



list

≡ 태그

Array

과정 소개

빅-오(O) 표기법

배열

버블 정렬

카운팅 정렬

baby gin

순열

탐욕 알고리즘(GREEDY)

배열 2

2차원 배열

부분집합

비트 연산자

검색

순차 검색

이진 검색

선택 정렬

Array

과정 소개

- 자료구조와 알고리즘

1. 알고리즘 : 유한한 단계를 통해 문제를 해결하기 위한 절차나 방법.

⇒ 주로 컴퓨터 용어로 쓰이며, 컴퓨터가 어떤 일을 수행하기 위한 단계적 방법

- 무엇이 좋은 알고리즘 인가?

1. 정확성 : 얼마나 정확하게 동작하는가
2. 작업량 : 얼마나 적은 연산으로 원하는 결과를 얻어내는가
3. 메모리 사용량 : 얼마나 적은 메모리를 사용하는 가
4. 단순성 : 얼마나 단순한가
5. 최적성 : 더 이상 개선할 여지없이 최적화되었는가

- 논리적 사고력 향상

- 성능 분석 필요

- 많은 문제에서 성능 분석의 기준으로 알고리즘 작업량을 비교한다.



예 : 1부터 100까지 합을 구하는 문제

알고리즘 1	알고리즘 2
$ \begin{array}{r} 1 \\ + 2 \\ + 3 \\ + 4 \\ \dots\dots \\ + 100 \\ \hline 5,050 \end{array} $	
100번의 연산 (덧셈 100번)	3번의 연산 (덧셈 1번, 곱셈 1번, 나눗셈 1번)

⇒ 알고리즘의 작업량을 표현할 때 시간복잡도로 표현한다.

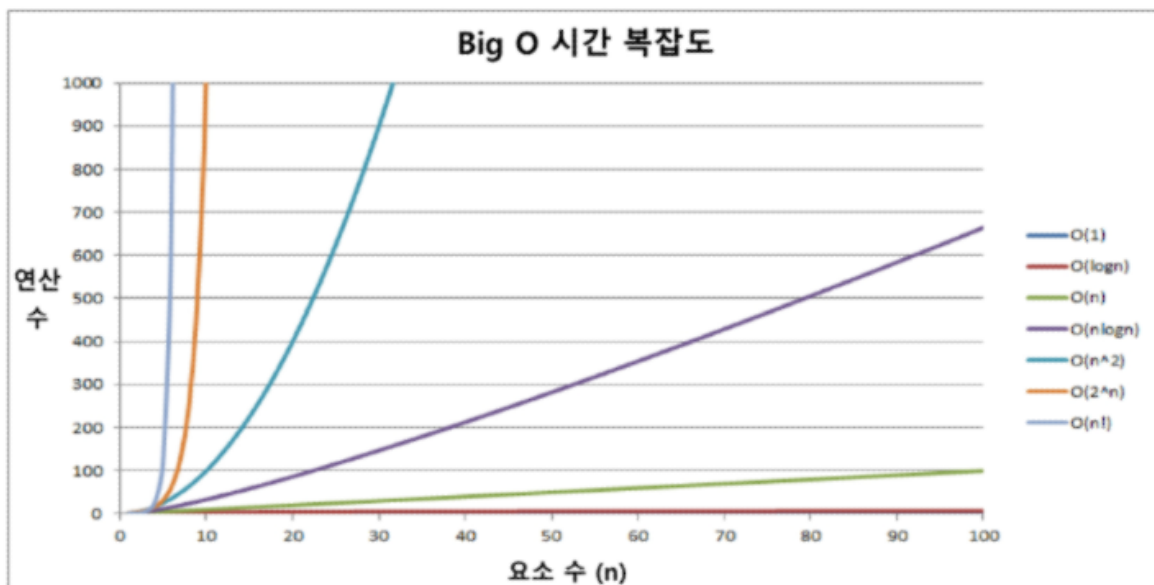
- 시간 복잡도 : 실제 걸리는 시간을 측정 or 실행되는 명령문의 개수를 계산

빅-오(O) 표기법

- 시간 복잡도 함수 중에서 가장 큰 영향력을 주는 n 에 대한 항만을 표시한다.
- 계수는 생략하여 표시한다.
- 예)

$O(3n + 2) = O(3n)$ (최고차항만 선택) = $O(n)$ (계수 3제거)

요소 수가 증가함에 따라, 각기 다른 시간 복잡도의 알고리즘은 아래와 같은 연산수를 보인다.



배열

- 정의

일정한 자료형의 변수들을 하나의 이름으로 열거하여 사용하는 자료구조

- 필요성

1. 여러 개의 변수가 필요할 때, 일일이 다른 변수명을 이용하여 자료에 접근하는 것은 매우 비효율적일 수 있다.
2. 배열을 사용하면 하나의 선언을 통해서 둘 이상의 변수를 선언할 수 있다.
3. 단순히 다수의 변수 선언을 의미하는 것이 아니라, 다수의 변수로써하기 힘든 작업을 할 수 있다.

