알고리즘과 자료구조 워크북

교과목명 : 알고리즘과 자료구조
차시명: 4차시 축소정복, 분할정복

◈ 담당교수: 신일훈 **(서울과학기술대학교)**

◉ 세부목차
- 축소정복, 분할정복의 개념
- 팩토리얼을 계산하는 알고리즘
- 피보나치 수열을 구하는 알고리즘
- 제곱근을 구하는 알고리즘

학습에 앞서

■ 학습개요

축소정복과 분할정복의 개념에 대해 배운다. 또한 이들 기법에서 주로 활용하는 재귀함수의 개념에 대해 공부한다. 축소정복과 분할정복을 활용하여 팩토리얼 계산, 피보나치 수열, 제곱근을 구하는 알고리즘을 각각 설계하고 파이썬으로 구현한다.

■ 학습목표

1	축소정복, 분할정복의 개념을 이해한다.
2	재귀함수의 개념 및 유의사항을 이해한다.
3	팩토리얼을 계산하는 알고리즘을 설계하고 표현할 수 있다.
4	피보나치 수열을 구하는 알고리즘을 설계하고 표현할 수 있다.
5	제곱근을 구하는 알고리즘을 설계하고 표현할 수 있다.

■ 주요용어

용어	해설
축소정복	원래 문제를 더 작은 문제로 축소해가며, 간단하게 풀 수 있는 작은 문제를 해결함으로써 원래 문제를 해결.
분할정복	원래 문제를 더 작은 부분 문제들로 분할하여, 부분 문제들을 해결함으로써 원래 문제를 해결
재귀함수	자기 자신을 호출하는 함수
팩토리얼	N! = 1 * 2 * ... * (N-1) * N
피보나치 수열	f(n) = f(n-1) + f(n-2), f(0) = 0, f(1) = 1

학습하기

1. 축소정복(decrease & conquer) 과 분할정복(divide & conquer) 개념

가. 축소정복 개념

- 원래 문제를 더 작은 문제로 축소하는 것을 반복하여, 간단하게 풀 수 있는 작은 문제를 해결함으로써 원래 문제를 해결하는 기법
- 예시
  - n에 대한 문제를 n-1에 대한 문제로 축소하여 문제를 해결하는 예시
    - 1~N의 합계 => 1~(N-1)의 합계로 축소 => 1~(N-2)의 합계로 축소 => … => 1~1의 합계로 축소
    - N! => (N-1)!로 축소 => (N-2)!로 축소 => … => 1! 로 축소
  - 솔루션이 존재하는 범위를 축소하여 문제를 해결하는 예시
    - 제곱근 구하기
    - 1~100 숫자 맞추기
  - 재귀(recursion) 또는 반복을 활용

나. 분할정복 개념

- 원래 문제를 더 작은 부분 문제들로 분할하여, 부분 문제들을 해결함으로써 원래 문제를 해결.
- 예시
  - 피보나치 수열
  - 병합(merge) 정렬
  - 재귀 또는 반복을 활용

2. 재귀 개념

가. 재귀함수

- 자기 자신을 호출하는 함수를 재귀 함수라고 한다
- 재귀 함수의 예

```
def rec_func(call_count) :
    if (call_count == 0) :
        return
    print(call_count)
    call_count -= 1
    rec_func(call_count)
```

- 파이썬으로 구현한 rec_func() 함수의 경우, 함수 안에서 자기 자신을 다시 호출하고 있음을 알 수 있다. 이런 함수를 재귀 함수라고 한다.
- 재귀 함수는 자기 자신을 다시 호출하므로 무한 반복될 우려가 있다.
- 따라서 재귀 함수는 반드시 종료 조건을 포함해야 한다.
- 위 함수의 경우 인자로 전달된 call_count가 0이면 더 이상 자신을 호출하지 않고 종료함을 알 수 있다.
- 이처럼 종료 조건을 반드시 포함해야만 무한반복에 빠지는 것을 방지할 수 있다.

