이코테(2021)

05 DFS & BFS

꼭 필요한 자료구조 기초

탐색이란 **많은 양의 데이터 중에서 원하는 데이터를 찾는 과정**을 의미한다.

대표적인 탐색 알고리즘으로 **DFS**와 **BFS**를 꼽을 수 있는데,

이 두 알고리즘의 원리를 제대로 이해해야 코딩 테스트의 탐색 문제 유형을 풀 수 있다.

자료구조란 **데이터를 표현하고 관리하고 처리하기 위한 구조**를 의미한다.

그 중, 스택과 큐는 자료구조의 기초 개념으로 다음의 두 핵심적인 함수로 구성된다.

- 삽입 : 데이터를 삽입한다.
- 삭제 : 데이터를 삭제한다.

또한, 실제로 사용할 땐, **오버플로**와 **언더플로**를 고민해야 한다.

- **오버플로** : **특정한 자료구조가 수용할 수 있는 데이터의 크기를 이미 가득 찬 상태에서 삽입연산을 수행할 때 발생**한다. 즉, **저장 공간을 벗어나 데이터가 넘쳐흐를 때 발생한다.**
- **언더플로** : **특정한 자료구조에 데이터가 전혀 들어 있지 않은 상태에서 삭제 연산을 수행하면 데이터가 전혀 없는 상태**이므로 언더플로가 발생한다.

스택

스택은 **선입후출(First In Last Out) 구조** 또는 **후입선출(Last In First Out) 구조**라고 한다.

예시로, 삽입(5) - 삽입(2) - 삽입(3) - 삽입(7) - 삭제() - 삽입(1) - 삽입(4) - 삭제()를 순서대로 표현 했을 때를 살펴보자.

- 5 - 2 - 3 - 7로 진행 후 7이 삭제
- 1과 4가 삽입
- 다시 4가 삭제
- 최종, 5 - 2- 3 - 1 이 남는다.

즉, **FILO or LIFO 구조**를 따르고 있다.

```
stack = []

stack.append(5)
stack.append(2)
stack.append(3)
stack.append(7)
stack.pop() # FILO 구조에 따라 7이 제거됨을 알 수 있다.
stack.append(1)
stack.append(4)
stack.pop()

print(stack)
print(stack[::-1])
```

`append()` 와 `pop()` 은 리스트의 **가장 뒤쪽에서 삽입과 삭제**를 수행한다.

큐

큐는 **선입선출(First In First Out) 구조**라고 한다.

예를 들어, 삽입(5) - 삽입(2) - 삽입(3) - 삽입(7) - 삭제() - 삽입(1) - 삽입(4) - 삭제()를 순서대로 표현 했을 때를 살펴보자.

- 7 - 3 - 2 -5 가 삽입
- 가장 먼저 들어온 5가 삭제
- 4 - 1 - 7 - 3 -2 삽입
- 가장 먼저 들어온 2가 삭제
- 최종 **4 - 1 - 7 - 3**

```
# 컬렉션 모듈에서 제공하는 deque 자료 구조 활용

from collections import deque

#큐 구현을 위해 deque 라이브러리 사용
queue = deque()

queue.append(5)
queue.append(2)
queue.append(3)
queue.append(7)
queue.popleft()
queue.append(1)
queue.append(4)
queue.popleft()
```

```
print(queue) # 먼저 들어온 원소 순서 대로 출력 -> 3714
queue.reverse()
print(queue) # 4173
```

`deque` 자료구조는 스택과 큐의 장점을 모두 채택한 것인데, **데이터를 넣고 빼는 속도가 리스트 자료형에 비해 효율적이며 queue 라이브러리를 이용하는 것보다 더 간단하다.**

재귀 함수

재귀 함수란 **자기 자신을 다시 호출하는 함수**

```
def recursive_function():
  print('재귀 함수를 호출합니다.')
  recursive_function()

recursive_function()
```

재귀 함수의 종료 조건

재귀 함수는 종료 조건을 꼭 명시해야 한다. 자칫 종료 조건을 명시하지 않으면 함수가 무한 호출될 수 있다.

```
def recursive_function(i):
  if i == 100:
    return
  print(i, '번째 함수에서', i+1, '번째 재귀 함수를 호출한다.')
  recursive_function(i+1)
  print(i,'번째 재귀 삼수를 종료합니다.')

recursive_function(1)
```

```
85 번째 함수에서 86 번째 재귀 함수를 호출한다.
86 번째 함수에서 87 번째 재귀 함수를 호출한다.
87 번째 함수에서 88 번째 재귀 함수를 호출한다.
88 번째 함수에서 89 번째 재귀 함수를 호출한다.
89 번째 함수에서 90 번째 재귀 함수를 호출한다.
90 번째 함수에서 91 번째 재귀 함수를 호출한다.
91 번째 함수에서 92 번째 재귀 함수를 호출한다.
92 번째 함수에서 93 번째 재귀 함수를 호출한다.
93 번째 함수에서 94 번째 재귀 함수를 호출한다.
94 번째 함수에서 95 번째 재귀 함수를 호출한다.
95 번째 함수에서 96 번째 재귀 함수를 호출한다.
96 번째 함수에서 97 번째 재귀 함수를 호출한다.
97 번째 함수에서 98 번째 재귀 함수를 호출한다.
98 번째 함수에서 99 번째 재귀 함수를 호출한다.
99 번째 함수에서 100 번째 재귀 함수를 호출한다.
99 번째 재귀 삼수를 종료합니다.
98 번째 재귀 삼수를 종료합니다.
97 번째 재귀 삼수를 종료합니다.
96 번째 재귀 삼수를 종료합니다.
95 번째 재귀 삼수를 종료합니다.
94 번째 재귀 삼수를 종료합니다.
93 번째 재귀 삼수를 종료합니다.
92 번째 재귀 삼수를 종료합니다.
91 번째 재귀 삼수를 종료합니다.
90 번째 재귀 삼수를 종료합니다.
89 번째 재귀 삼수를 종료합니다.
88 번째 재귀 삼수를 종료합니다.
87 번째 재귀 삼수를 종료합니다.
86 번째 재귀 삼수를 종료합니다.
85 번째 재귀 삼수를 종료합니다.
```

