1. 알고리즘

1.1. 알고리즘(algorithm)이란?

어떤 문제를 풀기 위한 절차나 방법
주어진 입력을 출력으로 만드는 과정
알고리즘의 각 단계는 구체적이고 명료해야 함

페르시아의 수학자 알-콰리즈미(780~850년경) - 이차방정식의 풀이법과 인수분해를 개발한 분
수학의 발전에 막대한 영향을 끼친 분
이 분의 이름에서 계산 방법을 뜻하는 알고리즘이라는 단어가 나옴

1.2. 절대값 계산

```
# 0번 : 절대값 계산

import math

#절대값 계산 함수(부호 판단 방식)
#입력: 실수값 a
#출력: a의 절대값
def abs_sign(a):
    if a>=0:
        return a
    else:
        return -a

#절대값 계산 함수(제곱-제곱근 방식)
def abs_square(a):
    b=a*a
    return math.sqrt(b)

print(abs_sign(5))
print(abs_sign(-3))
print()
print(abs_square(5))
print(abs_square(-3))
```

1.3. 합계 구하기

```
# 입력: n
# 출력: 1부터 n까지 연속한 숫자의 합계

def mysum(n):
    s = 0
    for i in range(1, n + 1):
        s = s + i
    return s

print(mysum(10))   # 1부터 10까지의 합(입력:10, 출력:55)
print(mysum(100))  # 1부터 100까지의 합(입력:100, 출력:5050)
```

가우스의 방법

```
# 입력: n
# 출력: 1부터 n까지의 숫자를 더한 값

def sum_n(n):
    return n * (n + 1) // 2 #정수몫

print(sum_n(10))   # 1부터 10까지 합(입력:10, 출력:55)
print(sum_n(100))  # 1부터 100까지 합(입력:100, 출력:5050)
```

계산횟수 비교
첫번째 방법 : 덧셈 n회
두번째 방법 : 덧셈,곱셈,나눗셈 각 1회(총 3회)

```
#연습문제1,

# 연속한 숫자 제곱의 합을 구하는 알고리즘
# 입력: n
# 출력: 1부터 n까지 연속한 숫자의 제곱을 더한 합

def sum_sq1(n):
    s = 0
    for i in range(1, n + 1):
        s = s + i * i
    return s

print(sum_sq1(10))    # 1부터 10까지 제곱의 합(입력:10, 출력:385)
print(sum_sq1(100))  # 1부터 100까지 제곱의 합(입력:100, 출력:338350)
```

위 코드를 아래 공식을 사용하여 풀 수 있습니다.

$n\ (n\ +1)(2n\ +1)/6$

```
# 연속한 숫자 제곱의 합 구하기 알고리즘
# 입력: n
# 출력: 1부터 n까지 연속한 숫자의 제곱을 더한 합

def sum_sq2(n):
    return n * (n + 1) * (2 * n + 1) // 6

print(sum_sq2(10))    # 1부터 10까지 제곱의 합(입력:10, 출력:385)
print(sum_sq2(100))  # 1부터 100까지 제곱의 합(입력:100, 출력:338350)
```

1.4. 최대값 구하기

```
# 입력: 숫자가 n개 들어 있는 리스트
# 출력: 숫자 n개 중 최대값

def find_max(a):
    n = len(a)    # 입력 크기 n
    max_v = a[0]  # 리스트의 첫 번째 값을 최대값으로 기억
    for i in range(1, n):  # 1부터 n-1까지 반복
        if a[i] > max_v:   # 이번 값이 현재까지 기억된 최대값보
다 크면
            max_v = a[i]   # 최대값을 변경
    return max_v

v = [17, 92, 18, 33, 58, 7, 33, 42]
print(find_max(v))

#비교를 n-1회 해야 함
#입력크기와 계산시간이 대체로 비례한다.
```

```
# 최대값의 위치 구하기
# 입력: 숫자가 n개 들어 있는 리스트
# 출력: 숫자 n개 중에서 최대값이 있는 위치(0부터 n-1까지의 값)

def find_max_idx(a):
    n = len(a)   # 입력 크기 n
    max_idx = 0  # 리스트 중 0번 위치를 일단 최대값 위치로 기억
    for i in range(1, n):
        if a[i] > a[max_idx]:  # 이번 값이 현재까지 기억된 최대값보다 크면
            max_idx = i        # 최대값의 위치를 변경
    return max_idx

v = [17, 92, 18, 33, 58, 7, 33, 42]
print(find_max_idx(v))
```

```
# 최소값 구하기
# 입력: 숫자가 n개 들어 있는 리스트
# 출력: 숫자 n개 중 최소값

def find_min(a):
    n = len(a)    # 입력 크기 n
    min_v = a[0]  # 리스트 중 첫 번째 값을 일단 최소값으로 기억
    for i in range(1, n):  # 1부터 n-1까지 반복
        if a[i] < min_v:   # 이번 값이 현재까지 기억된 최소값보다 작으면
            min_v = a[i]   # 최소값을 변경
    return min_v

v = [17, 92, 18, 33, 58, 7, 33, 42]
print(find_min(v))
```

