Big O

**□ 복잡도**

**-** Time Complexity(시간 복잡도) : 얼마나 빨리 수행하느냐?

- Space Complexity(공간 복잡도) : 메모리를 얼마나 쓰느냐?

**□ 표기법**

- 오메가(Ω) 표기법 : 실행 시간의 하한, 최선의 경우를 산정

- 세타(θ) 표기법 : 실행 시간의 평균, 최선과 최악 사이의 시간

- 오미크론(Ο, Big O) 표기법 : 실행 시간의 상한, 최악의 경우

가장 빠르게 증가하는 항만 고려

**□ O(n) : 수행 시간이 n에 정비례 증가**

|  |
| --- |
| def print\_items(n):  for i in range(n):  print(i) |

**□ O(n2) : 수행 시간이 n에 제곱 비례 증가**

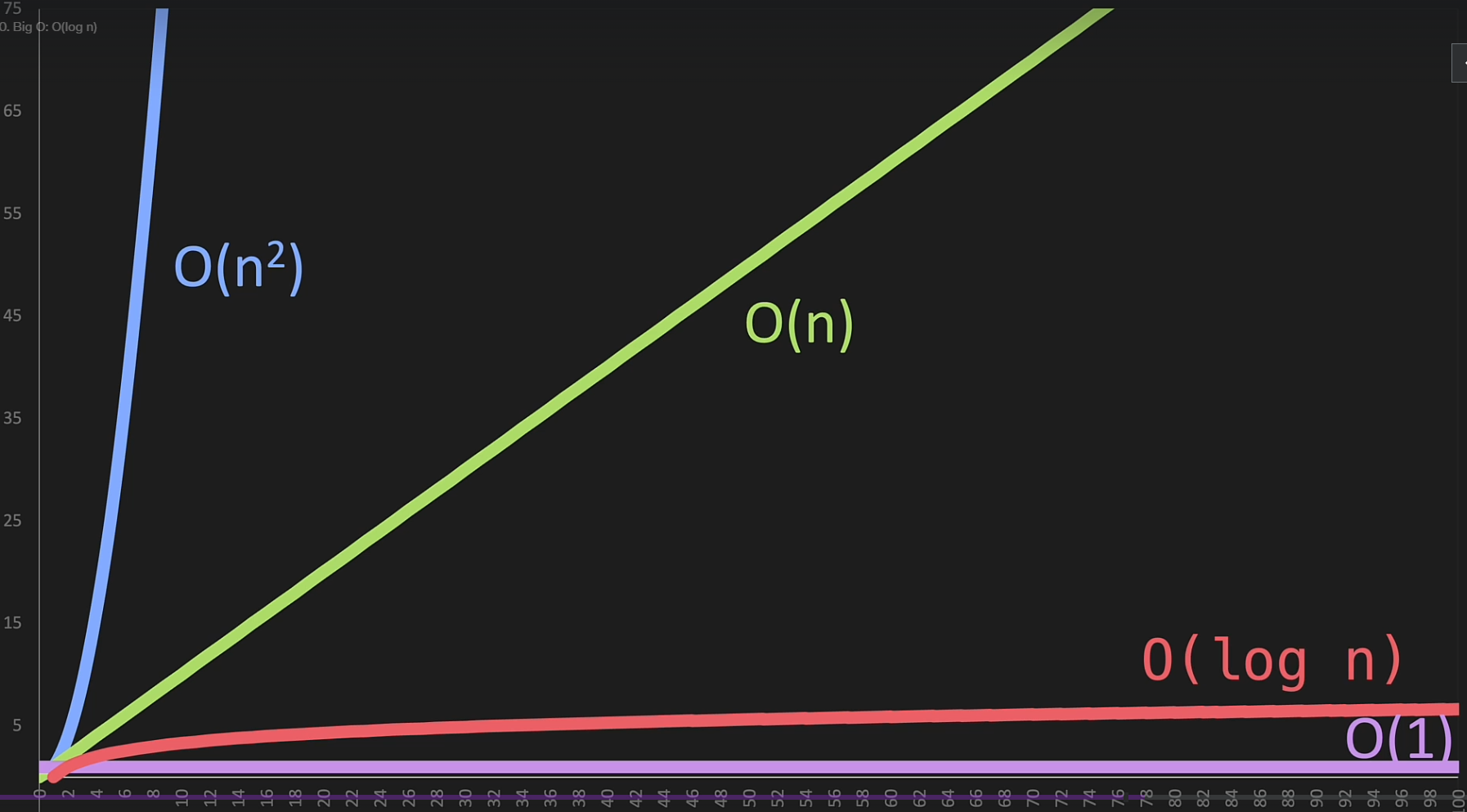
|  |
| --- |
| def print\_items(n):  for I in range(n):  for j in range(n):  print(I, j) |

**□ O(1) : 수행 시간이 n에 상관 없이 일정**

|  |
| --- |
| def add\_items(n):  return n+n |

**□ O(log n)**

**- 수행 시간이 log n 만큼 증가함**



**※ 적용 규칙**

- Drop Constants : 상수는 무시

. O(n + n) = O(2n) → O(n)

- Drop Non-Dominants : 지배적이지 않은 것은 무시

. O(n2 +n) → O(n2)