재귀 함수는 **스택 자료구조**를 이용한다. **함수를 계속 호출했을 때 가장 마지막에 호출한 함수가 먼저 수행을 끝내야 그 앞의 함수 호출이 종료되기 때문이다.**

재귀함수를 이용하는 대표적 예제는 **팩토리얼 문제**가 있다.

```
#반복적으로 구현한 n!
def factorial_iterative(n):
```

```
  result = 1
  # 1부터 n까지의 수를 차례대로 곱하기
  for i in range(1,n+1):
    result *= i
  return result

#재귀함수를 이용해 구현한 n!
def factorial_recursive(n):
  if n <= 1:
    return 1
  #n! = n * (n-1)!를 그대로 코드로 작성하면 된다.
  return n * factorial_recursive(n-1)

print(factorial_iterative(5))
print(factorial_recursive(5))
```

재귀 함수를 사용했을 때, **코드가 간결해짐을 알 수 있다.** 재귀 함수가 **수학의 점화식을 그대로 소스코드로 옮겼기 때문이다. 점화식은 특정한 함수를 자신보다 더 작은 변수에 대한 함수와의 관계로 표현한 것을 의미한다.**

팩토리얼을 수학적 점화식으로 표현해보자

1. n이 0 혹은 1일 때 : $factorial(n) = 1$
2. n이 1보다 클 때 : $factorial(n) = n * factorial(n-1)$

점화식의 종료 조건 : 'n이 0 혹은 1일 때' → 이를 고려하지 않으면 재귀함수가 무한히 반복된다.

따라서, 재귀 함수 내에서 특정 조건일 때 **더 이상 재귀적으로 함수를 호출하지 않고 종료하도록 if문을 이용하여 꼭 종료 조건을 구현해야 한다.**

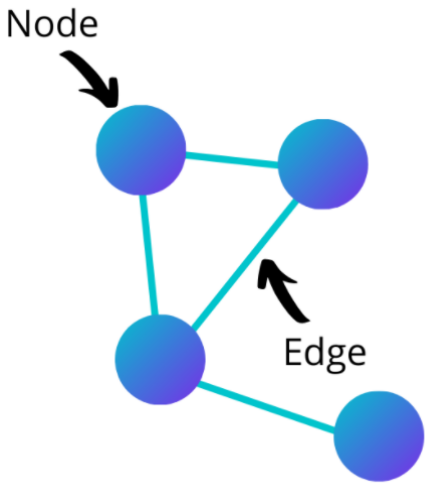
탐색 알고리즘 DFS & BFS

DFS

DFS는 Depth-First-Search, 깊이 우선 탐색이라고도 부르며, 그래프에서 깊은 부분을 우선적으로 탐색하는 알고리즘이다.

그래프의 기본 구조

그래프는 **노드** 와 **간선**으로 표현되며, 이 때 노드를 **정점**이라고 한다.



그래프 탐색이란 하나의 노드를 시작으로 다수의 노드를 방문하는 것을 말한다.

- **인접 행렬(Adjacency Matrix)** : 2차원 배열로 그래프의 연결 관계를 표현하는 방식

  → 모든 관계를 저장하므로 메모리가 낭비됨

- **인접 리스트(Adjacency List)** : 리스트로 그래프의 연결 관계를 표현하는 방식

  → 메모리를 효율적으로 사용하지만, 특정 두 노드가 연결되어 있는지에 대한 정보를 얻는 속도가 느리다.

DFS의 동작 과정

1. 탐색 시작 노드를 스택에 삽입하고 방문 처리를 한다.
2. 스택의 최상단 노드에 방문하지 않은 인접 노드가 있으면 그 인접 노드를 스택에 넣고 방문 처리를 한다. 방문하지 않은 인접 노드가 없으면, 스택에서 최상단 노드를 꺼낸다.
3. 2번의 과정을 더 이상 수행할 수 없을 때까지 반복한다.

BFS

BFS는 **Breadth-First-Search**, **너비 우선 탐색**이라고 부르며, **가까운 노드부터 탐색하는 알고리즘**이다.

BFS의 동작 과정

1. 탐색 시작 노드를 큐에 삽입하고 방문 처리를 한다.
2. 큐에서 노드를 꺼내 해당 노드의 인접 노드 중에서 방문하지 않은 노드를 모두 큐에 삽입하고 방문 처리를 한다.
3. 2번의 과정을 더 이상 수행할 수 없을 때까지 반복한다.