- 1차원 배열의 선언

별도의 선언이 없으면 변수에 처음 값을 할당할 때 생성

```
# 언패킹
```

```
arr = [1, 2, 3]  
print(*arr)
```

```
# 10칸 배열
```

```
arr = [0] * 10  
print(arr) #[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- 예제

❖ 배열 활용 예제 : Gravity

- 상자들이 쌓여있는 방이 있다. 방이 오른쪽으로 90도 회전하여 상자들이 중력의 영향을 받아 낙하한다고 할 때, 낙차가 가장 큰 상자를 구하여 그 낙차를 리턴 하는 프로그램을 작성하시오.
- 중력은 회전이 완료된 후 적용된다.
- 상자들은 모두 한쪽 벽면에 붙여진 상태로 쌓여 2차원의 형태를 이루며 벽에서 떨어져서 쌓인 상자는 없다.
- 상자의 가로, 세로 길이는 각각 1이다.
- 방의 가로길이는 100이며, 세로 길이도 항상 100이다.
- 즉, 상자는 최소 0, 최대 100 높이로 쌓을 수 있다.
- 상자가 놓인 가로 칸의 수 N, 다음 줄에 각 칸의 상자 높이가 주어진다.

• 코드

```
'''
입력값
9
7 4 2 0 0 6 0 7 0
7 8 2 0 0 6 0 7 0
'''

# 내코드
=> 안되는 거였음..

# N    = int(input()) # 상자가 쌓여있는 가로 길이
# arr = list(map(int, input().split()))
# maxi = arr.index(max(arr))
#
# result = 0
# for idx, value in enumerate(arr):
#     if idx > maxi and arr[maxi] > value:
#         result += 1
# print(result)
```

수업 코드

```
N = int(input())
arr = list(map(int, input().split()))
max_v = 0
for i in range(N-1):
    cnt = 0 # 오른쪽에 있는 더 낮은 높이의 개수
    for j in range(i+1, N):
        if arr[i] > arr[j]:
            cnt += 1
    if max_v < cnt:
        max_v = cnt
print(max_v)
```

버블 정렬

✔ 인접한 두 개의 원소를 비교하며 자리를 계속 교환하는 방식

✔ 정렬 과정

- 첫 번째 원소부터 인접한 원소끼리 계속 자리를 교환하면서 맨 마지막 자리까지 이동한다.
- 한 단계가 끝나면 가장 큰 원소가 마지막 자리로 정렬된다.
- 교환하며 자리를 이동하는 모습이 물 위에 올라오는 거품 모양과 같다고 하여 버블 정렬이라고 한다.

✔ 시간 복잡도

- $O(n^2)$

- 논리

❖ 배열을 활용한 버블 정렬

- 앞서 살펴 본 정렬 과정을 코드로 구현하면 아래와 같다. (오름차순)

BubbleSort(a, N)	# 정렬할 배열과 배열의 크기
for i : N-1 -> 1	# 정렬될 구간의 끝
for j : 0 -> i-1	# 비교할 원소 중 왼쪽 원소의 인덱스
if a[j] > a[j+1]	# 왼쪽 원소가 더 크면
a[j] <-> a[j+1]	# 오른쪽 원소와 교환

- 스켈레톤

```
for i : N - 1 → 1:
  for j : 0 → i - 1:
    if arr[j] > arr[j+1]:
      A[j] <-> A[j+1]
```

- 실제 코드

```
N = 6
arr = [7, 2, 5, 3, 1, 4]

# 오름차순

def asc(arr, N):
  for i in range(N-1, 0, -1): # for i : N - 1 -> 1,
    for j in range(i): # for j : 0 -> i - 1, j 비교할 !
      if arr[j] > arr[j+1]: # 오름차순은 큰 수를 오른쪽으로
```

```

        arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

# 내림차순
def desc(arr, N):
    for i in range(N - 1, 0, -1): # for i : N - 1-> 1,
        for j in range(i): # for j : 0 -> i - 1, j 비교할
            if arr[j] < arr[j + 1]: # 오름차순은 큰 수를 오른쪽으로
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

print(asc(arr, N))
print(desc(arr, N))

```

카운팅 정렬

- 항목들의 순서를 결정하기 위해, 집합에 각 항목이 몇 개 있는지 세서, 선형 시간에 정렬
- 시간 복잡도 $O(n+k)$ n : 리스트 길이 k : 정수의 최대값

⇒ 각 숫자의 개수를 저장할 수 있는 것이다...

- 1단계 (다른 알고리즘에서도 범용적으로 사용되는 단계)

- Data에서 각 항목들의 발생 회수를 세고, 정수 항목들로 직접 인덱스 되는 카운트 배열 counts에 저장한다.



Data에서 각 항목들의 발생 회수를 세고, 정수 항목들로 직접 인덱스 되는 counts(배열)에 저장

- counts 배열은?

- n개인 경우 : `counts = [0] * N`

- 마지막 인덱스가 n인 경우 : `counts = [0] * (N+1)`

⇒ `counts[x] += 1`


