Divide (or Decrease) and conquer

– 재귀적 기법을 중심으로 –

Divide (or Decrease) and conquer

- 개념
  - 원래 문제를 더 작은 문제로 쪼개어서 혹은 축소하여, 작은 문제들을 해결함으로써 원래 문제를 해결.
  - 재귀적 기법 또는 반복문 활용

Divide (or Decrease) and conquer

- 예시
  - N에 대한 문제를 N-1에 대한 문제로 축소
    - N! = (N-1)! * N
      - => f(n) = f(n-1) * n => 원래 문제를 더 작은 문제로 축소함
    - sum(1 ~ n) = sum(1 ~ n-1) + n
      - => f(n) = f(n-1) + n => 원래 문제를 더 작은 문제로 축소함

재귀 함수 활용

  - 해가 존재하는 공간을 반복적으로 축소
    - 1-100 사이의 숫자 맞추기 게임
    - 제곱근 구하기

재귀 (recursion) 알고리즘

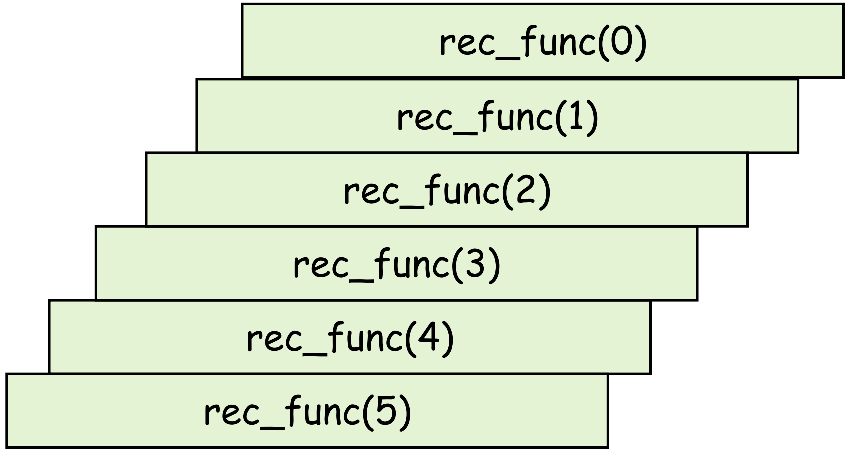
- 재귀 함수를 이용하는 알고리즘
- 재귀 함수
  - 자기 자신을 호출하는 함수를 재귀 함수라고 함

재귀 (recursion) 알고리즘

- 재귀 함수의 예

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

rec_func(5)
```



재귀를 활용한 DC

- 원리
  - $f(n)$의 해를 $f(n-1)$을 이용해 구하라.
- 예
  - 팩토리얼
    - $f(n) = n * f(n-1)$ and $f(1) = 1$
  - 1~n까지 합계
    - $f(n) = n + f(n-1)$ and $f(1) = 1$
  - 피보나치 수열
    - $f(n) = f(n-1) + f(n-2)$ and $f(0) = 1, f(1) = 1$
  - …

재귀를 활용한 DC

- 장점
  - 직관적이고 이해하기 쉬운 문제 해결 기법
  - 문제를 유사한 형태의 더 작은 문제로 축소하여 해결
- 단점
  - 과도한 함수 호출로 인한 stack overflow 가능성
  - 과도한 함수 호출로 인한 낮은 성능 (반복 알고리즘에 비해)

문제1

- N! (팩토리얼)을 계산하시오.

- 의사코드
  - $N*(N-1)*(N-2)*\cdots*1$
    - 반복문 활용하여 가능

  - $N! = N * (N-1)!$
    - 재귀 알고리즘 가능

문제1

- $N! = N * (N-1)!$
- 의사코드 (문제점)?