나. 재귀를 활용한 축소정복

- 원리
  - f(n)의 해를 f(n-1)을 이용하여 구한다.
- 예시

- 팩토리얼 구하기
    - f(n) = n * f(n-1) and f(1) = 1
- 1~n까지 합계
    - f(n) = n + f(n-1) and f(1) = 1

다. 재귀를 활용한 분할정복

- 예시
    - 피보나치 수열
        - f(n) = f(n-1) + f(n-2)
        - f(0) = 0, f(1) = 1

라. 장점과 단점

- 직관적이고 이해하기 쉬운 문제 해결 기법이다.
- 과도한 함수 호출로 인해 스택 오버플로우가 발생할 가능성이 있다.
- 반복을 활용한 프로그램에 비해, 과도한 함수 호출로 인해 성능이 떨어질 수 있다.

2. 축소정복의 적용1: 팩토리얼 계산

가. 문제

- N! (팩토리얼)을 계산하시오.

나. 아이디어

- N! = 1 * 2 * … (N-1) * N
- N에 대한 문제를 어떻게 N-1의 문제로 축소할 것인가가 관건이다.
- N에 대한 솔루션을 N-1을 활용한 솔루션으로 표현해야 한다.
    - N! = (N-1)! * N
    - N에 대한 솔루션이 N-1을 활용한 솔루션으로 표현됨.

다. 알고리즘(의사코드로 표현)

```
def my_fact(n) :
        if (n == 1) :
            return 1
        return (n * my_fact(n-1))
```

- my_fact() 함수는 n을 인자로 전달받아 n!을 재귀 용법으로 구하여 반환한다.
- 만약 인자 n이 1이라면 1!은 1이므로 바로 1을 반환하여 함수를 종료한다.
- 그렇지 않은 경우에는 n-1을 인자로 하여 자신을 다시 호출하고 여기에 n을 곱한 값을 반환한다.
- 위의 의사코드에서 my_fact(n)은 n!을 나타내고 my_fact(n-1)은 (n-1)!을 의미한다. 즉 n에 대한 솔루션이 n-1에 대한 솔루션으로 표현되어 있음을 알 수 있다.

라. 알고리즘 최악 시간복잡도

- 인자가 n인 경우, my_fact() 함수는 약 n번 재귀적으로 호출되며, 최종적으로 n이 1로 감소했을 때 재귀 호출을 종료한다. 즉 함수의 호출 횟수가 n에 비례하여 증가함을 알 수 있다.
- 따라서 알고리즘의 최악 시간복잡도는 O(n)이다.

마. 파이썬 구현 및 실행

```
def my_fact(n) :
        if (n == 1) :
            return 1
        return (n * my_fact(n-1))

print(my_fact(5))
```

- 파이썬 코드는 의사 코드와 거의 동일하다. 다만 my_fact() 함수를 테스트하는 코드가 추가되어 있다. 즉 테스트 코드에서는 5!을 출력한다.
- 다음은 이 파이썬 코드를 스파이더에서 실행한 결과 화면이다. 5!인 120이 올바르게 출력되었다.

```
In [32]: runfile('D:/data/lecture/
파이썬으로배우는자료구조와알고리즘/code/알고리즘/
untitled1.py', wdir='D:/data/lecture/
파이썬으로배우는자료구조와알고리즘/code/알고리즘')
120
```

3. 분할정복의 적용1: 피보나치 수열

가. 문제

- 피보나치 수열 fibo(N)을 구하시오.

나. 아이디어

- 피보나치 수열 공식:
  - fibo(n) = fibo(n-1) + fibo(n-2)
  - fibo(0) = 0
  - fibo(1) = 1
- 피보나치 수열은 답을 알고 있는 fibo(0), fibo(1)에서 시작하여 fibo(2), fibo(3), …, fibo(n)을 구하는 식으로 반복문을 활용하여 구할 수 있다.
- 하지만 여기서는 재귀를 적용하여 n에 대한 문제를 n-1과 n-2에 대한 문제로 분할하여 축소한다. 이 과정을 반복하면 결국에는 fibo(0)과 fibo(1)로 귀결되므로 fibo(n)을 구할 수 있다.