1.5. 동명이인 찾기

```
#집합
s=set()
s.add(1)
s.add(2)
s.add(2) #중복값은 제외됨
print(s)
print(len(s))
#순서에 관계없이 집합의 원소가 같은지 비교
print( {1,2} == {2,1})
```

```
# 두 번 이상 나온 이름 찾기
# 입력: 이름이 n개 들어 있는 리스트
# 출력: 이름 n개 중 반복되는 이름의 집합

def find_same_name(a):
    n = len(a)   # 리스트의 자료 개수를 n에 저장
    result = set()  # 결과를 저장할 집합(리스트로 하면 중복값이
발생함)
    for i in range(0, n - 1):      # 0부터 n-2까지 반복
        for j in range(i + 1, n):  # i+1부터 n-1까지 반복
            if a[i] == a[j]:       # 이름이 같으면
                result.add(a[i])   # 찾은 이름을 result에 추가
    return result

name = ["김철수", "박철수", "홍수민", "김철수", "김철수"]
print(find_same_name(name))

name2 = ["김미영", "박수진", "이민주", "김미영", "최숙희"]
print(find_same_name(name2))
```

```
# n명에서 두 명을 뽑아 짝을 만드는 모든 경우를 찾는 알고리즘
# 입력: n명의 이름이 들어 있는 리스트
# 출력: 두 명을 뽑아 만들 수 있는 모든 짝

def print_pairs(a):
    n = len(a)  # 리스트의 자료 개수를 n에 저장
    for i in range(0, n - 1):      # 0부터 n-2까지 반복
        for j in range(i + 1, n):  # i+1부터 n-1까지 반복
            if a[i] != a[j]:
                print(a[i], "-", a[j])

name = ["김철수", "박철수", "홍수민", "김철수", "김철수"]
print_pairs(name)
print()
name2 = ["김미영", "박수진", "이민주", "김미영", "최숙희"]
print_pairs(name2)
```

1.6. 팩토리얼

```
# 팩토리얼(계승)
# 연속한 숫자의 곱을 구하는 알고리즘
# 입력: n
# 출력: 1부터 n까지 연속한 숫자를 곱한 값

import math
print(math.factorial(10))

# 반복문을 사용한 코드
def fact(n):
    f = 1                       # 곱을 계산할 변수, 초깃값은 1
    for i in range(1, n + 1):  # 1부터 n까지 반복(n+1은 제외)
        f = f * i               # 곱셈 연산으로 수정
    return f

print(fact(1))   # 1! = 1
print(fact(5))   # 5! = 120
print(fact(10))  # 10! = 3628800

#n번의 곱셈이 필요함
```

```
#재귀호출 코드
def fact(n):
    if n <= 1: #종료 조건(종료 조건이 없으면 메모리 부족 에러 발생)
        return 1
    return n * fact(n - 1)

print(fact(1))   # 1! = 1
print(fact(5))   # 5! = 120
print(fact(10))  # 10! = 3628800
```

```
#1~n의 합 계산을 재귀호출로 처리
# 연속한 숫자의 합을 구하는 알고리즘
# 입력: n
# 출력: 1부터 n까지 연속한 숫자를 더한 합

def sum_n(n):
    if n == 0:
        return 0
    return sum_n(n - 1) + n

print(sum_n(10))   # 1부터 10까지의 합(입력:10, 출력:55)
print(sum_n(100))  # 1부터 100까지의 합(입력:100, 출력:5050)
```

```
#n개의 숫자 중에서 최대값 찾기(재귀호출)
# 최대값 구하기
# 입력: 숫자가 n개 들어 있는 리스트
# 출력: 숫자 n개 중 최대값

def find_max(a, n):  # 리스트 a의 앞부분 n개 중 최대값을 구하는 재귀 함수
    if n == 1:
        return a[0]
    max_n_1 = find_max(a, n - 1) # n-1개 중 최대값을 구함
    if max_n_1 > a[n - 1]:       # n-1개 중 최대값과 n-1번 위치 값을 비교
        return max_n_1
    else:
        return a[n - 1]

v = [17, 92, 18, 33, 58, 7, 33, 42]
print(find_max(v, len(v)))  # 함수에 리스트의 자료 개수를 인자로 추가하여 호출
```

1.7. 최대공약수

두 개 이상의 정수의 공통 약수 중에서 가장 큰 값
1. 두 수의 약수를 구한다.
2. 공통된 약수를 구한다.
3. 그 중 가장 큰 값을 구한다.