```
my_fact(n) {
        return (n * my_fact(n-1))
}
```

문제1

- 재귀 알고리즘을 작성할 때는 재귀 호출을 종료하는 base case를 잊지 말고 쓰자.
- 의사코드

```
my_fact(n) {
    if (n == 1) {
        return 1
    }
    return (n * my_fact(n-1))
}
```

문제1

- 파이썬 코드

문제2

- 피보나치 수열 *fibo(N)*을 구하시오.
- 의사코드
  - 반복 알고리즘으로 가능
  - 재귀 알고리즘으로 가능

문제2

- 의사코드

  fibo(n) = fibo(n-1) + fibo(n-2)

  base case?

  fibo(0) = 1

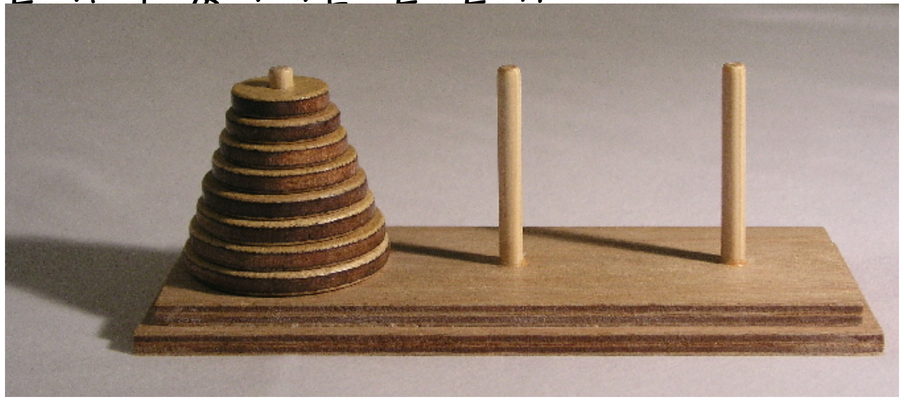
  fibo(1) = 1

문제2

- 파이썬 코드

문제3 (하노이의 탑 – 위키백과 참조)

- 세 개의 기둥과 이 기둥에 꽂을 수 있는 크기가 다양한 원판들이 있다. 한 기둥에 크기가 큰 원판부터 차례대로 쌓여 있다.
- 다음 조건을 만족시키면서, 한 기둥의 원판들을 모두 다른 한 기둥으로 이동시키시오.
  - 한 번에 한 개의 원판만 옮길 수 있다.
  - 큰 원판이 작은 원판 위에 있어서는 안 된다.



출처: 위키백과

문제3

- 원판이 한 개라면..
  - move disc 1 from pole A to pole C
- 원판이 두 개라면 (2 is larger than 1),
  - move disc 1 from pole A to pole B
  - move disc 2 from pole A to pole C
  - move disc 1 from pole B to pole C

문제3

- 원판이 N 개라면..
  - how to decrease or divide the problem?

문제3

- 원판이 N 개라면..
  - how to decrease or divide the problem?
    - 우리는 원판이 1개일 때의 solution을 알고 있다 (base case).
    - 원판이 N-1개일 때의 solution을 안다는 가정하에서, 원판이 N개일 때의 solution을 원판이 N-1개일 때의 solution을 이용하여 도출하면 됨 (재귀적 기법)
      - cf. $f(1) = 1$

        $f(n) = a*f(n-1) + b$

문제3

- 원판이 두 개라면,
  - move disc 1 from pole A to pole B
  - move disc 2 from pole A to pole C
  - move disc 1 from pole B to pole C
- 원판이 N 개라면..
  - move disc 1~(N-1) from pole A to pole B
  - move disc N from pole A to pole C
  - move disc 1~(N-1) from pole B to pole C

문제3

- 파이썬 코드 예시

문제4 (병합정렬)

- N개의 숫자를 저장한 파이썬 리스트를 내림차순으로 정렬하시오.

- 아이디어
  - 리스트를 두 개의 서브 리스트로 분할하여 각각을 정렬 (더 작은 문제로 쪼갬)
  - 각각의 정렬된 서브 리스트를 통합

문제4 (병합정렬)

- 의사코드 예시

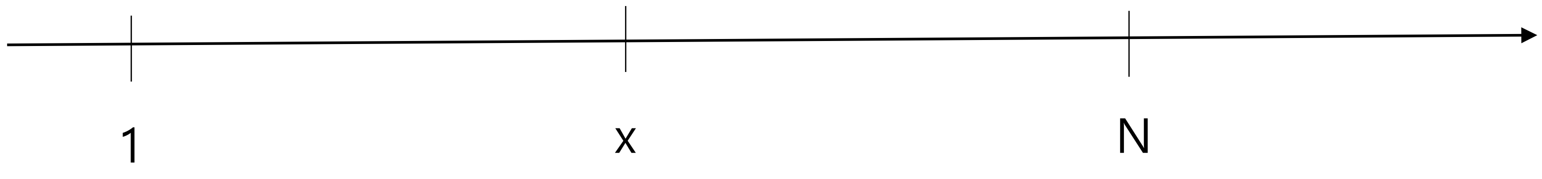
```
merge_sort(list)
{
      if (len(list) == 1) {
           return list
      }
      list1 = list[0:mid]             # mid is the middle point
      list2 = list[mid:end]
      list1 = merge_sort(list1)
      list2 = merge_sort(list2)
      result = merge(list1, list2)
      return result
 }
```

문제4 (병합정렬)

