알고리즘과 자료구조 워크북

교과목명 : 알고리즘과 자료구조
차시명: 5차시 분할정복과 동적 프로그래밍

◈ 담당교수: 신일훈 **(서울과학기술대학교)**

◉ 세부목차
- 병합정렬을 수행하는 알고리즘
- 분할정복의 문제점과 한계
- 동적 프로그래밍의 개념
- 동적 프로그래밍을 활용하여 피보나치 수열을 구하는 알고리즘

학습에 앞서

■ 학습개요

분할정복의 개념에 대한 이해를 바탕으로 이를 활용하여 병합정렬 알고리즘을 설계하고 구현한다. 또한 동일한 부분문제가 반복되는 경우, 이를 효율적으로 처리하지 못하는 분할정복의 문제점에 대해서 이해하고 이를 해결하기 위한 동적 프로그래밍 개념을 강의한다. 동적 프로그래밍을 활용하여 시간복잡도가 낮은 피보나치 수열을 구하는 알고리즘을 각각 설계하고 파이썬으로 구현한다.

■ 학습목표

1	병합정렬을 수행하는 알고리즘을 설계하고 표현할 수 있다.
2	분할정복의 문제점 및 한계를 이해한다.
3	동적 프로그래밍의 개념을 이해한다.
4	동적 프로그래밍을 적용하여 피보나치 수열을 구하는 알고리즘을 설계할 수 있다.

■ 주요용어

용어	해설
병합정렬	대상 리스트를 리스트의 크기가 1이 될 때까지 반으로 쪼개는 과정을 반복하고, 이후 이들을 병합하는 단계를 통해 정렬을 수행하는 알고리즘
동적프로그래밍	동일한 부분 문제가 반복될 때, 부분 문제들의 답을 메모리에 저장해 두고 이를 활용하는 기법. 시간복잡도를 낮출 수 있다.
딕셔너리	key와 value를 쌍으로 저장하는 파이썬 자료구조
time	현재 시간을 반환하는 메서드를 포함한 파이썬 클래스. 시간 측정 시 time() 함수를 사용한다.

학습하기

1. 분할정복의 적용: 병합정렬

가. 문제

- N개의 숫자를 저장한 파이썬 리스트를 내림차순으로 정렬하시오.

나. 아이디어

- 입력: list = [2, 5, 8, 3, 11, 7, 6, 9]
- 대상 리스트를 두 개의 서브 리스트로 분할한다. 가령 위의 예에서는 [2, 5, 8, 3], [11, 7, 6, 9]의 두 개의 리스트로 분할한다.
- 이러한 분할 작업을 리스트를 더 이상 분할할 수 없을 때까지 반복한다.
- 즉, 리스트는 원소가 하나만 남을 때까지 분할된다. 그리고 원소가 하나만 있으므로 자연히 정렬된 리스트가 된다.
- 분할이 끝나면, 분할된 정렬된 리스트를 정렬하여 통합하는 작업을 반복한다.
- 가령, 마지막 직전 단계에서 서브 리스트들은 [8, 5, 3, 2], [11, 9, 7, 6]로 정렬되어 있는데, 이를 통합하면 된다.
- 통합 작업은 각 리스트의 첫 번째 원소들을 비교하여 더 큰 값을 제거하고 이를 결과를 저장할 리스트에 추가하면 된다.
- 이를 반복하면 하나의 서브 리스트가 빈 상태가 되고, 남은 서브 리스트의 값들을 결과를 저장할 리스트에 추가하면 된다.
- 다음은 이를 반복하는 과정을 개념적으로 나타낸다.

1. list = [2, 5, 8, 3, 11, 7, 6, 9]
2. [2, 5, 8, 3] , [11, 7, 6, 9]
3. [2, 5], [8, 3], [11, 7], [6, 9]
4. [2], [5], [8], [3], [11], [7], [6], [9]
5. [5, 2], [8, 3], [11, 7], [9, 6]
6. [8, 5, 3, 2], [11, 9, 7, 6]
7. [11, 9, 8, 7, 6, 5, 3, 2]

다. 알고리즘(파이썬으로 표현)

```
def merge_sort(list) :
        length = len(list)
        if (length == 1) :
                return list
        start1 = 0
        start2 = length // 2

        list1 = list[start1:start2]
        list2 = list[start2:length]
        list1 = merge_sort(list1)
        list2 = merge_sort(list2)
        result = merge(list1, list2)
        return result
```