1. 두 수 중 더 작은 값을 i에 저장
2. i가 두 수의 공통된 약수인지 확인
3. 공통된 약수이면 그 값을 리턴하고 종료
4. 공통된 약수가 아니면 i를 1 감소시키고 2번으로 돌아간다.

예를 들어 4와 6의 최대공약수를 구한다면
1. 두 수 중 더 작은 값 4를 i에 저장
2. i가 두 수의 공통된 약수인지 확인 - 4는 i로 나누어떨어지지만 6으로는 나누어 떨어지지 않음
3. i를 1 감소시킨다.(3)
4. 4는 i(3)로 나누어떨어지지 않는다.
5. i를 1 감소시킨다.(2)
6. 4는 i(2)로 나누어 떨어지고 6도 i(2)로 나누어떨어지므로 2를 리턴하고 종료시킨다.

```
# 최대공약수 구하기
# 입력: a, b
# 출력: a와 b의 최대공약수

def gcd(a, b):
    i = min(a, b) # 두 수 중에서 최소값을 구하는 파이썬 함수
    while True:
        if a % i == 0 and b % i == 0:
            return i
        i = i - 1  # i를 1만큼 감소시킴

print(gcd(1, 5))    # 1
print(gcd(3, 6))    # 3
print(gcd(60, 24))  # 12
print(gcd(81, 27))  # 27
```

유클리드 알고리즘
수학자 유클리드가 발견한 최대공약수의 특징
1.a와 b의 최대공약수는 b와 a를 b로 나눈 나머지의 최대공약수와 같다.
gcd(a,b)==gcd(b,a%b)
2. 어떤 수와 0의 최대공약수는 자기 자신이다.
gcd(n,0)==n

gcd(10,5)==gcd(5,10%5)==gcd(5,0)==>5
gcd(20,4)==gcd(4,20%4)==gcd(4,0)==>4
어떤 두 수의 최대공약수를 구하기 위해 다시 다른 두 수의 최대공약수를 구하고 있다(재귀호출을 사용할 수 있는 경우)

```
# 최대공약수 구하기
# 입력: a, b
# 출력: a와 b의 최대공약수

def gcd(a, b):
    if b == 0:  # 종료 조건
        return a
    return gcd(b, a % b)  # 좀 더 작은 값으로 자기 자신을 호출

print(gcd(1, 5))    # 1
print(gcd(3, 6))    # 3
print(gcd(10, 5))   # 5
print(gcd(20, 4))   # 4
print(gcd(60, 24))  # 12
print(gcd(81, 27))  # 27
```

피보나치 수열 : 0과 1부터 시작해서 바로 앞의 두 수를 더한 값을 다음 값으로 추가하는 방식으로 만든 수열

0+1=1, 1+1=2, 1+2=3, 2+3=5

0,1,1,2,3,5

n번째 피보나치 수를 구하는 코드를 완성하시오. (0부터 시작)

```
# n번째 피보나치 수열 찾기
# 입력: n값(0부터 시작)
# 출력: n번째 피보나치 수열 값

def fibo(n):
    if n <= 1:
        return n  # n = 0 -> 0 | n = 1 -> 1
    #예를 들어 7번값=5번값 + 6번값
    #          n    n-2     n-1
    return fibo(n - 2) + fibo(n - 1)

print(fibo(0))
print(fibo(1))
print(fibo(2))
print(fibo(3))
print(fibo(7))
print(fibo(10))
```