```
merge(list1, list2)
{
    result is empty
    while (both list1 and list2 are not empty) {
        if(list1[0] > list2[0]) {
            result.append(list1.pop(0))
        }
        else {
            result.append(list2.pop(0))
        }
    }
    while(list1 is not empty) {
        result.append(list1.pop(0))
    }
    while(list2 is not empty) {
        result.append(list2.pop(0))
    }
    return result
}
```

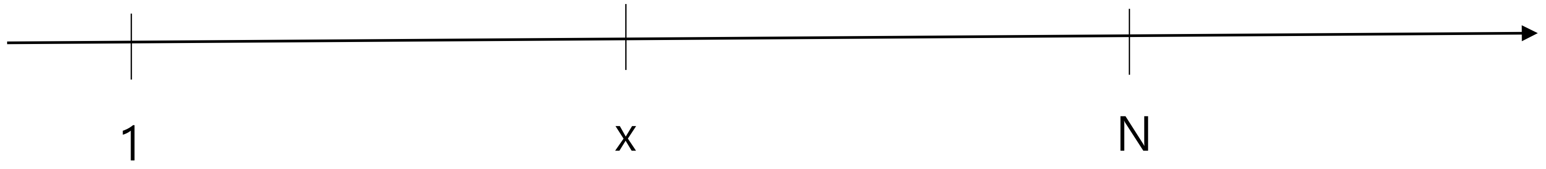
문제5 (반복을 활용한 DC)

- 양의 정수 N의 제곱근을 구하시오.
- 아이디어
  - 정확한 해를 구하지 못할 수 있음.
  - 우리는 근사값을 구해야 함.
  - 근사값과 해의 차이가 충분히 작다면 (가령 제곱의 결과 차이가 0.01이하), 이 값을 해로 간주
  - 해(x)가 위치하는 구간을 어떻게 좁혀갈 것인가?
    - 중간값을 해로 간주하여, 근사 해를 찾을 때까지 반복



문제5 (반복을 활용한 DC)

- 아이디어 (5의 제곱근)
  - 1과 5의 중간값 3을 해로 가정
  - 3의 제곱은 9이므로 5보다 큼. 즉 해는 3보다 작음을 알 수 있다.
  - 따라서 이번에는 기존 low인 1과 새로운 high인 3의 중간값인 2를 해로 가정
  - 2의 제곱은 4이므로 5보다 작음. 즉 해는 4보다 작음을 알 수 있다.
  - 따라서 이번에는 새로운 low인 2와 기존의 high인 3의 중간값인 2.5를 해로 가정
  - 위의 작업을 제곱의 결과 차이가 충분히 작을 때까지 반복하면 됨.



문제5 (반복을 활용한 DC)

- 의사코드

```
low = 1
high = number
ans = (low + high) / 2
while (1) {
        if (ans * ans – number) < 0.01) {
                break
        }

        if (ans * ans > number) {
                high = ans
                ans = (low + high) / 2
        }
        else if (ans * ans < number) {
                low = ans
                ans = (low + high) / 2

