# Algorithm (알고리즘) - 2

## 사전 발표 준비 및 전체적인 이론 학습

수업 자료의 글꼴 - "나눔스퀘어 라운드[https://hangeul.naver.com/fonts]"
수업 코드 - https://github.com/NET-Organization/24-2-Computer_Science_Study/tree/main/src

# 알고리즘

Divide and Conquer (분할과 정복)

Dynamic Programming (동적 프로그래밍)

Greedy Algorithm (탐욕 알고리즘)

Backtracking (되추적법)

Branch (분기한정법)

Sort (정렬)

Search (탐색)

# 알고리즘

Divide and Conquer (분할과 정복)

Dynamic Programming (동적 프로그래밍)

Greedy Algorithm (탐욕 알고리즘)

Backtracking (되추적법)

Branch (분기한정법)

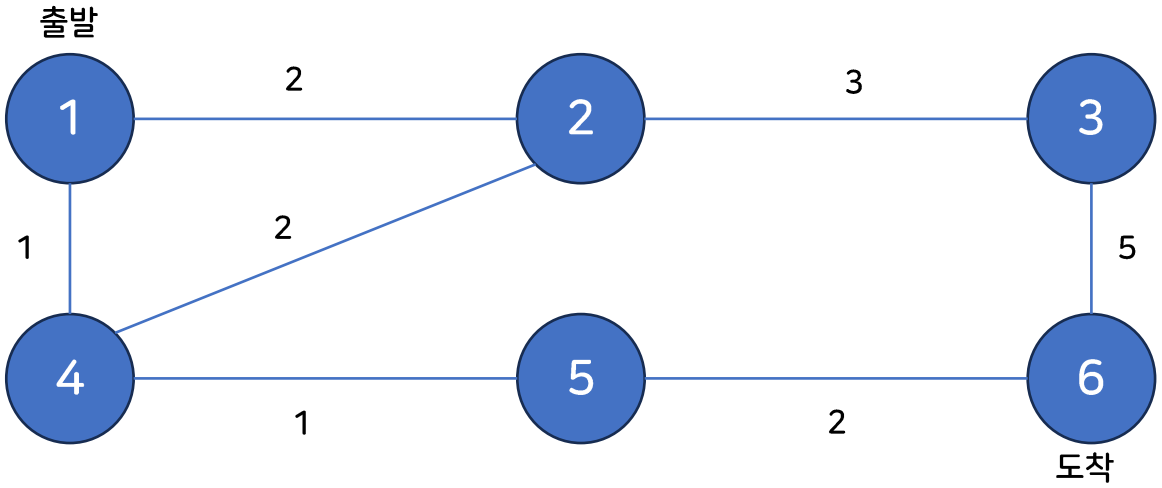
Sort (정렬)

Search (탐색)

# Dijkstra

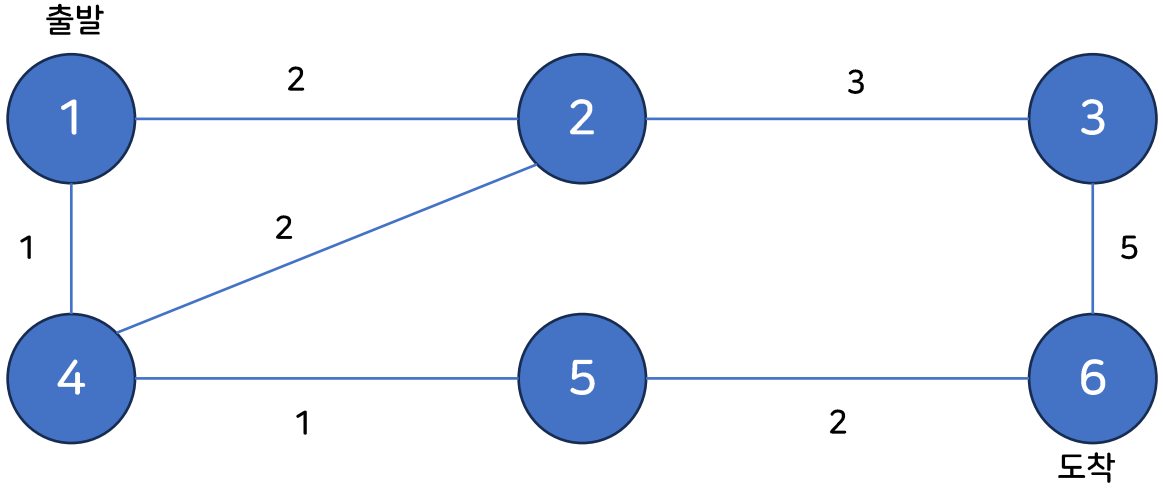
다익스트라 알고리즘은 그래프의 한 정점에서 다른 정점까지의 최단 경로를 구하는 알고리즘입니다.
해당 과정에서 도착 정점 뿐만 아니라 모든 다른 정점까지 최단 경로로 방문하여 각 정점까지의 최단 경로를 모두 찾습니다.

1. 출발 노드와 도착 노드를 세팅합니다
2. 최단 거리 테이블을 초기화 합니다.
3. 현재 위치한 노드의 인접 노드 중 방문하지 않은 노드를 구별하고, 방문하지 않은 노드 중 거리가 가장 짧은 노드를 선택합니다.
4. 해당 노드를 거쳐 다른 노드로 넘어가는 가중치를 계산해 최단거리 테이블을 업데이트합니다
5. 3,4번 과정을 반복합니다.



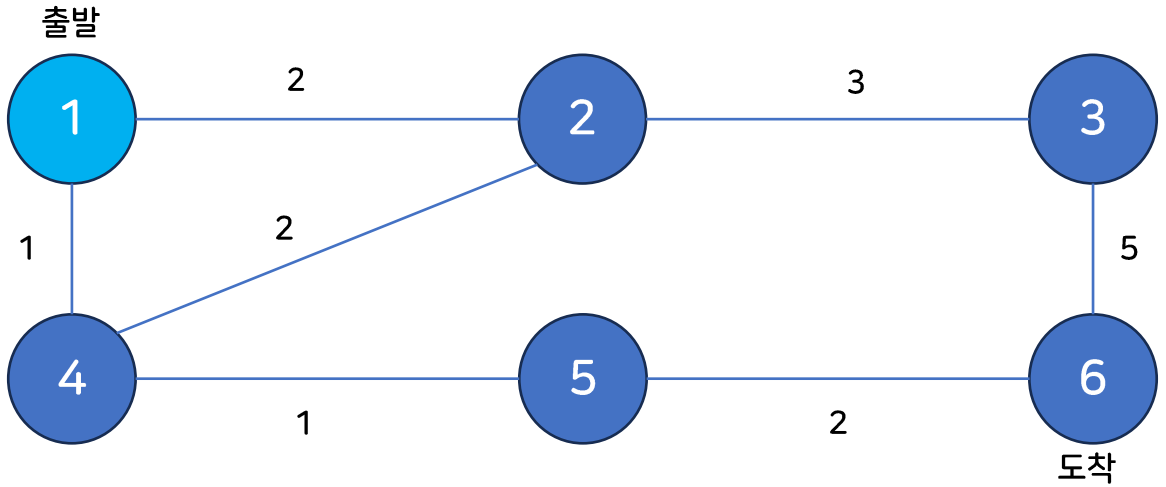

| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | inf | inf | inf | inf | inf | inf |

노드별 거리를 무한으로 초기화 한다.




| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | 0 | inf | inf | inf | inf | inf |

출발 노드의 거리를 0으로 설정한다.