1.8. 순차탐색

#순차 탐색(Linear Search)
#리스트에 특정한 값이 있는지 찾아서 그 위치를 리턴, 찾는 값이 없으면 -1을 리턴

```
# 리스트에서 특정 숫자 위치 찾기
# 입력: 리스트 a, 찾는 값 x
# 출력: 찾으면 그 값의 위치, 찾지 못하면 -1

def search_list(a, x):
    n = len(a)  # 입력 크기 n
    for i in range(0, n):  # 리스트 a의 모든 값을 차례로
        if x == a[i]:      # x 값과 비교하여
            return i       # 같으면 위치 돌려줍니다.

    return -1   # 끝까지 비교해서 없으면 -1을 돌려줍니다.

v = [17, 92, 18, 33, 58, 7, 33, 42]
print(search_list(v, 42))
print(search_list(v, 33))
print(search_list(v, 900))  # -1(900은 리스트에 없음)

print(v.index(42))
print(v.index(33))
print(v.index(900))
```

#실행횟수는 경우에 따라 다름
최선의 경우 , 평균적인 경우, 최악의 경우
보수적인 관점에서 최악의 경우로 계산하면 n번

#1개만이 아닌 여러개를 모두 리턴하고 없으면 []을 리턴하는 코드

```
# 리스트에서 특정 숫자 위치 전부 찾기
# 입력: 리스트 a, 찾는 값 x
# 출력: 찾는 값의 위치 번호가 담긴 리스트, 찾는 값이 없으면 빈 리스트 []

def search_list(a, x):
    n = len(a)  # 입력 크기 n
    result = []  # 새 리스트를 만들어 결과값을 저장
    for i in range(0, n):     # 리스트 a의 모든 값을 차례로
        if x == a[i]:         # x값과 비교하여
            result.append(i)  # 같으면 위치 인덱스값을 결과 리스트에 추가

    return result  # 만들어진 결과 리스트를 돌려줌

v = [17, 92, 16, 33, 58, 7, 33, 42]
print(search_list(v, 92))   # [1]
print(search_list(v, 33)) # [3, 6] (33은 리스트에 두 번 나옴)
print(search_list(v, 900))  # []    (900은 리스트에 없음)
```

```
#학생번호와 이름이 있을 때 학생번호를 입력하면 해당하는 이름을 출력하는 코드
# 학생 번호에 해당하는 학생 이름 찾기
# 입력: 학생 번호 리스트 s_no, 학생 이름 리스트 s_name, 찾는 학생 번호 find_no
# 출력: 해당하는 학생 이름, 학생 이름이 없으면 물음표 "?"

def get_name(s_no, s_name, find_no):
    n = len(s_no)  # 입력 크기 n
    for i in range(0, n):
        if find_no == s_no[i]: # 학생 번호가 찾는 학생 번호와 같으면
            return s_name[i] # 해당하는 학생 이름을 결과로 반환

    return "?"  # 자료가 없으면 물음표 반환

nums = [39, 14, 67, 105]
names = ["김철수", "최상호", "이미영", "한선영"]
print(get_name(nums, names, 67))
print(get_name(nums, names, 70))
```

1.9. 거품정렬

```
a=[2,3,5,7,9,1]
b=sorted(a)
print(a) #a는 정렬되지 않음
print(b) #정렬된 값을 b에 저장
```

```
a=[2,3,5,7,9,1]
b=a.sort()
print(a) #a 자체가 정렬됨
print(b) #리턴값은 없음
```

```
# 거품 정렬
# 입력: 리스트 a
# 출력: 없음(입력으로 주어진 a가 정렬됨)

# 매우 느리지만 코드가 단순하여 자주 사용됨
# 원소의 이동이 거품이 수면으로 올라오는 듯한 모습을 보이기에 거품정렬이라고 함

def bubble_sort(a):
    for i in range(0,len(a)):
        for j in range(i+1,len(a)):
            if a[i] > a[j]:
                a[i],a[j]=a[j],a[i]
            print(a)

    return a

d = [2, 4, 5, 1, 3]
print(bubble_sort(d))
```