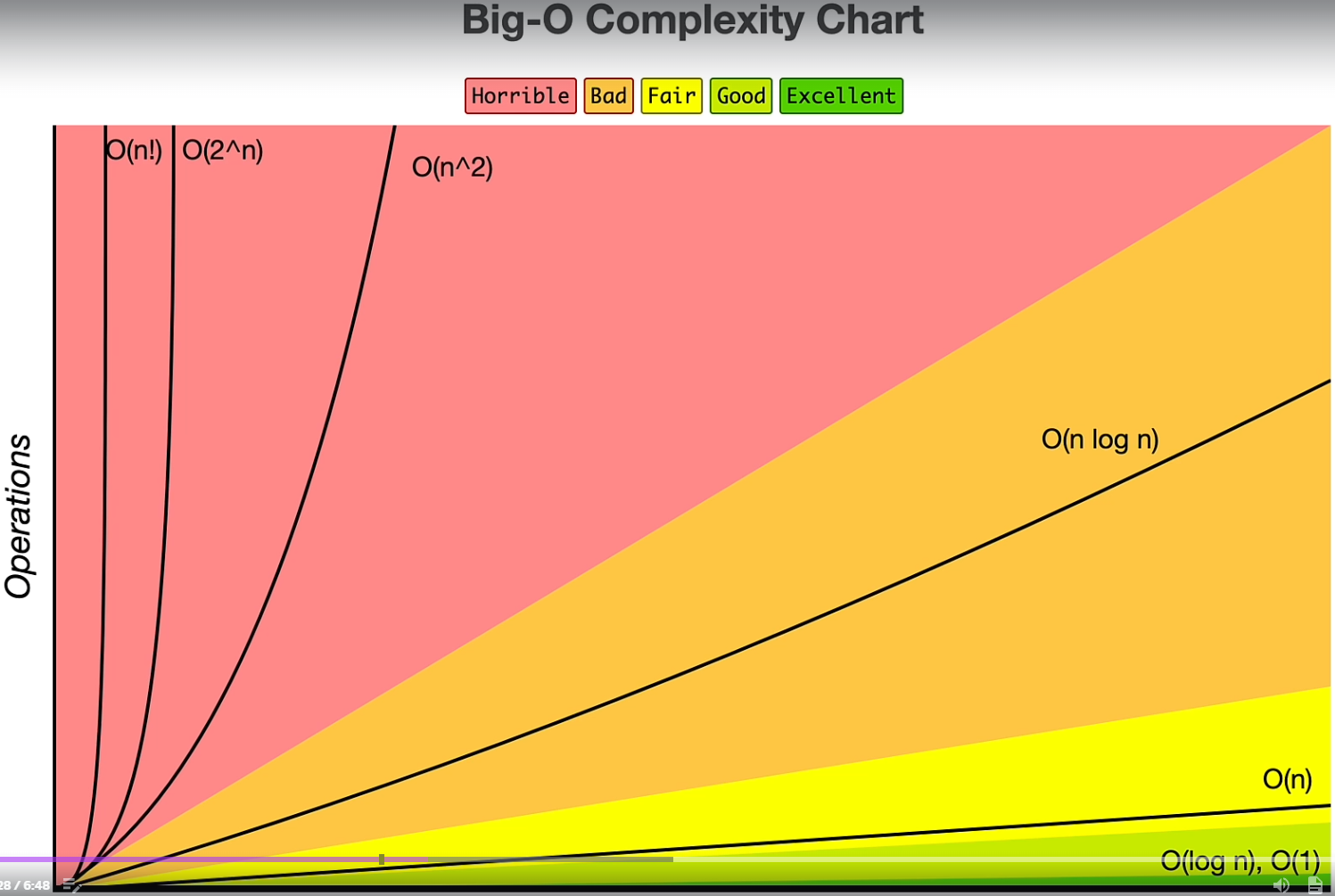
**※ List**

- append(), pop() : O(1) → 마지막에 넣고 빼기 때문

- insert() : O(n) → 인덱스를 다시 정렬해야 함

- my\_list[1] : O(1) → 인덱스를 이용해 바로 찾아감





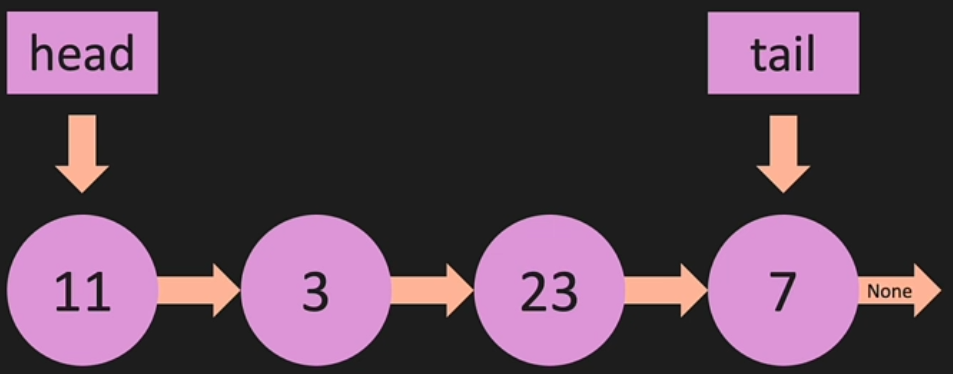
**※ 1초 기준(Python 기준)**

- 2,000만번 연산

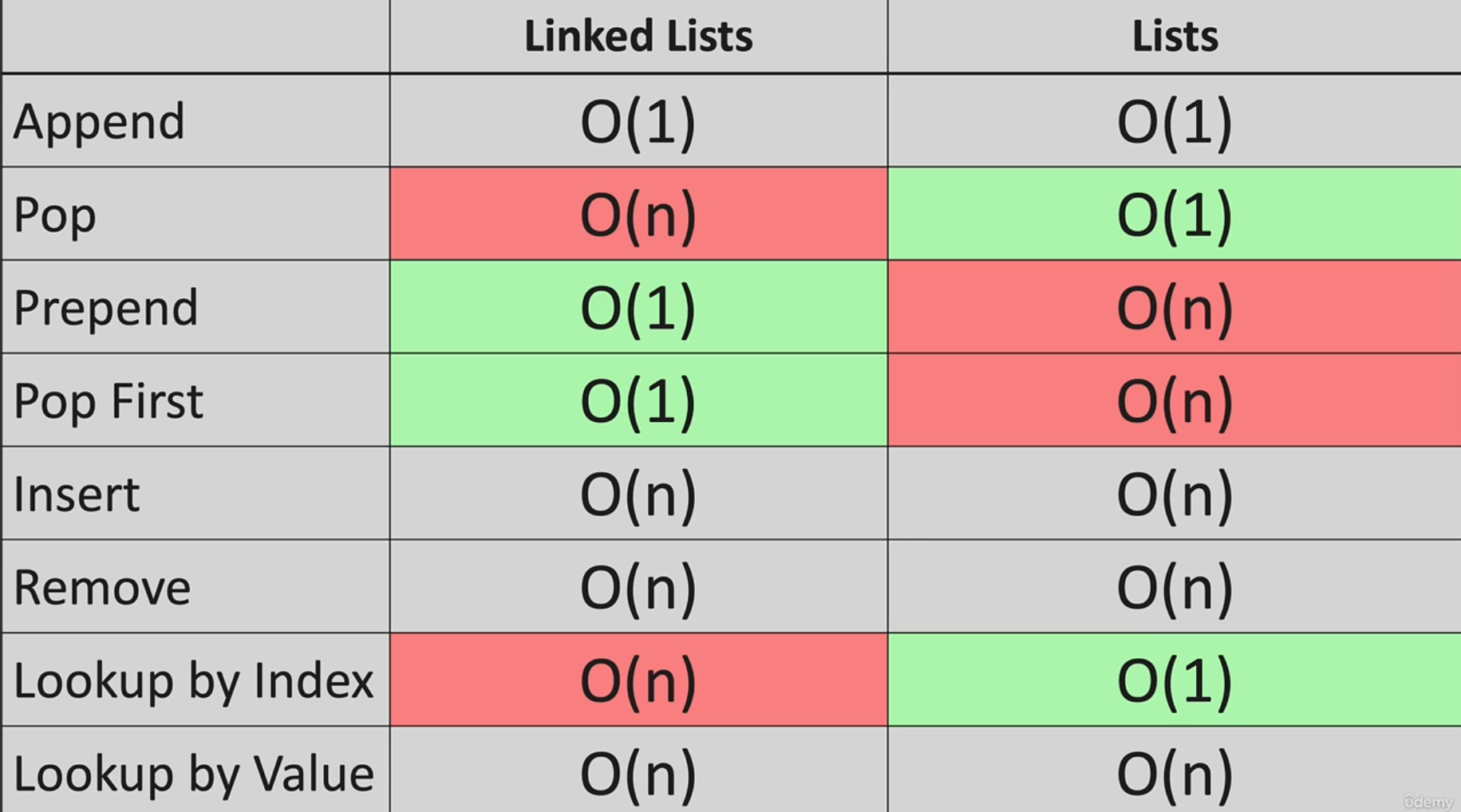
- N = 1,000,000일 경우 O(NlogN) 알고리즘일 경우 20,000,000