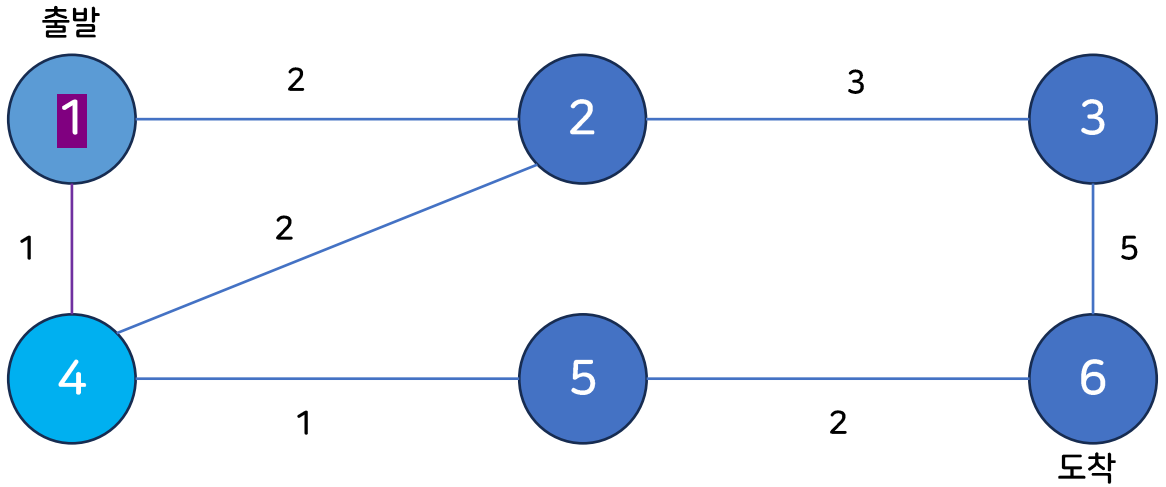



| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | 0 | Min(inf,2) | inf | Min(inf,1) | inf | inf |
| 업데이트 | 0 | 2 | inf | 1 | inf | inf |

1번 노드와 인접된 노드는 2번과 4번 노드이므로 그곳까지 가는 거리를 기존의 거리값과 비교해 최솟값으로 업데이트한다.

또한 1번은 인접노드와의 거리를 모두 업데이트 하였으므로 방문 표시를 한다.

최근 업데이트한 테이블에서 최단거리 노드는 4이므로 4를 다음 노드로 택한다

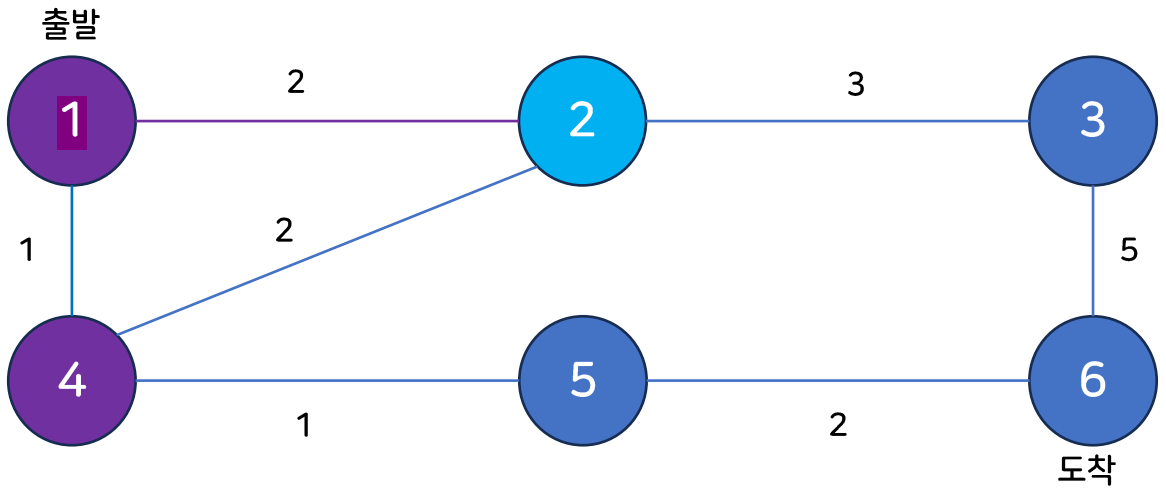



| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | 0 | Min(2, 1+2) | inf | 1 | Min(inf,1+1) | inf |
| 업데이트 | 0 | 2 | inf | 1 | 2 | inf |

4번노드에서 시작해 갈 수 있는 노드는 2와 5 두가지가 존재한다.

2번 노드의 경우 – 1에서 바로 가는 거리2, 4를 거쳐 가는 거리 1+2 둘의 최솟값으로 업데이트하여 2로 업데이트 된다.
5번 노드의 경우 – 4에서 바로 가는 거리 1+1과 기존 값인 무한과의 최솟값으로 업데이트하여 1로 업데이트 한다.

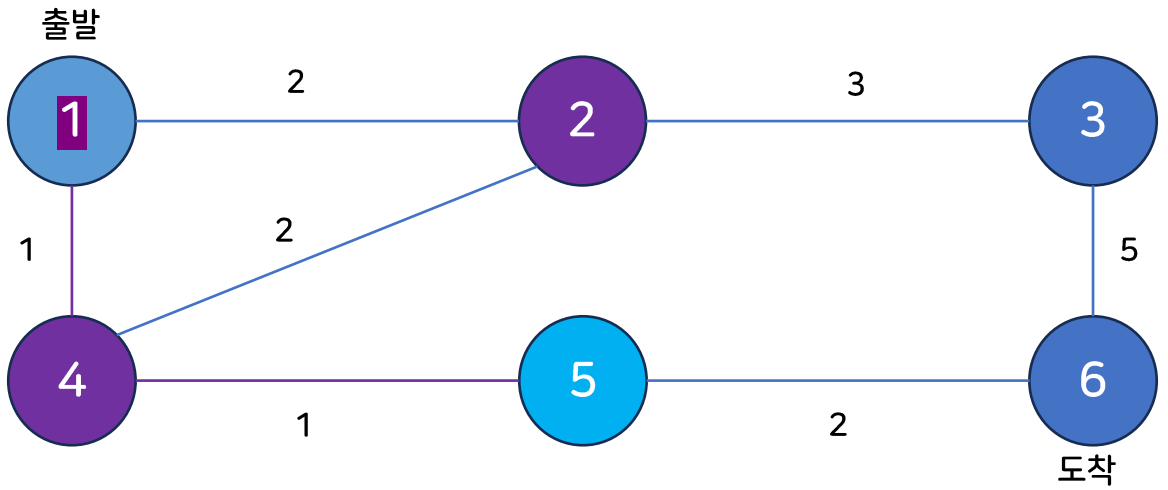
현재 거리가 가장 작은 노드는 2와5 이므로 편의상 인덱스가 작은 2로 이동한다




| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | 0 | 2 | Min(inf,2+3) | Min(1,2+2) | 2 | inf |
| 업데이트 | 0 | 2 | 5 | 1 | 2 | inf |

2번 노드에서 갈 수 있는 노드는 4와3이 있고 이전 슬라이드와 같은 방식으로 거리를 업데이트한다.