```
counts = [0] * (k+1)
for x in Data:
    counts[x] += 1

for i : 1-> k
    counts[i] <- counts[i-1] + counts[:]
```

- 2단계

- 정렬된 집합에서, 각 항목의 앞에 위치할 항목의 개수를 반영하기 위해 counts 원소조정.

- 정렬된 집합에서 각 항목의 앞에 위치할 항목의 개수를 반영하기 위해 counts의 원소를 조정한다.

DATA	0	4	1	3	1	2	4	1
	0	1	2	3	4			
COUNTS	1	3	1	1	2			
								
COUNTS	1	4	5	6	8			

⇒ 누적한 값

- 3단계

1. counts[1]을 감소시키고 Temp에 1을 삽입한다.

• counts[1]을 감소시키고 Temp에 1을 삽입한다.

	J = 7 ↓							
DATA	0	4	1	3	1	2	4	1
COUNTS	1	4->3	5	6	8			
TEMP				1				

2. counts[4]를 감소시키고 temp에 4를 삽입한다.

• counts[4]를 감소시키고 temp에 4를 삽입한다.

	J = 6 ↓							
DATA	0	4	1	3	1	2	4	1
COUNTS	1	3	5	6	8->7			
TEMP				1				4

3. counts[2]를 감소시키고 temp에 2를 삽입한다.

• counts[2]를 감소시키고 temp에 2를 삽입한다.

	J = 5 ↓							
DATA	0	4	1	3	1	2	4	1
COUNTS	1	3	5->4	6	7			
TEMP				1	2			4

4. counts[1]을 감소시키고 temp에 1을 삽입한다.

- counts[1]을 감소시키고 temp에 1을 삽입한다.

J = 4
↓

DATA	0	4	1	3	1	2	4	1
COUNTS	1	3->2	4	6	7			
TEMP			1	1	2			4

- counts[4]를 감소시키고 temp에 4를 삽입한다.

- counts[4]를 감소시키고 temp에 4를 삽입한다.

J = 1
↓

DATA	0	4	1	3	1	2	4	1
COUNTS	1	1	4	5	7->6			
TEMP		1	1	1	2	3	4	4

- counts[0]을 감소시키고 temp에 0을 삽입한다.

- counts[0]을 감소시키고 temp에 0을 삽입한다.

J = 0
↓

DATA	0	4	1	3	1	2	4	1
COUNTS	1->0	1	4	5	6			
TEMP	0	1	1	1	2	3	4	4

- Temp 업데이트 완료하고 정렬 작업을 종료한다.

- 결론

```

def Counting_Sort(DATA, TEMP, k)
# DATA [] -- 입력 배열(0 to k)
# TEMP [] -- 정렬된 배열.
# COUNTS [] -- 카운트 배열.

COUNTS = [0] * (k+1)

for i in range (0, len(DATA)) :
    COUNTS[DATA[i]] += 1

for i in range (1, k+1) :
    COUNTS[i] += COUNTS[i-1]

for i in range (len(TEMP)-1, -1, -1) :
    COUNTS[DATA[i]] -= 1
    TEMP[COUNTS[DATA[i]]] = DATA[i]

```

- 실습

```

N = 6
K = 9 # 0~K
data = [7,2,4,5,1,3] # 0~9, K = 9
counts = [0] * 9
temp = [0] * N # 정렬된 결과 저장
# counts 배열에 기록하기
for x in data:
    counts[x] += 1
# counts 누적합 구하기
for i in range(1, K):
    counts[i] = counts[i-1] + counts[i]
# data의 마지막 원소부터 정렬하기
for i in range(N-1, -1, -1): # N-1 -> 0 인덱스...
    counts[data[i]] -= 1 # 개수를 인덱스로 변환(남은 개수 계산)
    temp[counts[data[i]]] = data[i]
print(*temp)

```

baby gin

• 설명

- 0~9 사이의 숫자 카드에서 임의의 카드 6장을 뽑았을 때, 3장의 카드가 연속적인 번호를 갖는 경우를 run이라 하고, 3장의 카드가 동일한 번호를 갖는 경우를 triplet이라고 한다.
- 그리고, 6장의 카드가 run과 triplet로만 구성된 경우를 baby-gin으로 부른다.
- 6자리의 숫자를 입력 받아 baby-gin 여부를 판단하는 프로그램을 작성하라.

- 완전 검색
 - 모든 경우의 수를 나열해보고 확인하는 기법이다.
 - Brute-force 혹은 generate-and-test 기법이라고도 불리 운다.
 - 모든 경우의 수를 테스트한 후, 최종 해법을 도출한다.
 - 일반적으로 경우의 수가 적을 때 유용하다.
- 모든 경우의 수를 생성하기 때문에, 수행 속도는 느리지만, 해답을 찾지 못할 확률이 작다
- 주어진 문제를 풀 때, 완전 검색으로 풀고, 성능을 개선하는 것이 바람직하다.
- 완전 검색으로 baby-gin을 푼다면
 1. 6개의 숫자로 만들 수 있는 모든 숫자 나열
 2. 입력으로[2,3,5,7,7,7]을 받았을 경우, 아래와 같이 순열을 생성할 수 있다. → 앞3자리, 뒤3자리 잘라서 run과 triple 여부 테스트, 최종적으로 baby-gin 판단.

순열

- 서로 다른 것들 중, 몇개를 뽑아서 한줄로 나열하는 것
- nPr 로 표현

$nPn = n!$ (Factorial) // $nPr = > n * (n-1) * (n-2) * \dots * (n-r+1)$

- {1,2,3}을 포함하는 모든 순열을 생성하려면