Linked List

**□ 개요**

****

**□ Big O**

****

**□ Code**

|  |
| --- |
| class Node:  def \_\_init\_\_(self, value):  self.value = value  self.next = None    class LinkedList:  def \_\_init\_\_(self, value):  new\_node = Node(value)  self.head = new\_node  self.tail = new\_node  self.length = 1  def append(self, value):  new\_node = Node(value)  if self.length == 0:  self.head = new\_node  self.tail = new\_node  else:  self.tail.next = new\_node  self.tail = new\_node  self.length += 1  return True  def pop(self):  if self.length == 0:  return None  temp = self.head  pre = self.head  while(temp.next):  pre = temp  temp = temp.next  self.tail = pre  self.tail.next = None  self.length -= 1  if self.length == 0:  self.head = None  self.tail = None  return temp  def prepend(self, value):  new\_node = Node(value)  if self.length == 0:  self.head = new\_node  self.tail = new\_node  else:  new\_node.next = self.head  self.head = new\_node  self.length += 1  return True  def pop\_first(self):  if self.length == 0:  return None  temp = self.head  self.head = self.head.next  temp.next = None  self.length -= 1  if self.length == 0:  self.tail = None  return temp  def get(self, index):  if index < 0 or index >= self.length:  return None  temp = self.head  for \_ in range(index):  temp = temp.next  return temp    def set\_value(self, index, value):  temp = self.get(index)  if temp:  temp.value = value  return True  return False    def insert(self, index, value):  if index < 0 or index > self.length:  return False  if index == 0:  return self.prepend(value)  if index == self.length:  return self.append(value)  new\_node = Node(value)  temp = self.get(index - 1)  new\_node.next = temp.next  temp.next = new\_node  self.length += 1  return True  def remove(self, index):  if index < 0 or index >= self.length:  return None  if index == 0:  return self.pop\_first()  if index == self.length - 1:  return self.pop()  pre = self.get(index - 1)  temp = pre.next  pre.next = temp.next  temp.next = None  self.length -= 1  return temp  def reverse(self):  temp = self.head  self.head = self.tail  self.tail = temp  after = temp.next  before = None  for \_ in range(self.length):  after = temp.next  temp.next = before  before = temp  temp = after |

Stack

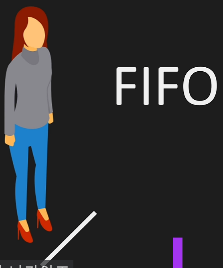
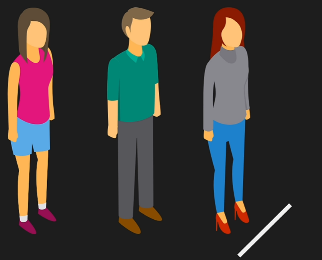
**□ 개요**

**-** 선입후출

**□ Code**

|  |
| --- |
| Stack = []  stack.append(5) # 삽입  Stack.pop() # 삭제 |

Queue

**□ 개요**

**-** 선입선출

**□ Code**

|  |
| --- |
| from collections import deque  queue = deque()  queue.append(5) # 삽입  queue.popleft() # 삭제 |

Recursion

**□ 개요**

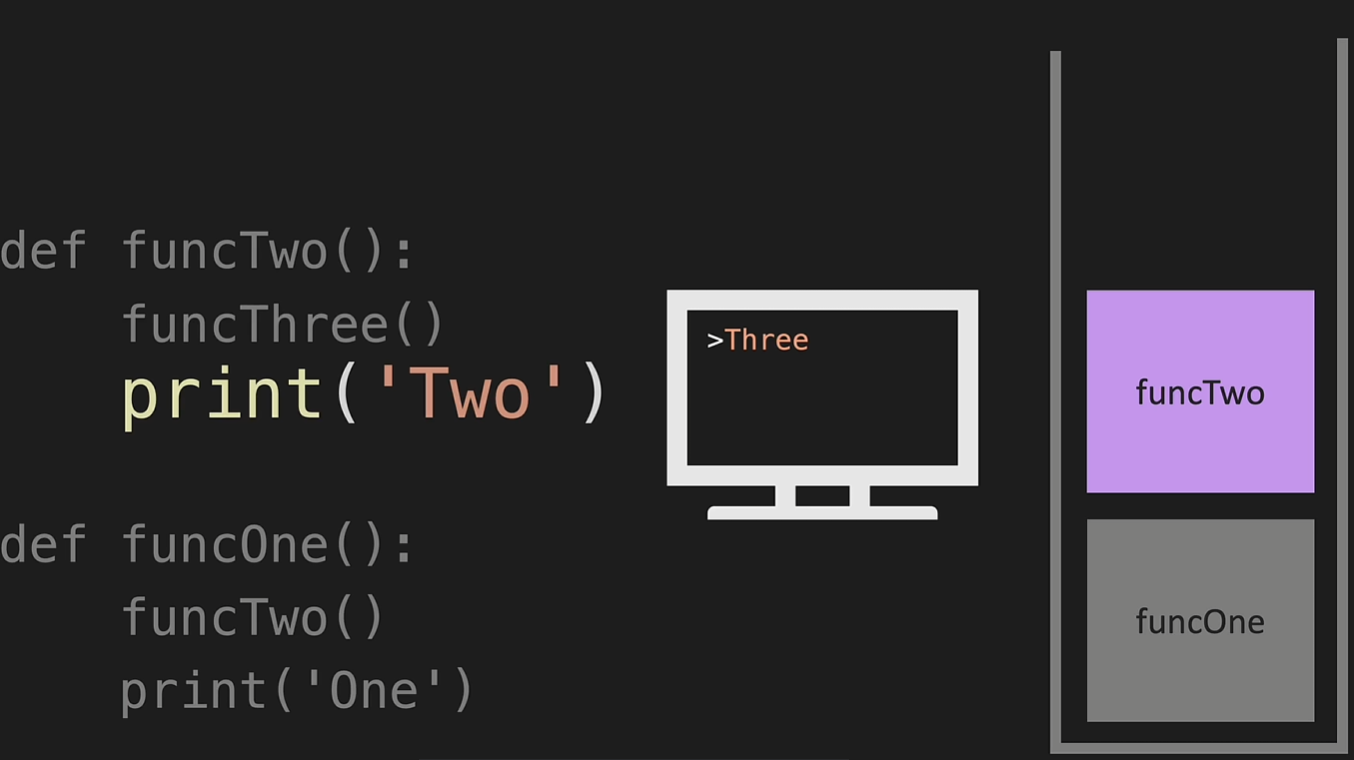
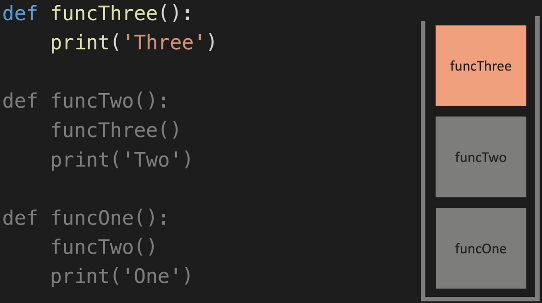
- 자기자신을 호출하는 함수

