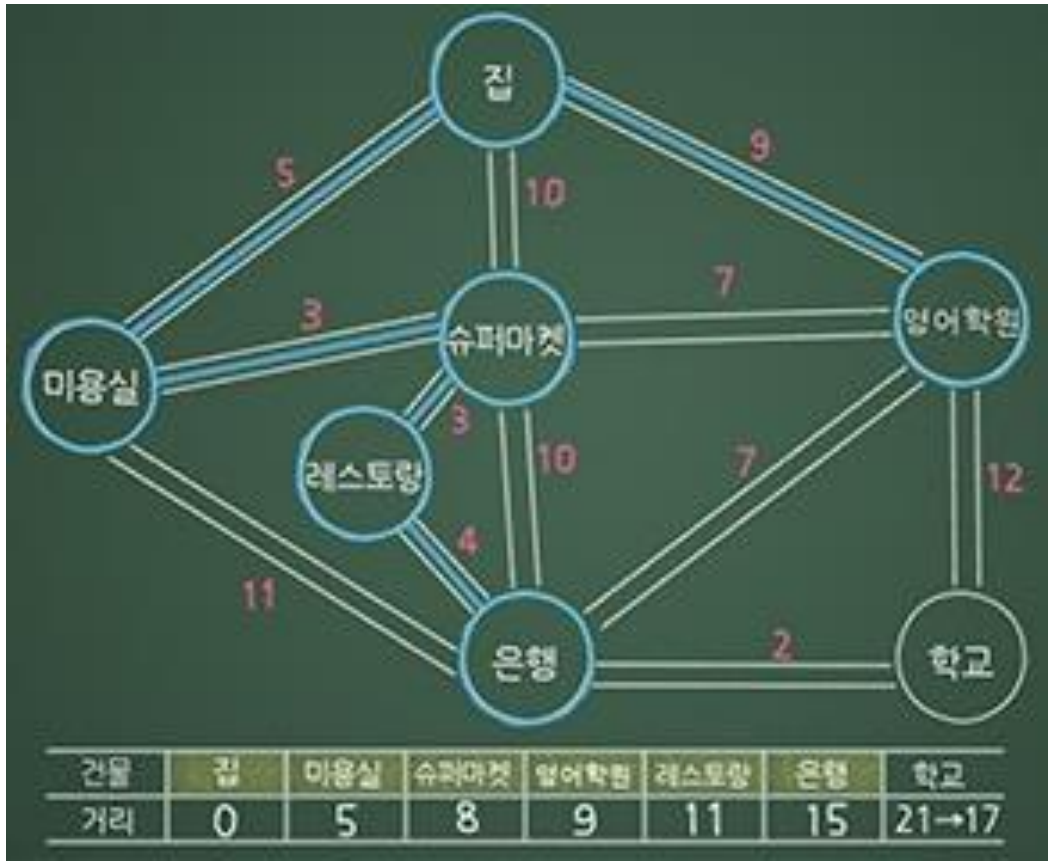
- 과정6



건물	집	미용실	슈퍼마켓	영어학원	레스토랑	은행	학교
거리	0	5	8	9	11	15	21→17

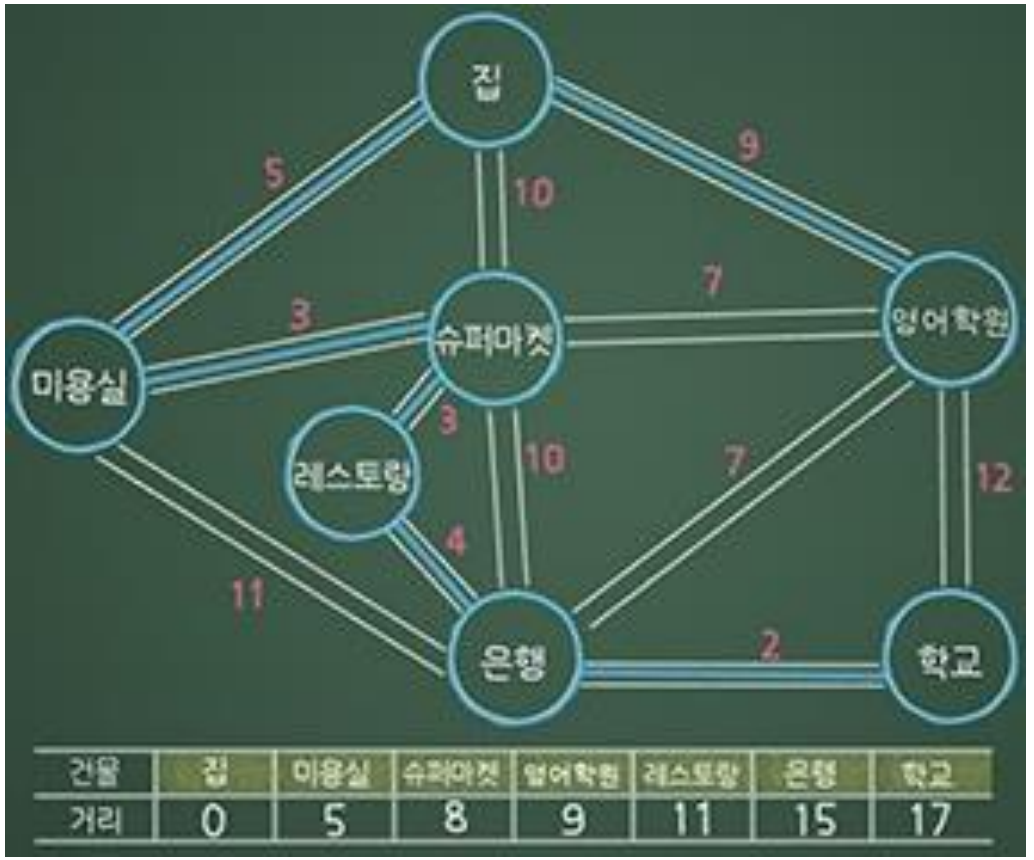
다익스트라(dijkstra) 알고리즘

- 과정7



건물	집	미용실	슈퍼마켓	영어학원	레스토랑	은행	학교
거리	0	5	8	9	11	15	21→17

- 과정8



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

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
    }
};