```
for i in range(1,4):
    for i2 in range(1,4):
        if i2 != i1:
            for i3 in range(1,4):
                if i3 != i1 and i3 != i2:
                    print(i1,i2,i3)
```

탐욕 알고리즘(GREEDY)

- 탐욕 알고리즘은 최적해를 구하는데 사용되는 근시안적인 방법
 - 여러 경우 중 하나 결정할 때, 최적을 선택해가면서 최종적인 해답에 도달
 - 지역적으로는 최적이지만, 최종적으로 만들어도 그게 최적은 아닐 수 있다.
-
- 동작 과정
 1. 해 선택: 최적 해를 구한 뒤, 이를 부분 집합에 추가한다.
 2. 실행 가능성 검사 : 새로운 부분해 집합이 실행 가능한지를 확인한다.
 3. 해 검사 : 새로운 부분해 집합이 문제의 해가 되는지를 확인한다. / 아직 전체 문제의 해가 완성되지 않았다면 1)의 해선택 부터 다시

ex) 거스름돈 줄이기

어떻게 해야 지폐와 동전의 개수를 최소화 할 수 있을까?

1) 해 선택 : 여기에서는 멀리 내다볼 것 없이 가장 좋은 해를 선택, 단위가 큰 걸로만 만들면 동전 개수 줄어듬 => 현재 고를 수 있는 가장 단위가 큰 동전을 하나 골라 거스름돈에 추가

한다.

2) 실행 가능성 검사 : 거스름돈이 손님에게 내드려야 할 액수를 초과하는지 확인.

3) 해 검사 : 거스름돈 문제의 해는 당연히 거스름돈이 손님에게 내드려야 하는 액수와 일치
→ 모자라면 1)로 돌아가서 추가할 동전 고른다.

- baby - gin을 탐욕 알고리즘으로 풀어본다면

❖ 풀이

ex) 444345

	1	2	3	4	5	6	7	8	9	0
counts			1	4	1					
run 조사 후 run 데이터 완전 삭제										
	1	2	3	4	5	6	7	8	9	0
counts				3						
										Baby-gin!

ex) 333456

	1	2	3	4	5	6	7	8	9	0
counts										

❖ 풀이

ex) 444456

	1	2	3	4	5	6	7	8	9	0
counts				4	1	1				
triplet 조사 후 triplet 데이터 완전 삭제										
	1	2	3	4	5	6	7	8	9	0
counts				1	1	1				
										Baby-gin!

⇒ triple을 먼저 찾고 run을 찾는 것이 바람직 할 것이라는 점을 깨닫는다.

- 실습

```

num = 456789
c = [0] * 12 # counts할 lst

## 자주 사용되는 코드

for i in range(6):
    c[num % 10] += 1
    num //= 10

i = 0
tri = run = 0
while i < 10:
    if c[i] == 3: # triplete 조사 후 데이터 삭제
        c[i] -= 3
        tri += 1
        continue
    if c[i] >= 1 and c[i+1] >= 1 and c[i+2] >= 1 : #run 조사후
        c[i] -= 1
        c[i+1] -= 1
        c[i+2] -= 2
        run += 1
        continue
    i += 1

if run + tri == 2 :
    print("Baby Gin")
else:
    print("Lose")

```

배열 2

2차원 배열

- 1차원 리스트를 묶어놓은 LIST
- 2차원 리스트의 선언 : 새로길이(행의 개수) 와 가로길이(열의 개수)를 필요로 한다.
- python에서는 데이터 초기화를 통해 변수선언과 초기화가 가능하다.