- merge_sort() 함수는 인자로 전달받은 list를 병합정렬을 재귀적으로 수행하여 내림차순으로 정렬한 후에 정렬된 리스트를 반환하는 함수이다.
- len() 함수는 리스트의 원소 개수를 반환한다. 즉 원소가 하나인 경우는 이미 정렬된 상태이므로 리스

트를 바로 반환한다.
- // 연산자는 나눗셈의 몫을 구하는 연산자이다.
- start1은 대상 리스트를 분할할 때 첫째 서브리스트의 시작 인덱스를 저장하고 start2는 둘째 서브리스트의 시작 인덱스를 저장한다.
- list1 = list[start1:start2] => start1부터 start2-1까지의 원소들을 추출하여 list1으로 저장한다.
- list2 = list[start2:length] => start2부터 length-1까지의 원소들을 추출하여 list2로 저장한다.
- 즉 위의 두 구문을 통해 대상 list가 list1, list2라는 서브리스트로 분할된다.
- 분할된 서브리스트들을 인자로 하여 merge_sort()를 다시 재귀적으로 호출함으로써, 각각의 서브리스트들을 내림차순으로 정렬한다.
- 정렬된 list1, list2를 인자로 하여 merge() 함수를 호출하여 두 리스트를 내림차순으로 정렬하여 통합한다. 즉 merge() 함수는 정렬된 리스트들을 인자로 전달받아, 이들을 정렬된 리스트로 통합하여 반환한다.
- 다음은 파이썬으로 구현된 merge() 함수의 알고리즘이다.

```
def merge(list1, list2) :
        result = []
        while (len(list1) > 0 and len(list2) > 0) :
                if(list1[0] > list2[0])  :
                        result.append(list1.pop(0))
                else :
                        result.append(list2.pop(0))

        while (len(list1) > 0) :
                result.append(list1.pop(0))

        while (len(list2) > 0) :
                result.append(list2.pop(0))

        return result
```

- result는 결과를 저장할 리스트이다. 비어 있는 상태로 초기화된다.
- 통합을 수행할 두 리스트에 모두 원소가 있는 동안에는 각각의 첫 번째 값을 비교하여 더 큰 값을 해당 리스트에서 제거하고 (pop(0)), result 리스트에 추가하는 작업을 반복한다. (append())
- 따라서 result 리스트에는 가장 큰 값부터 차례로 저장된다.
- 두 리스트 중 하나가 비게 되면, while 문을 빠져 나온다.
- 이후, 두 리스트 중 원소가 남아 있는 리스트에서 원소들을 첫째 값부터 차례로 제거하여 (pop(0)) 제거한 값을 result에 추가한다.
- 따라서 result에는 정렬된 통합 리스트가 저장된다.

라. 알고리즘 최악 시간복잡도
- merge() 함수를 보면 연산의 회수가 두 리스트의 원소의 개수의 합에 비례하여 증가한다.
- 따라서 1개의 원소로 분할된 서브리스트들을 2개의 원소를 가진 리스트들로 통합하는 시간복잡도가 N에 비례한다.
- 또한 2개의 원소로 분할된 서브리스트들을 4개의 원소를 가진 리스트들로 통합하는 시간복잡도가 N에 비례한다.
- 이러한 통합이 logN에 비례하여 일어나므로 통합하는 최악의 시간복잡도는 O(NlogN)이다.
- 분할하는 연산은 N에 비례하여 증가한다.
- 따라서 전체 알고리즘의 최악 시간복잡도는 O(NlogN)이다.

마. 파이썬 구현 및 실행

```
list = [3, 5, 7, 1, 8, 2]
print(merge_sort(list))
```

- 파이썬 코드는 의사 코드와 동일하다. 다만 테스트하는 코드를 추가하면 된다.
- 다음은 이 파이썬 코드를 스파이더에서 실행한 결과 화면이다. 정렬된 리스트가 올바르게 출력되었다.

```
In [43]: runfile('D:/data/lecture/
파이썬으로배우는자료구조와알고리즘/code/알고리즘/
untitled1.py', wdir='D:/data/lecture/
파이썬으로배우는자료구조와알고리즘/code/알고리즘')
[8, 7, 5, 3, 2, 1]
```