다. 알고리즘(의사코드로 표현)

```
def fibo(n) :
        if (n == 0 or n == 1) :
            return n
        return (fibo(n-1) + fibo(n-2))
```

- fibo() 함수는 n을 인자로 전달받아 피보나치 수열을 재귀 용법으로 구하여 반환한다.
- 만약 인자 n이 0이나 1이라면 바로 n을 반환하여 함수를 종료한다.
- 그렇지 않은 경우에는 n-1을 인자로 하여 자신을 다시 호출한 값과 n-2를 인자로 하여 자신을 다시 호출한 값을 더한 값을 반환한다. (피보나치 수열의 정의)

라. 알고리즘 최악 시간복잡도

- n이 충분히 클 경우, fibo(n)은 fibo(n-1), fibo(n-2)의 호출을 수반하며, 다시 fibo(n-1)이 호출되면 fibo(n-2), fibo(n-3)의 호출이 수반된다. 즉 함수를 거칠 때마다 2 개의 함수가 호출됨을 알 수 있다. 즉 인자가 n이라면 함수의 호출 회수가 2^n에 비례하여 증가한다.
- 따라서 알고리즘의 최악 시간복잡도는 $O(2^n)$이다.

마. 파이썬 구현 및 실행

```
def fibo(n) :
        if (n == 0 or n == 1) :
            return n
        return (fibo(n-1) + fibo(n-2))

print(fibo(10))
```

- 파이썬 코드는 의사 코드와 거의 동일하다. 다만 fibo() 함수를 테스트하는 코드가 추가되어 있다. 즉 테스트 코드에서는 fibo(10)을 출력한다.
- 다음은 이 파이썬 코드를 스파이더에서 실행한 결과 화면이다. 55가 올바르게 출력되었다.

```
In [34]: runfile('D:/data/lecture/
파이썬으로배우는자료구조와알고리즘/code/알고리즘/
untitled1.py', wdir='D:/data/lecture/
파이썬으로배우는자료구조와알고리즘/code/알고리즘')
55
```

4. 축소정복의 적용2: 제곱근 계산

가. 문제

- 양의 정수 N의 제곱근을 구하시오.

나. 아이디어

- 제곱근은 무리수일 수 있고, 컴퓨터는 무리수를 표현할 수 없으므로 정확한 해를 구할 수는 없다. 즉, 근사값을 구해야 한다.
- 근사값과 정확한 해의 차이가 충분히 작다면 이 근사값을 제곱근으로 간주한다.
- 다만 정확한 제곱근 값은 알 수 없으므로 근사값을 제곱한 값과 N의 차이가 충분히 작다면, 해당 근사값을 제곱근으로 간주한다.

- 관건은 오차가 적은 근사값을 어떻게 최대한 빨리 찾을 것인가이다.
- 축소 정복을 적용하여 해가 존재하는 구간을 좁혀 나가는 방법을 선택한다.
- 즉, N이 1 이상의 정수라면, 제곱근은 1~N사이에 존재한다.
- 따라서 이 구간의 중간값을 해로 추정하여 이 값을 제곱한 값과 N을 비교한다. 만약 N이 더 작다면 정확한 해는 중간값을 기준으로 왼쪽 구간에 존재한다는 의미이며 N이 더 크다면 정확한 해가 중간값을 기준으로 오른쪽 구간에 존재한다는 의미이다.
- 따라서 각각의 경우에 구간의 범위를 수정하여 새로운 중간값을 구한다.
- 그리고 이 중간값을 새로운 근사값으로 간주하여 다시 오차를 계산한다.
- 이러한 작업을 오차가 충분히 작아질 때까지 반복하면 정확한 해에 가까운 근사값을 구할 수 있다.

다. 알고리즘(의사코드로 표현)

```
def my_sqrt(num) :
        if (num >= 1) :
                sol = my_sqrt_rec(1, num, num)
        elif (num > 0) :
                # 생략
        else :
                sol = -1
        return sol

def my_sqrt_rec(low, high, num) :
        ans = (low + high) / 2
        diff = ans*ans - num
        if (abs(diff) <  0.0001) :
                return ans
        elif (diff > 0) :
                return my_sqrt_rec(low, ans, num)
        else :
                return my_sqrt_rec(ans, high, num)
```