        }
}
```

문제6 (knapsack - 재귀)

- 각각 무게가 $weight_i$, 가치가 val_i인 n개의 물건들을 배낭에 넣으려고 한다. 배낭에 넣을 수 있는 최대 무게는 W이다. 배낭에 넣은 물건들의 가치를 최대로 하는 물건의 조합을 구하시오.

문제6 (knapsack - 재귀)

- 아이디어 (재귀 기법으로 해결해보자.. $f(n) = f(n-1)\cdots$)
  - n-1개의 아이템에 대한 솔루션이 존재한다고 가정하고, 이를 이용하여 n개의 아이템에 대한 솔루션을 설계하자.
    - FindOptValue(N, W) : N개의 아이템, 최대 무게 W 조건에서 가치를 최대로 하는 solution
    - FindOptValue(N, W)를 N-1개의 아이템에 대한 함수로 기술해야 함.

    - FindOptValue(1, 10) : 1개의 아이템이 있고 최대 무게가 10일 때 가치를 최대로 하는 solution
    - FindOptValue(2, 15) : 2개의 아이템이 있고 최대 무게가 15일 때 가치를 최대로 하는 solution
    - …
    - FindOptValue(N-1, W2) : N-1개의 아이템이 있고 최대 무게가 W2일 때 가치를 최대로 하는 solution

문제6 (knapsack - 재귀)

- 아이디어 (재귀 기법으로 해결해보자.. $f(n) = f(n-1)\cdots$)
  - 우리는 n-1개의 아이템에 있을 때, 가치를 최대로 하는 솔루션을 알고 있다고 가정
    - FindOptValue(n-1, W)
  - 아이템의 개수가 n-1개에서 n개로 증가할 때, 가능한 경우의 수는 2가지임
    - n번째 아이템을 선택하는 경우
    - n번째 아이템을 선택하지 않는 경우
  - 우리는 각각의 경우에 대해 가치를 최대로 하는 solution을 구하고, 이 중 가치가 더 큰 solution을 최종 solution으로 선택하면 됨.

문제6 (knapsack – 재귀)

- 아이디어 (재귀 기법으로 해결해보자.. $f(n) = f(n-1)\cdots$)
  1. n번째 아이템을 선택하는 경우에 가치를 최대로 하는 solution
     - FindOptValue(n, W) : W는 허용 가능한 최대 무게.

     - 마지막 물건을 선택하므로 기존의 최종 solution에 마지막 item의 무게와 가치가 더해진다…
     - 따라서 item이 n–1개일 때, 허용 가능한 최대 무게는 (W – n번째 item의 무게)
     - 최종 가치는 item이 n–1개일 때의 최대 가치에 n번째 item의 가치를 더해 주어야 함.

     - FindOptValue(n, W) = FindOptValue(n–1, W–weight$_n$) + val$_n$ ———————— A

     - 단, 이 때 W–weight$_n$ 가 0보다 작다면, 마지막 item은 선택할 수 없음. 즉, A를 버려야 함.

문제6 (knapsack - 재귀)

- 아이디어 (재귀 기법으로 해결해보자.. $f(n) = f(n-1)\cdots$)

2. n번째 아이템을 선택하지 않는 경우에 가치를 최대로 하는 solution
   - FindOptValue(n, W) : W는 허용 가능한 최대 무게.

   - 마지막 물건을 선택하지 않으므로 최종 solution에 마지막 item의 무게와 가치가 더해지지 않는다…
   - 즉, item이 n-1개이고, 허용 가능한 최대 무게가 W일 때의 solution과 동일함

   - FindOptValue(n, W) = FindOptValue(n-1, W) ------------ B

문제6 (knapsack – 재귀)

- 아이디어 (재귀 기법으로 해결해보자.. $f(n) = f(n-1)\cdots$)
  - FindOptValue(n, W) = max(A, B)
  - A = FindOptValue(n–1, W–weight$_n$) + val$_n$ when W–weight$_n$ >= 0
  - B = FindOptValue(n–1, W)

문제6 (knapsack - 재귀)

- 아이디어 (재귀 기법으로 해결해보자.. $f(n) = f(n-1)\cdots$)
  - Base case?
    - 1개의 아이템만 있는 경우.
    - 이 때는 아이템의 weight가 주어진 무게 상한을 초과하지 않으면 해당 아이템 포함하여 가치 계산. 아니면 포함하지 않음. 즉 가치가 0이 됨.

문제6 (knapsack - 재귀) – 파이썬 구현 /단계

- item 클래스 정의
  - Item 클래스는 각각의 아이템을 표현
  - 아이템의 이름, 가치, 무게를 저장
  - 생성자만 있으며 여타 아이템 관련 메서드는 없음.