4번 노드는 이미 방문하였으며 그 다음 노드는 3,5 중에 값이 작은 5를 택한다.



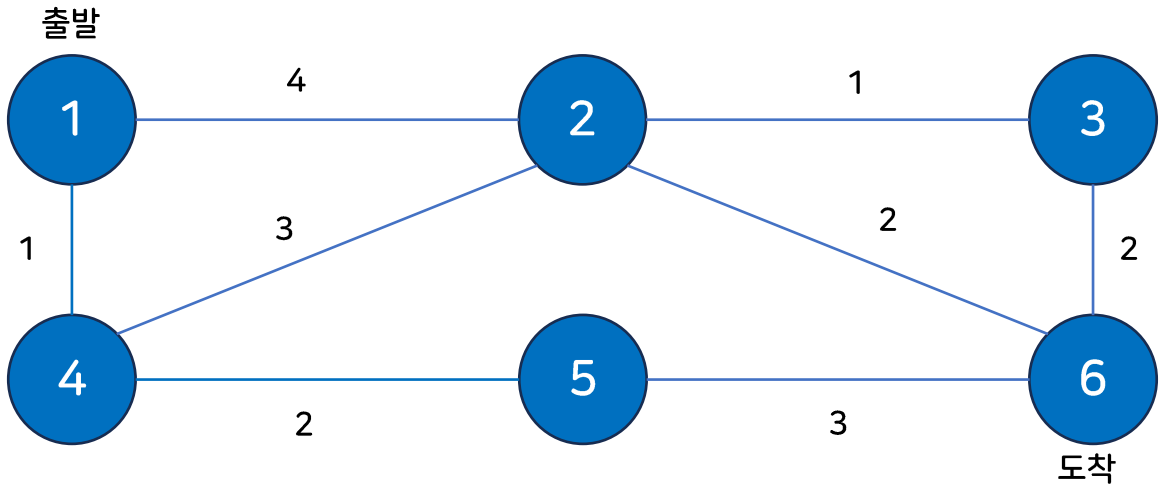

| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | 0 | 2 | 5 | 1 | 2 | Min(inf,2+2) |
| 업데이트 | 0 | 2 | 5 | 1 | 2 | 4 |

5번 노드에서 갈 수 있는 노드는 6으로 거리를 업데이트한다.

아직 방문하지 않은 노드에는 3과 6이 존재하는데, 거리가 가장 짧은 6번 노드로 이동한다.

6번 노드에 도착하였으므로 최종 비용은 4이다.

학습한 다익스트라 알고리즘을 사용해 최단거리를 구해보세요!!




| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | | | | | | |
| 업데이트 | | | | | | |

```python
class dijkstra () :

    def __init__ (self, W) :
        self.w = W
        self.n = len(W)

        self.f = set()
        self.touch = [0] * self.n
        self.length = [0] * self.n
        self.save_length = [0] * self.n

        for i in range (1, self.n) : self.length[i] = self.w[0][i]

        self.do_dijkstra()

    def do_dijkstra (self) :
        inf = 1000

        for i in range (1, self.n) :
            min_length = inf

            for j in range(1, self.n) :
                if (0 <= self.length[j] < min_length) :
                    min_length = self.length[j]
                    vnear = j

            edge = (self.touch[vnear], vnear)
            self.f.add(edge)

            for k in range (1, self.n) :
                if ((self.length[vnear] + self.w[vnear][k]) < self.length[k]) :
                    self.length[k] = self.length[vnear] + self.w[vnear][k]
                    self.touch[k] = vnear

            self.save_length[vnear] = self.length[vnear]
            self.length[vnear] = -1
```

생성자 부분 코드 설명

- W 파라미터를 통해 가중치 행렬을 저장합니다
- Self.n에는 가중치의 길이를 입력 받습니다
- F에는 결과로 저장할 최단경로의 집합을 저장합니다
- Touch에는 각 노드에 대해 직전에 연결된 노드를 저장합니다
- Length에는 출발점에서 각 노드까지의 현재까지 계산된 거리를 저장합니다
- Save_length에는 최종적으로 계산된 각 노드까지의 최단거리를 저장합니다