- my_sqrt 함수는 num을 인자로 전달받아 num의 제곱근을 구하여 반환한다. 이 때 num이 1이상의 수일 때만 제곱근을 구한다.
- my_sqrt_rec()는 my_sqrt() 함수에 의해 호출된다.
- my_sqrt_rec()는 세 개의 인자를 전달받는데, 첫째, 둘째 인자는 해가 존재하는 구간의 하한값과 상한값을 의미한다. 세 번째 인자는 제곱근을 구할 대상 수이다.
- my_sqrt_rec() 함수에서 ans는 답이 존재하는 구간의 중간값을 의미하고, diff는 중간값의 제곱과 num의 오차를 계산한다.
- abs() 함수는 절대값을 반환한다.
- 오차가 0.0001보다 작으면 현재의 근사값을 정답으로 반환하다.
- 오차가 이보다 큰 경우에는 오차의 부호에 따라 중간값을 기준으로 왼쪽 구간, 혹은 오른쪽 구간에 대해, my_sqrt_rec()를 다시 호출하여 근사값 추정하는 작업을 다시 반복한다.

라. 알고리즘 최악 시간복잡도

- 재귀 함수가 호출되는 회수가 n과 직접적인 연관이 없다. 따라서 최악 시간복잡도를 계산할 수 없다.

마. 파이썬 구현 및 실행

```
def my_sqrt(num) :
        if (num >= 1) :
                sol = my_sqrt_rec(1, num, num)
        elif (num > 0) :
                # 생략
        else :
                sol = -1
        return sol

def my_sqrt_rec(low, high, num) :
        ans = (low + high) / 2
        diff = ans*ans - num
        if (abs(diff) <  0.0001) :
                return ans
        elif (diff > 0) :
                return my_sqrt_rec(low, ans, num)
        else :
                return my_sqrt_rec(ans, high, num)

print(my_sqrt(4))
print(my_sqrt(1))
print(my_sqrt(2))
```

- 파이썬 코드는 의사 코드와 거의 동일하다. 다만 my_sqrt() 함수를 테스트하는 코드가 추가되어 있다. 즉 테스트 코드에서는 4, 1, 2의 제곱근을 출력한다.
- 다음은 이 파이썬 코드를 스파이더에서 실행한 결과 화면이다. 정답에 가까운 근사값이 출력되었음을 알 수 있다.

```
In [49]: runfile('D:/data/lecture/
파이썬으로배우는자료구조와알고리즘/code/알고리즘/
untitled1.py', wdir='D:/data/lecture/
파이썬으로배우는자료구조와알고리즘/code/알고리즘')
2.000001907348633
1.0
1.414215087890625
```

연습문제

1. 축소정복을 설명하시오.

정답 : 축소정복은 원래 문제를 더 작은 문제로 축소해가며, 작은 문제를 해결함으로써 원래 문제를 해결하는 기법이다.

해설 : 정답 참조

2. 재귀함수를 정의하시오.

정답 : 재귀함수는 자기 자신을 다시 호출하는 함수이다.

해설 : 재귀함수는 자기 자신을 다시 호출하는 함수이며, base case를 정의하지 않으면 무한 반복될 수 있으므로 유의해야 한다.

3. 1~N까지의 합계를 구하는 알고리즘을 축소정복을 사용하여 설계하고 의사코드로 표현하시오.

정답 :

```
def cal_sum(num) :
        if (num == 1) :
                return 1
        return num + cal_sum(num-1)
```

해설 : f(n)을 f(n-1)로 표현하면 된다. 이 때 base case를 잊지 말아야 한다.

정리하기

1. 축소정복은 원래 문제를 더 작은 문제로 축소해가며, 작은 문제를 해결함으로써 원래 문제를 해결하는 기법이다.
2. 분할정복은 원래 문제를 더 작은 부분 문제들로 분할하고, 이들 부분 문제들을 해결함으로써 원래 문제를 해결하는 기법이다.
3. 재귀 함수는 자신을 호출하는 함수이다. Base case를 사용하지 않으면 무한반복할 수 있다.
4. 팩토리얼을 계산하는 알고리즘을 축소정복 전략을 활용하여 설계할 수 있다.
5. 피보나치 수열을 구하는 알고리즘을 분할정복 전략을 활용하여 설계할 수 있다.
6. 제곱근을 구하는 알고리즘을 축소정복 전략을 활용하여 설계할 수 있다.

참고자료

- 파이썬 알고리즘, 최영규, 2021, 생능출판

다음 차시 예고

- 분할정복과 동적프로그래밍에 대해 학습한다.