- stack 이용

- Return문이 꼭 있어야 함(종료 조건)

**□ Call Stack**

- 호출되는 순서에 따라 Stack에 쌓였다가 제거됨



**□ Code**

|  |
| --- |
| def factorial(n):  if n == 1:  return 1  return n \* factorial(n-1)  factorial(5) |

DFS

**□ 개요**

- Depth First Search, 깊이 우선 탐색

- 스택 또는 재귀함수 사용

**□ 알고리즘**

① 시작 노드 스택에 저장

② 스택의 최상단 노드를 꺼내서 방문하지 않았다면 방문처리

연결된 노드를 스택에 삽입, 방문한 노드는 Pass

③ 2를 반복

**□ Code**

|  |
| --- |
| graph = {1:[2,3,8],  2:[1,7],  3:[1,4,5],  4:[3,5],  5:[3,4],  6:[7],  7:[2,6,8],  8:[1,7]} |

|  |
| --- |
| # Stack 사용  from collections import deque  stack = deque()  visited = set()  def dfs(graph, start):  stack.append(start)  while stack:  node = stack.pop()  if node not in visited:  visited.add(node)  stack.extend(graph[node])  print(node)  dfs(graph, 1) |

|  |
| --- |
| # 재귀 용법  visited = set()  def dfs(graph, node):  visited.add(node)  print(node)  for i in graph[node]:  if i not in visited:  dfs(graph, i)  dfs(graph, 1) |

BFS

**□ 개요**

- Breadth First Search, 너비 우선 탐색

- 큐 사용

**□ 알고리즘**

① 시작 노드 큐에 저장

② 스택의 최상단 노드를 꺼내서 방문하지 않았다면 방문처리

연결된 노드를 큐에 삽입, 방문한 노드는 Pass

③ 2를 반복

**□ Code**

|  |
| --- |
| from collections import deque  queue = deque()  visited = set()  def bfs(graph, start):  queue.append(start)  while queue:  node = queue.popleft()  if node not in visited:  visited.add(node)  queue.extend(graph[node])  print(node)  bfs(graph, 1) |

Binary Search

**□ 개요**

- 정렬된 데이터 전제조건

- 시작점 index, 끝점 index를 가지고 중간점을 찾아가는 방식

- O(logN), 탐색 범위가 1,000만 이상일 경우 고려할 것

**□ 알고리즘**

① 중간 index를 구한다(실수이면 소수점 버림)

② 중간값과 Target을 비교하여 같으면 Return

중간값이 Target 보다 크면 끝점 index를 중간으로 이동

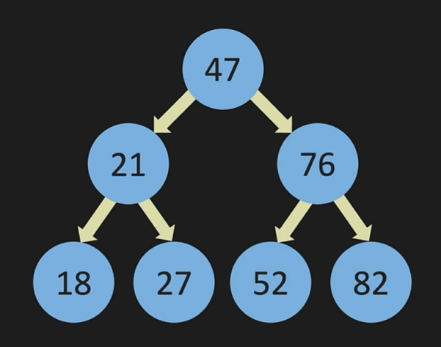
중간값이 Target 보다 작으면 시작점 Index를 중간으로 이동

③ 1~2를 반복

**□ Code**

|  |
| --- |
| def binary\_search(array, target, start, end):  while start <= end:  mid = (start + end) // 2 # 주의, index 합을 2로 나눈 몫  if array[mid] == target:  return mid  elif array[mid] > target:  end = mid - 1 # mid 이전 index  else:  start = mid + 1 # mid 다음 index  return None  array = [1,3,5,7,9,11,13,15,17,19]  binary\_search(array, 7, 0, 9) |

Binary Search Trees

**□ 개요**

- 비교 노드보다 작으면 왼쪽,

크면 오른쪽에 배치

- O(logN)

**□ 알고리즘**

- 비교 노드보다 작으면 왼쪽,

**□ Code**

|  |
| --- |
| class Node:  def \_\_init\_\_(self, value):  self.value = value  self.left = None  self.right = None    class BinarySearchTree:  def \_\_init\_\_(self):  self.root = None  def insert(self, value):  new\_node = Node(value)  if self.root is None:  self.root = new\_node  return True  temp = self.root  while (True):  if new\_node.value == temp.value:  return False  if new\_node.value < temp.value:  if temp.left is None:  temp.left = new\_node  return True  temp = temp.left  else:  if temp.right is None:  temp.right = new\_node  return True  temp = temp.right  def contains(self, value):  temp = self.root  while temp is not None:  if value < temp.value:  temp = temp.left  elif value > temp.value:  temp = temp.right  else:  return True  return False |