- 반복문을 통해 출발점 노드에서 다른 노드까지의 초기 거리를 세팅합니다

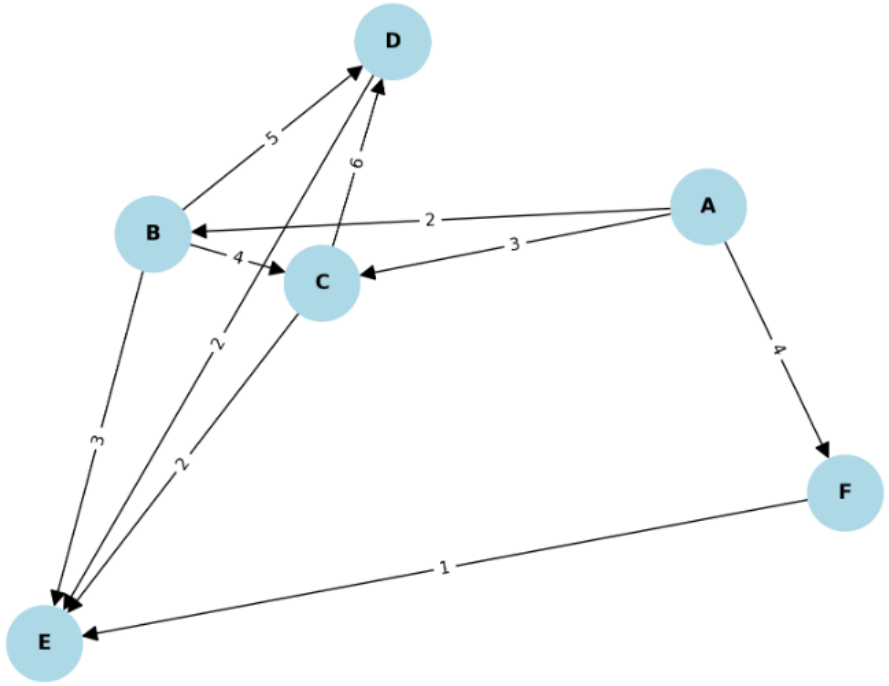
다익스트라 부분 코드 설명

- 반복문을 돌며 아직 처리되지 않은 노드중에서 가장 가까운 노드(vnear)를 찾습니다
- 최단 경로 집합에 vnear로 가는 간선을 추가합니다
- Vnear를 경유하면서 각 노드로 가는 경로가 더 짧은지 확인하고 갱신합니다
- Vnear까지의 최단 경로를 저장하고 그 노드를 처리 완료 합니다.

```python
if (__name__ == "__main__") :

    inf = 1000
    w = [[0, 2, 3, inf, inf, 4],
         [inf, 0, 4, 5, 3, inf],
         [inf, inf, 0, 6, 2, inf],
         [inf, inf, inf, 0, 2, inf],
         [inf, inf, inf, inf, 0, inf],
         [inf, inf, inf, inf, 1, 0]]

    D = dijkstra(w)
    node = ['A', 'B', 'C', 'D', 'E', 'F']

    print(D.f)
    print(D.save_length)

    for i in range (1, len(w)) :
        path = []
        current = i
        while (current != 0) :
            path.insert(0, node[current])
            current = D.touch[current]
        path.insert(0, node[0])
        distance = D.save_length[i]
        print(f"Path '{ ' → '.join(path):}' :: Distance : { distance }")
```

## 샘플 동작 과정

Path 'A → B' :: Distance : 2
Path 'A → C' :: Distance : 3
Path 'A → F' :: Distance : 4
Path 'A → B → E' :: Distance : 5
Path 'A → B → D' :: Distance : 7



# 알고리즘

Divide and Conquer (분할과 정복)

Dynamic Programming (동적 프로그래밍)

Greedy Algorithm (탐욕 알고리즘)

**Backtracking (되추적법)**

Branch (분기한정법)

Sort (정렬)

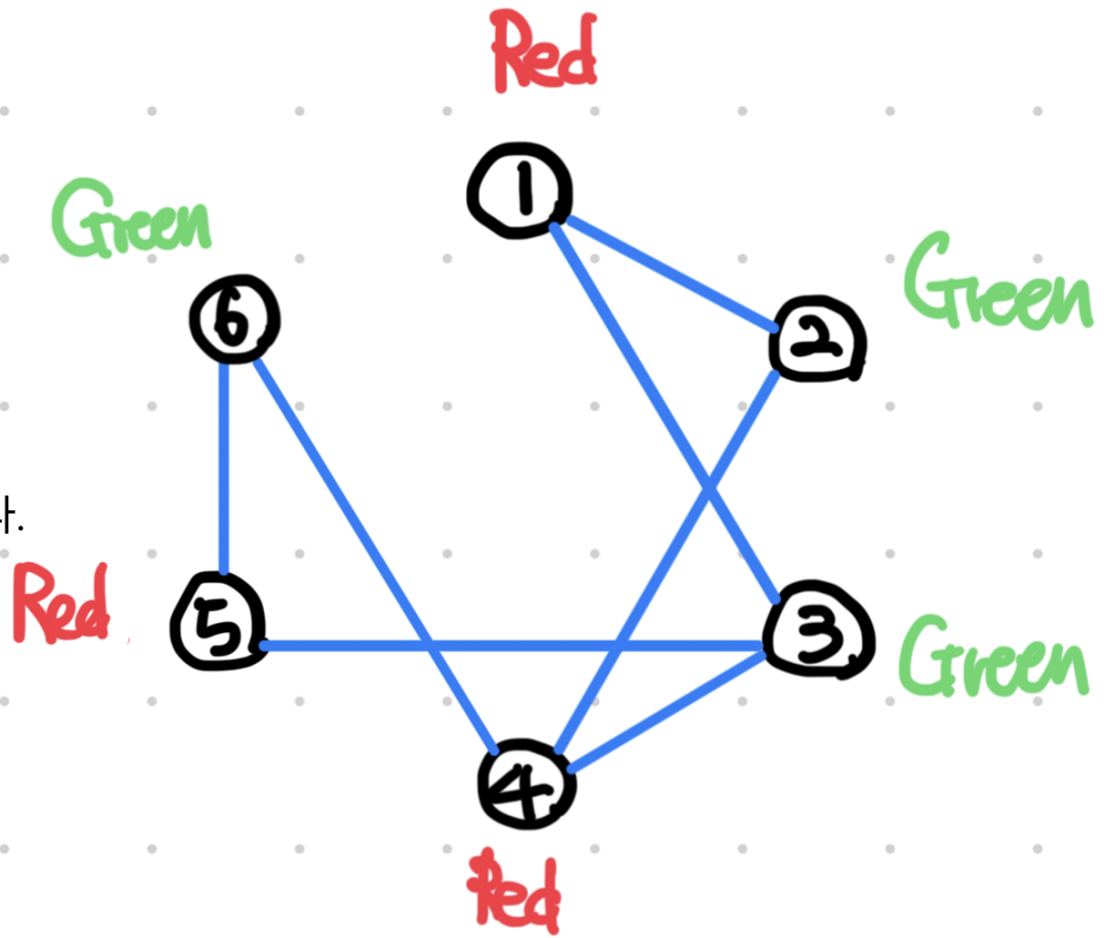
Search (탐색)

# N-coloring

N-Coloring (지구본 색칠하기)

- 인접한 노드는 다른 색을 칠한다.

1. 매번 색칠할 때마다 제약 조건을 확인한다.
   - 제약 조건 : 인접한 노드에 같은 색이 없다.
   - 제약 조건에 위배되면
     (Back Tracking) : 해당 경로 탐색 중단
   - 아니면 계속 색칠하기
2. 색을 모두 칠했으면 Solutions에 추가

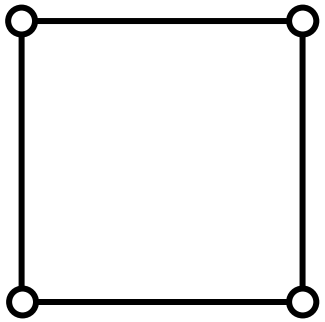


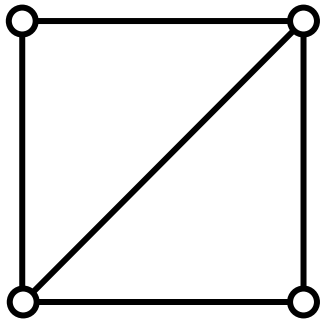
# Planar Graph

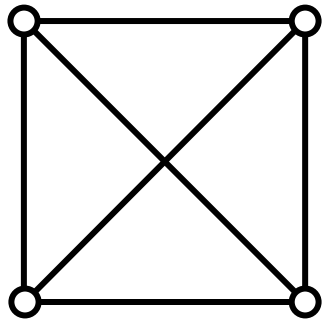
평면상으로 그래프를 그렸을 때, 두 변이 꼭짓점 이외에 만나지 않도록 그릴 수 있는 그래프

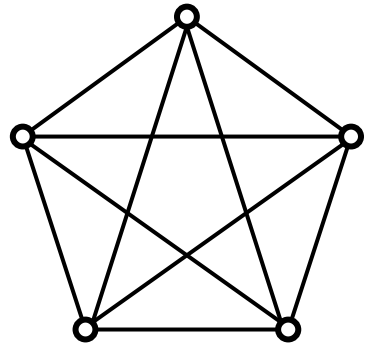
→ 교차점이 꼭짓점 이외에 없다.

