Stack

1.물건을 쌓아 올리듯 자료를 쌓아 올린 형태의 자료구조
2.마지막에 삽입한 자료를 먼저 꺼내는 구조.(후입선출, LIFO)
3.스택에 저장된 자료는 선형구조를 가짐.
  (데이터간 1:1관계. 데이터-데이터-데이터
   포인터 등을 사용하여 자료를 연결하면 그 결과가 자료에 일직선상에 표시되거나 하나의 원상에 표시되는 구조.
   이중연결리스트는 1:1관계가 아니잖아욧?!이라고 할수 있잖아. 이런 정의로는.)

자료구조

1.구현시 선형으로 저장할 저장소 필요.
-C언어 : arr
-Python : list
2.저장소 자체를 스택이라 부르 수 있음
3.스택에서 마지막 삽입된 원소의 위치를 top이라고 부름

연산

1.삽입(push): 저장소에 자료를 저장
2.삭제(pop):저장소에서 자료를 꺼냄. 꺼낸 자료는 삽입한 자료의 역순으로 꺼낸다.(LIFO)
3.isEmpty : 스택이 공백인지 아닌지를 확인하는 연산
4.peek:스택의 top에 있는 item(원소)을 반환하는 연산

구현하기

practice.py | practice2.py | practic3.py

```python
class Stack:
    def __init__(self):
        self.stack = []

    def push(self,data):
        self.stack.append(data)

    def pop(self):
        if self.stack:
            return self.stack.pop()
        else:
            print("stack is empty")
            return

    def isEmpty(self):
        if self.stack:
            return True
        else:
            return False

    def peek(self):
        if self.stack:
            return stack[-1]
        else:
            return False

stack = Stack()

stack.push(1)
stack.push(2)
stack.push(3)

for _ in range(3):
    print("pop item =>", stack.pop())
```

스택 구현 방법에 따른 장단점

리스트

장점 : 구현 용이
단점 : 리스트 크기를 변경하는 작업(append)은 내부적으로 큰
overhead 발생을 시키는 작업으로, 많은 시간 소요
=> 배열 및 동적 연결리스트로 스택구현

동적 연결리스트

장점 : 리스트 단점 극복
단점 : 리스트보다는 구현 복잡

Stack의 응용

1.괄호검사
2.함수 호출 관리(-재귀함수)
3.memorization
4.DP
5.DFS

1.괄호검사

2 Stack의 응용 1 괄호검사

괄호의 종류

- 대괄호 (‘[’, ‘]’), 중괄호 (‘{’, ‘}’), 소괄호 (‘(’, ‘)’)

조건

1. 왼쪽 괄호의 개수와 오른쪽 괄호의 개수가 같아야 함
2. 같은 괄호에서 왼쪽 괄호는 오른쪽 괄호보다 먼저 나와야 함
3. 괄호 사이에는 포함 관계만 존재함

잘못된 괄호 사용의 예

(a(b)	a(b)c)	a{b(c[d]e}f)

2.재귀함수 호출

2 Stack의 응용 2 Function call

함수 호출 관리

프로그램에서의 함수 호출과 복귀에 따른 수행 순서를 관리

가장 마지막에 호출된 함수가 가장 먼저 실행을 완료하고 복귀하는
후입선출 구조이므로, 후입선출 구조의 스택을 이용하여 수행순서 관리

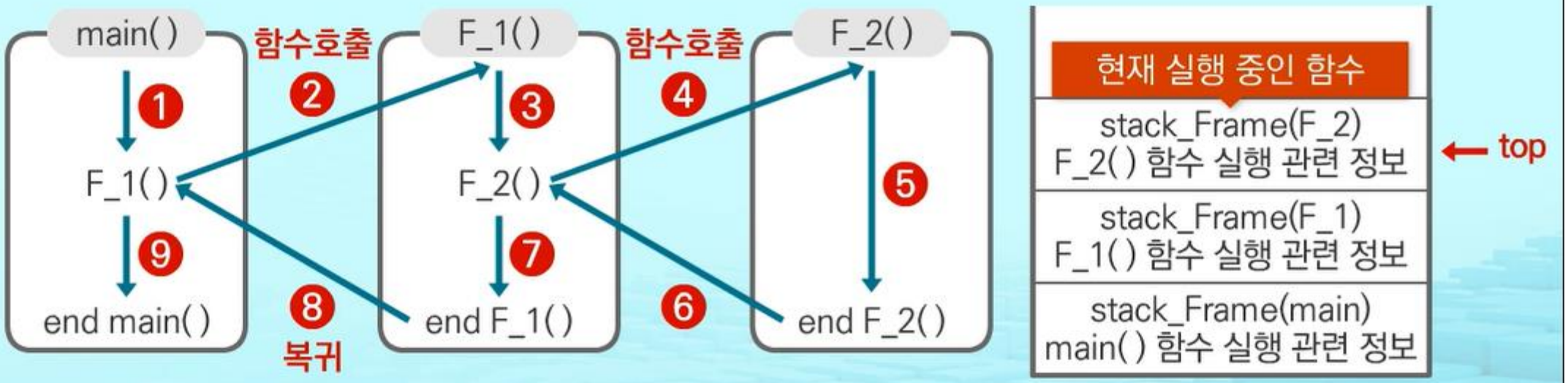
함수 호출이 발생하면 호출한 함수 수행에 필요한 지역변수, 매개변수 및
수행 후 복귀할 주소 등의 정보를 스택 프레임에 저장하여 시스템 스택에 삽입