// 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
int dist[100001];
// 직전 노드 저장
int from[100001];
```

개선된 다익스트라 알고리즘 $O((V+E)log(V))$

```
void dijkstra(int start) {
    // 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 최단거리
        int now_dist = pq.top().cost;
        pq.pop();

        // 현재 노드가 이미 처리된 적이 있는 노드라면 무시
        if (dist[now_node] < now_dist) {
            printf(" %d(%d)노드 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 < graph[now_node].size(); i++) {
            // i번째 노드(graph[now_node][i])를 목적지로 설정
            noco target = graph[now_node][i];
            // 목적지까지 거리 = 현재 노드까지 거리 + 목적지까지 비용
            int tar_dist = now_dist + target.cost;
            // 현재 노드를 거쳐가는 거리가 다른 방법보다 짧으면,
            if (tar_dist < dist[target.node]) {
                // target까지의 최단거리 테이블 업데이트
                dist[target.node] = tar_dist;
                // 같은 내용 힙트리에도 삽입
                pq.push({target.node, tar_dist});
                // 목적지의 직전 노드가 now_node임을 저장
                from[target.node] = now_node;

                printf(" %d(%d)노드 업데이트\n",
                        target.node, tar_dist);
            }
        }
    }
}
```

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

플로이드-워셜(Floyd-Warshall) 알고리즘

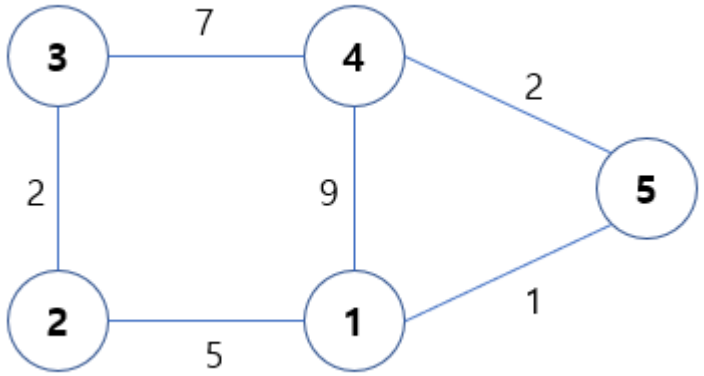
- 특징
  - 다익스트라가 하나의 정점에서 다른 모든 정점까지의 최단 거리를 구하는 알고리즘이었다면,
  - 플로이드-워셜은 모든 정점에서 다른 모든 정점까지의 최단 경로를 모두 구하는 알고리즘이다.
  - 플로이드-워셜 알고리즘은 다익스트라와는 달리 음의 간선도 사용가능
  - 정점이 N개일 때 모든 정점에 대하여 $O(N^2)$의 연산이 필요하므로 총 시간 복잡도는 $O(N^3)$이 된다

- 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

  - $D_{ab} = \min(D_{ab}, D_{ak} + D_{kb})$

플로이드-워셜(Floyd-Warshall) 알고리즘

- 초기 그래프

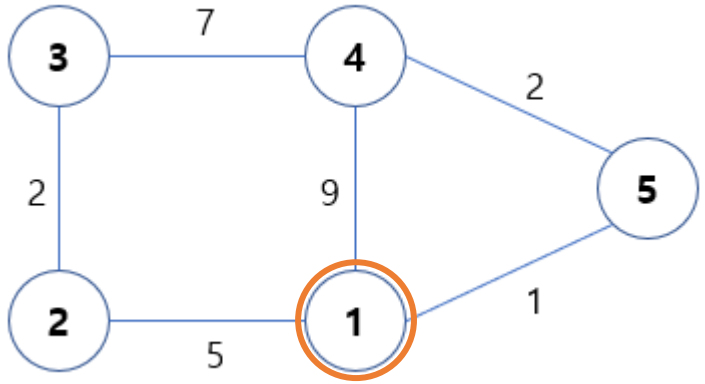


- 초기 인접행렬

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

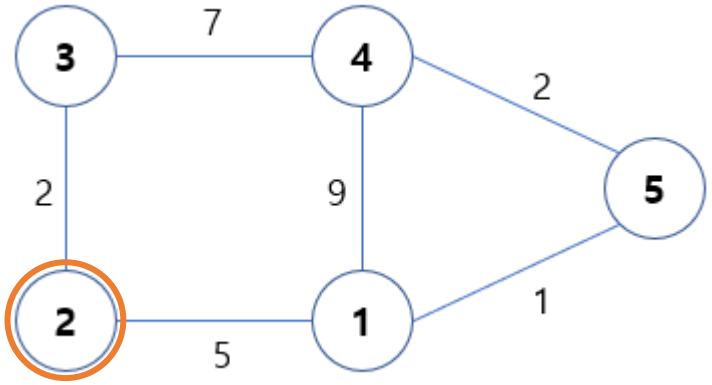
- Round1

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

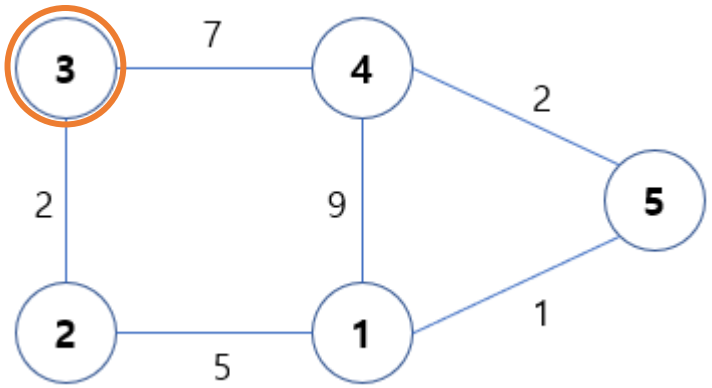
- Round2

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

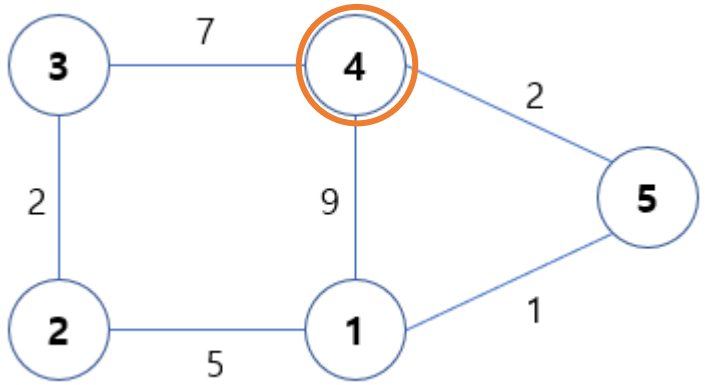
- Round3

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

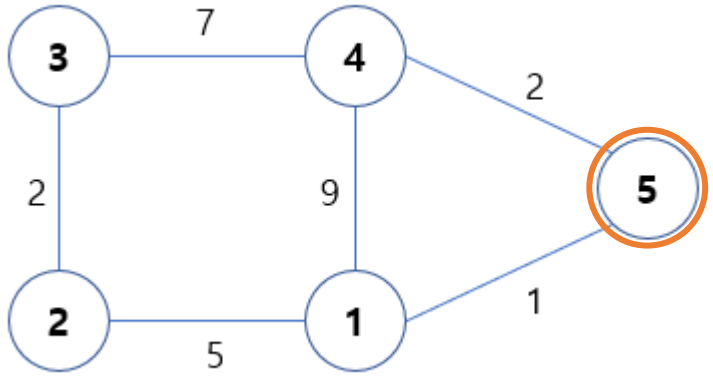
- Round4

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

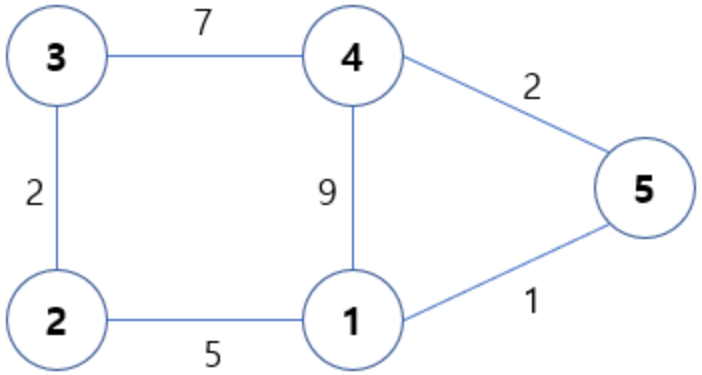
- Round5

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
}
```

