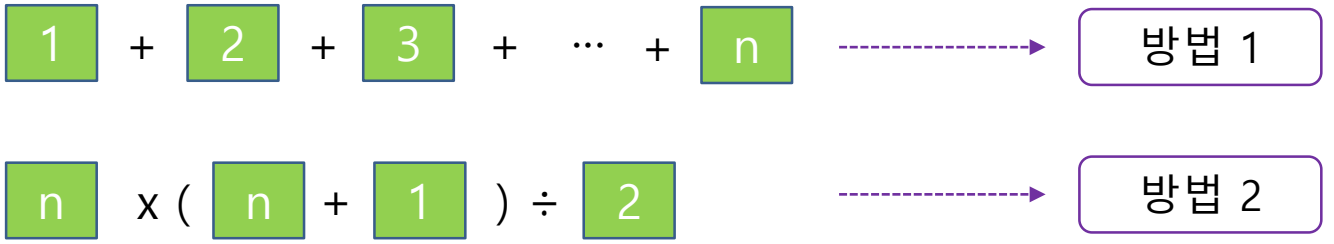
8장. 알고리즘과 자료구조

알고리즘 계산 복잡도

- 1부터 n까지의 합을 구하기




```python
def sum_n(n):
    total = 0  # 합계
    for i in range(1, n + 1):
        total += i
    return total

def sum_n2(n):
    total = (n * (n + 1)) // 2
    return total
```

```python
print(sum_n(10))
print(sum_n2(10))
```

알고리즘 계산 복잡도

➢ 계산 복잡도

- **입력 크기와 계산 횟수**
  - 첫 번째 알고리즘 : 덧셈 n번
  - 두 번째 알고리즘 : 덧셈, 곱셈, 나눗셈(총 3번)

- **대문자 O표기법(Big O) : 계산 복잡도 표현**
  - O(n) : 필요한 계산횟수가 입력 크기 n과 비례할 때
  - O(1) : 필요한 계산횟수가 입력 크기 n과 무관할 때
- **판단**
  - 두번째 방법이 계산 속도가 더 빠름

재귀 함수(recursive function)

- **재귀 함수(recursive function)**

  어떤 함수 안에서 자기 자신을 부르는 것을 말한다.

  재귀호출은 무한 반복하므로 **종료 조건**이 필요함

```
def func(입력 값):
    if 입력값이 충분히 작으면: #종료 조건
        return 결과값
    else:      # 더 작은 값으로 호출
        return 결과값
```

재귀 함수(recursive function)

- 재귀 호출: 다시 돌아가 부르기

```
def sos(n):
  if n <= 0:
    return ''
  else:
    print("Help me!")
    return sos(n - 1)

  '''
  print("Help me!")
  n -= 1
  if n > 0:
    sos(n)
  '''

sos(1)
sos(4)
```

팩토리얼(factorial)

- 팩토리얼을 구하는 재귀 함수

```
def facto(n):
    gob = 1
    for i in range(1, n + 1):
        gob *= i
    return gob

def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * facto(n - 1)
```

팩토리얼(factorial)