함수의 실행이 끝나면 시스템 스택의 top 원소(스택 프레임)를
삭제(pop)하면서 프레임에 저장되어있던 복귀주소를 확인하고 복귀

함수 호출과 복귀에 따라 이 과정을 반복하여
전체 프로그램 수행이 종료되면 시스템 스택은 공백 스택이 됨

2 Stack의 응용 2 Function call

함수 호출 수행 순서



2 Stack의 응용 2 Function call

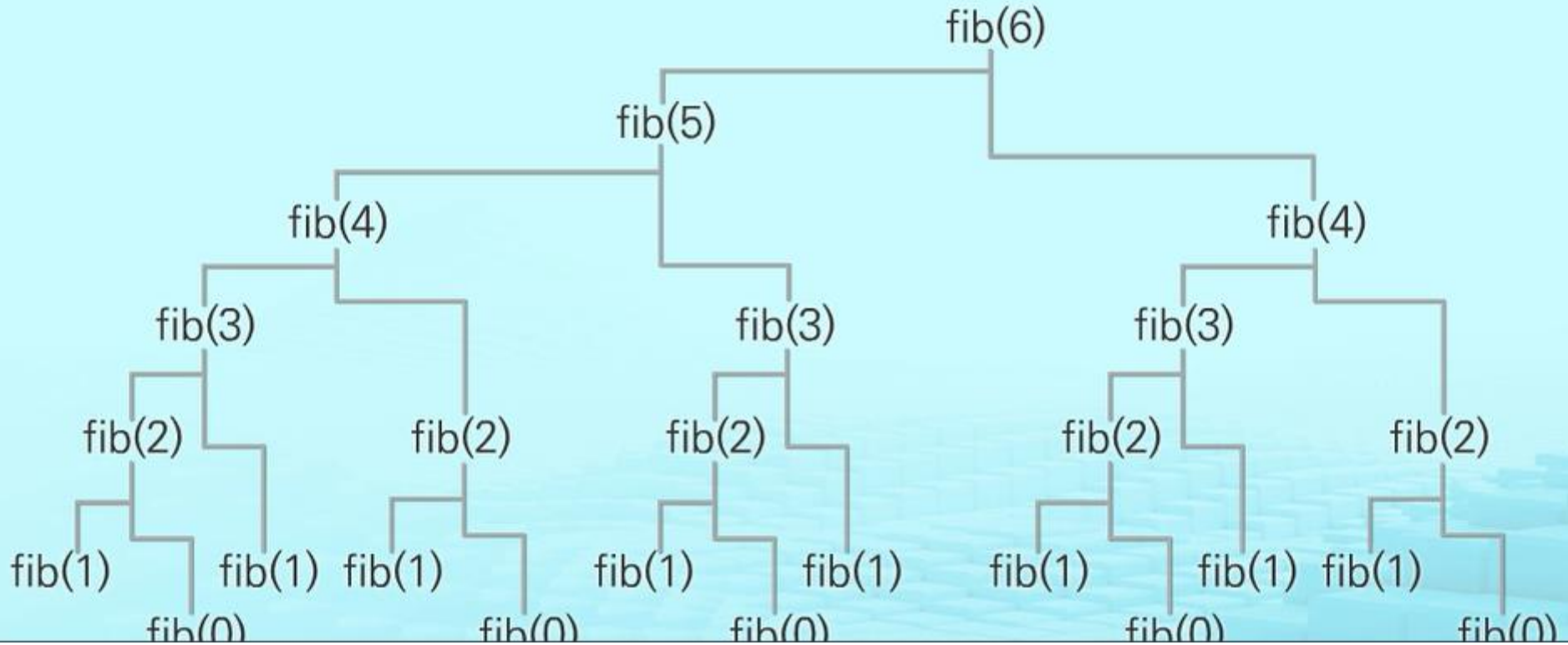
재귀 호출

1. 자기 자신을 호출하여 순환 수행되는 것
2. 함수에서 실행해야 하는 작업의 특성에 따라 일반적인 호출방식보다 재귀 호출 방식을 사용하여 함수를 만들면 프로그램의 크기를 줄이고 간단하게 작성할 수 있음
3. 디버깅이 어렵고 잘못 작성하게 되면 수행 시간이 많이 소요됨