```
arr = [[0,1,2,3],[4,5,6,7]] (2행 4열의 2차원 List)
```

arr	0	1	2	3
	4	5	6	7

```
N = int(input())
arr = [list(map(int, input().split())) for _ in range(N)]

N = int(input())
arr = [list(map(int, input())) for _ in range(N)]
```

- 2차원 배열의 선언

```
3
1 2 3
4 5 6
7 8 9

N = int(input())

arr = [list(map(int, input().split())) for _ in range(N)]

# [[1,2,3],[4,5,6],[7,8,9]]
```

```
arr2 = [[0] * N for _ in range(N)]
```

이렇게 하면 안된다 !

```
arr3 = [[0]*N]*N
print(arr3)
arr3[0][0] = 1
print(arr3)
```

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
[[1, 0, 0], [1, 0, 0], [1, 0, 0]] 다 바뀌게 된다.
```

- 2차원 배열의 순회

행 우선 순회

```
for i in range(n):
    for j in range(m):
        f(array[i][j])
```

열 우선 순회

```
for j in range(m):
    for i in range(n):
        f(array[i][j])
```

지그재그 순회

```
for i in range(n):
    for j in range(m):
        f(array[i][j + (m-1-2*j) * (i%2) # 짝수행에서만 동작])
```

2차원 배열의 접근

• 지그재그 순회

i 행의 좌표

j 열의 좌표

for i in range(n) :

for j in range(m) : 0 1 2 3

f(array[i][j] + (m-1-2*j) * (i%2))

$(m-1-j-i) \times 0$ (짝수행)
 $\times 1$ (홀수행)

- 델타를 이용한 2차 배열의 탐색
 - 2차 배열의 한 좌표에서 4방향의 인접 배열 요소를 탐색하는 방법
 - 인덱스(i, j)인 칸의 상하좌우 칸(ni,nj)

델타를 이용한 2차 배열 탐색

```

N = 5
di = [0, 1, 0, -1]
dj = [1, 0, -1, 0]
arr = [[0] * N for _ in range(N)]
for i in range(N):
    for j in range(N):
        for k in range(4):
            ni = i + di[k]
            nj = j + dj[k]
            if 0 <= ni < N and 0 <= nj < N:
                print(arr[ni][nj], end = ' ')
        print()

i = 3
j = 4
    
```

```

for k in range(4):
    ni = i + di[k]
    nj = j + dj[k]
    print(ni, nj)

# 한 좌표에서 4방향의 인접 배열 요소를 탐색하게 된다.

N = 5
arr = [[0] * N for _ in range(N)]
for i in range(N):
    for j in range(N):
        for di, dj in [[0,1], [1,0], [0,-1], [-1,0]]:
            ni, nj = i + di, j + dj
            if 0 <= ni < N and 0 <= nj < N:
                print(arr[ni][nj])

```

- 전치행렬



```

# 전치행렬
arr = [[1,2,3],[4,5,6],[7,8,9]]
for i in range(3):
    for j in range(3):
        if i < j: # if 문이 없다면 2번 바뀌서 전치행렬이 가능하지 않는다.
            arr[i][j], arr[j][i] = arr[j][i], arr[i][j]

```

부분집합

- 유한 개의 정수로 이루어진 집합이 있을 때, 이 집합의 부분집합 중에서 그 집합의 원소를 모두 더한 값이 0 이 되는 경우가 있는지를 알아내는 문제
- 부분집합의 수
 - 원소가 n 개일 때, 공집합을 포함한 부분집합의 수는 2^n 개 이다.
 - 이는 각 원소를 부분집합에 포함시키거나 포함시키지 않는 2가지 경우를 모든 원소에 적용한 경우의 수와 같다.
- 예시코드

```
bit = [0,0,0,0]
for i in range(2):
    bit[0] = i
    for j in range(2):
        bit[1] = j
        for k in range(2):
            bit[2] = k
            for l in range(2):
                bit[3] = l
                print(bit)
```

비트 연산자

비트 연산자

1. & 비트 단위로 AND 연산을 한다.
2. | 비트 단위로 OR 연산을 한다.
3. << 피연산자의 비트 열을 왼쪽으로 이동
4. >> 피연산자의 비트 열을 오른쪽으로 이동

<< 연산자

$1 \ll n : 2^n$ 즉, 원소가 n 개일 경우의 모든 부분집합의 수를 의미한다.

& 연산자

$i \& (1 \ll j) : i$ 의 j 번째 비트가 1인지 아닌지를 검사한다.