Graph

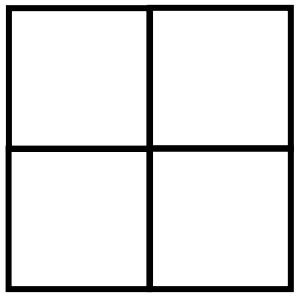


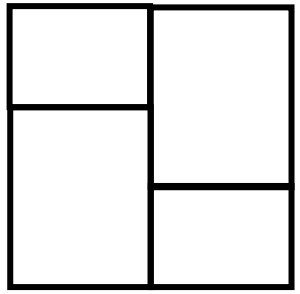


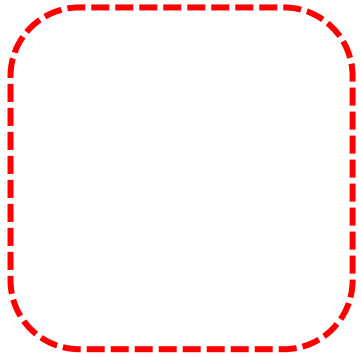


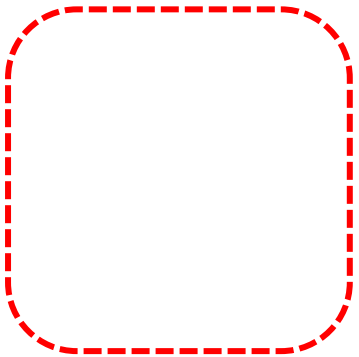


Map









Planar Graph N-Coloring → (  )개면 된다.

# 알고리즘

Divide and Conquer (분할과 정복)

Dynamic Programming (동적 프로그래밍)

Greedy Algorithm (탐욕 알고리즘)

Backtracking (되추적법)

**Branch (분기한정법)**

Sort (정렬)

Search (탐색)

# 0-1 KnapSack

정해진 최대 무게(W)에 넘치지 않고, 최대 가치로 물건을 담는 문제

```
W = 16                  # 최대무게
p = [40, 30, 10, 50]    # 각 물건의 가치 (profit)
w = [2, 5, 5, 10]       # 각 물건의 무게 (weight)
```

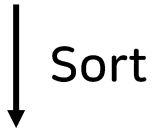
Dynamic Programming → (  )개의 Node

Backtracking → (  ) 검사 (3글자)

Branch and Bound → (  )

→ 한계; 가능한 경우의 최대치

| P (profit) | W (weight) | p/w |
|---|---|---|
| 40 | 2 | 20 |
| 30 | 5 | 6 |
| 10 | 5 | 2 |
| 50 | 10 | 5 |




| P (profit) | W (weight) | p/w |
|---|---|---|
| 40 | 2 | 20 |
| 30 | 5 | 6 |
| 50 | 10 | 5 |
| 10 | 5 | 2 |

| P (profit) | W (weight) | p/w |
|---|---|---|
| 40 | 2 | 20 |
| 30 | 5 | 6 |
| 50 | 10 | 5 |
| 10 | 5 | 2 |

Bound = (지금까지의 profit) + (W가 넘지 않는 선에서의 최대 profit) + (남은 무게) * (p/w)

→ 잘라서 구겨 넣는다.

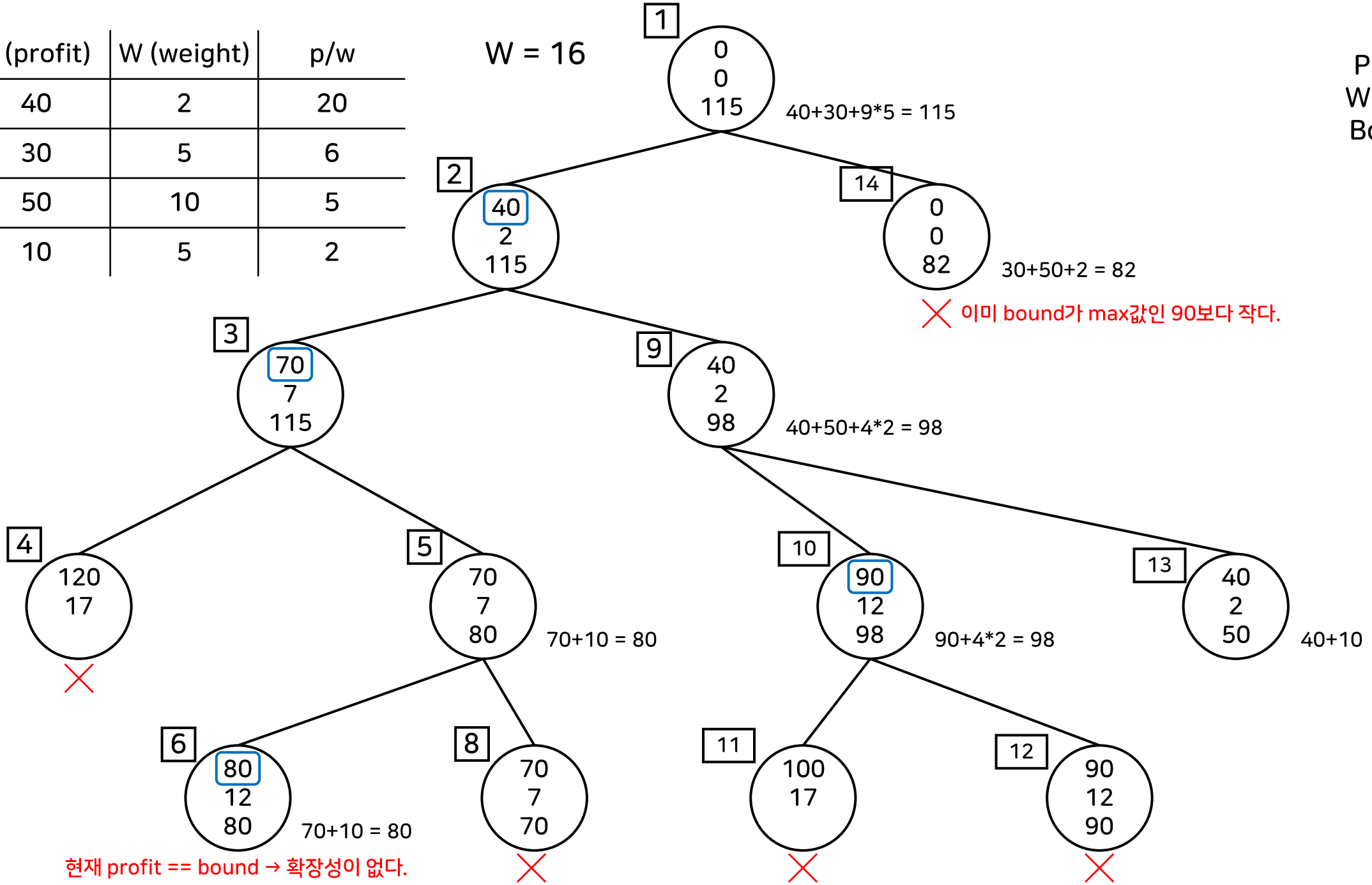
W = 16

Profit → 40 30 9 * 5 = 45 Bound = 115

Weight → 2 5 16 – 7 = 9

| P (profit) | W (weight) | p/w |
|---|---|---|
| 40 | 2 | 20 |
| 30 | 5 | 6 |
| 50 | 10 | 5 |
| 10 | 5 | 2 |