1.10. 선택정렬

운동장에서 키 순서대로 서기
초등학교에서 학생 10명을 키 순서대로 세우기
키가 제일 작은 학생을 맨 앞에 세우고
남은 9명의 학생 중 제일 작은 학생을 두번째에 세우고

```
# 쉽게 설명한 선택 정렬
# 입력: 리스트 a
# 출력: 정렬된 새 리스트

# 주어진 리스트에서 최소값의 위치를 돌려주는 함수
def find_min_idx(a):
    n = len(a)
    min_idx = 0
    for i in range(1, n):
        if a[i] < a[min_idx]:
            min_idx = i
    return min_idx

def sel_sort(a):
    result = []  # 새 리스트를 만들어 정렬된 값을 저장
    while a:     # 주어진 리스트에 값이 남아있는 동안 계속
        min_idx = find_min_idx(a)  # 리스트에 남아 있는 값 중 최소값의 위치
        value = a.pop(min_idx)    # 찾은 최소값을 빼내어 value에 저장
        result.append(value)   # value를 결과 리스트 끝에 추가
    return result

d = [2, 4, 5, 1, 3]
print(sel_sort(d))
```

```
# 선택 정렬(일반적인 코드)
# 입력: 리스트 a
# 출력: 없음(입력으로 주어진 a가 정렬됨)

def sel_sort(a):
    n = len(a)
    for i in range(0, n - 1):  # 0부터 n-2까지 반복
        # i번 위치부터 끝까지 자료 값 중 최소값의 위치를 찾음
        min_idx = i
        for j in range(i + 1, n):
            if a[j] < a[min_idx]:
                min_idx = j
            print(i,j,a)
        # 찾은 최소값을 i번 위치로
        a[i], a[min_idx] = a[min_idx], a[i]  #두 값을 바꾸기
        print(i,j,a)


d = [2, 4, 5, 1, 3]
sel_sort(d)
print(d)
```

장점 : 이해하기 쉽고 간단한 알고리즘
단점 : 입력 크기가 커질수록 정렬에 시간이 많이 걸린다.

```
# 내림차순 선택 정렬
# 입력: 리스트 a
# 출력: 없음(입력으로 주어진 a가 정렬됨)

def sel_sort(a):
    n = len(a)
    for i in range(0, n - 1):
        max_idx = i  # 최소값(min) 대신 최대값(max)을 찾아야 함
        for j in range(i + 1, n):
            if a[j] > a[max_idx]:  # 부등호 방향 뒤집기
                max_idx = j
        a[i], a[max_idx] = a[max_idx], a[i]

d = [2, 4, 5, 1, 3]
sel_sort(d)
print(d)
```

1.11. 삽입정렬

한 명이 맨 앞에 서고
나머지 9명 중 한 명과 비교해서 앞 또는 뒤에 서고(앞에 서면 삽입)
나머지 8명 중 한 명과 비교해서 맨 앞, 중간, 뒤에 서고

[2 4 5 1 3]
2를 세우고
4의 위치는 2 뒤
[2 4]
5를 꺼내서 위치는 맨 뒤
[2 4 5]
1을 꺼내보니 위치는 맨 앞
[1 2 4 5] 삽입
3을 꺼내보니 위치는 세번째
[1 2 3 4 5] 삽입

```
# 쉽게 설명한 삽입 정렬
# 입력: 리스트 a
# 출력: 정렬된 새 리스트

# 리스트 r에서 v가 들어가야 할 위치를 돌려주는 함수
def find_ins_idx(r, v):
    # 이미 정렬된 리스트 r의 자료를 앞에서부터 차례로 확인하여
    for i in range(0, len(r)):
        # v 값보다 i번 위치에 있는 자료 값이 크면
        # v가 그 값 바로 앞에 놓여야 정렬 순서가 유지됨
        if v < r[i]:
            return i
    # 적절한 위치를 못 찾았을 때는
    # v가 r의 모든 자료보다 크다는 뜻이므로 맨 뒤에 삽입
    return len(r)

def ins_sort(a):
    result = []  # 새 리스트를 만들어 정렬된 값을 저장
    while a:     # 기존 리스트에 값이 남아 있는 동안 반복
        value = a.pop(0) # 기존 리스트에서 한 개를 꺼냄
        ins_idx = find_ins_idx(result, value)  # 꺼낸 값이 들
어갈 적당한 위치 찾기
        print(ins_idx, value)
        result.insert(ins_idx, value)  # 찾은 위치에 값 삽입
(이후 값은 한 칸씩 밀려남)
    return result

d = [2, 4, 5, 1, 3]
print(ins_sort(d))
```

```
# 삽입 정렬(일반적인 코드)
# 입력: 리스트 a
# 출력: 없음(입력으로 주어진 a가 정렬됨)

def ins_sort(a):
    n = len(a)
    for i in range(1, n):  # 1부터 n-1까지
        # i번 위치의 값을 key로 저장
        key = a[i]
        # j를 i 바로 왼쪽 위치로 저장
        j = i - 1
        # 리스트의 j번 위치에 있는 값과 key를 비교해 key가 삽입
될 적절한 위치를 찾음
        while j >= 0 and a[j] > key:
            a[j + 1] = a[j]  # 삽입할 공간이 생기도록 값을 오른
쪽으로 한 칸 이동
            j -= 1
        a[j + 1] = key  # 찾은 삽입 위치에 key를 저장
        print(a)

d = [5, 1, 3, 2, 4]
ins_sort(d)
print(d)
```

```
[5 1 3 2 4]
5
1 5
1 3 5
1 2 3 5
1 2 3 4 5
```

```
# 내림차순 삽입 정렬
# 입력: 리스트 a
# 출력: 없음(입력으로 주어진 a가 정렬됨)

def ins_sort(a):
    n = len(a)
    for i in range(1, n):
        key = a[i]
        j = i - 1
        while j >= 0 and a[j] < key:  # 부등호 방향 뒤집기
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key

d = [2, 4, 5, 1, 3]
ins_sort(d)
print(d)
```