- 비트 연산자를 활용하는 방법

```
## 비트 연산자

def f(arr, N):
    for i in range(1, 1 << N):
        s = 0
        for j in range(N):
            if i & (1 << j):
                s += arr[j]
                # print(arr[j], end=' ')
        #print(s)
        if s == 0:
            return True
    return False

N = int(input())
arr = list(map(int, input().split()))

print(f(arr, N))
```

- 이해하기

```
N = 10
arr = [-7, -5, 2, 3, 8, -2, 4, 6, 9, 0]
print(arr)
```

```

print(2**10) # 모든 부분집합의 개수
print(2 ** N)
print(1 << N)
print(bin(1024))

for i in range(1 << N):
    lst = []
    print(i, '번째 경우의 수')
    for j in range(N):
        # print(1 << j) # 비트가 각각의 고유 번호가 된다.
        # i번째 경우의 수에, j번째 요소가 포함 되어있는 부분 집합인지 확
        # i번째가 3번째라면 -> 0b 0011
        # j번째 요소 (0번째, 1번째, 2번째...) -> 0b 0001, 0010, 00
        if i & (1 << j):
            lst.append(arr[j])
    print(lst)

```

```

N = 100
# arr = [-7, -5, 2, 3, 8, -2, 4, 6, 9, 0]
arr = list(range(1,101))
# 부분집합의 합이 55 미만인 경우만 모은 리스트
print(arr)
print(2**10) # 모든 부분집합의 개수
print(2 ** N)
print(1 << N)
print(bin(1024))

for i in range(1, 1 << N):
    lst = []
    print(i, '번째 경우의 수')
    for j in range(N-1, -1, -1):
        # print(1 << j) # 비트가 각각의 고유 번호가 된다.
        # i번째 경우의 수에, j번째 요소가 포함 되어있는 부분 집합인지 확
        # i번째가 3번째라면 -> 0b 0011
        # j번째 요소 (0번째, 1번째, 2번째...) -> 0b 0001, 0010, 00

```

```
        if i & (1 << j):
            lst.append(arr[j])
            if sum(lst) >= 55:
                break
    if sum(lst) < 55:
        print(lst)
```

검색

- 원하는 항목을 찾는 작업
- 목적하는 탐색 키를 가진 항목을 찾는 것

순차 검색

- 정의
 - 가장 간단하고 직관적인 검색, 배열, 연결리스트 등 순차구조에서 원하는 항목을 찾는다.
 - 알고리즘이 단순하지만, 검색 대상의 수가 많은 경우에는 수행시간이 급격히 증가하여 비효율적이다.
- 2가지 경우
 - 정렬되어 있거나, 안되어 있거나
- 검색과정
 - 첫 원소부터 순서대로 검색 대상과 키 값이 같은 원소가 있는지 비교하며 찾는다.
 - 키 값이 동일한 원소를 찾으면 그 원소의 인덱스를 반환한다.
 - 자료구조의 마지막에 이를 때까지 검색 대상을 찾지 못하면 검색 실패

1. 정렬되어 있지 않은 경우

- 예시

예) 2를 검색하는 경우

	4	9	11	23	2	19	7
<검색 과정>							
① 4 ≠ 2	4	9	11	23	2	19	7
② 9 ≠ 2	4	9	11	23	2	19	7
③ 11 ≠ 2	4	9	11	23	2	19	7
④ 23 ≠ 2	4	9	11	23	2	19	7
⑤ 2 = 2	4	9	11	23	2	19	7

검색 성공!

- 첫번째 원소를 찾을 때 1번 비교, 두번째 2번 ...
- 평균 비교 횟수 : $(n+1)/2$

구현 예

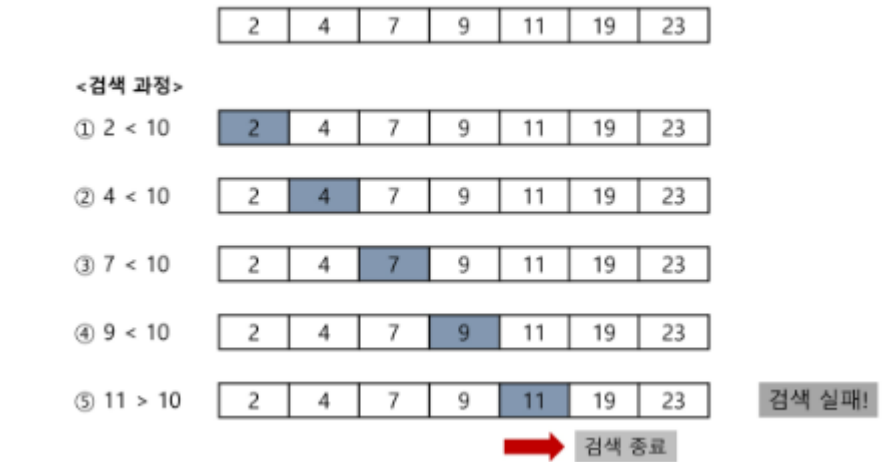
```
def sequential_search(a, n, key)
    i <- 0
    while i < n and a[i] != key :
        i <- i+1
    if i < n : return i
    else : return -1
```

2. 정렬된 경우

- 자료를 순차적으로 검색하면서 키 값을 비교하고, 원소의 키 값이 검색 대상의 키 값보다 크면, 찾는 원소가 없다는 것이므로 더 이상 검색하지 않고 검색을 종료한다.