2. 동적 프로그래밍 개념

가. 피보나치 수열을 분할정복으로 해결할 때의 문제점

- 피보나치 수열을 구하는 과정 (재귀)
  - f(10) = f(9) + f(8)
  - f(9) = f(8) + f(7)
  - f(8) = f(7) + f(6)
  - …
  - => 동일한 부분 문제를 반복하여 풀고 있음을 알 수 있음.
  - 가령, f(9)를 구할 때 f(8)의 답을 구하지만, f(10)을 계산할 때, f(8)을 다시 계산함
- 동일한 부분 문제를 반복하여 풀기 때문에 시간복잡도가 높음. => $O(\log 2^N)$
- 실제 분할정복으로 구현한 피보나치 수열을 구하는 프로그램을 실행하면, N이 커질수록 실행 시간이 급격하게 증가함을 확인할 수 있다. 가령 f(40, f(50)을 구하는데 굉장히 오랜 시간이 걸리는 것을 확인할 수 있다.
- 동적 프로그래밍은 이를 해결하기 위한 기법이다.

나. 동적 프로그래밍 개념

- 동적 프로그래밍에서는 이미 구한 정답 f(K)를 메모리에 저장하여 추후 다시 계산하는 과정 없이 활용함으로써(memoization) 시간복잡도를 개선한다.
- 즉, 동적 프로그래밍은 분할정복과 마찬가지로 원 문제를 더 작은 부분 문제로 쪼개어 부분 문제들을 해결함으로써 원 문제를 해결한다.
- 다만 동일한 부분 문제가 반복되는 경우, 이미 답을 구한 부분 문제의 답을 메모리에 기억해 둠으로써(memoization), 추후 동일한 부분 문제를 다시 풀어야할 때 저장된 답을 활용함으로써 시간복잡도를 개선한다.
- 가령 피보나치 수열 예제에서는 이미 구한 피보나치 수열의 정답을 저장해두고 추후 동일한 수열을 필요로할 때 다시 계산하는 과정 없이, 저장된 값을 활용한다.

3. 동적 프로그래밍을 활용한 피보나치 수열

가. 문제

- 피보나치 수열을 동적 프로그래밍을 활용하여 구하시오.

나. 아이디어

- 계산한 피보나치 수열의 결과값을 메모리에 저장해야 한다.
- 파이썬에서는 리스트 또는 딕셔너리를 활용할 수 있는데, 여기서는 딕셔너리를 활용하도록 한다.

- 딕셔너리는 (key, value)의 집합이며, key를 통해 value를 찾을 수 있다.
- 따라서 fibo(N)에서 N을 key로, fibo(N)을 value로 저장하면 된다.

다. 알고리즘(파이썬으로 표현)

```
def fibo(num) :
        dict = {}
        return fibo_dict(num, dict)
```

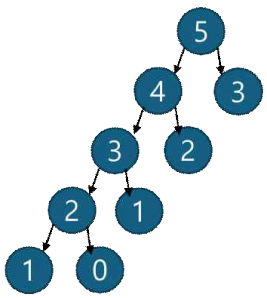
- fibo() 함수는 피보나치 수열을 구하여 반환한다.
- 사용할 딕셔너리 dict를 비어 있는 상태로 초기화한다.
- fibo_dict() 함수를 호출하여 피보나치 수열을 구한다.

- 다음은 fibo_dict() 함수의 알고리즘이다.

```
def fibo_dict(num, dict) :
        if (num == 0 or num == 1) :
                return num
        if (num in dict) :
                return dict[num]
        else :
                dict[num] = fibo_dict(num-1, dict) + fibo_dict(num-2, dict)
                return dict[num]
```

- 만약 인자가 0 또는 1이라면 바로 0 또는 1을 반환한다.
- 그렇지 않은 경우에는 먼저 num이 이미 정답을 알고 있는 수인지를 확인한다.
- num in dict 구문은 num에 해당하는 key가 dict에서 발견되면 참을 반환한다. 따라서 if문이 만족되는 경우는 이미 피보나치 수열을 구한 경우이다. 이 경우에는 구한 수열값이 dict의 value로 저장되어 있으므로 이를 반환한다. 즉 재귀 호출을 통해 피보나치 수열을 계산하는 과정이 생략된다.
- 그 외의 경우에는 아직 해당 피보나치수를 구하지 못한 상태이므로, 분할정복을 이용하여 피보나치수를 구한다. 중요한 것은 구한 값을 dict에 저장해 놓아야 한다는 것이다.