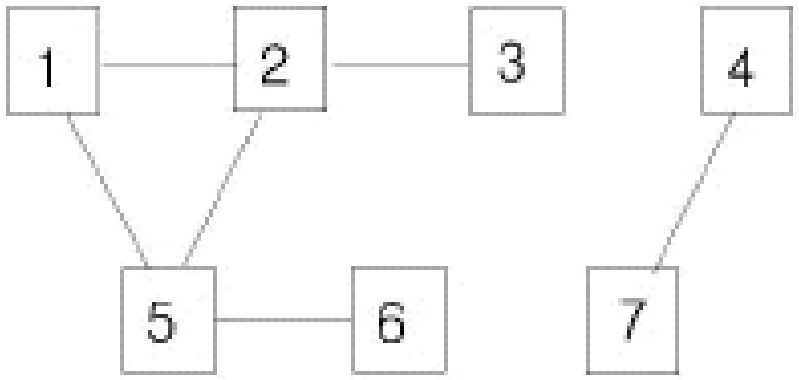
f\t	1	2	3	4	5
1	0	5	INF	9	1
2	5	0	2	INF	INF
3	INF	2	0	7	INF
4	9	INF	7	0	2
5	1	INF	INF	2	0

```
    // 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

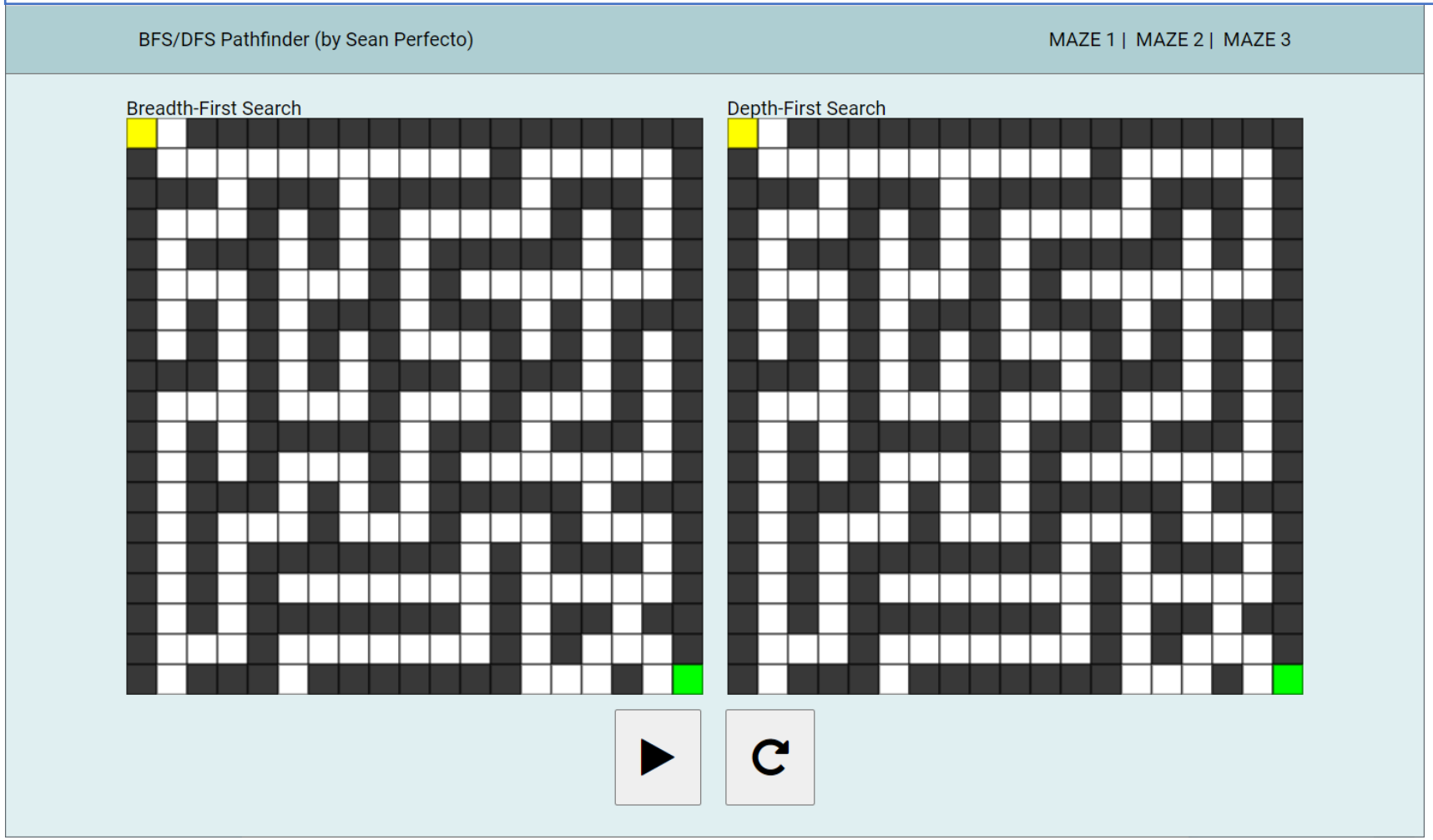
- 출처: 한국정보올림피아드(2004 지역본선 초등부 3번)

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;
}
```