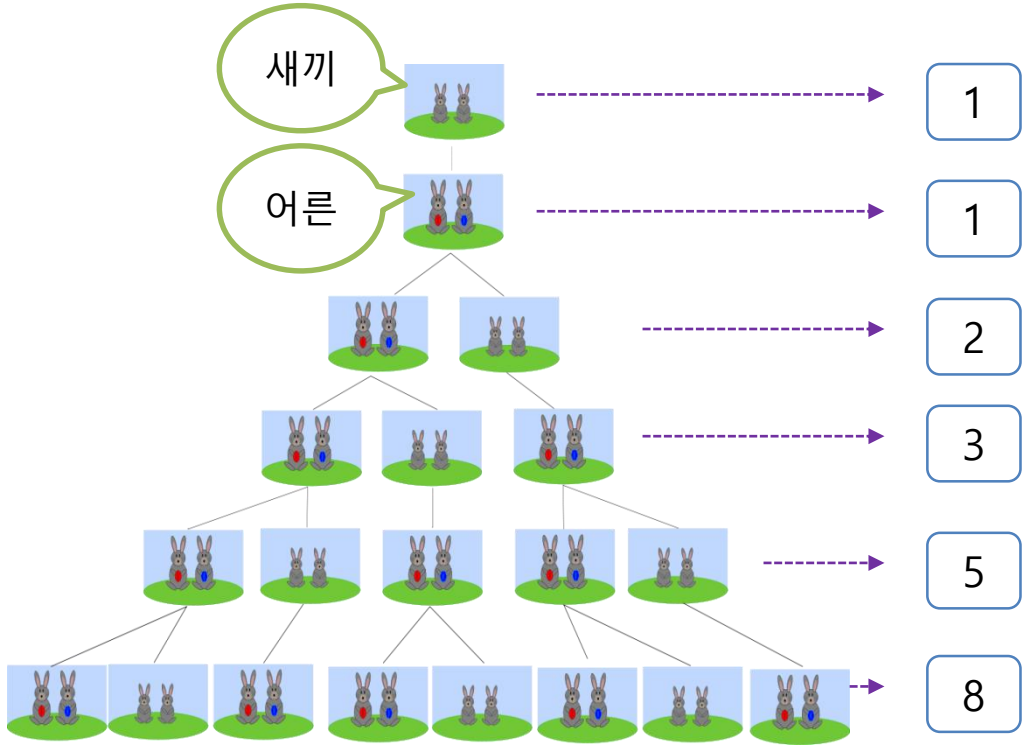
- 팩토리얼을 구하는 재귀 함수

```
'''
  n=4, 4 * facto(3) - 4 * 6
  n=3, 3 * facto(2) - 3 * 2
  n=2, 2 * facto(1) - 2 * 1
  n=1, 1 * facto(0) - 1 * 1

'''


# facto() 호출
print(facto(1))
print(facto(4))

# factorial() 호출
print(factorial(1))
print(factorial(4))
```

피보나치 수열

- **피보나치(Fibonacci) 수열**

  수학에서 피보나치 수는 첫째 및 둘째 항이 1이며, 그 뒤의 모든 항은 바로 앞 두 항의 합인 수열이다. 처음 여섯 항은 각각 1, 1, 2, 3, 5, 8이다.



첫번째 달에 새로 태어난 토끼 한쌍이 있고, 둘째달에 토끼가 커서 그대로 어른토끼 한쌍, 세째달에는 새끼를 한쌍 낳아 어른, 새끼 두쌍, 네째달에는 어른이 새끼를 낳고, 새끼는 어른이 되어 총 세쌍, 이렇게 계속 새끼를 낳고, 죽지 않는다는 가정을 세우면 피보나치의 수가 된다.

피보나치 수열

- **피보나치 수열 – 재귀 함수**

```
# 재귀 호출 - 피보나치 수열
# 1, 1, 2, 3, 5, 8...
# 세째항 = 첫째항 + 둘째항
def fibo(n):
    if n <= 2:
        return 1
    else:
        return fibo(n - 2) + fibo(n - 1)

'''
  n = 1  fibo(1) = 1
  n = 2  fibo(2) = 1
  n = 3  fibo(3) = fibo(1) + fibo(2) = 1 + 1 = 2
  n = 4  fibo(4) = fibo(2) + fibo(3) = 1 + 2 = 3
'''

print(fibo(1))
print(fibo(2))
print(fibo(3))
print(fibo(4))
```

정렬 – 버블정렬

- 리스트의 정렬 – sort() 함수

```
a = [60, 5, 33, 12, 97, 24]

#1. 내장 함수 sort()
a.sort() # 오름차순 정렬
print(a)
a.sort(reverse=True) #내림 차순
print(a)
```

정렬 – 버블정렬

- **버블 정렬**

  리스트에서 인접한 두 개의 요소를 비교하여 자리를 바꾸어 정렬하는 알고리즘이다.

```
a = [41, 8, 36, 77, 15]

for i in range(0, 5):
  for j in range(0, 4):
    if a[j] > a[j+1]:
      a[j], a[j+1] = a[j+1], a[j]
```

정렬 – 버블정렬

- 버블 정렬

```
'''
  i=0, j=0, 41>8, [8 41 36 77 15]
       j=1, 41>36, [8 36 41 77 15]
       j=2, 41>77, 교환없음
       j=3, 77>15, [8 36 41 15 77]
  i=1, j=0, 8>36, 교환없음
       j=1, 36>41, 교환없음
       j=2, 41>15, [8 36 15 41 77]
  i=2, j=0, 8>36, 교환없음
       j=1, 36>15, [8 15 36 41 77] - 오름차순 정렬
  i=3, 교환없음
  i=4, 교환없음
  i=5, 반복종료
'''

print(a)

# 전체 요소 출력
for val in a:
  print(val, end=' ')
```