- 예시

예) 10을 검색하는 경우



- 정렬이 되어있으므로, 검색 실패를 반환하는 경우 평균 비교 회수가 반으로 줄어듦

구현 예

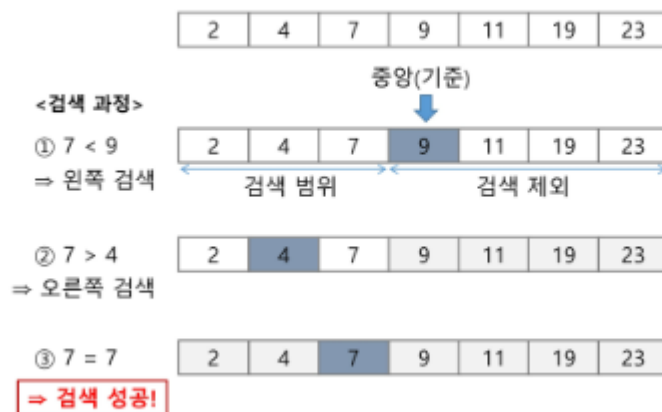
```
def sequentialSearch2(a, n, key)
    i <- 0
    while i < n and a[i] < key :
        i <- i + 1
    if i < n and a[i] == key :
        return i
    else :
        return -1
```

```
for i in range(N):
    if a[i] == key:
        return i
    else:
        break
```

이진 검색

- 자료의 가운데에 있는 항목의 키 값과 비교하여 다음 검색의 위치를 결정하고 검색을 계속 진행하는 방법
 - 목적 키를 찾을 때까지 이진 검색을 순환적으로 반복 수행함으로써 검색 범위를 반으로 줄여가면서 보다 빠르게 검색을 수행함.
- 이진 검색을 하기 위해서는 자료가 정렬된 상태여야 한다.
- 검색 과정
 1. 자료의 중앙에 있는 원소를 고른다
 2. 중앙 원소의 값과 찾고자 하는 목표값을 비교한다.
 3. 중앙 원소의 값보다 작으면 자료의 왼쪽 반에 대해서 새로 검색 수행
 4. 크다면 자료 오른쪽 반에 대해서 새로 검색 수행
 5. 찾고자 하는 값을 찾을 때 까지 1~3의 과정 반복

예) 이진 검색으로 7을 찾는 경우



- 코드

```

def binarySearch(a, N, key)
    start = 0
    end = N-1
    while start <= end :
        middle = (start + end)//2
        if a[middle] == key : # 검색 성공
            return true
        elif a[middle] > key :
            end = middle - 1
        else :
            start = middle + 1
    return false          # 검색 실패

```

```

def binary_search(arr, N, key):
    start = 0    # 구간 초기화
    end = N - 1
    while start <= end:
        middle = (start + end) // 2 # 중앙원소 인덱스
        if arr[middle] == key:      # 검색 성공
            return middle
        elif arr[middle] > key:     # 중앙값이 키보다 크면
            end = middle - 1
        else:
            start = middle + 1      # 키보다 작으면
    return -1                      # 탐색 실패

```

- 재귀 함수 이용
 - 이진 검색을 구현할 수 있다.

❖ 재귀 함수 이용

- 아래와 같이 재귀 함수를 이용하여 이진 검색을 구현할 수도 있다.
- 재귀 함수에 대해서는 나중에 더 자세히 배우도록 한다.

```
def binarySearch2(a, low, high, key) :  
    if low > high : # 검색 실패  
        return False  
    else :  
        middle = (low + high) // 2  
        if key == a[middle] : # 검색 성공  
            return True  
        elif key < a[middle] :  
            return binarySearch2(a, low, middle-1, key)  
        elif a[middle] < key :  
            return binarySearch2(a, middle+1, high, key)
```

선택 정렬

- 인덱스

DataBase에서 유래한 인덱스, 테이블에 대한 동작 속도를 높여주는 자료구조를 일컫는다.

⇒ Look up table 등의 용어를 사용하기도 한다.

인덱스를 저장하는데 필요한 디스크 공간은 보통 테이블을 저장하는데 필요한 디스크 보다 작다.

array index	name	original index
0	Barbara	1
1	Florence	2
2	Melissa	0
3	Shara	3

이름 인덱스 배열

array index	id	original index
0	1041	0
1	7334	2
2	19467	1
3	27500	3

id 인덱스 배열



array index	id	name	number	...
0	1041	Melissa	428-849-0471	...
1	19467	barbara	672-511-7155	...
2	7334	Florence	586-122-3241	...
3	27500	shara	459-729-8167	...
...