1.12. 퀵정렬

```
#합계계산
def mysum(arr):
  total = 0
  for x in arr:
    total += x
  return total

mysum([1, 2, 3, 4])
```

```
#합계계산(재귀호출)
def mysum(items):
  if items == []:
    return 0
  return items[0] + mysum(items[1:])

mysum([1, 2, 3, 4])

# 1+mysum(2,3,4)
#   2+mysum(3,4)
#     3+mysum(4)
#       4+0
```

```
#갯수세기(재귀)
def mycount(items):
  if items == []:
    return 0
  return 1 + mycount(items[1:])

mycount([1, 2, 3, 4])
```

```
#퀵정렬
def quicksort(array):
    #배열의 사이즈가 0 또는 1인 경우는 그대로 리턴함
  if len(array) < 2:
    return array
  else: #배열의 사이즈가 2보다 큰 경우
    pivot = array[0] #배열의 기준원소를 0번으로 선택
    # 기준원소보다 작은 값들
    less = [i for i in array[1:] if i <= pivot]
    # 기준원소보다 큰 값들
    greater = [i for i in array[1:] if i > pivot]
    return quicksort(less) + [pivot] + quicksort(greater)

quicksort([-10, -15, 2, 3, 0, 9, 11, 16, 1])
```

1.13. 이진탐색

500페이지 책에서 200페이지를 찾으려면?
중간쯤 펴니 240페이지
앞부분 중간쯤 펴니 120페이지
또 중간쯤 펴니 200 페이지

미리 정렬되어 있다는 가정 하에 가능

호텔 호수 찾는 방법

204호라면 중간쯤 보니 206호 앞 라인 찾아간다.

[1,4,9,16,25,36,49,64,81]

먼저 중간 위치를 찾는다.
찾는 값과 중간 위치값 비교
같으면 종료
찾는 값이 중간 위치값보다 크면 오른쪽 탐색
찾는 값이 중간 위치값보다 작으면 왼쪽 탐색

```
# 리스트에서 특정 숫자 위치 찾기
# 입력: 리스트 a, 찾는 값 x
# 출력: 찾으면 그 값의 위치, 찾지 못하면 -1
def binary_search(a, x):
    # 탐색할 범위를 저장하는 변수 start, end
    # 리스트 전체를 범위로 탐색 시작(0 ~ len(a)-1)
    start = 0
    end = len(a) - 1
    cnt=0
    while start <= end: # 탐색할 범위가 남아 있는 동안 반복
        mid = (start + end) // 2  # 탐색 범위의 중간 위치
        if x == a[mid]:   # 발견!
            return mid, a[mid], cnt #인덱스, 찾은값, 검사횟수
        elif x > a[mid]:  # 찾는 값이 더 크면 오른쪽으로 범위를
좁혀 계속 탐색
            start = mid + 1
        else:#찾는 값이 더 작으면 왼쪽으로 범위를 좁혀 계속 탐색
            end = mid - 1
        cnt+=1
    return -1  # 찾지 못했을 때

d = [1, 4, 9, 16, 25, 36, 49, 64, 81]
print(binary_search(d, 36))
print(binary_search(d, 50))
```

자료가 1000개 있을 경우 순차탐색은 최악의 경우 1000번 모두 비교해야 함, 이진탐색은 10번만 비교하면 됨

주민등록번호 탐색을 순차탐색으로 한다면 최악의 경우 5000만번 비교를 해야 함

이진탐색으로는 26번 정도를 비교하면 됨

이진 탐색을 하려면 자료를 미리 정렬해야 함

1.14. 자료구조(큐와 스택)