W = 16



# 알고리즘

Divide and Conquer (분할과 정복)

Dynamic Programming (동적 프로그래밍)

Greedy Algorithm (탐욕 알고리즘)

Backtracking (되추적법)

Branch (분기한정법)

Sort (정렬)

Search (탐색)

# Sort (정렬)

김혜란

- 목록 안에 저장된 요소들을 특정한 순서대로 정렬하는 알고리즘

## 1. Bubble sort

- 매번 인접한 두 요소를 비교하여 더 큰 값을 오른쪽으로 이동시키는 방식으로 작동
- 매 반복마다 가장 큰 값이 배열의 끝으로 이동하며, 전체 배열이 정렬될 때까지 수행
- 단순한 만큼 비효율적
- 시간 복잡도는 $O(n^2)$


## 2. Exchange sort

- 배열의 각 요소를 차례대로 탐색하면서, 다른 요소들과 비교하여 더 작은 값을 찾을 때마다 교환하는 방식
- 배열에서 첫 번째 원소를 선택하고, 그 원소와 배열의 나머지 원소들을 하나씩 비교 -> 만약 더 작은 원소를 찾으면 두 원소를 교환 -> 이걸 반복해서 배열의 끝까지 비교와 교환을 반복하면 배열이 정렬됨
- 간단하지만 비효율적
- 시간 복잡도는 $O(n^2)$

## 3. Insertion sort

- 정렬을 진행할 원소의 index보다 낮은 곳에 있는 원소들을 탐색하며 알맞은 위치에 삽입
- 왼쪽으로 이동하면서, 자신보다 큰 값이 있는 경우 계속 교환하다가, 작은 값이 나오면 그 자리에 멈추고 삽입
- 두 번째 index부터 시작한다. 그 이유는 첫 번째 index는 비교할 원소가 없기 때문
- 맨 왼쪽 index까지 탐색하지 않아도 된다는 장점이 있다
- 최악의 경우 $O(n^2)$의 시간 복잡도

## 4. Selection sort

- 배열에서 최소값을 반복적으로 찾아 정렬하는 알고리즘
- 시간복잡도 $O(n2)$에 해당하는 비효율적인 알고리즘
- 모든 경우의 수를 전부 확인

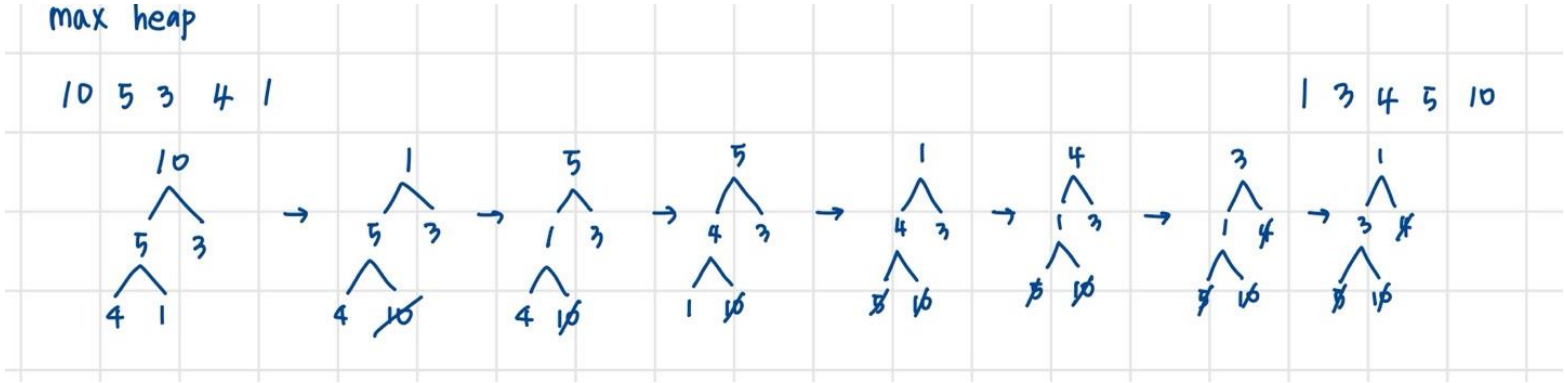
1. 주어진 배열에서 최소값을 찾는다.
2. 최소값을 맨 앞의 값과 바꾼다.
3. 바꿔준 맨 앞 값을 제외한 나머지 원소를 동일한 방법으로 바꿔준다

- 최소 선택 정렬과, 최대 선택 정렬이 있다. 최소는 위와 같이 오름차순으로 정렬하는 것이고 최대는 위와 반대로 내림차순으로 정렬하는 것

## 5. Heap sort

- 완전 이진 트리를 기반으로 하는 정렬 알고리즘
- 평균 시간 복잡도가 O(n log n)
- 제자리 정렬(In-place Sorting)로 추가 메모리를 사용하지 않고 정렬할 수 있다는 장점이 있음
- 정렬과정

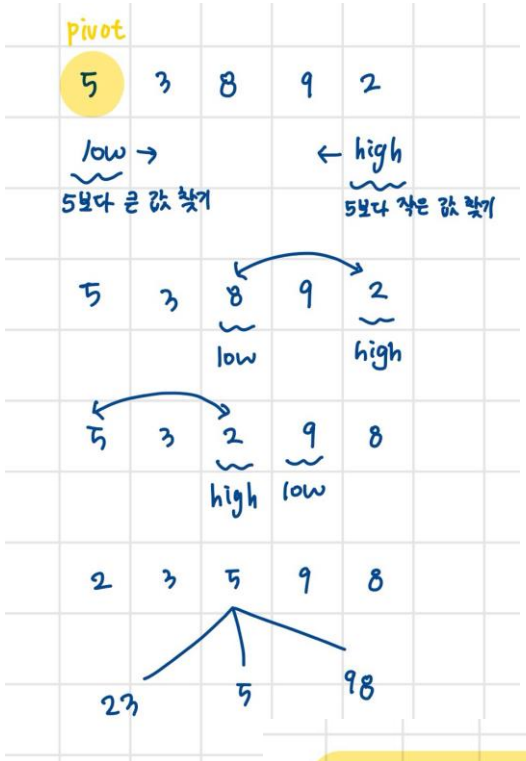
1. 배열의 루트(가장 큰 값)를 배열의 마지막 값과 교환
2. 배열의 크기를 1씩 줄여가며 나머지 배열을 다시 힙 구조로 재정비(heapify)
3. 이 과정을 배열의 모든 요소에 대해 반복.



