재귀(Recursion, 순환)

1

재귀 (recursion, 순환)

- 문제의 해를 부분문제(subproblem: 작은 크기의 입력에 대한 동일한 문제)의 해를 이용하여 해결하는 방법
- recursive definition

예 1: 0이상의 정수 n에 대한 n! 정의

n! = 1 if n = 0 // base case

n! = n * (n – 1)! if n > 0 // recursive case

2

n!을 구하는 recursive function (파이썬)

- recursive function: 자기 자신을 호출하는 함수

```
# python
def fact(n):
# Precondition(사전조건): n >= 0.
    if n == 0:    #  base case
        return  1
    else:        # recursive case(general case)
        return  n * fact(n - 1)
```

수행시간 분석

n! 계산에서 기본연산: 두 정수 곱셈

T(n): fact(n)을 수행할 때 기본연산 수

T(n) = 0 if n = 0이면

= T(n-1) + 1 if n > 0이면

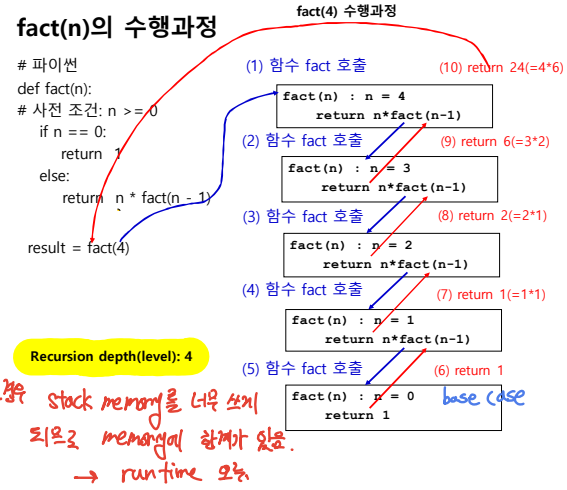
위의 점화식으로부터 T(n)을 구한다

T(n) = T(n-1) + 1
= [T(n-2) + 1] + 1
= T(n-2) + 2
...
= T(n-k) + k

n-k = 0이면, k = n이다.
T(n) = T(0) + n = n
O(n)

3

fact(n)의 수행과정



너무 깊을경우 stack memory를 너무 쓰게 되므로 memory의 한계가 있음.
→ runtime 오류

4

n!을 구하는 recursive function(C)

```
// C
int fact(int n )
// Precondition(사전조건): n >= 0.
{
    if (n == 0)  //  base case
        return  1 ;
    else                  //  recursive case(general case)
        return  n * fact(n - 1) ;
}
```

수행시간 분석

n! 계산에서 기본연산: 두 정수 곱셈

T(n): fact(n)을 수행할 때 기본연산 수

T(n) = 0 if n = 0이면

= T(n-1) + 1 if n > 0이면

위의 점화식으로부터 T(n)을 구한다

T(n) = T(n-1) + 1
= [T(n-2) + 1] + 1
= T(n-2) + 2
...
= T(n-k) + k

n-k = 0이면, k = n이다.
T(n) = T(0) + n = n
O(n)

5

2. gcd (최대공약수, greatest common divisor)

- 양의 정수 a,b의 최대공약수 gcd(a,b)의 재귀적 정의

gcd(a,b) = b a가 b로 나누어지면 // base case

= gcd(b,a%b) 그렇지 않으면 // recursive case

```
# python
def gcd(a, b):
#   if (a%b == 0):
#       return b
    if b == 0:
        return a
    else:
        return gcd(b,a%b)
```

수행시간 분석

기본연산: %

대략적인 분석:

a ≥ b라 하자.

gcd(a,b) = gcd(b,a%b) = gcd(a%b,b%(a%b))

a%b ≤ a/2이다. (그 이유는?)

gcd를 2번 호출하면 gcd 첫번째 매개변수가 a/2보다 같거나 작게 됨

gcd(a,b)의 시간복잡도

$O(\log_2 \max(a,b))$

앞으로 log의 base 2는 생략

6

gcd (C)

- 양의 정수 a,b의 최대공약수 gcd(a,b)의 재귀적 정의

gcd(a,b) = b a가 b로 나누어지면 // base case
= gcd(b,a%b) 그렇지 않으면 // recursive case

```
// C
int gcd(int a, int b)
{
//  if (a%b == 0)
//      return b;
   if (a%b == 0)
      return a;
   else
      return gcd(b,a%b);
}
```

7

3. x^n 계산

- 0이상 정수 n에 대하여 x^n 계산

```
# python
def exp1(x, n):
    result = 1
    for i in range(1,n+1):
        result *= x;

    return result
```

수행시간 O(n)

```
// C
int exp1(int x, int n)
{
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= x;

    return result;
}
```

수행시간 O(n)

8

x^n 계산

- 0이상 정수 n에 대하여 x^n

x^n 의 재귀적 정의 (n ≥0):

$x^n = 1$ if n = 0

$= x * x^{n-1}$ if n > 0

```
// C
int exp2(float x, int n) {
// n은 0이상 정수
   if (n == 0)
      return 1;
   else
      return x*exp2(x,n-1);
}
```

```
# python
def exp2(x,n):
# n은 0이상 정수
    if n == 0:
        return 1
    else:
        return x*exp2(x,n-1)
```