미로 탈출 (BFS vs DFS)

https://seanperfecto.github.io/BFS-DFS-Pathfinder/



미로 탈출

- 문제

  정진이는 벽과 길로 만들어진 N행 M열(세로 N칸, 가로 M칸) 크기의 미로에 갇혀 있다.

  미로에서는 한 번에 한 칸씩만 이동할 수 있다. 벽은 0으로, 길은 0아닌 수로 표시된다.

  정진이가 미로에서 탈출하기 위해 이동하여야 하는 최소 칸수를 구하시오. 시작 칸과 마지막 칸도 이동거리에 포함시킨다.

입력 예	출력 예
7 8 1 0 1 0 0 0 0 1 1 0 1 0 0 1 1 1 1 1 8 1 1 1 0 1 0 1 0 1 0 0 0 1 1 1 0 1 0 1 1 1 1 0 0 1 0 1 0 9 0 0 0 1 1 1 0 1	11

- 입력

  첫 번째 줄에 두 정수 N, M이 주어진다.

  (4 ≤ N, M ≤ 200)

  다음 N개의 줄에는 각각 M개의 정수로 미로의 정보가 주어진다. 숫자는 각각 다음을 의미한다.

  (0: 벽, 1: 길, 8: 시작위치, 9: 도착위치)

- 출력

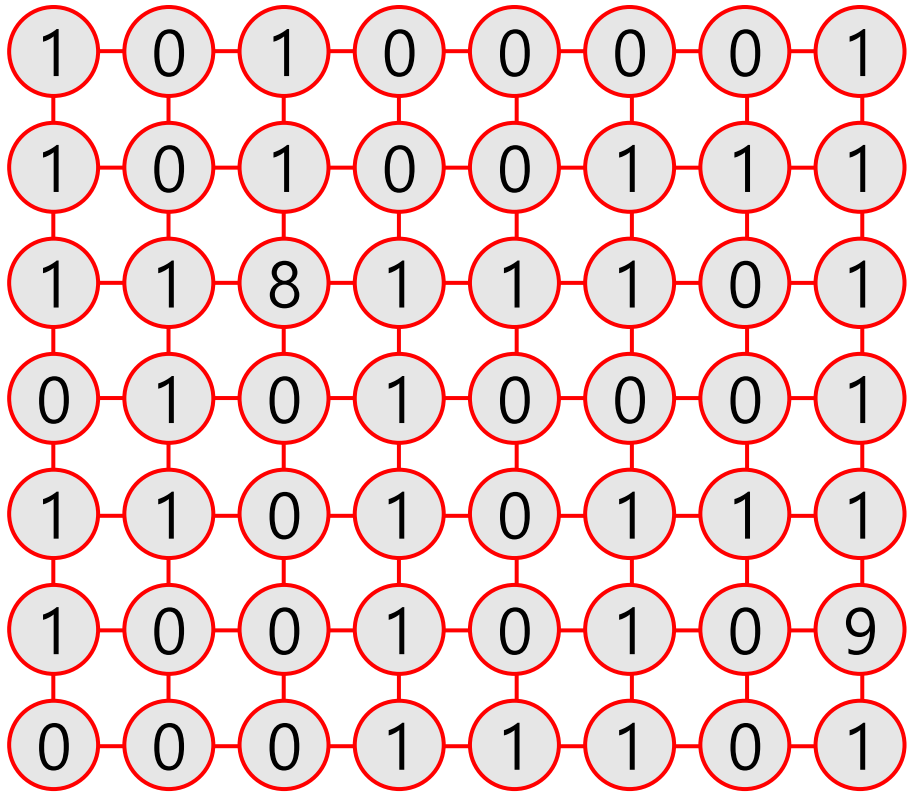
  첫 번째 줄에 최소 이동 칸의 개수를 출력한다.