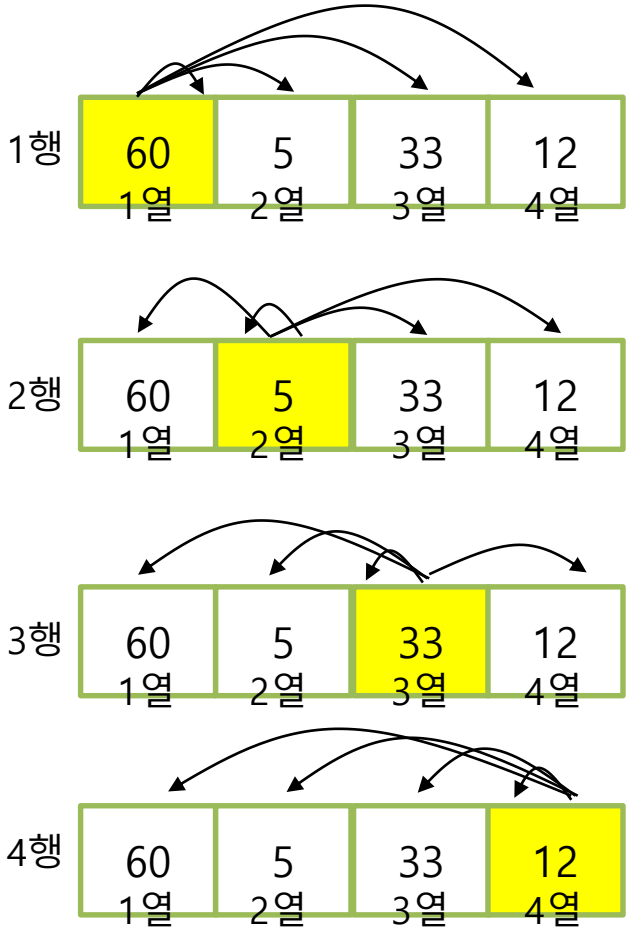
정렬 – 버블정렬

- 버블 정렬 – 함수로 구현

```
def sort_bubble(a):
    n = len(a)
    for i in range(0, n):
        for j in range(0, n-1):
            if a[j] > a[j+1]:
                a[j], a[j+1] = a[j+1], a[j]
    return a

a = [41, 8, 36, 77, 15]

# a 출력
print(sort_bubble(a)) #[8, 15, 36, 41, 77]
```

순위 - ranking

♥ 순위 정하기



60이 60, 5, 33, 12와 대소비교

5가 60, 5, 33, 12와 대소비교

33이 60, 5, 33, 12와 대소비교

12가 60, 5, 33, 12와 대소비교

순위 - ranking

♥ 순위 정하기

```
score = [60, 5, 33, 12, 97, 24]
rank = [1, 1, 1, 1, 1, 1]  # 순위를 모두 1로 설정
n = len(score)

for i in range(0, n):
    for j in range(0, n):
        if score[i] < score[j]:
            rank[i] = rank[i] + 1  # 순위 1증가

print(rank)
```