회문 찾기
거꾸로 읽어도 내용이 같은 낱말이나 문장

역삼역, 기러기, 일요일, 사진사
mom, wow, noon, level, radar

큐(queue) - 줄서기, First In First Out
    입력-인큐(enqueue), 출력-디큐(dequeue)

스택(stack) - 접시쌓기, Last In First Out
    입력-푸시(push), 출력-팝(pop)

큐에서 꺼낸 문자들이 스택에서 꺼낸 문자들과 같다면 그 문장은 회문

```
# 주어진 문장이 회문인지 아닌지 찾기(큐와 스택의 특징을 이용)
# 입력: 문자열 s
# 출력: 회문이면 True, 아니면 False

def palindrome(s):
    # 큐와 스택을 리스트로 정의
    qu = []
    st = []
    # 1단계: 문자열의 알파벳 문자를 각각 큐와 스택에 넣음
    for x in s:
        # 해당 문자가 공백, 특수문자, 숫자가 아니면
        # 큐와 스택에 각각 그 문자를 추가
        if x.isalpha():
            qu.append(x.lower())
            st.append(x.lower())
    # 2단계: 큐와 스택에 들어 있는 문자를 꺼내면서 비교
    while qu:  # 큐에 문자가 남아 있는 동안 반복
        #pop(0) 맨 앞의 글자를 출력시킴, 리스트의 맨 앞의 자료를 빼내면 0번 위치가 비게 되므로 남은 자료를 전부 한 칸씩 당겨주는 처리가 필요함
        #pop() 맨 뒤의 글자를 출력시킴
        if qu.pop(0) != st.pop():  # 큐와 스택에서 꺼낸 문자가 다르면 회문이 아님
            return False
    return True

print(palindrome("study"))
print(palindrome("Wow"))
print(palindrome("나는 학교에 간다"))
print(palindrome("기러기"))
```

1.15. 자료구조(딕셔너리)

동명이인 찾기
리스트에서 이름을 하나씩 꺼낸다.
주어진 이름이 딕셔너리에 있으면 카운트를 1증가시킨다.
주어진 이름이 딕셔너리에 없으면 그 이름을 키로 하는 항목을 새로 만들고 1을 저장
만들어진 딕셔너리에서 등장횟수가 2이상인 이름을 찾아 출력

```
# 두 번 이상 나온 이름 찾기
# 입력: 이름이 n개 들어 있는 리스트
# 출력: n개의 이름 중 반복되는 이름의 집합

def find_same_name(a):
    # 1단계: 각 이름의 등장 횟수를 딕셔너리로 만듦
    name_dict = {}
    for name in a:  # 리스트 a에 있는 자료들을 차례로 반복
        if name in name_dict:     # 이름이 name_dict에 있으면
            name_dict[name] += 1  # 등장 횟수를 1 증가
        else:                     # 새 이름이면
            name_dict[name] = 1   # 등장 횟수를 1로 저장

    # 2단계: 만들어진 딕셔너리에서 등장 횟수가 2 이상인 것을 결과에 추가
    result = []  # 결과값을 저장할 리스트
    for name in name_dict:  # 딕셔너리 name_dict에 있는 자료들을 차례로 반복
        if name_dict[name] >= 2:
            result.append(name)

    return result

name = ["Tom", "Jerry", "Mike", "Tom"]  # 대소문자 유의: 파이썬은 대소문자를 구분함
print(find_same_name(name))

name2 = ["김철수", "이혜진", "송민호", "이혜진", "김철수"]
print(find_same_name(name2))
```

```
# 학생 번호에 해당하는 학생 이름 찾기(dict 이용)
# 입력: 학생 명부 딕셔너리 s_info, 찾는 학생 번호 find_no
# 출력: 해당하는 학생 이름, 해당하는 학생 번호가 없으면 물음표
"?"

def get_name(s_info, find_no):
    if find_no in s_info:
        return s_info[find_no]
    else:
        return "?"  # 해당하는 번호가 없으면 물음표

sample_info = {
    39: "김철수",
    14: "홍명수",
    67: "이혜성",
    105: "송미진"
}

print(get_name(sample_info, 105))
print(get_name(sample_info, 777))
```