06 정렬

기준에 따라 데이터를 정렬

정렬 알고리즘 개요

정렬이란 **데이터를 특정한 기준에 따라서 순서대로 나열**하는 것

선택 정렬

선택정렬은 **매번 가장 작은 데이터를 선택해 맨 앞에 있는 데이터와 바꾸고, 그 다음 작은 데이터를 선택해 앞에서 두 번째 데이터와 바꾸는 과정을 반복하는 것이다.**

```
array = [7,5,9,0,3,1,6,2,4,8]

for i in range(len(array)):
  min_index = i
  for j in range(i+1,len(array)):
    if array[min_index] > array[j]:
      min_index = j
  array[i], array[min_index] = array[min_index], array[i]

print(array)
```

`Swap`은 파이썬에선 간단하게 리스트 내 두 원소 위치를 변경할 수 있다.

```
array =[3,5]

array[0],array[1] = array[1], array[0]
```

그러나, C++에선 별도 저장공간이 필요하다.

```
int a = 3;
int b = 5;

int temp = a
a = b;
b = temp;
```

선택 정렬의 시간 복잡도

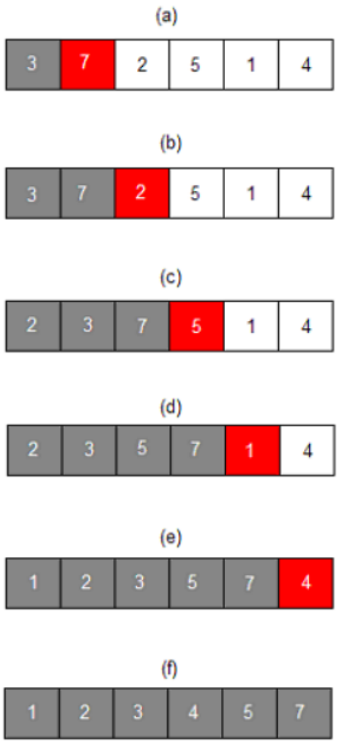
연산 횟수는 N + (N-1) + (N-2) + ... + 2로 볼 수 있으며 N*(N+1)/2번의 연산을 수행한다. 그러므로 빅오표기법으로 간단히 O(N^2)으로 표현할 수 있다.

삽입 정렬

삽입 정렬은 **특정한 데이터를 적절한 위치에 '삽입'한다는 의미**이다.

특정한 데이터가 적절한 위치에 들어가기 이전에, 그 앞까지의 데이터는 이미 정렬되어 있다고 가정한다.

→ 삽입될 데이터보다 작은 데이터를 만나면 그 위치에 멈춘다. **즉, 더 이상 데이터를 살펴볼 필요 없이 그 자리에 삽입되면 되는 것이다.**



삽입 정렬의 시간 복잡도

삽입 정렬의 시간 복잡도는 O(N^2)인데, 정렬이 거의 되어 있는 상황에서는 최선의 경우 O(N)의 시간 복잡도를 가진다.

퀵 정렬

지금까지 배운 정렬 알고리즘 중에 가장 많이 사용되는 알고리즘이다.

퀵 정렬은 기준을 설정한 다음 큰 수와 작은 수를 교환한 후 리스트를 반으로 나누는 방식으로 동작한다.

퀵 정렬에서는 **피벗**이 사용된다. 큰 숫자와 작은 숫자를 교환할 때, 교환하기 위한 **'기준'**을 바로 피벗이라고 표현한다.

호어 분할 방식

- 리스트에서 첫 번째 데이터를 피벗으로 설정한다.

퀵 정렬의 소스 코드

```
import sys
sys.setrecursionlimit(10 ** 4)

array = [7,5,9,0,3,1,6,2,4,8]

def quick_sort(array,start,end):
  if start >= end:
```

```
    return
  pivot = start
  left = start + 1
  right = end
  while left <= right:
    while left <= end and array[left] <= array[pivot]:
      left += 1
    while right > start and array[right] >= array[pivot]:
      right -= 1
    if left > right:
      array[right], array[pivot] = array[pivot], array[right]
    else:
      array[left], array[right] = array[right], array[left]
  quick_sort(array,start,right-1)
  quick_sort(array,right+1,end)

quick_sort(array,0,len(array)-1)
print(array)
```

파이썬의 장점을 살린 퀵 정렬의 소스코드

```
array = [7,5,9,0,3,1,6,2,4,8]

def quick_sort(array):
  if len(array) <=1:
    return array

  pivot = array[0]
  tail = array[1:]

  left_side = [x for x in tail if x <= pivot]
  right_side = [x for x in tail if x > pivot]

  return quick_sort(left_side) + [pivot] + quick_sort(right_side)

print(quick_sort(array))
```

퀵 정렬의 시간 복잡도

퀵 정렬의 시간 복잡도 : $O(NlogN)$

즉, 데이터의 개수가 많을수록 선택 정렬, 삽입 정렬에 비해 압도적으로 빠르게 동작하리라 추측할 수 있다.