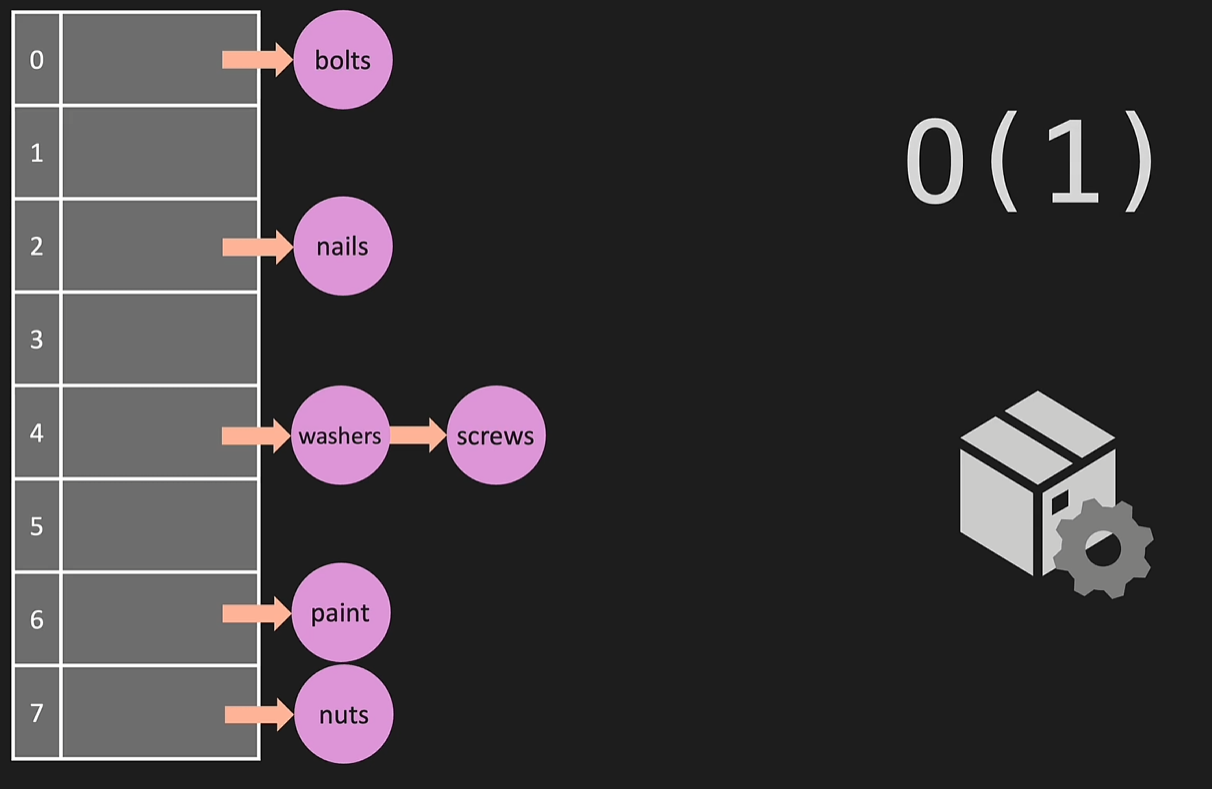
Hash Tables

**□ 개요**

- 한쪽 방향으로만 수행됨

- 결정론적 : 매번 동일한 결과를 도출함

**□ Big O**

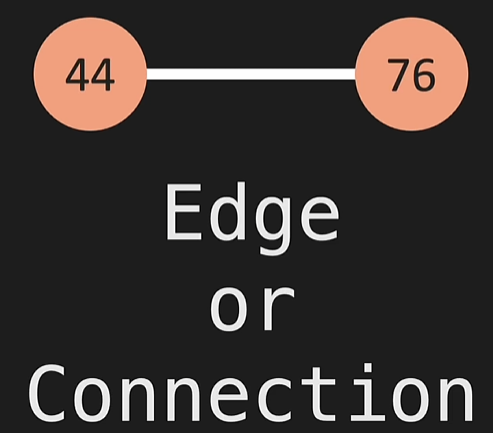


**□ Code**

|  |
| --- |
| class HashTable:  def \_\_init\_\_(self, size = 7):  self.data\_map = [None] \* size    def \_\_hash(self, key):  my\_hash = 0  for letter in key:  my\_hash = (my\_hash + ord(letter) \* 23) % len(self.data\_map)  return my\_hash  def print\_table(self):  for i, val in enumerate(self.data\_map):  print(i, ": ", val)    def set\_item(self, key, value):  index = self.\_\_hash(key)  if self.data\_map[index] == None:  self.data\_map[index] = []  self.data\_map[index].append([key, value])    def get\_item(self, key):  index = self.\_\_hash(key)  if self.data\_map[index] is not None:  for i in range(len(self.data\_map[index])):  if self.data\_map[index][i][0] == key:  return self.data\_map[index][i][1]  return None  def keys(self):  all\_keys = []  for item in self.data\_map:  if item is not None:  for k in item:  all\_keys.append(k[0])  return all\_keys |