3.Memoization

3 Memoization 2 피보나치 수를 구하는 재귀 함수

피보나치 수열의 재귀 함수 Call Tree



3 Memoization 3 Memoization이란?

메모이제이션(Memoization)의 의미

- 컴퓨터 프로그램을 실행할 때 이전에 계산한 값을 메모리에 저장해서 매번 다시 계산하지 않도록 하여 전체적인 실행속도를 빠르게 하는 기술
- DP(동적계획법)의 핵심이 되는 기술

Memoization 단어의 의미

- 글자 그대로 해석하면 '메모리에 넣기 (to put in memory)'라는 의미
- '기억되어야 할 것'이라는 뜻의 라틴어 Memorandum에서 파생
- 흔히 '기억하기', '암기하기'라는 뜻의 Memorization과 혼동하지만, 정확한 단어는 Memoization으로 동사형은 memoize임

4.DP

4 DP(동적 계획법) 1 DP(동적 계획법) 알고리즘

Dynamic Programming의 약자

그리디 알고리즘과 같이
최적화 문제를 해결하는 알고리즘

DP (동적 계획법) 알고리즘

- 먼저 입력 크기가 작은 부분 문제들을 모두 해결한 후에 그 해들을 이용하여 보다 큰 크기의 부분 문제들을 해결
- 최종적으로 원래 주어진 입력의 문제를 해결

4 DP(동적 계획법) 2 DP(동적 계획법)를 적용한 피보나치 수

피보나치 수를 구하는 함수에 DP 적용하기

부분 문제의 답으로부터 본 문제의 답을 얻을 수 있는
최적 부분 구조로 이루어져 있어 DP를 적용할 수 있음

1. 문제를 부분 문제로 분할
2. 부분 문제로 나누는 일을 끝냈으면 가장 작은 부분 문제부터 해를 구함
3. 그 결과는 테이블에 저장하고, 테이블에 저장된 부분 문제의 해를 이용하여 상위 문제의 해를 구함

4 DP(동적 계획법) 2 DP(동적 계획법)를 적용한 피보나치 수

DP의 구현 방식

recursive 방식 : fibo1()

재귀적 구조는 내부에
시스템 호출 스택을 사용하는
overhead가 발생할 수 있음

iterative 방식 : fibo2()

Memoization을 재귀적 구조에
사용하는 것보다 반복적 구조로 DP를
구현한 것이 성능 면에서 보다 효율적

5.DFS

5 DFS(깊이 우선 탐색) 1 DFS(깊이 우선 탐색)이란?

비선형구조인 그래프 구조는 그래프로 표현된 모든 자료를
빠짐없이 검색하는 것이 중요

- 깊이 우선 탐색
  (Depth First Search, DFS)
- 너비 우선 탐색
  (Breadth First Search, BFS)

5 DFS(깊이 우선 탐색) 1 DFS(깊이 우선 탐색)이란?

DFS(깊이 우선 탐색) 방법

시작 정점의 한 방향으로 갈 수 있는 경로가 있는 곳까지 깊이 탐색

더 이상 갈 곳이 없게 되면, 가장 마지막에 만났던 갈림길 간선이 있는 정점으로 되돌아옴

다른 방향의 정점으로 탐색을 계속 반복하여 결국 모든 정점을 방문하여 순회

가장 마지막에 만났던 갈림길의 정점으로 되돌아가서
다시 깊이 우선 탐색을 반복해야 하므로 후입선출 구조의 스택을 사용

5 DFS(깊이 우선 탐색) 2 DFS(깊이 우선 탐색) 알고리즘

시작 정점 v를 결정하여 방문

① 정점 v에 인접한 정점 중에서

방문하지 않은 정점 w가 있으면,
정점 v를 스택에 push하고
정점 w를 방문

w를 v로 하여 다시 ①를 반복

방문하지 않은 정점이 없으면,
탐색의 방향을 바꾸기 위해서
스택을 pop하여 받은
가장 마지막 방문 정점을 v로 하여
다시 ①을 반복