계수 정렬

계수 정렬은 **특정한 조건이 부합할 때만 사용할 수 있지만 매우 빠른 정렬 알고리즘**이다.

데이터의 개수가 N, 데이터 중 최댓값이 K일 때, 계수 정렬은 최악의 경우에도 수행 시간 **O(N+K)**를 보장한다. 다만, '**데이터의 크기 범위가 제한되어 정수 형태로 표현할 수 있을**

때'만 사용할 수 있다.

계수 정렬을 이용할 땐 '**모든 범위를 담을 수 있는 크기의 리스트를 선언**'해야 한다.

계수 정렬의 시간 복잡도

시간 복잡도 : O(N + K)

계수 정렬의 공간 복잡도

데이터의 크기가 한정되어 있고, 데이터의 크기가 많이 중복되어 있을수록 유리하며 항상 사용할 수는 없다.

계수 정렬의 공간복잡도는 : O(N + K)이다.

파이썬의 정렬 라이브러리

파이썬은 기본 정렬 라이브러리인 `sorted()` 를 제공받는다. `sorted()` 는 퀵 정렬과 동작 방식이 비슷한 병합 정렬을 기반으로 만들어졌으며, 병합 정렬은 일반적으로 퀵 정렬보다 느리지만 최악의 경우에도 **시간 복잡도 $O(NlogN)$을 보장한다는 특징**이 있다.

리스트가 1개 있을 때도, 내부 원소를 바로 정렬할 수도 있다. 리스트 객체의 내장 함수인 `sort()` 를 이용하는 것인데, 이를 이용하면, **별도의 정렬된 리스트가 반환되지 않고, 내부 원소가 바로 정렬된다.**

정렬 라이브러리의 시간 복잡도

1. **정렬 라이브러리로 풀 수 있는 문제**

   단순히 정렬 기법을 알고 있는지에 대한 문제로, 기본 정렬 라이브러리 방법을 숙지하고 있으면 풀 수 있다.

2. **정렬 알고리즘의 원리에 대해서 물어보는 문제**

   선택 정렬, 삽입 정렬, 퀵 정렬 등의 원리를 알고 있어야 문제를 풀 수 있다.

3. **더 빠른 정렬이 필요한 문제**

   퀵 정렬은 풀 수 없고, 계수 정렬 등의 다른 정렬 알고리즘을 이용하거나 문제에서 기존에 알려진 알고리즘의 구조적인 개선을 거쳐야 한다.

07 이진 탐색

범위를 반씩 좁혀나가는 탐색

순차 탐색

리스트 안에 있는 특정한 데이터를 찾기 위해 앞에서부터 데이터를 하나씩 차례대로 확인하는 방법

→ 정렬되지 않은 리스트에서 데이터를 찾아야할 때 사용한다.

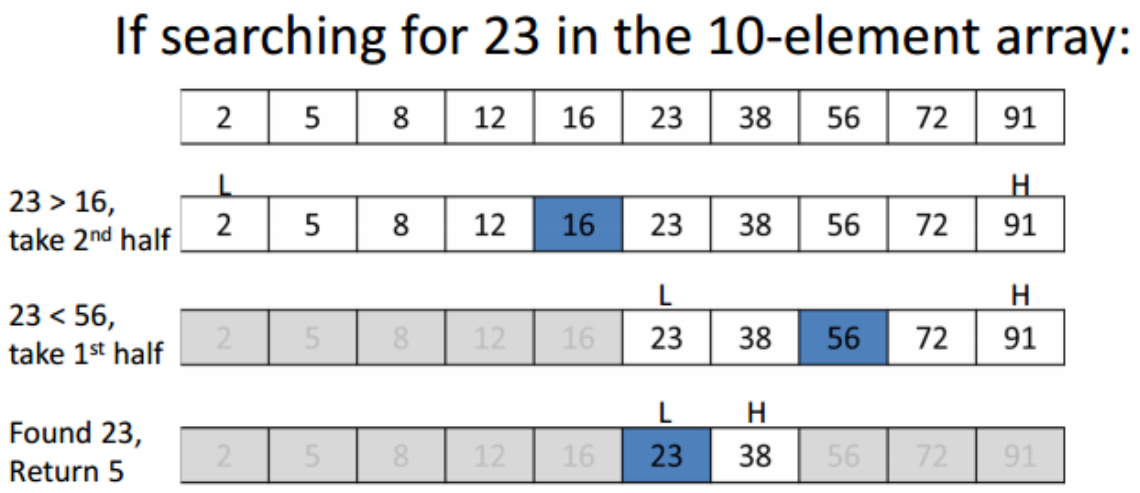
1. 리스트에 특정 값의 원소가 있는지 체크할 때 사용한다.
2. 리스트 자료형에서 특정한 값을 가지는 원소의 개수를 세는 `count()` 메서드를 이용할 때 사용한다.

