프로그래밍 언어 HW1: Python

B743014 양혜진

April 2, 2021

1 과제1: Binary Search(이진 탐색)

```
def binary_search(list, target, low, high):
    if low >= high:
        return -1
    mid = (low + high) // 2
    if list[mid] > target:
        return binary_search(list, target, low, mid)
    if list[mid] == target:
        return mid
    if list[mid] < target:
        return binary_search(list, target, mid + 1, high)
```

이진 탐색 알고리즘은 오름차순으로 정렬된 리스트에서 특정한 값의 위치를 찾는 알고리즘이다.

- 함수의 이름: binary_search()
- 함수의 파라미터: list, target, low, high
  - list: 탐색을 수행할 리스트
  - target: 탐색할 값
  - low: 최솟값의 인덱스(함수 최초 시작 시 0)
  - high: 최댓값의 인덱스(함수 최초 시작 시 len(list) - 1)
- 함수 동작 방식 설명: 리스트의 중간에 있는 값을 골라 pivot 이라는 변수에 저장한다. 그 다음 pivot 보다 작은 값은 lesser 리스트에 저장하고, pivot 보다 큰 값은 greater 리스트에 저장하고, pivot 과 같은 값은 equal 리스트에 저장한다. 그 다음, lesser 리스트와 greater 리스트를 quick_sort() 함수의 인자로 전달하여 재귀 호출함으로써, lesser 과 greater 리스트들이 원소 한 개만 남을 때 까지 다시 더 큰값과 작은 값으로 분할한 다음, 합쳐진 값을 반환할 수 있도록 한다.

2 과제2: Quick Sort(퀵 정렬)

```
def quick_sort(list):
    if len(list) <= 1:
        return list
    pivot = list[len(list) // 2]
    lesser, equal, greater = [], [], []
    for i in list:
```

```
        if i < pivot:
            lesser.append(i)
        elif i > pivot:
            greater.append(i)
        else:
            equal.append(i)
    return quick_sort(lesser) + equal + quick_sort(greater)
```

퀵 정렬은 분할 정복(divide and conquer) 방법을 통해 리스트를 정렬하는 알고리즘이다. 합병 정렬(Merge Sort)과 달리 리스트를 비균등한 크기로 분할한다.

- 함수의 이름: quick_sort()
- 함수의 파라미터: list
  - list: 탐색을 수행할 리스트
- 함수 동작 방식 설명: 리스트의 중간에 있는 값을 골라 pivot 이라는 변수에 저장한다. 그 다음 pivot 보다 작은 값은 lesser 리스트에 저장하고, pivot 보다 큰 값은 greater 리스트에 저장하고, pivot 과 같은 값은 equal 리스트에 저장한다. 그 다음, lesser 리스트와 greater 리스트를 quick_sort() 함수의 인자로 전달하여 재귀 호출함으로써, lesser 과 greater 리스트들이 원소 한 개만 남을 때 까지 다시 더 큰값과 작은 값으로 분할한 다음, 합쳐진 값을 반환할 수 있도록 한다.

3 과제3: Merge Sort(합병 정렬)

```
def merge_sort(list):
    if len(list) < 2:
        return list
    mid = len(list) // 2
    low = merge_sort(list[:mid])
    high = merge_sort(list[mid:])

    merged = []
    l = h = 0
    while l < len(low) and h < len(high):
        if low[l] < high[h]:
            merged.append(low[l])
            l += 1
        else:
            merged.append(high[h])
            h += 1
    merged += low[l:]
    merged += high[h:]
    return merged
```

합병 정렬은 분할 정복(divide and conquer) 방법을 통해 리스트를 정렬하는 알고리즘이다. 퀵 정렬(Quick Sort)과 달리 리스트를 균등한 크기로 분할한다.

- 함수의 이름: merge_sort()
- 함수의 파라미터: list

  - list: 탐색을 수행할 리스트

- 함수 동작 방식 설명: 리스트의 중간 인덱스 값을 구해, 리스트를 두 개의 균등한 크기로 분할하고, 분할된 리스트를 merge_sort()의 인자로 전달하여 재귀 호출함으로써 리스트의 원소가 한 개만 남을 때까지 리스트를 분할하여 반환한다. 반환된 값은 차례로 low와 high 리스트에 들어가는데, low 와 high 리스트는 값이 크거나 작은 리스트가 아니라, 기존 의 리스트를 앞과 뒤로 나눈 리스트이다. low 와 high 에 값이 할당되면, 두 원소의 값의 크기를 비교하여 merge 리스트에 정렬하여 저장한 다음 반환하도록 한다.

4 과제4: Binary Tree(이진 트리)

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class BinaryTree():
    def __init__(self):
        self.root = None

    def preorder_traverse(self):
        print("Preorder Traverse")
        self._preorder_traverse(self.root)

    def _preorder_traverse(self, node):
        if node is None: return
        print(node.data)
        self._preorder_traverse(node.left)
        self._preorder_traverse(node.right)

    def inorder_traverse(self):
        print("Inorder Traverse")
        self._inorder_traverse(self.root)

    def _inorder_traverse(self, node):
        if node is None: return
        self._inorder_traverse(node.left)
        print(node.data)
        self._inorder_traverse(node.right)

    def postorder_traverse(self):
        print("Postorder Traverse")
        self._postorder_traverse(self.root)