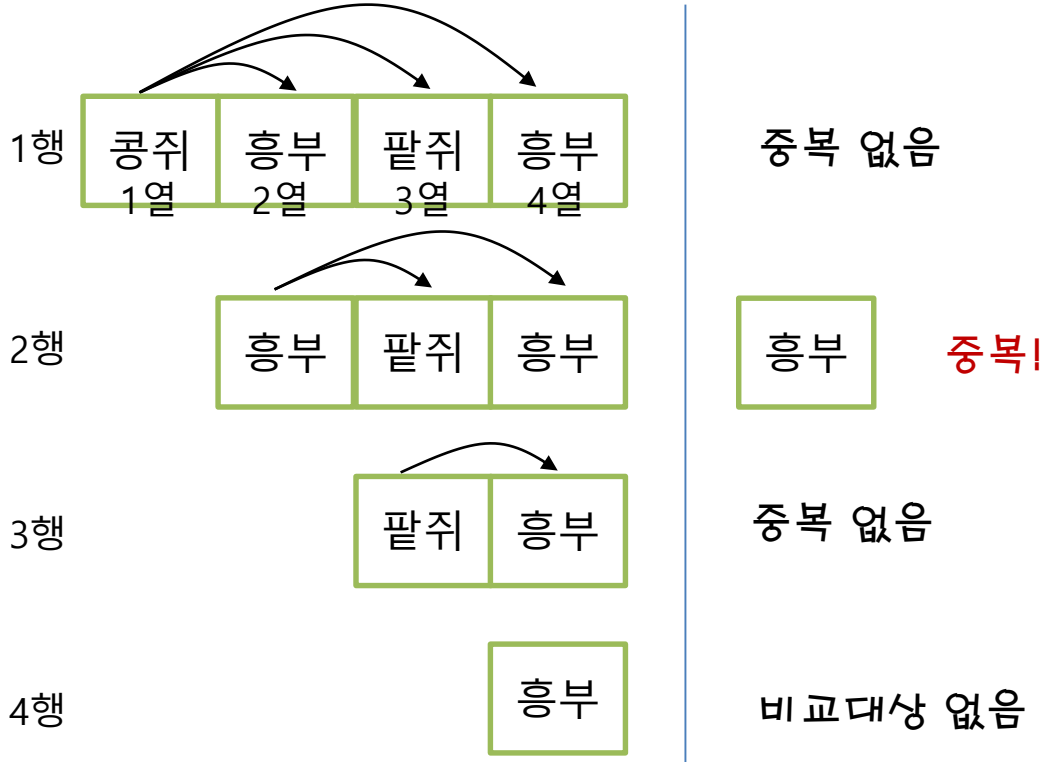
순위 - ranking

♥ 순위 정하기

```
'''
i=0 j=0 score[0] < score[0] False  rank[0] 1
    j=1 score[0] < score[1] False  rank[0] 1
    j=2 score[0] < score[2] False  rank[0] 1
    j=3 score[0] < score[3] False  rank[0] 1
    j=4 score[0] < score[4] True   rank[0] 2
    j=5 score[0] < score[5] False  rank[0] 2(순위 확정)
i=1 j=0 score[1] < score[0] True  rank[1] 2
    j=1 score[1] < score[1] False  rank[1] 2
    j=2 score[1] < score[2] True  rank[1] 3
    j=3 score[1] < score[3] True  rank[1] 4
    j=4 score[1] < score[4] True  rank[1] 5
    j=5 score[1] < score[5] True  rank[1] 6(순위 확정)
i=2 j=0 score[2] < score[0] True  rank[2] 2
    j=1 score[2] < score[1] False  rank[2] 2
    j=2 score[2] < score[2] False  rank[2] 2
    j=3 score[2] < score[3] False  rank[2] 2
    j=4 score[2] < score[4] True  rank[2] 3
    j=5 score[2] < score[5] False  rank[2] 3(순위 확정)
'''
```

동명이인 찾기 – 중복 검사

♥ 같은 이름 찾기



동명이인 찾기 – 중복 검사

♥ 같은 이름 찾기

```
# 리스트로 구현하기
def find_same_name(a):
    same_name = []
    n = len(a)
    for i in range(0, n-1):
        for j in range(i + 1, n):
            if a[i] == a[j]:
                same_name.append(a[i])
    return same_name

name = ['콩쥐', '흥부', '팥쥐', '흥부']
result = find_same_name(name)
print(result)
```

```
"""
  i=0,
        j=1, a[0] == a[1], False
        j=2, a[0] == a[2], False
        j=3, a[0] == a[3], False
  i=1,
        j=2, a[1] == a[2], False
        j=3, a[1] == a[3], True, 중복
  i=2,
        j=3, a[2] == a[3], False,
  i=3, 반복 종료
"""
```

동명이인 찾기 – 중복 검사

♥ *같은 이름 찾기*

```
# set(집합)으로 구현하기
def find_same_name(a):
    same_name = set()
    n = len(a)
    for i in range(0, n-1):
        for j in range(i + 1, n):
            if a[i] == a[j]:
                same_name.add(a[i])
    return same_name

name = ['콩쥐', '흥부', '팥쥐', '흥부']
result2 = find_same_name(name)
print(result2)
```