시간 복잡도가 O(N)인 순차 탐색 소스코드

```
def sequential_search(n,target,array):
  for i in range(n):
    if array[i] == target:
      return i + 1
print("생성할 원소 개수를 입력한 다음 한 칸 띄고 찾을 문자열을 입력하세요.")
input_data = input().split()
n = int(input_data[0])
target = input_data[1]

print("앞서 적은 원소 개수만큼 문자열을 입력하세요. 구분은 띄어쓰기 한 칸으로 합니다.")
array = input().split()

#순차 탐색 수행 결과 출력
print(sequential_search(n,target,array))
```

이진 탐색 : 반으로 쪼개면서 탐색하기

찾으려는 데이터와 중간점 위치에 있는 데이터를 반복적으로 비교해서 원하는 데이터를 찾는 것



전체 데이터 개수는 10개지만, 이진 탐색을 사용해 총 3번의 탐색으로 원소를 찾을 수 있다.

한 번 확인할 때마다 확인하는 원소의 개수가 절반씩 줄어든다는 점에서 **시간복잡도가 O(logN)**이다.

재귀 함수로 구현한 이진 탐색 소스코드

```
def binary_search(array,target,start,end):
  if start > end:
    return None
  mid = (start + end) //2
  if array[mid] ==target:
    return mid
  elif array[mid] > target:
    return binary_search(array,target,start,mid-1)
  else:
    return binary_search(array,target,mid+1,end)

n, target = map(int,input().split())

array = list(map(int,input().split()))


result = binary_search(array,target,0,n-1)
if result == None:
  print('원소가 존재하지 않습니다')
else:
  print(result+1)
```

08 다이나믹 프로그래밍

- 다이나믹 프로그래밍은 **메모리를 적절히 사용하여 수행 시간 효율성을 비약적으로 향상시키는 방법**이다.
- 이미 계산된 결과는 별도의 메모리 영역에 저장하여 다시 계산하지 않도록 한다.
- 다이나믹 프로그래밍의 구현은 일반적으로 두 가지 방식(탑다운과 보텀업)으로 구성된다.
- 다이나믹 프로그래밍은 **동적 계획법**이라고 부른다.
- 일반적인 프로그래밍 분야에서의 동적이란 어떤 의미를 가질까?
    - 자료구조에서 동적 할당은 '**프로그램이 실행되는 도중에 실행에 필요한 메모리를 할당하는 기법**'을 의미한다.
    - 반면에 다이나믹 프로그래밍에서 '다이나믹'은 별다른 의미 없이 사용된 단어이다.

다이나믹 프로그래밍의 조건

- 다이나믹 프로그래밍은 문제가 다음의 조건을 만족할 때 사용할 수 있다.

1. **최적 부분 구조**
    - 큰 문제를 작은 문제로 나눌 수 있으며 작은 문제의 답을 모아서 큰 문제를 해결할 수 있다.
2. **중복되는 부분 문제**
    - 동일한 작은 문제를 반복적으로 해결해야 한다.

피보나치 수열

- 피보나치 수열 다음과 같은 형태의 수열이며, 다이나믹 프로그래밍으로 효과적으로 계산할 수 있다.

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...$$

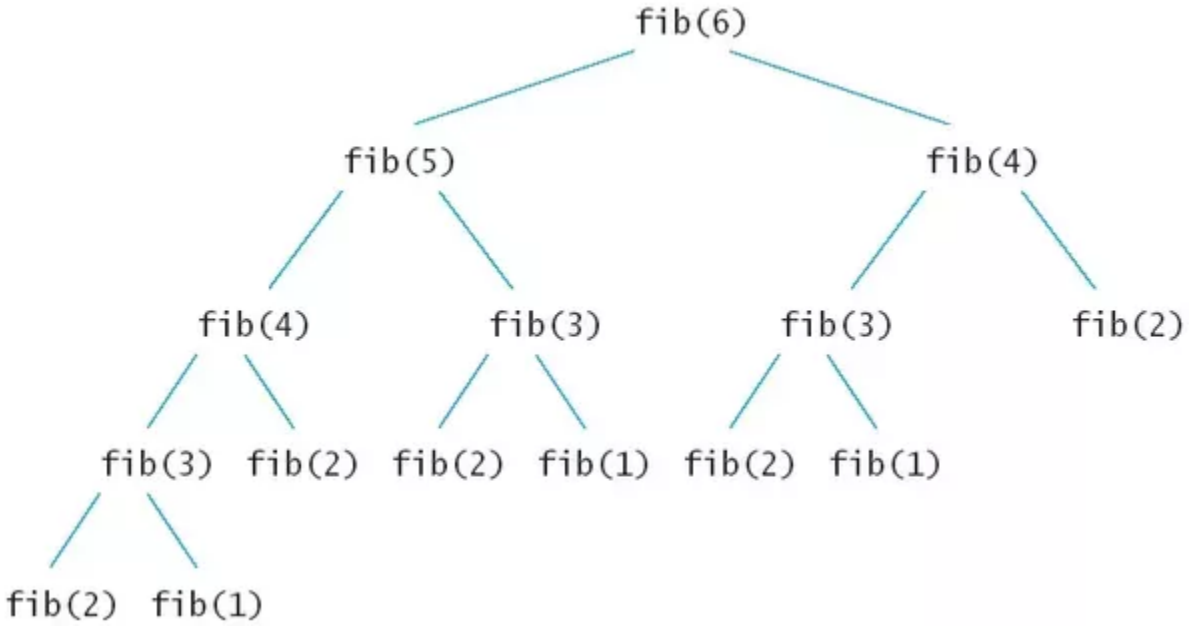
- **점화식**이란 인접한 항들 사이의 관계식을 의미한다.
- 피보나치 수열을 점화식으로 표현하면 다음과 같다.

$$a_n = a_{n-1} + a_{n-2}, a_1 = 1, a_2 = 1$$

- 피보나치 수열이 계산되는 과정은 다음과 같이 표현할 수 있다.
    - 프로그래밍에서는 이러한 수열을 배열이나 리스트를 이용해 표현한다.