원본 데이터 배열

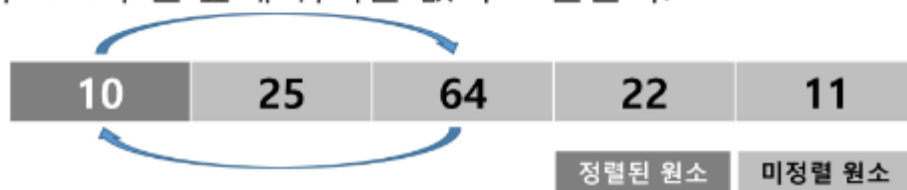
- 선택 정렬이란?
 - 셀렉션 알고리즘을 전체 자료에 적용한 것
- 과정
 1. 주어진 리스트 중 최소값 찾기
 2. 그 값을 리스트의 맨 앞에 위치한 값과 교환
 3. 맨 처음 위치를 제외한 나머지 리스트를 대상으로 위 과정 반복

✓ 정렬 과정

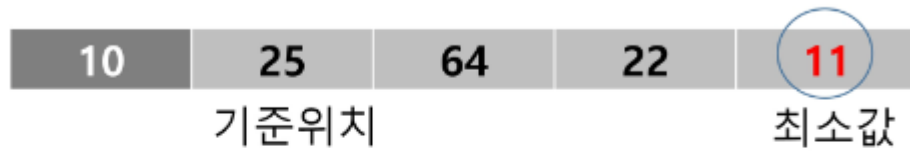
① 주어진 리스트에서 최소값을 찾는다.



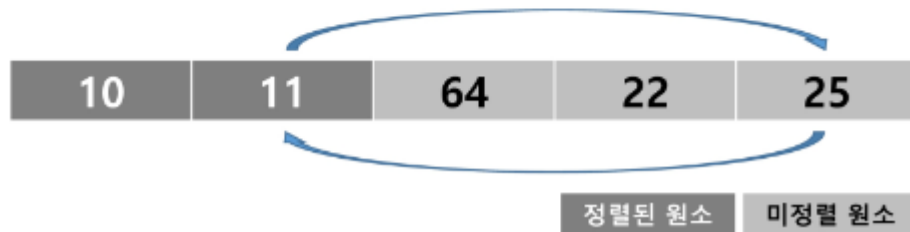
② 리스트의 맨 앞에 위치한 값과 교환한다.



③미정렬 리스트에서 최소값을 찾는다.



④리스트의 맨 앞에 위치한 값과 교환한다.



⑦미정렬 리스트에서 최소값을 찾는다.



⑧리스트의 맨 앞에 위치한 값과 교환한다.



- 미정렬원소가 하나 남은 상황에서는 마지막 원소가 가장 큰 값을 갖게 되므로, 실행을 종료하고 선택 정렬이 완료된다.

- 알고리즘

✔ 알고리즘

```
def SelectionSort(a[], n)
    for i from 0 to n-2
        a[i],...,a[n-1] 원소 중 최소값 a[k] 찾음
        a[i]와 a[k] 교환
```

- 선택정렬

✔ 선택 정렬

```
def selectionSort(a, N) :
    for i in range(N-1) :
        minIdx = i
        for j in range(i+1, N) :
            if a[minIdx] > a[j] :
                minIdx = j
        a[i], a[minIdx] = a[minIdx], a[i]
```

- 선택 알고리즘이란?
 - 저장되어 있는 자료로부터 k번째로 큰 or 작은 원소 찾는 법
- 선택 과정
 - 정렬 알고리즘으로 자료 정렬
 - 원하는 순서의 원소 가져오기
- 알고리즘

- 1번부터 k번째까지 작은 원소들을 찾아 배열의 앞쪽으로 이동시키고, 배열의 k번째를 반환한다.
- k가 비교적 작을 때 유용하며 $O(kn)$ 의 수행시간을 필요로 한다.

```
def select(arr, k):
    for i in range(0, k):
        minIndex = i
        for j in range(i+1, len(arr)):
            if arr[minIndex] > arr[j]:
                minIndex = j
        arr[i], arr[minIndex] = arr[minIndex], arr[i]
    return arr[k-1]
```

- 비교

✓ 학습한 정렬 알고리즘의 특성을 다른 정렬들과 비교해보자.

알고리즘	평균 수행시간	최악 수행시간	알고리즘 기법	비고
버블 정렬	$O(n^2)$	$O(n^2)$	비교와 교환	코딩이 가장 쉽다.
카운팅 정렬	$O(n+k)$	$O(n+k)$	비교환 방식	n이 비교적 작을 때만 가능하다.
선택 정렬	$O(n^2)$	$O(n^2)$	비교와 교환	교환의 회수가 버블, 삽입정렬보다 작다.
퀵 정렬	$O(n \log n)$	$O(n^2)$	분할 정복	최악의 경우 $O(n^2)$ 이지만, 평균적으로는 가장 빠르다.
삽입 정렬	$O(n^2)$	$O(n^2)$	비교와 교환	n의 개수가 작을 때 효과적이다.
병합 정렬	$O(n \log n)$	$O(n \log n)$	분할 정복	연결리스트의 경우 가장 효율적인 방식