미로 탈출

- 지도 데이터

```
7 8
1 0 1 0 0 0 0 1
1 0 1 0 0 1 1 1
1 1 8 1 1 1 0 1
0 1 0 1 0 0 0 1
1 1 0 1 0 1 1 1
1 0 0 1 0 1 0 9
0 0 0 1 1 1 0 1
```

- 그래프로 해석
  - 지도데이터를 그래프로보고 탐색기법 적용



미로 탈출 (초기설계)

```
#include <stdio.h>
#include <algorithm>
#include <queue>
#define  MAX_INT 0x7fffffff
using namespace std;

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

int n, m;         // n행 m열
int M[201][201]; // 지도 정보
int Sa, Sb;  //출발지 a행 b열
int Ga, Gb;  //목적지 a행 b열

// 이동할 네 가지 방향 정의, 아래와 같이 적으면
// 아래, 오른쪽, 위쪽, 왼쪽 순서로 탐색하게 됨.
int da[] = {1, 0,-1, 0}; // 행(세로) 방향
int db[] = {0, 1, 0,-1}; // 열(가로) 방향

bool safe(int a, int b) {  // a행 b열
    return (0<=a && a<n) && (0<=b && b<m);
}
```

```
void input() {
   // (n행 m열)
   scanf("%d %d", &n, &m);
   // 2차원 리스트의 맵 정보 입력 받기
   for(int a=0; a<n; a++) {
       for(int b=0; b<m; b++) {
          int c;
          scanf("%d", &c);

          if(c==8) // 출발지 설정
             Sa=a, Sb=b;
          else if(c==9) // 목적지 설정
             Ga=a, Gb=b;

          // 모든 길을 -1로 변경
          // a행 b열에 삽입
          if(c>=1)
             M[a][b]=-1;
          else
             M[a][b]=0;
       }
   }
}
```

```
// 현재 맵에서 탐색 상태를 출력
void output(const char* title, int dist) {
    // 제목출력
    printf("\n[%s] (%d)\n", title, dist);

    for(int a = 0; a < n; a++) {
        for(int b = 0; b < m; b++) {
            printf("%3d", M[a][b]);
        }
        printf("\n");
    }
}

void dfs(int a, int b, int d) {
}

int main(void) {
    input();
    output("initial state",-1);
    // dfs 또는 bfs 하나의 함수만 호출할 것!
    dfs(Sa, Sb, 1);  // Sa행 Sb열에서 DFS시작
    //bfs(Sa, Sb, 1);  //Sa행 Sb열에서 BFS시작
    output("last state", -1);
    return 0;
}
```

미로 탈출 - DFS로 구현

- DFS 알고리즘

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

- 초기맵 상태

Sa, Sb = (2, 2)
Ga, Gb = (5, 7)

	0	1	2	3	4	5	6	7
0	-1	0	-1	0	0	0	0	-1
1	-1	0	-1	0	0	-1	-1	-1
2	-1	-1	**-1**	-1	-1	-1	0	-1
3	0	-1	0	-1	0	0	0	-1
4	-1	-1	0	-1	0	-1	-1	-1
5	-1	0	0	-1	0	-1	0	**-1**
6	0	0	0	-1	-1	-1	0	-1

- DFS 구현

```
// a행, b열에서 dfs, 현재까지 계산한 거리는 d
void dfs(int a, int b, int d) {
    ?      //① a행 b열에 거리 기록
    //② 방문한 것으로 표시 이거 안해도 됨?
    //④ 목적지에 도착하면 맵상태와 도달거리 출력

                ?

    //③ 4방향으로 dfs
    // da, db배열을 이용하여 for문으로 간략화 가능
                                    // ↓

                                    // →

                ?                   // ↑

                                    // ←
}
```

미로 탈출 - DFS로 구현

- 결과 고찰

```
7 8
1 0 1 0 0 0 0 1
1 0 1 0 0 1 1 1
1 1 8 1 1 1 0 1
0 1 0 1 0 0 0 1
1 1 0 1 0 1 1 1
1 0 0 1 0 1 0 9
0 0 0 1 1 1 0 1

[initial state] (-1)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 -1 -1 -1
-1 -1 -1 -1 -1 -1  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1
```

```
[DFS success] (13)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 -1 -1 -1
-1 -1  1  2 -1 -1  0 -1
 0 -1  0  3  0  0  0 -1
-1 -1  0  4  0 10 11 12
-1  0  0  5  0  9  0 13
 0  0  0  6  7  8  0 -1

[last state] (-1)
 5  0  3  0  0  0  0 16
 4  0  2  0  0 17 16 15
 3  2  1  2 19 18  0 14
 0  3  0  3  0  0  0 13
 5  4  0  4  0 10 11 12
 6  0  0  5  0  9  0 13
 0  0  0  6  7  8  0 -1
```

- 엥? 최적해가 아닌데…
- 왜 최적해를 못 찾지?
- 원래 전체탐색이 진행되어야 하는데…

- DFS 구현

```
// a행, b열에서 dfs, 현재까지 계산한 거리는 d
void dfs(int a, int b, int d) {
    M[a][b] = d;  //① a행 b열에 거리 기록
    //② 방문한 것으로 표시 이거 안해도 됨?
    //④ 목적지에 도착하면 맵상태와 도달거리 출력
    if(a==Ga && b==Gb) {
        output("DFS success", d);
        return;
    }

    //③ 4방향으로 dfs
    // da, db배열을 이용하여 for문으로 간략화 가능
    if(safe(a+1, b) && M[a+1][b]==-1) // ↓
        dfs(a+1, b, d+1);
    if(safe(a, b+1) && M[a][b+1]==-1) // →
        dfs(a, b+1, d+1);
    if(safe(a-1, b) && M[a-1][b]==-1) // ↑
        dfs(a-1, b, d+1);
    if(safe(a, b-1) && M[a][b-1]==-1) // ←
        dfs(a, b-1, d+1);
}
```