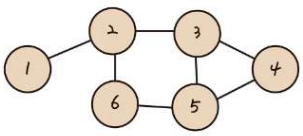
1.16. 그래프

친구의 친구 찾기
sns에서 제공하는 친구관계, 친구 추천 기능에 활용되는 알고리즘

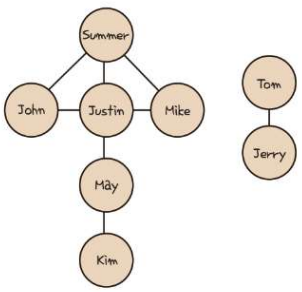
친구(일촌) - 두 사람이 서로 친구 요청을 수락한 경우
친구(친척) - 직접 아는 친구들과 그 친구들의 친구들, 즉 직간접으로 아는 모든 사람을 포함하는 개념
친밀도(촌수) - 서로 몇 단계를 거쳐 아는지를 나타내는 숫자

그래프 탐색 알고리즘 : 그래프에 연결된 모든 노드를 탐색하는 알고리즘

그래프 : 노드(둥그라미)와 엣지(연결선)의 집합



친구관계 그래프



```
# 친구 리스트에서 자신의 모든 친구를 찾는 알고리즘
# 입력: 친구 관계 그래프 g, 모든 친구를 찾을 자신 start
# 출력: 모든 친구의 이름

def print_all_friends(g, start):
    qu = []       # 기억 장소1: 앞으로 처리해야 할 사람들을 큐에 저장
    done = set()  # 기억 장소2: 이미 큐에 추가한 사람들을 집합에 기록(중복 방지)

    qu.append(start)  # 자신을 큐에 넣고 시작
    done.add(start)   # 집합에도 추가

    while qu:           # 큐에 처리할 사람이 남아 있는 동안
        p = qu.pop(0)   # 큐에서 처리 대상을 한 명 꺼내
        print(p)        # 이름을 출력하고
        for x in g[p]:  # 그의 친구들 중에
            if x not in done:  # 아직 큐에 추가된 적이 없는 사람을
                qu.append(x)   # 큐에 추가하고
                done.add(x)    # 집합에도 추가

# 친구 관계 리스트
# A와 B가 친구이면
# A의 친구 리스트에도 B가 나오고, B의 친구 리스트에도 A가 나옴
fr_info = {
    '김철수': ['홍수찬', '김순기', '이미영'],
    '김순기': ['김철수', '한도훈'],
    '한도훈': ['이미리', '김철수', '김순기', '설진수'],
    '홍수찬': ['이미리', '최진우'],
    '이미영': ['홍수찬', '이미리'],
```

```
    '설진수': ['한도훈'],
    '최진우': ['설진수'],
    '이미리': ['이미영']
}

print_all_friends(fr_info, '한도훈') #한도훈의 친구들
print()
print_all_friends(fr_info, '김철수')
```

```
# 친구 리스트에서 자신의 모든 친구를 찾고 친구들의 친밀도를 계산하는 알고리즘
# 입력: 친구 관계 그래프 g, 모든 친구를 찾을 자신 start
# 출력: 모든 친구의 이름과 자신과의 친밀도

def print_all_friends(g, start):
    qu = []       # 기억 장소 1: 앞으로 처리해야 할 (사람 이름, 친밀도) 튜플을 큐에 저장
    done = set()  # 기억 장소 2: 이미 큐에 추가한 사람을 집합에 기록(중복 방지)

    qu.append((start, 0))  # (사람 이름, 친밀도) 정보를 하나의 튜플로 묶어 처리(자기 자신 친밀도: 0)
    done.add(start)        # 집합에도 추가

    while qu:              # 큐에 처리할 사람이 남아 있는 동안
       # 큐에서 (사람 이름, 친밀도) 정보를 p와 d로 각각 꺼냄
        (p, d) = qu.pop(0)
        print(p, d)        # 사람 이름과 친밀도를 출력
        for x in g[p]:     # 친구들 중에
            # 아직 큐에 추가된 적이 없는 사람을
            if x not in done:
                #튜플을 사용하기 위해 이중괄호를 사용함
                # 친밀도를 1 증가시켜 큐에 추가하고
                qu.append((x, d + 1))
                done.add(x)           # 집합에도 추가
```

```
fr_info = {
    '김철수': ['홍수찬', '김순기', '이미영'],
    '김순기': ['김철수', '한도훈'],
    '한도훈': ['이미리', '김철수', '김순기', '설진수'],
    '홍수찬': ['이미리', '최진우'],
    '이미영': ['홍수찬', '이미리'],
    '설진수': ['한도훈'],
    '최진우': ['설진수'],
    '이미리': ['이미영']
}

print_all_friends(fr_info, '한도훈')
print()
print_all_friends(fr_info, '김철수')
```