## 6. Quick sort

- 분할정복법과 재귀를 사용해 정렬하는 알고리즘
- 평균 시간 복잡도는 O(n log n), 매우 빠름
- 피봇(Pivot)이라는 개념이 사용 -> 정렬 될 기준 원소(보통 배열의 첫 번째 값이나 중앙 값을 선택)

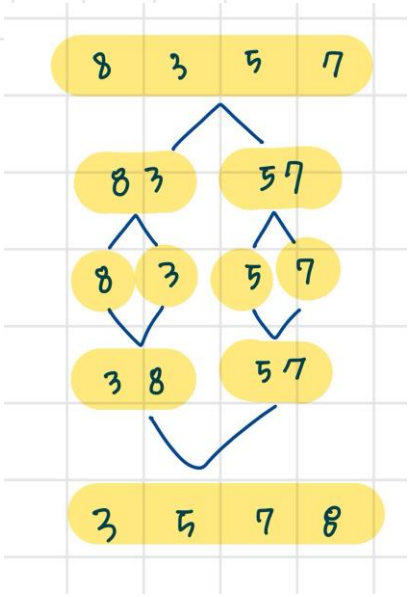
1. 피봇 선택
2. Low는 피봇보다 큰 값 찾으면 멈춤, high는 피봇보다 작은 값 찾으면 멈춤
3. Low, high 둘 다 멈췄고 low < high 면 둘이 교환
4. High < low 면 high랑 피벗 교환
5. 분할 후 같은 과정 반복




## 7. Merge sort

- 분할정복 방식으로 작동하는 알고리즘
- 배열을 원소가 하나만 남을 때 까지 계속 이분할 한 다음, 대소관계를 고려하여 다시 재배열 하며 원래 크기의 배열로 병합
- 시간 복잡도 $O(nlogn)$

1. 리스트의 길이가 0 또는 1이면 이미 정렬된 것으로 본다.
2. 그렇지 않은 경우에는 리스트를 이분할 한다.
3. 각 부분 리스트를 재귀적으로 합병 정렬을 이용해 정렬한다.
4. 두 부분 리스트를 다시 하나의 정렬된 리스트로 합병한다.



# 알고리즘

Divide and Conquer (분할과 정복)

Dynamic Programming (동적 프로그래밍)

Greedy Algorithm (탐욕 알고리즘)

Backtracking (되추적법)

Branch (분기한정법)

Sort (정렬)

Search (탐색)

# Search (검색)

김혜란

## Linear Search

처음부터 끝까지 하나씩 순서대로 비교하며 원하는 값을 찾아내기

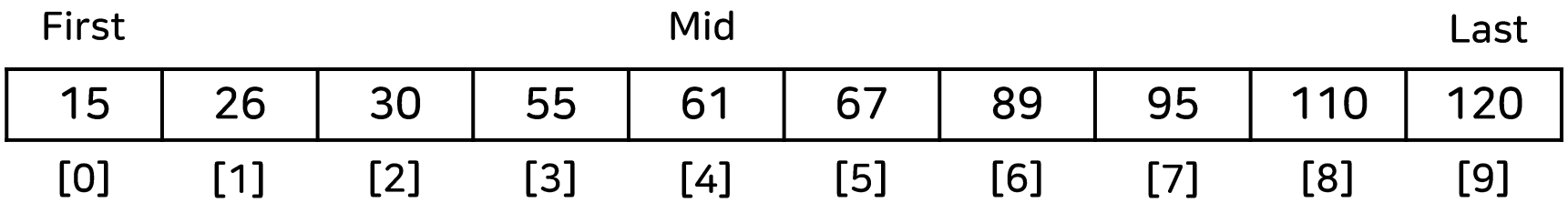
시간복잡도 O( )

## Binary Search

1. First Mid Last 로 구성
2. 찾는 값이 mid 보다 크면 first = mid + 1, mid 보다 작으면 last = mid - 1, mid랑 동일하면 종료
3. Last < First 되면 종료

시간복잡도 O( )

Binary Search (45)