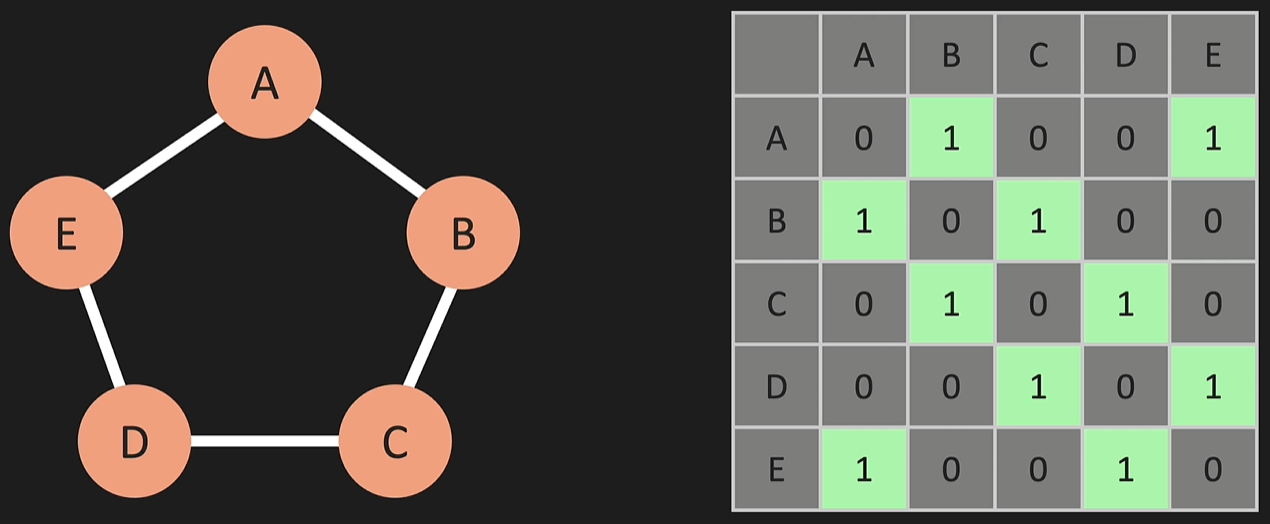
Graph

**□ 개요**

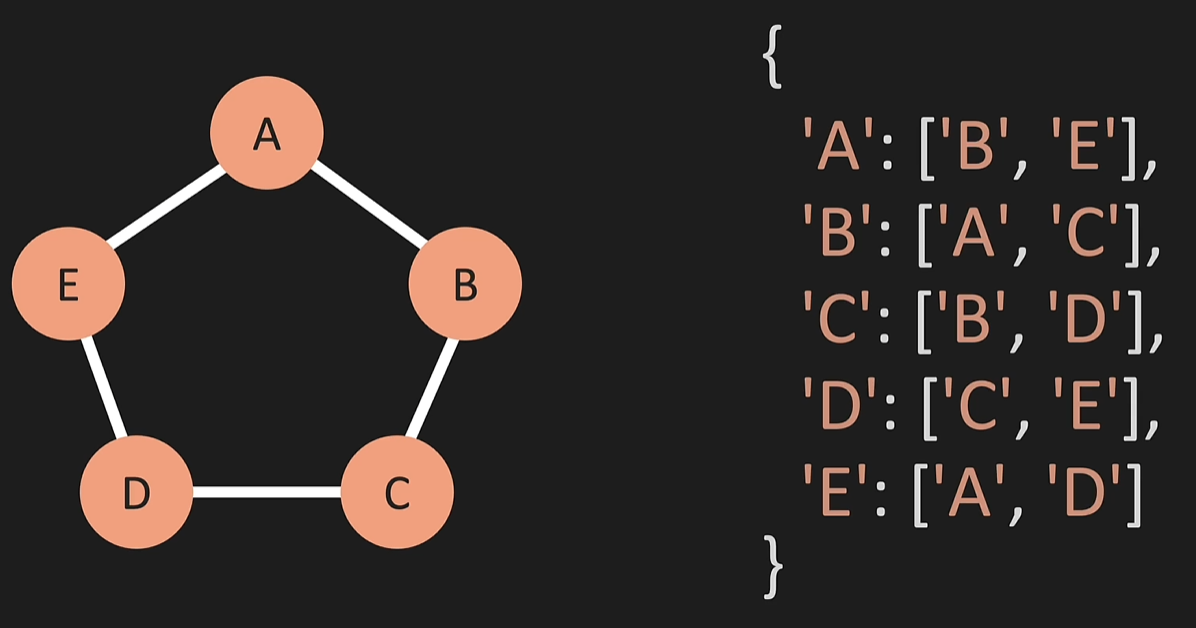
 

**□ Adjacency Matrix vs Adjacency List**

- Adjacency Matrix(인접행력) : 메모리 소비가 심함

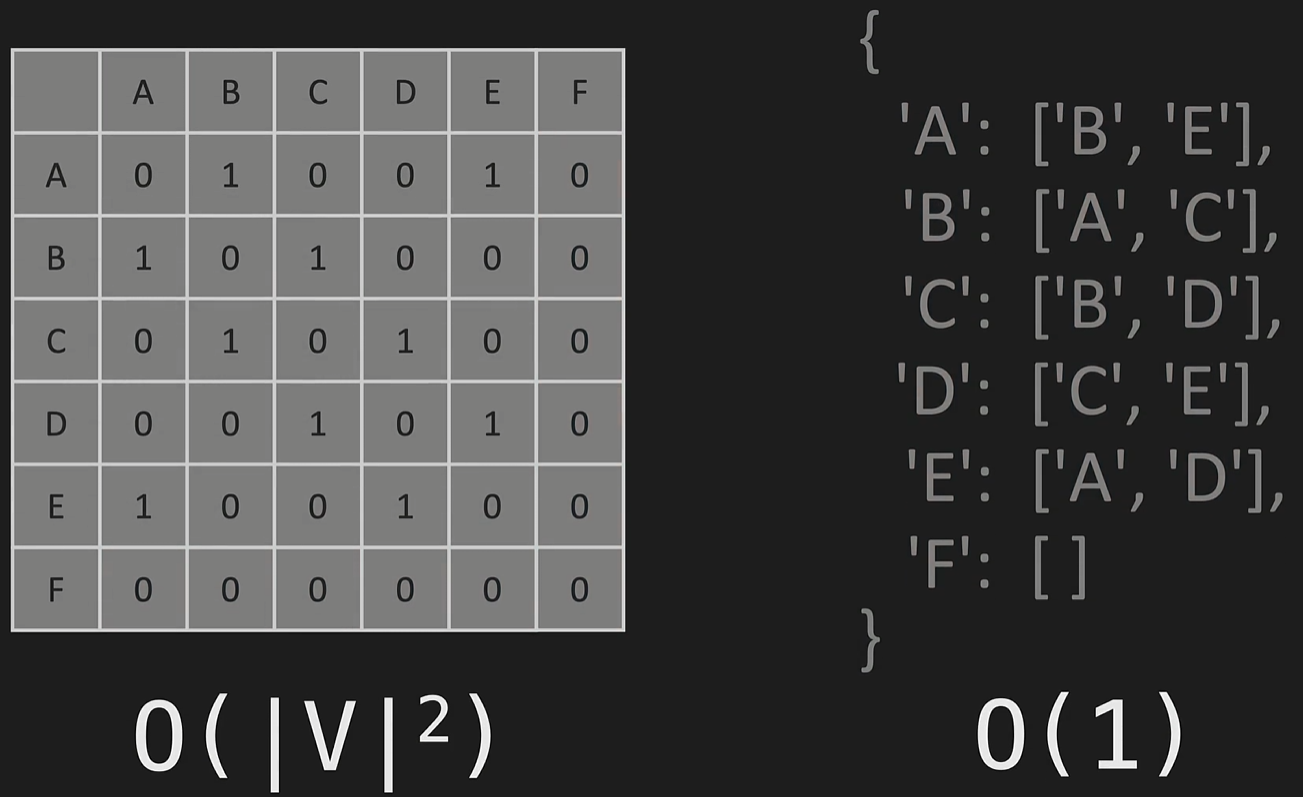


- Adjacency list(인접리스트) : 앞으로 사용할 형태

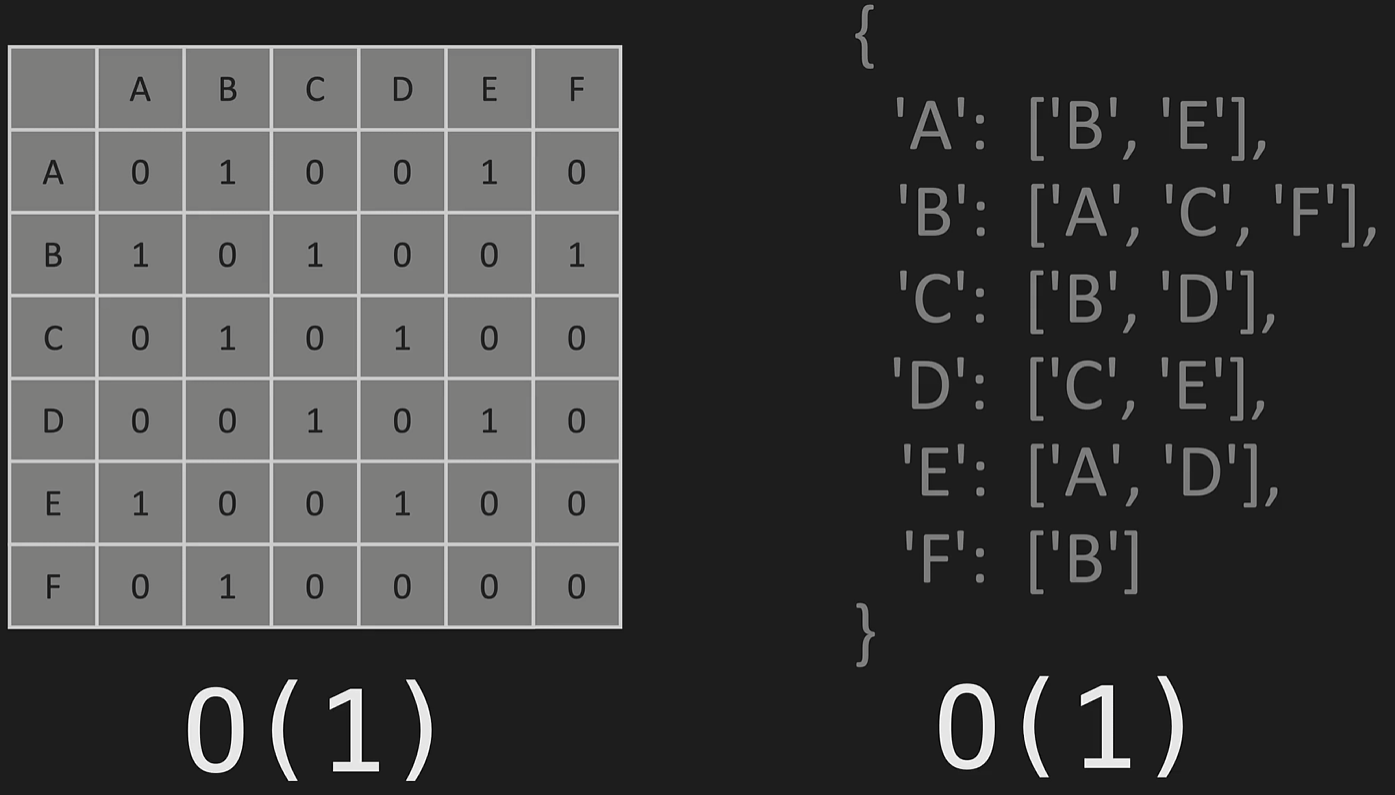


**□ Big O**

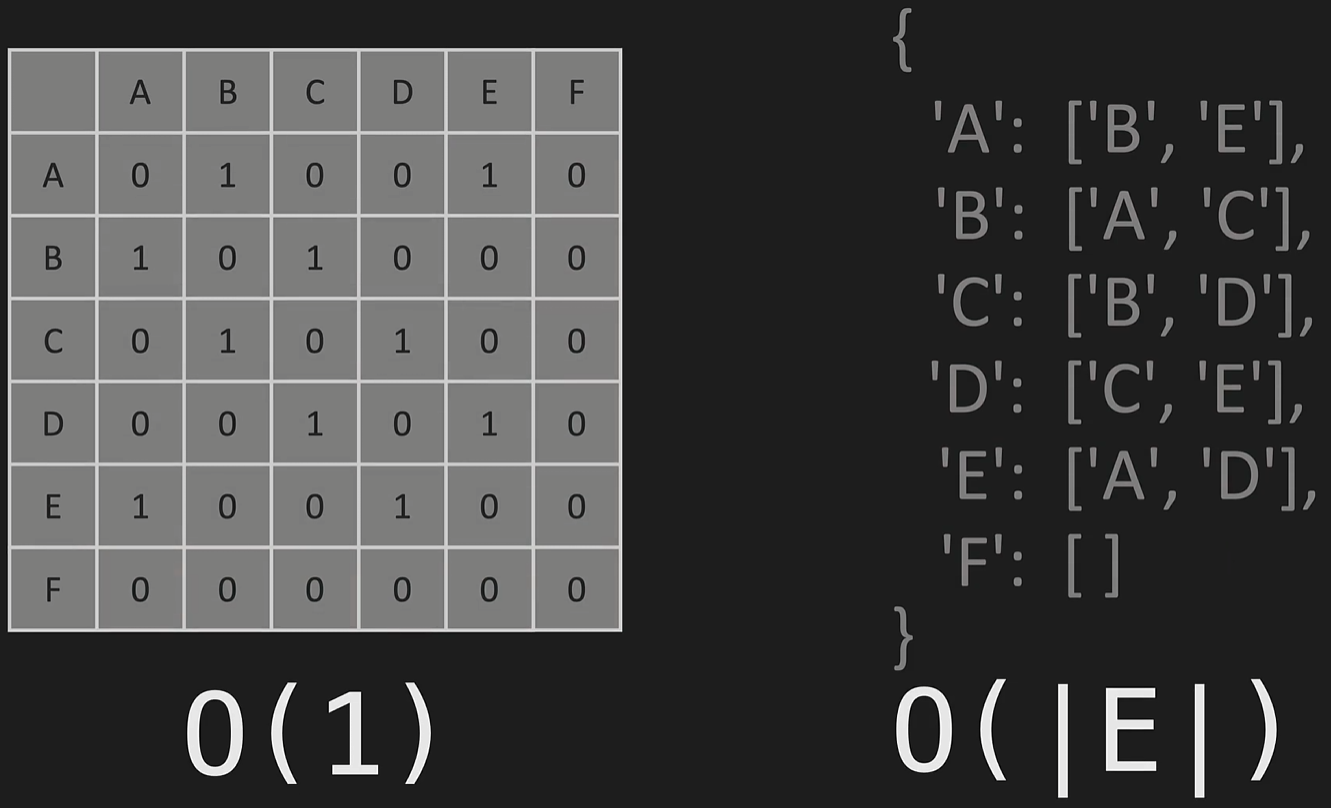
- Node 추가



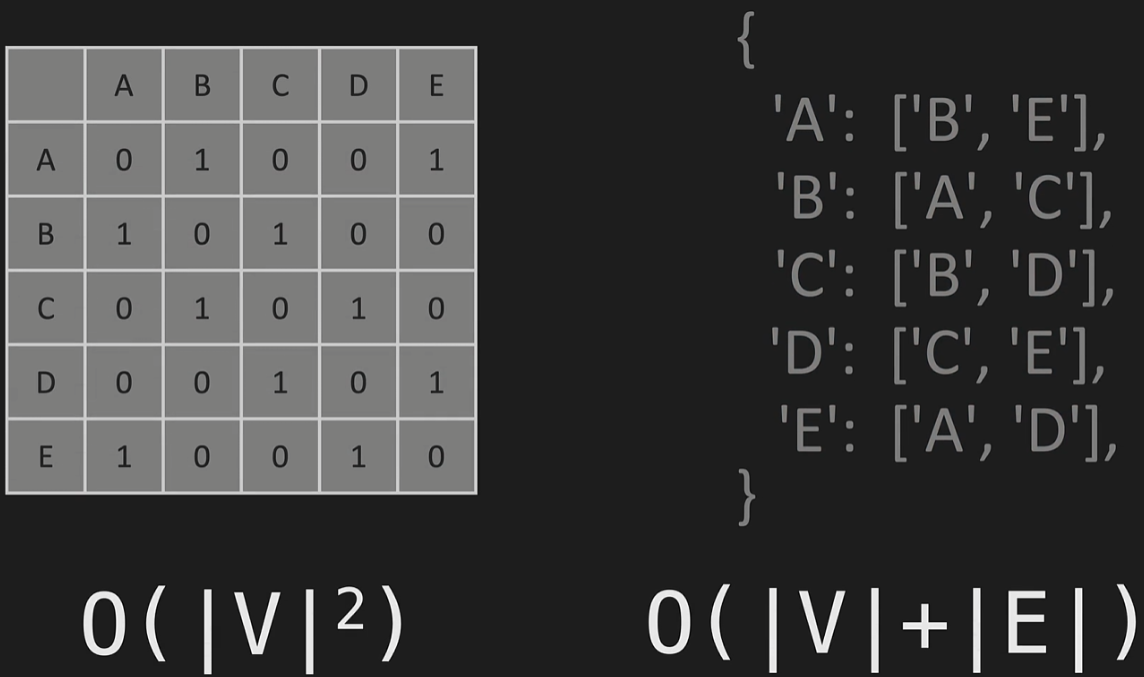
- Edge 추가



- Edge 삭제



- Node 삭제



**□ Code**

|  |
| --- |
| class Graph:  def \_\_init\_\_(self):  self.adj\_list = {}  def print\_graph(self):  v\_list = []  for vertex in self.adj\_list:  v\_list.append(vertex)  v\_list.sort()  for v in v\_list:  print(v, ':', self.adj\_list[v])  def add\_vertex(self, vertex):  if vertex not in self.adj\_list.keys():  self.adj\_list[vertex] = []  return True  return False  def add\_edge(self, v1, v2):  if v1 in self.adj\_list.keys() and v2 in self.adj\_list.keys():  self.adj\_list[v1].append(v2)  self.adj\_list[v2].append(v1)  return True  return False  def remove\_edge(self, v1, v2):  if v1 in self.adj\_list.keys() and v2 in self.adj\_list.keys():  try:  self.adj\_list[v1].remove(v2)  self.adj\_list[v2].remove(v1)  except ValueError:  pass  return True  return False  def remove\_vertex(self, v1):  if v1 in self.adj\_list:  for i in self.adj\_list[v1]:  self.adj\_list[i].remove(v1)  self.adj\_list.pop(v1)  return True  return False |

heapq vs PriorityQueue

**□ heapq**

- 완전 이진트리의 일종, 항상 Root Node를 제거