미로 탈출 - DFS로 구현

- 이전 버전

```
// a행, b열에서 dfs, 현재까지 계산한 거리는 d
void dfs(int a, int b, int d) {
    M[a][b] = d;  //① a행 b열에 거리 기록
    //② 방문한 것으로 표시 이거 안해도 됨?
    //④ 목적지에 도착하면 맵상태와 도달거리 출력
    if(a==Ga && b==Gb) {
        output("DFS success", d);
        return;
    }

    //③ 4방향으로 dfs
    // da, db배열을 이용하여 for문으로 간략화 가능
    if(safe(a+1, b) && M[a+1][b]==-1) // ↓
        dfs(a+1, b, d+1);
    if(safe(a, b+1) && M[a][b+1]==-1) // →
        dfs(a, b+1, d+1);
    if(safe(a-1, b) && M[a-1][b]==-1) // ↑
        dfs(a-1, b, d+1);
    if(safe(a, b-1) && M[a][b-1]==-1) // ←
        dfs(a, b-1, d+1);
}
```

- 방향 탐색 배열이용 업데이트

```
// 이동할 네 가지 방향 정의, 아래와 같이 적으면
// 아래, 오른쪽, 위쪽, 왼쪽 순서로 탐색하게 됨.
int da[] = {1, 0,-1, 0}; // 행(세로) 방향
int db[] = {0, 1, 0,-1}; // 열(가로) 방향

// a행, b열에서 dfs, 현재까지 계산한 거리는 d
void dfs(int a, int b, int d) {
    M[a][b] = d;
    if(a==Ga && b==Gb) {
        output("DFS success", d);
        return;
    }

    //③ 4방향으로 dfs
    for(int i=0; i<4; i++) {
        int na = a+da[i], nb = b+db[i];
        if(safe(na, nb) && M[na][nb]==-1) {
            dfs(na, nb, d+1);
        }
    }
}
```

미로 탈출 - DFS로 구현

- DFS 구현 업데이트

```
// 최소거리(최적해)를 저장하기 위한 변수 셋팅
            ?
// a행, b열에서 dfs, 현재까지 계산한 거리는 d
void dfs(int a, int b, int d) {
    M[a][b] = d;
    if(a==Ga && b==Gb) {
        // 더 짧은 방법을 찾으면 최소거리 업데이트
            ?
        output("DFS success", d);
        return;
    }
    //③ 4방향으로 dfs 방법 업데이트, 백트랙시 길 복원
                    ?
}
```

- 결과 확인

```
[initial state] (-1)
 -1  0 -1  0  0  0  0 -1
 -1  0 -1  0  0 -1 -1 -1
 -1 -1 -1 -1 -1 -1  0 -1
  0 -1  0 -1  0  0  0 -1
 -1 -1  0 -1  0 -1 -1 -1
 -1  0  0 -1  0 -1  0 -1
  0  0  0 -1 -1 -1  0 -1

[DFS success] (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 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

[last state] (-1)
 -1  0 -1  0  0  0  0 -1
 -1  0 -1  0  0 -1 -1 -1
 -1 -1  1 -1 -1 -1  0 -1
  0 -1  0 -1  0  0  0 -1
 -1 -1  0 -1  0 -1 -1 -1
 -1  0  0 -1  0 -1  0 -1
  0  0  0 -1 -1 -1  0 -1
```

미로 탈출 – DFS 최종 구현

```
int min_dist=MAX_INT;
// a행, b열에서 dfs, 현재까지 계산한 거리는 d
void dfs(int a, int b, int d) {
    M[a][b] = d;

    // 목적지에 도달하면,
    if(a==Ga && b==Gb) {
        if(min_dist > d)
            min_dist=d;
        output("DFS search success", d);
        return;
    }

    int noway=0;  // 현재 위치에서 이동 불가능 방향 개수
    for(int i=0; i<4; i++) {
        int na = a+da[i];
        int nb = b+db[i];

        if(safe(na, nb) && M[na][nb]==-1) {
            dfs(na, nb, d+1);
            M[na][nb]=-1;  // 백트랙할 경우 길복원
        }
        else
            noway++; // 이동불가 카운터 증가

        if(noway==4) // 4방향 모두 길이 막혀있으면,
            output("DFS search fail", -1);
    }
}
```

```
[initial state] (-1)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 -1 -1 -1
-1 -1 -1 -1 -1 -1  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[DFS search success] (13)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 -1 -1 -1
-1 -1  1  2 -1 -1  0 -1
 0 -1  0  3  0  0  0 -1
-1 -1  0  4  0 10 11 12
-1  0  0  5  0  9  0 13
 0  0  0  6  7  8  0 -1

[DFS search fail] (-1)
-1  0 -1  0  0  0  0 16
-1  0 -1  0  0 -1 -1 15
-1 -1  1  2 -1 -1  0 14
 0 -1  0  3  0  0  0 13
-1 -1  0  4  0 10 11 12
-1  0  0  5  0  9  0 -1
 0  0  0  6  7  8  0 -1

[DFS search fail] (-1)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 17 16 15
-1 -1  1  2 19 18  0 14
 0 -1  0  3  0  0  0 13
-1 -1  0  4  0 10 11 12
-1  0  0  5  0  9  0 -1
 0  0  0  6  7  8  0 -1
```

```
[DFS search success] (11)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0  5  6  7
-1 -1  1  2  3  4  0  8
 0 -1  0 -1  0  0  0  9
-1 -1  0 -1  0 -1 -1 10
-1  0  0 -1  0 -1  0 11
 0  0  0 -1 -1 -1  0 -1

[DFS search fail] (-1)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0  5  6  7
-1 -1  1  2  3  4  0  8
 0 -1  0 19  0  0  0  9
-1 -1  0 18  0 12 11 10
-1  0  0 17  0 13  0 -1
 0  0  0 16 15 14  0 -1

[DFS search fail] (-1)
-1  0 -1  0  0  0  0  8
-1  0 -1  0  0  5  6  7
-1 -1  1  2  3  4  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[DFS search fail] (-1)
-1  0  3  0  0  0  0 -1
-1  0  2  0  0 -1 -1 -1
-1 -1  1 -1 -1 -1  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1
```

```
[DFS search fail] (-1)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 -1 -1 -1
-1  2  1 -1 -1 -1  0 -1
 0  3  0 -1  0  0  0 -1
 5  4  0 -1  0 -1 -1 -1
 6  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[DFS search fail] (-1)
 5  0 -1  0  0  0  0 -1
 4  0 -1  0  0 -1 -1 -1
 3  2  1 -1 -1 -1  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[last state] (-1)
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 -1 -1 -1
-1 -1  1 -1 -1 -1  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1
```