피보나치 수열의 시간 복잡도 분석

- 단순 재귀 함수로 피보나치 수열을 해결하면 지수 시간 복잡도를 가지게 된다.
- 다음과 같이 f(2)가 여러번 호출되는 것을 확인할 수 있다. (**중복되는 부분 문제**)



피보나치 수열의 시간 복잡도 분석

- 빅오 표기법을 기준으로 f(30)을 계산하기 위해 약 10억가량의 연산을 수행해야 한다
- f(100)을 계산하면 비현실적으로 많은 연산을 수행해야한다.

피보나치 수열의 효율적인 해법 : 다이나믹 프로그래밍

- 다이나믹 프로그래밍의 사용 조건을 만족하는지 확인한다.
  1. 최적 부분 구조 : 큰 문제를 작은 문제로 나눌 수 있다.
  2. 중복되는 부분 문제 : 동일한 작은 문제를 반복적으로 해결한다.
- 피보나치 수열은 다이나믹 프로그래밍의 사용 조건을 만족한다.

메모이제이션

- 메모이제이션은 다이나믹 프로그래밍을 구현하는 방법 중 하나이다.
- 한 번 계산한 결과를 메모리 공간에 메모하는 기법이다.
  - 같은 문제를 다시 호출하면 메모했던 결과를 그대로 가져온다.
  - 값을 기록해 놓는다는 점에서 **캐싱**이라고도 한다.

탑다운 VS 보텀업

- 탑다운(메모이제이션) 방식은 **하향식**이라고도 하며 보텀업 방식은 **상향식**이라고 한다.

  탑다운 방식

```
#한 번 계산된 결과를 메모이제이션하기 위한 리스트 초기화
d = [0] * 100

#피보나치 함수를 재귀함수로 구현(탑다운 다이나믹 프로그래밍)
def fibo(x):
  # 종료 조건(1혹은 2일 때 1을 반환)
  if x == 1 or x == 2:
    return 1
  # 이미 계산한 적 있는 문제라면 그대로 반환한다.
  if d[x] != 0:
    return d[x]
  # 아직 계산하지 않은 문제라면 점화식에 따라서 피보나치 결과 반환
  d[x] = fibo(x-1) + fibo(x-2)
  return d[x]

print(fibo(99))
```

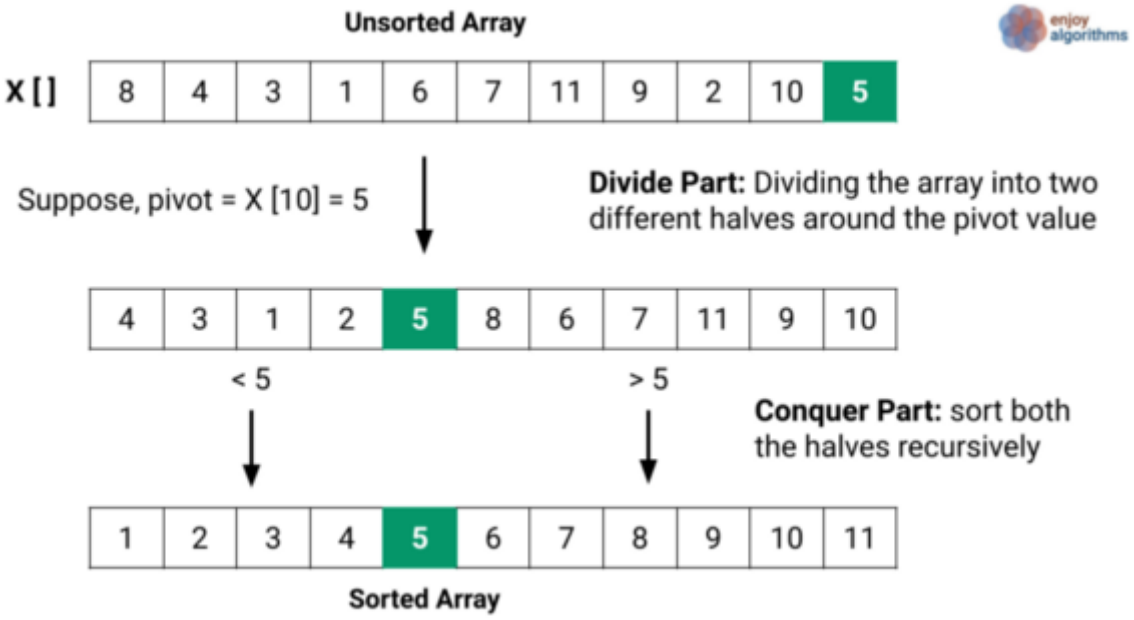
  보텀업 방식

```
# 앞서 계산된 결과를 저장하기 위한 DP 테이블 초기화
d = [0] * 100

# 첫 번째 피보나치 수와 두 번째 피보나치 수는 1
d[1] = 1
d[2] = 1
n = 99

#피보나치 함수 반복문으로 구현(보텀업 다이나믹 프로그래밍)
for i in range(3,n+1):
  d[i] = d[i-1] + d[i-2]

print(d[n])
```