- 최소힙 : Root Node가 가장 작은값

부모 노드로 올라가면서 더 작은 경우 위치 교체(상향식)

- 최대힙 : Root Node가 가장 큰 값

- O(logn)

**□ heapq vs PriorityQueue**

- heapq는 모듈, PriorityQueue는 클래스(객체생성 필요)

- heapq가 더 빠름

**□ Code(heapq)**

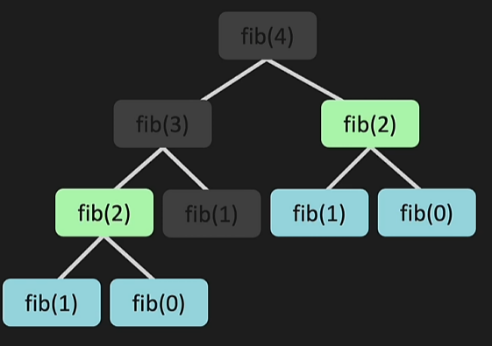
|  |
| --- |
| import heapq  def heapsort(arr):  q = []  result = []  for value in arr:  heapq.heappush(q, value)  for i in range(len(q)):  result.append(heapq.heappop(q))  return result  arr = [9,6,5,1,7]  heapsort(arr) |

|  |
| --- |
| heapq.nlargest(2, arr)  heapq.nsmallest(2, arr)  heapq.heapify(arr) # heap 구조로 변경(이진구조) |

**□ Code(PriorityQueue)**

|  |
| --- |
| from queue import PriorityQueue  def priority(arr):  q = PriorityQueue()  result = []  for value in arr:  q.put(value)  while q.empty() != True:  result.append(q.get())  return result  arr = [9,6,5,1,7]  priority(arr) |

Dynamic Programming

**□ 개요**

- Memorization 기법 사용

(한번 계산한 값을 메모리에 저장)

**□ 조건**

- 큰문제를 작은 문제로 나눌 수 있다

- 작은 문제에서 구한 정답은 그것을 포함하는 큰 문제에서 동일하다

**□ Example : 피보나치 수열**

- 피보나치 수열 :이전 2개 값의 합

****

- 재귀적인 방법

. 기하급수적으로 늘어나기 때문에 O(2^n) 형태

. 이런 방식은 비효율적이기 때문에 Memorization 사용

|  |
| --- |
| def fib(n):  if n == 0 or n == 1:  return n  return fib(n-1) + fib(n-1)  fibo(20) |

- Memorization 사용 (Top Down 방식)

. Subprogram의 결과값을 미리 저장함

. O(n)으로 계산 가능

|  |
| --- |
| memo = [0] \* 100  def fibo(x):  if x == 1 or x == 2:  return 1  if memo[x] != 0:  return memo[x]  memo[x] = fibo(x-1) + fibo(x-2)  return memo[x]  fibo(99) |