    def _postorder_traverse(self, node):
        if node is None: return
        self._postorder_traverse(node.left)
        self._postorder_traverse(node.right)
        print(node.data)
```

이진 트리(Binary Tree)는 각각의 노드가 최대 두 개의 자식 노드를 가지는 트리형 자료구조이다. 자식 노드를 각각 왼쪽 자식 노드와 오른쪽 자식노드라고 한다.

- 클래스의 이름: quick_sort()
- 클래스의 메서드: preorder_traverse,inorder_traverse, postorder_traverse, _preorder_traverse, _inorder_traverse, _postorder_traverse
  - preorder_traverse: 이진 트리를 전위 순회하는 함수
    부모 노드(root) -> 왼쪽 자식 노드(left) -> 오른쪽 자식 노드(right) 순서로 탐색
  - _preorder_traverse: 이진 트리 전위 순회를 재귀적 방식으로 구현하기 위해 선언한 함수. 재귀를 시작하기 전에 "Preorder Traverse" 를 출력한다.
  - inorder_traverse: 이진 트리를 중위 순회하는 함수
    왼쪽 자식 노드(left) -> 부모 노드(root) -> 오른쪽 자식 노드(right) 순서로 탐색
  - _inorder_traverse: 이진 트리 중위 순회를 재귀적 방식으로 구현하기 위해 선언한 함수. 재귀를 시작하기 전에 "Inorder Traverse" 를 출력한다.
  - postorder_traverse: 이진 트리를 후위 순회하는 함수
    왼쪽 자식 노드(left) -> 오른쪽 자식 노드(right) -> 부모 노드(root)순서로 탐색
  - _postorder_traverse: 이진 트리 후위 순회를 재귀적 방식으로 구현하기 위해 선언한 함수. 재귀를 시작하기 전에 "Inorder Traverse" 를 출력한다.
- 함수 동작 방식 설명: 왼쪽 자식 노드(left)를 먼저 출력하고 오른쪽 자식 노드(right)를 그 다음에 출력하는 순서는 같으나, 이 사이에 부모 노드(root)의 데이터를 어느 시점에 출력할지 정하는 것에 따라 전위, 중위, 후위 순회 가 정해진다.

5 과제5: Acrivity Selection Problem(활동 선택 문제)

```
NUM = 0; START = 1; END = 2

def lecture_sort(lectures, count):
    if len(lectures) <= 1:
        return lectures
    pivot = lectures[count // 2][END]
    lesser, equal, greater = [], [], []
    for i in range(count):
        if lectures[i][END] < pivot:
            lesser.append(lectures[i])
        elif lectures[i][END] > pivot:
            greater.append(lectures[i])
        else:
            equal.append(lectures[i])
    return lecture_sort(lesser, len(lesser))
            + equal + lecture_sort(greater, len(greater))

def activity_selection(lectures, count):
    result = [lectures[0][NUM]]
    before = 0
    for i in range(1, count):
        if lectures[i][START] >= lectures[before][END]:
```

```
            result.append(lectures[i][NUM])
            before = i
    return result
```

활동 선택 문제(Activity Selection Problem)은 시작(start)과 끝(end) 시간이 있는 n 개의 활동이 주어졌을 때, 한 번에 수행할 수 있는 최대 활동 수를 선택하는 문제로, 탐욕 알고리즘(Greedy Algorithm) 을 사용하여 풀 수 있다. 활동 선택 문제는 종료 시간이 가장 빠른 활동부터 선택하는 것이 핵심이다.

- 함수의 이름: activity_selection()
- 함수의 파라미터: lectures, count
  - lectures: 강의 번호, 시작 시간, 종료 시간이 담긴 리스트들의 리스트(이중배열)
  - count: 강의의 개수
- 함수 동작 방식 설명: 종료 시간이 가장 빠른 활동부터 차례대로 값을 비교하는 것이 좋기 때문에 각 활동들을 종료 시간 기준 오름차순으로 정렬한다. 그 다음, 정렬된 활동 중에서 첫 번째 활동을 선택하고, 나머지 활동들 중 시작 시간이 바로 직전에 선택한 활동의 종료시간보다 크거나 같은 경우 활동을 선택한다.

6 과제6: Pattern Matching Algorithm(패턴 매칭 알고리즘)

```
import re

count = int(input())

data_set = []
for i in range(count):
    data_set.append(input())

for data in data_set:
    if re.fullmatch('(100+1+|01)+', data): print("DANGER")
    else: print("PASS")
```

정규 표현식 또는 정규식은 특정한 규칙을 가진 문자열의 집합을 표현하는 데 사용하는 형식 언어이다. 문자열에서 특정한 패턴이 매칭하는지를 검사하는 정규 표현식을 파이썬으로 구현해보았다. 파이썬으로 정규표현식을 사용하기 위해서는 regex 라이브러리인 re 모듈을 import 해야한다. 과제6의 패턴을 정규표현식을 사용하지 않고 파이썬 코드로 작성한다면 아래와 같다.

- 사용한 정규표현식(regex) 메타 문자(meta characters)
  - | : or 과 동일한 의미로 사용된다. 만약 'A|B' 라는 정규식이 있다면 A 또는 B 라는 의미이다.
  - + : 최소 1번 이상 반복될 때 사용한다. * 는 반복 횟수가 0부터이지만, + 는 반복 횟수가 1부터 시작이다.
- 사용한 컴파일된 패턴(pattern) 객체의 메서드
  - fullmatch() : 문자열의 처음부터 남는 부분 없이 정규식과 매치되는지 조사한다.

```
def data_encryption_analyis(data):
    i = 0
    while i < len(data):
        if len(data[i:]) >= len("100") and data[i:i+3] == "100":
            i += len("100")
            while i < len(data) and data[i] == '0': i+= 1
            if i == len(data): print("PASS"); return
            if data[i] == '1':
                while i < len(data) and data[i] == '1': i+= 1
            else: print("PASS"); return
            if i < len(data)-1 and data[i-2:i+2] == "1100": i -= 1
            continue
        if len(data[i:]) >= len("01") and data[i:i+2] == "01":
            i += len("01"); continue
        print("PASS"); return
    print("DANGER")
```