```
class Item(object):
    def __init__(self, name, value, weight):
        self.name = name
        self.value = value
        self.weight = weight
```

문제6 (knapsack - 재귀) – 파이썬 구현 2단계

- Knapsack 클래스 정의
  - 멤버변수
    - 아이템들을 저장한 items 리스트. self.items[]
    - 허용 가능한 무게의 최대값. self.max_weight
    - 가치를 최대로 할 때의 가치. self.max_value
  - 메소드
    - 생성자

문제6 (knapsack - 재귀) – 파이썬 구현 2단계

- Knapsack 클래스 정의

```
class Knapsack(object):
    def __init__(self, names, values, weights, max_weight):
        self.items = []
        self.max_weight = max_weight
        self.max_value = 0
        self.opt_case = 0

        for i in range(len(names)) :
            item = Item(names[i], values[i], weights[i])            # Item 객체 생성
            self.items.append(item)
```

문제6 (knapsack – 재귀) – 파이썬 구현 3단계

- 최대가치 메서드 정의

```
…

class Knapsack(object):
   …

   def findOptCaseRecursive(self):
        chosen_items = []        # 선택된 item들의 index들을 저장

        (chosen_items, max_value) = self.__findOptCaseRecursive(len(self.items), self.max_weight, 
chosen_items)

        return (chosen_items, max_value)
```

문제6 (knapsack - 재귀)

```
class Knapsack(object):
    …

    def __findOptCaseRecursive(self, count, max_weight, chosen_items):
        last_item = self.items[count-1]

        if (count == 1):    # base case
            if (last_item.weight <= max_weight):  # select this item
                chosen_items.append(count-1)
                value = last_item.value
                return(chosen_items, value)
            else:
                return(chosen_items, 0)
```

문제6 (knapsack - 재귀)

```
def __findOptCaseRecursive(self, count, max_weight, chosen_items):
    …
    chosen_items2 = list(chosen_items)

    if (max_weight >= last_item.weight) :
        chosen_items, value1 = self.__findOptCaseRecursive(count-1, max_weight - last_item.weight, chosen_items)      # A
    else :
        value1 = -1
    chosen_items2, value2 = self.__findOptCaseRecursive(count-1, max_weight, chosen_items2)   # B
```

문제6 (knapsack - 재귀)

```
def __findOptCaseRecursive(self, count, max_weight, chosen_items):
    ...

    if (value1 + last_item.value >= value2 and value1 != -1):
        chosen_items.append(count-1)
        return (chosen_items, value1 + last_item.value)
    else :
        return (chosen_items2, value2)
```

문제6 (knapsack - 재귀)

```
names = ['0', '1', '2', '3', '4']
values = [10, 30, 20, 14, 23]
weights = [5, 8, 3, 7, 9]
max_weight = 20

knapsack = Knapsack(names, values, weights, max_weight)

(chosen_items, max_value) = knapsack.findOptCaseRecursive()
print(chosen_items, max_value)
```

Summary

- Divide and conquer 또는 decrease and conquer 알고리즘
  - 재귀적 문제 해결 기법 활용
    - f(n)의 해를 f(n-1)을 이용하여 도출
    - Don't forget the base cases
    - 예시
      - 팩토리얼, 피보나치 수열
      - 하노이의 탑
      - knapsack
    - 프로그래밍이 상대적으로 쉬우나, 성능은 상대적으로 낮음 (성능이 중요한 요소라면, 반복문으로 대체)
  - 반복을 통한 알고리즘 설계

homework candidate

- 1부터 n까지의 합계를 구하는 함수를 재귀적 기법으로 작성하시오.

homework candidate

- 0-1 사이의 실수에 대해 제곱근 구하는 함수를 작성하시오.