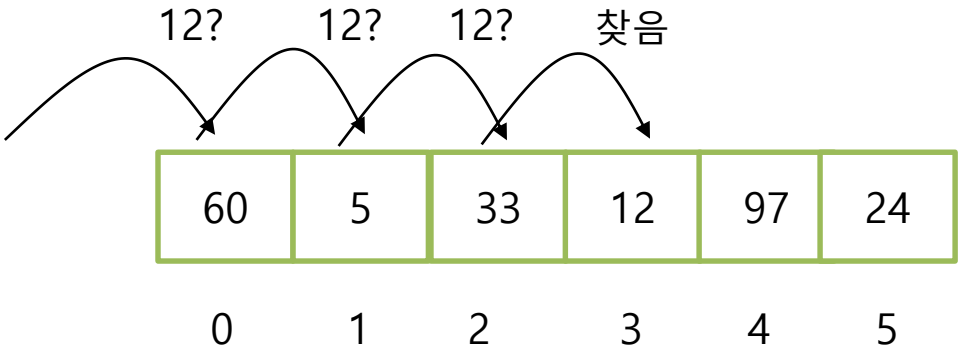
```
['흥부']
{'흥부'}
```

순차 탐색

➢ 순차탐색

주어진 리스트에 특정한 값이 있는지 찾아 그 위치를 돌려주는 알고리즘

리스트에 있는 첫번째 자료부터 하나씩 비교하면서 같은 값이 나오면 그 위치를 돌려주고, 리스트 끝까지 찾아도 같은 값이 나오지 않으면 -1을 반환함



순차 탐색

```
n = len(v)
for i in range(0, n):
    if 12 == v[i]:
        print("찾음")
```

```
def search_list(a, x):
    n = len(a)
    for i in range(0, n): # 모든 값을 차례로
        if x == a[i]:     # x값과 비교하여 같으면
            return i      # 위치를 돌려줌
    return -1

v = [60, 5, 33, 12, 97, 24, 12]

print(search_list(v, 5))
print(search_list(v, 12))
print(search_list(v, 111))
```

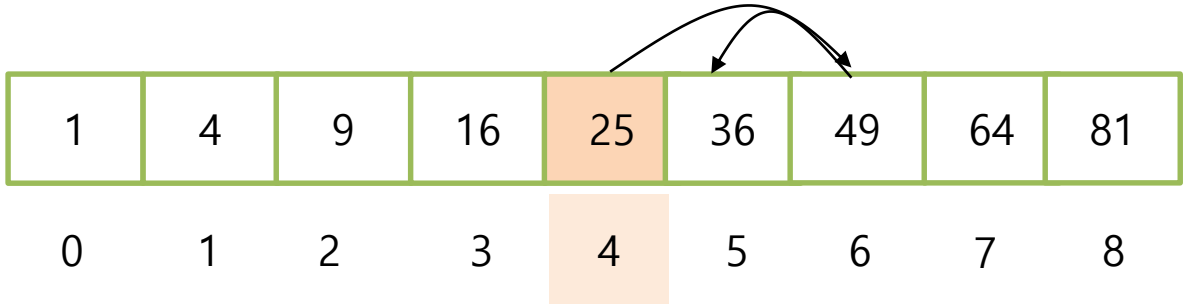
이분 탐색

➢ 이분 탐색

자료가 크기 순서대로 정렬된 리스트에서 특정한 값이 있는지 찾아 그 위치를 돌려주는 알고리즘

☆ 탐색 과정

1. 중간 위치를 찾는다.
2. 찾는 값과 중간 위치값을 비교한다.
3. 찾는 값이 중간 위치 값보다 크면 중간 위치의 오른쪽을 대상으로 탐색하고, 작으면 왼쪽을 대상으로 탐색한다.



이분 탐색

```python
def binary_search(a, x):
    start = 0
    end = len(a) - 1

    while start <= end:  #탐색할 범위가 있는 동안 반복
        mid = (start + end) // 2  # 탐색 범위의 중간 위치
        if x == a[mid]:  #찾음
            return mid
        elif x > a[mid]: #찾는 값이 더 크면 오른쪽 범위를 좁혀 계속 탐색
            start = mid + 1
        else:            #찾는 값이 더 작으면 왼쪽 범위를 좁혀 계속 탐색
            end = mid - 1

    return -1  #찾지 못함
```

이분 탐색

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

'''
1차 검색
1. mid=4 (0+8)/2, d[4]=25, 중간 위치값
2. 36>25 같지 않음 -> 25의 오른쪽 범위 탐색
3. mid=6 (5+8)/2  d[6]=49, 중간 위치값
4. 36<49 같지 않음 -> 49위 왼쪽 범위 탐색
5. 36 한 개 있음. 찾음. 위치번호 5를 결과값으로 반환함
'''
```