- 다이나믹 프로그래밍의 전형적인 형태는 **보텀업 방식**이다.
  - 결과 저장용 리스트는 **DP 테이블**이라고 부른다.
- 엄밀히 말하면 메모이제이션은 이전에 계산된 결과를 일시적으로 기록해 놓는 넓은 개념을 의미한다.
  - 따라서 메모이제이션은 다이나믹 프로그래밍에 국한된 개념은 아니다.
  - 한 번 계산된 결과를 담아 놓기만 하고 다이나믹 프로그래밍을 위해 활용하지 않을 수도 있다.

다이나믹 프로그래밍 VS 분할 정복

- 다이나믹 프로그래밍과 분할 정복은 모두 **최적 부분 구조**를 가질 때 사용할 수 있다.
  - 큰 문제를 작은 문제로 나눌 수 있으며 작은 문제의 답을 모아서 큰 문제를 해결할 수 있는 상황
- 다이나믹 프로그래밍과 분할 정복의 차이점은 **부분 문제의 중복**이다.
  - 다이나믹 프로그래밍 문제에서는 각 부분 문제들이 서로 영향을 미치며 부분 문제가 중복된다.
  - 분할 정복 문제에서는 동일한 부분 문제가 반복적으로 계산되지 않는다.
- **분할 정복**의 대표적인 예시인 퀵 정렬을 살펴보자.
  - 한 번 기준 원소가 자리를 변경해서 자리를 잡으면 그 기준의 원소의 위치는 바뀌지 않는다.
  - 분할 이후에 해당 피벗을 다시 처리하는 부분 문제는 호출하지 않는다.



다이나믹 프로그래밍 문제에 접근하는 방법

- 주어진 문제가 다이나믹 프로그래밍 유형임을 파악하는 것이 중요하다.
- 가장 먼저 그리디, 구현, 완전 탐색 등의 아이디어로 문제를 해결할 수 있는지 검토할 수 있다.
    - 다른 알고리즘으로 풀이 방법이 떠오르지 않으면 다이나믹 프로그래밍을 고려해보자.
- 일단 재귀 함수로 비효율적인 완전 탐색 프로그램을 작성한 뒤에 (탑다운) 작은 문제에서 구한 답이 큰 문제에서 그래도 사용될 수 있으면, 코드를 개선하는 방법을 사용할 수 있다.
- 일반적인 코딩 테스트 수준에서는 기본 유형의 다이나믹 프로그래밍 문제가 출제되는 경우가 많다.

09 최단 경로

01 가장 빠른 길 찾기

가장 빠르게 도달하는 방법

최단 경로 문제는 가장 짧은 경로를 찾는 알고리즘이며, 보통 그래프를 이용해 표현한다.

각 지점은 그래프에서 '노드'로 표현되고, 지점간 연결된 도로는 그래프에서 '간선'으로 표현된다.

최단 거리 알고리즘의 종류

- 다익스트라 최단 경로 알고리즘
- 플로이드 워셜 알고리즘
- 벨만 포드 알고리즘

다익스트라 최단 경로 알고리즘

다익스트라 최단 경로 알고리즘은 그래프에서 여러 개의 노드가 있을 때, 특정한 노드에서 출발해서 다른 노드로 가는 각각의 최단 경로를 구해주는 알고리즘이다.

매번 '**가장 비용이 적은 노드**'를 선택해서 임의의 과정을 반복하기 때문에 기본적으로 그리디 알고리즘으로 분류된다.

1. 출발 노드를 설정한다.
2. 최단 거리 테이블을 초기화한다.
3. 방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드를 선택한다.
4. 해당 노드를 거쳐 다른 노드로 가는 비용을 계산하여 최단 거리 테이블을 갱신한다.
5. 위 과정에서 3, 4번을 반복한다.

다익스트라 알고리즘을 구현하는 방법

1. 구현하기 쉽지만 느리게 동작하는 코드
2. 구현하기에 조금 더 까다롭지만 빠르게 동작하는 코드

→ **한 단계당 하나의 노드에 대한 최단 거리**를 확실히 찾을 수 있다.

따라서 마지막 노드에 대해서는 해당 노드를 거쳐 다른 노드로 가는 경우를 확인할 필요가 없다.

다익스트라 알고리즘의 특징

- **그리디 알고리즘** : **매 상황에서 방문하지 않은 가장 비용이 적은 노드**를 선택해 임의의 과정을 반복한다.
- 단계를 거치며 **한 번 처리된 노드의 최단 거리**는 고정되어 더 이상 바뀌지 않는다.
    - **한 단계당 하나의 노드에 대한 최단 거리를 확실히 찾는 것**으로 이해할 수 있다.

- 다익스트라 알고리즘을 수행한 뒤에 테이블에 각 노드까지의 최단 거리 정보가 저장된다.
    - 완벽한 형태의 최단 경로를 구하려면 소스코드에 추가적인 기능을 더 넣어야 한다.