미로 탈출 - BFS로 구현

- BFS 알고리즘

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

- 초기 맵 상태

Sa, Sb = (2, 2)
Ga, Gb = (5, 7)

	0	1	2	3	4	5	6	7
0	-1	0	-1	0	0	0	0	-1
1	-1	0	-1	0	0	-1	-1	-1
2	-1	-1	**-1**	-1	-1	-1	0	-1
3	0	-1	0	-1	0	0	0	-1
4	-1	-1	0	-1	0	-1	-1	-1
5	-1	0	0	-1	0	-1	0	**-1**
6	0	0	0	-1	-1	-1	0	-1

```
// bfs(시작행a, 시작열b, 거리d)
void bfs(int sa, int sb, int d) {
    ?                  // 시작위치에 거리 d표시, 방문표시 안함?
    queue<vertex> q;
    ?                  // 큐에 현재 위치 삽입

    while(!q.empty()) {  // 큐가 빌 때까지 반복
        ?              // 큐의 첫 번째 정점 a행
                       // 큐의 첫 번째 정점 b열
                       // 큐에서 첫 번째 항목 삭제

        // 현재 위치에서 4가지 방향으로의 위치 확인
        for(int i = 0; i < 4; i++) {
            ?          // na: next a
                       // nb: next b
            if (safe(na, nb) && M[na][nb] == -1) {
                ?      // 다음 위치에 거리 표시
                // 목적지에 도달하면 성공 출력하고 탈출
                ?

                ?      // 다음 위치를 큐에 삽입
            }
        }
    }
    output("BFS search fail", -1); // 여기까지 오면 탐색 실패임
}
```

미로 탈출 - BFS로 구현

```
// bfs(시작행a, 시작열b, 거리d)
void bfs(int sa, int sb, int d) {
    M[sa][sb] = d;      // 시작위치에 거리 d표시
    queue<vertex> q;
    q.push({sa, sb});  // 큐에 현재 위치 삽입

    while(!q.empty()) {  // 큐가 빌 때까지 반복
        int a = q.front().a; // 큐의 첫 번째 정점 a행
        int b = q.front().b; // 큐의 첫 번째 정점 b열
        q.pop();             // 큐에서 첫 번째 항목 삭제
        // 현재 위치에서 4가지 방향으로의 위치 확인
        for(int i=0; i<4; i++) {
            int na=a+da[i];  // na: next a
            int nb=b+db[i];  // nb: next b
            if (safe(na, nb) && M[na][nb] == -1) {
                M[na][nb] = M[a][b] + 1;  // 다음 위치에 거리 표시
                // 목적지에 도달하면 성공 출력하고 탈출
                if(na==Ga && nb==Gb) {
                    output("BFS search success", M[na][nb]);
                    return;
                }
                q.push({na, nb});    // 다음 위치를 큐에 삽입
            }
        }
    }
    output("BFS search fail", -1); // 여기까지 오면 탐색 실패임
}
```

- 결과 확인

```
7 8
1 0 1 0 0 0 0 1
1 0 1 0 0 1 1 1
1 1 8 1 1 1 0 1
0 1 0 1 0 0 0 1
1 1 0 1 0 1 1 1
1 0 0 1 0 1 0 9
0 0 0 1 1 1 0 1

[initial state] (-1)
 -1  0 -1  0  0  0  0 -1
 -1  0 -1  0  0 -1 -1 -1
 -1 -1 -1 -1 -1 -1  0 -1
  0 -1  0 -1  0  0  0 -1
 -1 -1  0 -1  0 -1 -1 -1
 -1  0  0 -1  0 -1  0 -1
  0  0  0 -1 -1 -1  0 -1
```

```
[BFS search success] (11)
  5  0  3  0  0  0  0  8
  4  0  2  0  0  5  6  7
  3  2  1  2  3  4  0  8
  0  3  0  3  0  0  0  9
  5  4  0  4  0 10 11 10
  6  0  0  5  0  9  0 11
  0  0  0  6  7  8  0 -1

[last state] (-1)
  5  0  3  0  0  0  0  8
  4  0  2  0  0  5  6  7
  3  2  1  2  3  4  0  8
  0  3  0  3  0  0  0  9
  5  4  0  4  0 10 11 10
  6  0  0  5  0  9  0 11
  0  0  0  6  7  8  0 -1
```