## Task

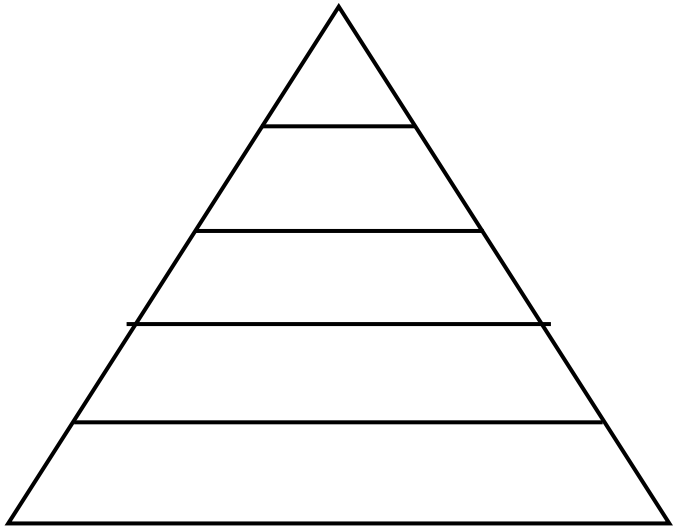
T/F : 지식을 판단하기 위함

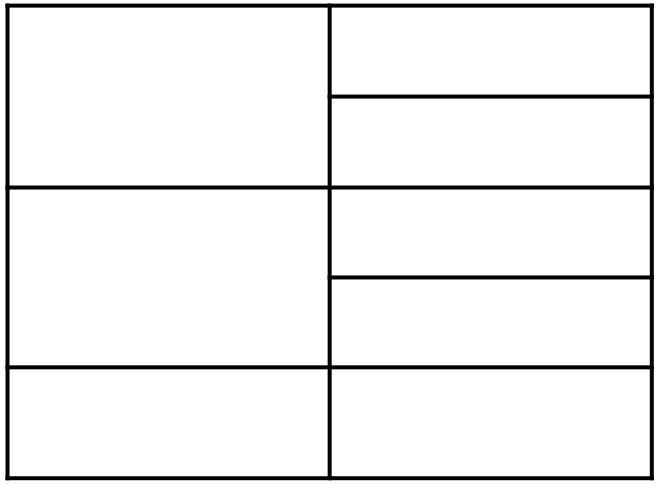
T/F : 지식을 판단하기 위함

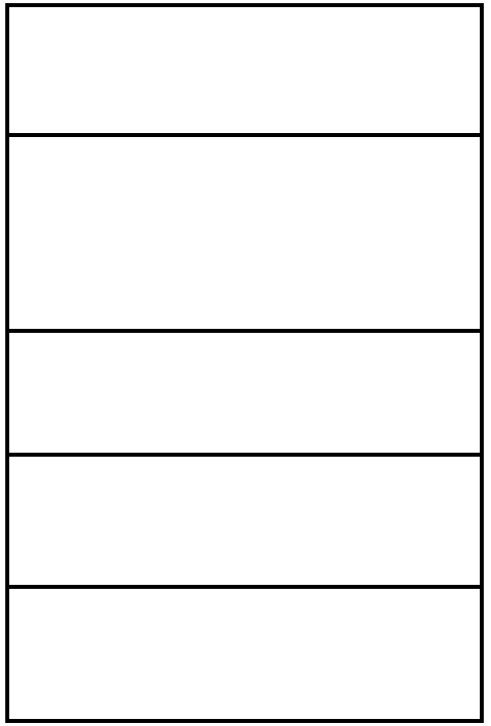
T/F : 지식을 판단하기 위함

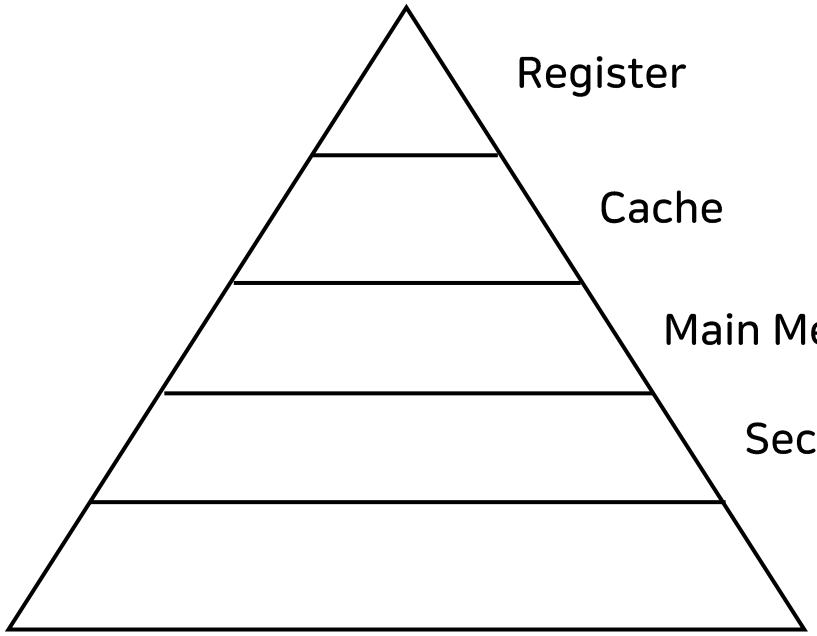
) 알고리즘 전체

# 컴퓨터구조





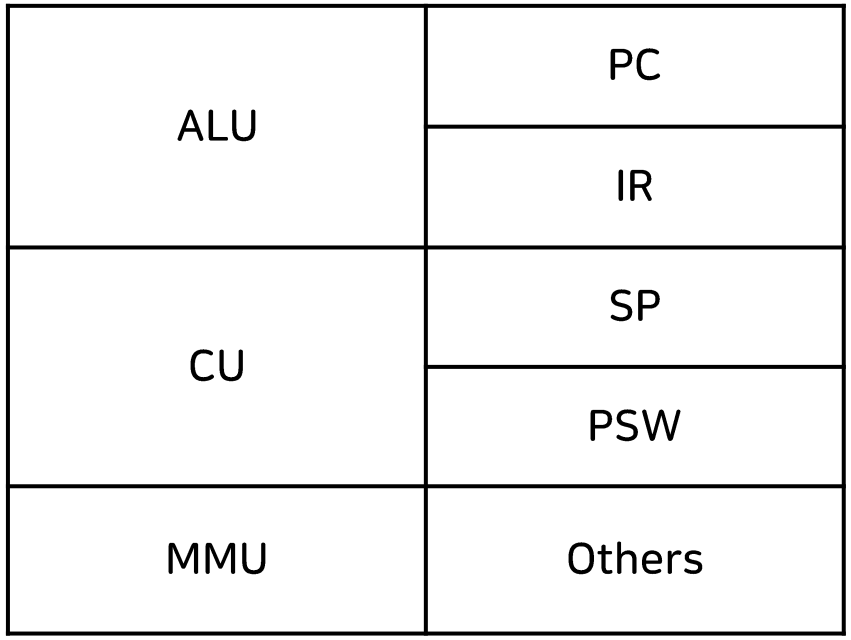





Memory Architecture

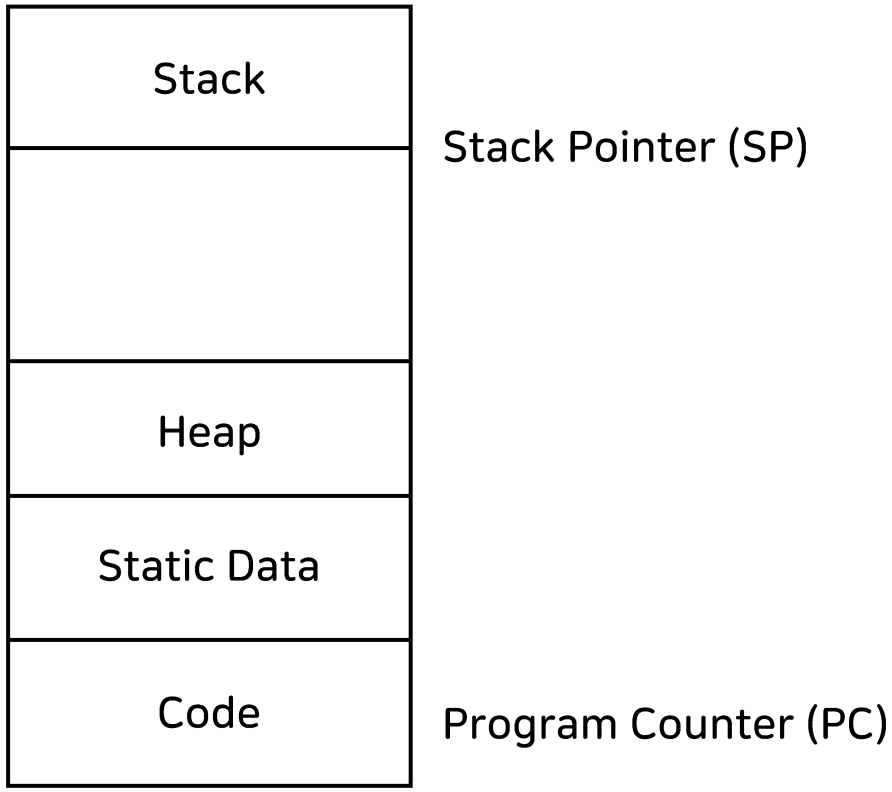
SRAM vs DRAM

ROM vs RAM




CPU Architecture

Pipelining
: Fetch, Decode, Execute, Write Back




Process Address Space

What is Process?

Process Control Block (PCB)