방법 1. 간단한 다익스트라 알고리즘

다익스트라 알고리즘은 $O(N^2)$의 시간 복잡도를 가지며, 다익스트라에 의해서 처음 고안되었던 알고리즘이다.

```
import sys
input = sys.stdin.readline
INF = int(1e9)

# 노드의 개수, 간선의 개수를 입력받기
n, m = map(int, input().split())

# 시작 노드 번호를 입력받기

start = int(input())

# 각 노드에 연결되어 있는 노드에 대한 정보를 담는 리스트를 만들기
graph = [[] for i in range(n+1)]

# 방문한 적이 있는지 체크하는 목적의 리스트를 만들기
visited = [False] * (n+1)

# 최단 거리 Table을 모두 무한으로 초기화
distance = [INF] * (n+1)

# 모든 간선 정보를 입력받기
for _ in range(m):
  a,b,c = map(int,input().split())
  # a -> b 로 가는 비용이 c다.
  graph[a].append((b,c))

# 방문하지 않은 노드 중, 가장 최단 거리가 짧은 노드의 번호를 반환한다.

def get_smallest_node():
  min_value = INF
  # 가장 최단거리가 짧은 노드(인덱스)
  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

def dijkstra(start):
  # 시작 노드에 대해 초기화
  distance[start] = 0
  visited[start] = True
  for j in graph[start]:
    distance[j[0]] = j[1]
```

```
  # 시작 노드를 제외한 전체 n-1개의 노드에 대해 반복
  for i in range(n-1):
    # 현재 최단 거리가 가장 짧은 노드를 꺼내서, 방문 처리
    now = get_smallest_node()
    visited[now] = True
    # 현재 노드와 연결된 다른 노드를 확인한다.
    for j in graph[now]:
      cost = distance[now] + j[1]
      # 현재 노드를 거쳐서 다른 노드로 이동하는 거리가 더 짧은 경우
      if cost < distance[j[0]]:
        distance[j[0]] = cost

dijkstra(start)

# 모든 노드로 가기 위한 최단 거리를 출력한다.
for i in range(1,n+1):
  # 도달할 수 없는 경우, INF를 출력한다.
  if distance[i] == INF:
    print("INFINITY")
  # 도달할 수 있는 경우 거리를 출력한다.
  else:
    print(distance[i])
```

다익스트라 알고리즘 : 간단한 구현 방법 성능 분석

- 총 $O(V)$번에 걸쳐서 최단 거리가 가장 짧은 노드를 매번 선형 탐색해야 한다.
- 따라서 전체 시간 복잡도는 $O(V^2)$이다.
- 일반적으로 코딩 테스트의 최단 경로 문제에서 전체 노드의 개수가 5000개 이하라면 이 코드로 문제를 해결할 수 있다.
    - 하지만 노드의 개수가 10000개를 넘어가는 문제라면 어떻게 해야할까?

우선순위 큐(Priority Queue)

- 우선순위가 가장 높은 데이터를 가장 먼저 삭제하는 자료구조이다.
- 예를 들어 여러 개의 물건 데이터를 자료구조에 넣었다가 가치가 높은 물건 데이터부터 꺼내서 확인해야 하는 경우에 우선순위 큐를 이용할 수 있다.
- Pyhton, C++, Java를 포함한 대부분의 프로그래밍 언어에서 **표준 라이브러리 형태**로 지원한다.

자료구조	추출되는 데이터
스택(Stack)	가장 나중에 삽입된 데이터
큐(Queue)	가장 먼저 삽입된 데이터
우선순위 큐(Priority Queue)	가장 우선순위가 높은 데이터

- 우선순위 큐를 구현하기 위해 사용하는 자료구조 중 하나이다.
- 최소 힙과 최대 힙이 있다.
- 다익스트라 최단 경로 알고리즘을 포함해 다양한 알고리즘에서 사용된다.

다익스트라 알고리즘 : 개선된 구현 방법

- 단계마다 방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드를 선택하기 위해 힙 자료구조를 이용한다.
- 다익스트라 알고리즘이 동작하는 기본 원리는 동일하다.
  - 현재 가장 가까운 노드를 저장해 놓기 위해서 힙 자료구조를 추가적으로 이용한다는 점이 다르다.
  - 현재의 최단 거리가 가장 짧은 노드를 선택해야 하므로 최소 힙을 사용한다.
- 힙 자료구조를 이용하는 다익스트라 알고리즘의 시간 복잡도는 $O(ElogV)$이다.
- 노드를 하나씩 꺼내 검사하는 반복문은 노드의 개수 V 이상의 횟수로는 처리되지 않는다.
  - 결과적으로 현재 우선순위 큐에서 꺼낸 노드와 연결된 다른 노드들을 확인하는 총 횟수는 최대 간선의 개수만큼 연산이 수행될 수 있다.
- 직관적으로 전체 과정은 E개의 원소를 우선순위 큐에 넣었다가 모두 빼내는 연산과 매우 유사하다.
  - 시간 복잡도를 $O(ElogE)$로 판단할 수 있다.
  - 중복 간선을 포함하지 않는 경우에 이를 $O(ElogV)$로 정리할 수 있다.
    - $O(ElogE)$ → $O(ElogV^2)$ → $O(2ElogV)$ -> $O(ElogV)$