미로 탈출 - BFS로 구현

- 탐색 과정을 보여주는 업데이트

```
if (safe(na, nb) && M[na][nb] == -1) {
    int new_d = M[a][b] + 1;  // 새로운 거리 계산

    // 새롭게 계산된 거리가 지금 거리보다 멀어지면,
    if(new_d > d) { // 현재 맵상태 일단 출력하고
        output("BFS searched new distance", -1);
        d = new_d;
    }
    M[na][nb] = new_d; // 다음 위치에 거리 표시
    // 목적지에 도달하면 성공 출력하고 탈출
    if(na==Ga && nb==Gb) {
        output("BFS search success", M[na][nb]);
        return;
    }
    q.push({na, nb});    // 다음 위치를 큐에 삽입
}
```

```
[BFS searched new distance]
-1  0 -1  0  0  0  0 -1
-1  0 -1  0  0 -1 -1 -1
-1 -1  1 -1 -1 -1  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[BFS searched new distance]
-1  0 -1  0  0  0  0 -1
-1  0  2  0  0 -1 -1 -1
-1  2  1  2 -1 -1  0 -1
 0 -1  0 -1  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[BFS searched new distance]
-1  0  3  0  0  0  0 -1
-1  0  2  0  0 -1 -1 -1
 3  2  1  2  3 -1  0 -1
 0  3  0  3  0  0  0 -1
-1 -1  0 -1  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[BFS searched new distance]
-1  0  3  0  0  0  0 -1
 4  0  2  0  0 -1 -1 -1
 3  2  1  2  3  4  0 -1
 0  3  0  3  0  0  0 -1
-1  4  0  4  0 -1 -1 -1
-1  0  0 -1  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1
```

```
[BFS searched new distance]
 5  0  3  0  0  0  0 -1
 4  0  2  0  0  5 -1 -1
 3  2  1  2  3  4  0 -1
 0  3  0  3  0  0  0 -1
 5  4  0  4  0 -1 -1 -1
-1  0  0  5  0 -1  0 -1
 0  0  0 -1 -1 -1  0 -1

[BFS searched new distance]
 5  0  3  0  0  0  0 -1
 4  0  2  0  0  5  6 -1
 3  2  1  2  3  4  0 -1
 0  3  0  3  0  0  0 -1
 5  4  0  4  0 -1 -1 -1
 6  0  0  5  0 -1  0 -1
 0  0  0  6 -1 -1  0 -1

[BFS searched new distance]
 5  0  3  0  0  0  0 -1
 4  0  2  0  0  5  6  7
 3  2  1  2  3  4  0 -1
 0  3  0  3  0  0  0 -1
 5  4  0  4  0 -1 -1 -1
 6  0  0  5  0 -1  0 -1
 0  0  0  6  7 -1  0 -1

[BFS searched new distance]
 5  0  3  0  0  0  0  8
 4  0  2  0  0  5  6  7
 3  2  1  2  3  4  0  8
 0  3  0  3  0  0  0 -1
 5  4  0  4  0 -1 -1 -1
 6  0  0  5  0 -1  0 -1
 0  0  0  6  7  8  0 -1
```

https://algo.datahub.pe.kr/src/maze/bfs/2.%20maze_bfs_show_process.cpp

미로 탈출 DFS + BFS 전체 소스코드

```
//written by akapo@naver.com
#include <stdio.h>
#include <algorithm>
#include <queue>
#define  MAX_INT 0x7fffffff
using namespace std;

typedef struct {
    int a, b;
} vertex;

int n, m;
int M[201][201]; // 지도 정보
int Sa, Sb;  //출발지 a행 b열
int Ga, Gb;  //목적지 a행 b열

// 이동할 네 가지 방향 정의, 아래와 같이 적으면
// 아래, 오른쪽, 위쪽, 왼쪽 순서로 탐색하게 됨.
int da[] = {1, 0,-1, 0}; // 행 방향
Int db[] = {0, 1, 0,-1}; // 열 방향

bool safe(int a, int b) {  // a행 b열
    return (0<=a && a<n) && (0<=b && b<m);
}

void input() {
    // N, M을 공백을 기준으로 구분하여 입력 받기(n행 m열)
    scanf("%d %d", &n, &m);
    // 2차원 리스트의 맵 정보 입력 받기
    for(int a=0; a<n; a++) {
        for(int b=0; b<m; b++) {
            int c;
            scanf("%d", &c);

            if(c==8) Sa=a, Sb=b;    // 출발지 설정
            else if(c==9) Ga=a, Gb=b; // 목적지 설정

            // 출발지 목적지 포함 모든 길을 -1로 변경함
            if(c>=1) M[a][b]=-1;
            else M[a][b]=0; // a행 b열에 삽입
        }
    }
}
```

```
// 현재 맵에서 탐색 상태를 출력
void output(const char* title, int dist) {
    if(dist > 0)
        printf("\n[%s] (%d)\n", title, dist);
    else
        printf("\n[%s]\n", title);

    for(int a=0; a<n; a++) {
        for(int b = 0; b < m; b++) {
            printf("%3d", M[a][b]);
        }
        printf("\n");
    }
}
// bfs(시작행a, 시작열b, 거리d)
void bfs(int sa, int sb, int d) {
    M[sa][sb] = d;
    queue<vertex> q;
    q.push({sa, sb});

    while(!q.empty()) {  // 큐가 빌 때까지 반복하기
        int a = q.front().a;
        int b = q.front().b;
        q.pop();

        // 현재 위치에서 4가지 방향으로의 위치 확인
        for(int i=0; i<4; i++) {
            int na=a+da[i]; // na: next a
            int nb=b+db[i]; // nb: next b

            if (safe(na, nb) && M[na][nb] == -1) {
                M[na][nb] = M[a][b] + 1;

                if(na==Ga && nb==Gb) { // 목적지에 도달하면,
                    output("BFS search success", M[na][nb]);
                    return;
                }
                q.push({na, nb});
            }
        }
    }
    output("BFS search fail", -1);
}
```

```
int min_dist=MAX_INT;
// a행, b열에서 dfs, 현재까지 계산한 거리는 d
void dfs(int a, int b, int d) {
    M[a][b] = d;

    // 목적지에 도달하면,
    if(a==Ga && b==Gb) {
        if(min_dist > d)
            min_dist=d;
        output("DFS search success", d);
        return;
    }

    int noway=0;  // 현재 위치에서 이동 불가능 방향 개수
    for(int i=0; i<4; i++) {
        int na = a+da[i];
        int nb = b+db[i];

        if(safe(na, nb) && M[na][nb]==-1) {
            dfs(na, nb, d+1);
            M[na][nb]=-1;  // 백트랙할 경우 길복원
        }
        else
            noway++; // 이동불가 카운터 증가

        if(noway==4) // 4방향 모두 길이 막혀있으면,
            output("DFS search fail", -1);
    }
}

int main(void) {
    input();
    output("initial state",-1);
    // dfs 또는 bfs 하나의 함수만 호출할 것!
    // Sa행 Sb열에서 DFS시작
    dfs(Sa, Sb, 1);
    // Sa행 Sb열에서 BFS시작
    //bfs(Sa, Sb, 1);
    output("last state",-1);
    return 0;
}
```