- 점화식 사용 (Bottom Up 방식)

|  |
| --- |
| fib\_list = [0, 1] # DP 테이블  def fibo(n):  for index in range(2, n+1):  next\_fib = fib\_list[index-1] + fib\_list[index-2]  fib\_list.append(next\_fib)  return fib\_list[n]  fibo(20) |

Dijkstra

**□ 개요**

- 특정 노드에서 시작해서 다른 노드로 가는 최단경로 계산

- 음의 간선이 없을 때 적용 가능

- 최단거리 노드를 구하기위해 우선순위 큐 사용

(PriorityQueue or Heapq)

- O(ElogV) \* E : 간선, V : 노드

**□ 알고리즘**

① 출발 노드 설정

② 최단거리 테이블 초기화

③ 방문하지 않은 노드 중 최단거리 노드 선택

④ 해당 노드를 거쳐 다른 노드로 가는 비용을 계산하여

최단거리 테이블 갱신

⑤ 3과4를 반복

**□ Code(간단한 알고리즘)**

|  |
| --- |
| # 입력받기 및 초기화  INF = int(1e9) # 10억(무한)  n,m = map(int, input().split())  start = int(input())  graph = [[] for i in range(n+1)]  visited = [False] \* (n+1)  distance = [INF] \* (n+1)  for \_ in range(m):  a,b,c = map(int, input().split())  graph[a].append((b,c))  # 방문하지 않은 노드 주 가장 최단거리가 짧은 노드 번호 반환  def get\_smallest\_node():  min\_value = INF  index = 0  for i in range(1, n+1):  if distance[i] < min\_value and not visited[i]:  min\_value = distance[i]  index = i  return index  # 다익스트라  def dijkstra(start):  distance[start] = 0  visited[start] =0  for i in graph[start]:  distance[i[0]] = i[1]  for i in range(n-1):  now = get\_smallest\_node()  visited[now] = True  for j in graph[now]:  cost = distance[now] + j[1]  if cost < distance[j[0]]:  distance[j[0]] = cost  dijkstra(start) |

Implementation

**□ 개요**

- 알고리즘을 소스코드로 바꾸는 과정

- 풀이를 떠올리는 것은 쉽지만 소스코드로 옮기기 어려운 문제