라. 알고리즘 최악 시간복잡도



- 위 그림은 fibo(5)를 구할 때 함수가 호출되는 것을 나타낸다. 즉 fibo(5)는 아직 답을 구하지 못했으므로 재귀 호출을 통해 fibo(4), fibo(3)이 각각 호출된다.
- 먼저 fibo(4)가 호출되면, 아직 답을 구하지 못했으므로 역시 fibo(3), fibo(2)가 호출된다.
- fibo(3)이 호출되면 아직 답을 구하지 못했으므로 역시 fibo(2), fibo(1)이 호출된다.
- fibo(2)도 아직 답을 구하지 못했으므로 fibo(1), fibo(0)이 호출되며, fibo(1)은 1을 반환하고 fibo(0)은 0을 반환한다.
- fibo(2)가 계산되었으므로 이 값이 dict에 저장된다.
- 이후 fibo(3)도 계산되어 결과가 dict에 저장된다.
- fibo(4)를 구할 때 fibo(2)가 호출되지만 이 때는 재귀 호출을 하지 않고 dict에 저장된 값이 활용된다.

그리고 fibo(4)는 dict에 저장된다.

- fibo(5)를 구할 때 fibo(3)의 결과가 필요한데, 이 때는 재귀 호출을 하지 않고 dict에 저장된 값이 활용된다.
- 결과적으로 memoization에 의해 함수 호출 회수가 N에 비례하여 증가함을 알 수 있다. 즉 최악의 시간복잡도는 O(N)이다.

마. 파이썬 구현 및 실행

```
import time
start = time.time()
res = fibo(30)
end = time.time()
print(res, end-start)
start = time.time()
res = fibo(40)
end = time.time()
print(res, end-start)
```

- 파이썬으로 구현된 테스트 코드에서는 fibo(30), fibo(40을 출력하고 있으며 이를 계산하는데 걸리는 시간을 함께 출력한다.
- 다음은 이 파이썬 코드를 스파이더에서 실행한 결과 화면이다. 시간복잡도가 N에 비례하므로 실제 실행 시간이 0으로 출력됨을 알 수 있다. (동적 프로그래밍을 사용하지 않으면 훨씬 더 큰 값이 출력됨)

```
In [60]: runfile('D:/data/lecture/
파이썬으로배우는자료구조와알고리즘/code/알고리즘/
untitled1.py', wdir='D:/data/lecture/
파이썬으로배우는자료구조와알고리즘/code/알고리즘')
832040 0.0
102334155 0.0
```

연습문제

1. 동적 프로그래밍과 분할정복의 차이점을 설명하시오.

정답 : 두 기법은 모두 원 문제를 부분 문제들로 나누고 부분 문제들을 해결함으로써 원 문제를 해결한다. 동적 프로그래밍은 동일한 부분 문제가 반복될 때 부분 문제의 해답을 메모리에 저장하고 이를 활용함으로써 알고리즘의 시간복잡도를 낮춘다.

해설 : 정답 참조

2. 분할정복으로 설계한 피보나치 수열의 최악의 시간복잡도를 구하시오.

정답 : $O(2^n)$
해설 : n이 1씩 감소할 때마다 2배의 피보나치 수로 확장된다. 따라서 연산의 수가 2^n 에 비례하여 증가한다.

3. 동적 프로그래밍을 활용한 피보나치 수열의 최악의 시간복잡도를 구하시오.

정답 : O(n)

해설 : 한번 계산한 피보나치 수를 다시 계산하지 않으므로 연산의 수는 n에 비례하여 증가한다.

정리하기

1. 병합정렬은 대상 리스트를 리스트의 크기가 1이 될 때까지 반으로 쪼개는 과정을 반복하고, 이후 이들을 병합하는 단계를 통해 정렬을 수행하는 정렬 알고리즘이다. 분할정복을 활용하는 예이다.
2. 피보나치 수 알고리즘을 분할정복으로 설계하면 동일한 부분 문제를 반복적으로 계산하는 오버헤드가 있다. 이를 해결하기 위한 기법이 동적 프로그래밍이다.
3. 동적 프로그래밍은 동일한 부분 문제가 반복될 때 이에 대한 해답을 메모리에 저장해 두고 추후 이를 활용함으로써 알고리즘의 시간복잡도를 낮춘다.
4. 피보나치 수열을 구하는 알고리즘을 동적 프로그래밍을 활용하여 설계할 수 있다.

참고자료

- 파이썬 알고리즘, 최영규, 2021, 생능출판

다음 차시 예고

- 탐욕 알고리즘에 대해 학습한다.