수행시간 분석

T(n): exp2(x,n)을 수행할 때 곱셈 수

T(n) = 0 n = 0이면

= T(n-1) +1 n > 0이면

T(n): O(n)

9

$x^{1025} = x \cdot x^{1024} = x \cdot (x^{512})^2$

$x^n = x \cdot x^{n-1}$

x^n 계산

x^n 의 재귀적 정의 (n ≥0):

$x^n = 1$ if n = 0 base

$= (x^{n/2})^2$ if n is even 짝수

$= x * x^{n-1}$ if n is odd 홀수

```
# python
def exp3(x, n):# n은 0이상 정수
    if n == 0:
        return 1
    elif (n % 2 == 0):
        temp = exp3(x,n/2)
        return temp*temp
    else:
        return x*exp3(x,n-1)
```

```
// C
int exp3(int x, int n)
{
// n은 0이상 정수
    if (n == 0)
        return 1;
    else if (n % 2 == 0) {
        int temp = exp3(x,n/2);
        return temp*temp;
    }
    else
        return x*exp3(x,n-1);
}
```

10

x^n 계산

```
# python
def exp3(x, n):
# n은 0이상 정수
    if n == 0:
        return 1
    elif (n % 2 == 0):
        temp = exp3(x,n/2)
        return temp*temp
    else:
        return x*exp3(x,n-1)
```

수행시간 대략적 분석

T(n): exp3(x,n)을 수행할 때 두 정수 곱셈 수

T(n) = 0 n = 0이면

≤ T(n/2) + 2 n > 0이면

T(n) ≤ T(n/2) + 2

≤ [T(n/2²)+ 2] + 2

≤ T(n/2²) + 2 + 2

≤ [T(n/2³)+ 2] + 2 + 2

≤ T(n/2³) + 2 + 2 + 2

…

≤ T(n/2^k) + 2*k

$n/2^k = 1$ 이면, 즉 $2^k = n$이다 $(k = \log_2 n)$

T(n) ≤ T(1) + 2*log₂ n

(T(1) = 1)

T(n) ≤ 2*log₂ n + 1

T(n) : O(log n)

11

x^n 계산

- 0이상 정수 n에 대하여 x^n

x^n 의 재귀적 정의:

$x^n = 1$ if n = 0

$= (x^2)^{n/2}$ if n is even

$= x * (x^2)^{(n-1)/2}$ if n is odd

```
def exp4(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return exp4(x*x,n/2)
    else:
        return x*exp4(x*x,(n-1)/2)
```

```
int exp4(int a, int n)
{
    if (n == 0)
        return 1;
    else if (n % 2 == 0)
        return exp4(x*x,n/2);
    else
        return x*exp4(x*x,(n-1)/2);
}
```

12

4. 이진탐색 (binary search)

- 정렬되어 있는 리스트 A에서 item과 같은 원소의 위치를 찾아라.

```
# python
def binarySearch(A, item, left, right): # 리스트의 A[left] 부터 A[right] 까지 최소값 위치
    if left <= right:
        mid = (left + right)//2
        if item == A[mid]:
            return mid
        elif item < A[mid]:
            return binarySearch(A, item, left, mid-1)
        else:
            return binarySearch(A, item, mid+1, right)
    else:
        return -1
```

binarySearch(A,item,0,n-1)을 호출
수행시간: O(log n)

4. 이진탐색 (binary search)

- 정렬되어 있는 배열 A[0..n-1]에서 item과 같은 원소의 위치를 찾아라.

```
// C
int binarySearch(int A[], int item, int left, int right)
{
    if (left <= right) {
        int mid = (left + right)/2;
        if (item == A[mid])
            return mid;
        else if (item < A[mid])
            return binarySearch(A, x, first, mid-1);
        else
            return binarySearch(A, x, mid+1, last);
    }
    else
        return -1;
}
```

binarySearch(A,item,0,n-1)을 호출
수행시간: O(log n)

5. 하노이 탑(Hanoi Tower) 문제

- 세 개의 막대기 1, 2, 3이 있고 서로 다른 크기의 n개의 원반들이 막대기 1에 크기순서(위에서부터 아래로 크기가 증가하는 순서)대로 놓여 있다. 다음 규칙을 지키면서 막대기 1에 있는 모든 원반들을 막대기 3으로 옮겨라.

  규칙 1: 한번에 막대기의 맨 위에 있는 한 장의 원반만을 다름 막대기 위로 옮길 수 있다.

  규칙 2: 큰 원반은 절대로 작은 원반 위에 놓여질 수 없다.

- 알고리즘

단계 1) 막대기 1의 가장 큰 원반(가장 아래에 있는 원반)을 제외한 나머지 n –1 개의 원반을 막대기 2로 옮긴다 (막대기 3을 이용).

단계 2) 막대기 1의 원반(가장 큰 원반)을 막대기 3으로 옮긴다.

단계 3) 막대기 2에 놓여 있는 n-1개의 원반을 막대기 3으로 옮긴다.

하노이 탑(Hanoi Tower) 문제

- 프로그램

```
# python
def hanoiTower(n, source, dest, temp):
    if (n == 1):
        print( "Move a disk from peg %d to peg  %d" % (source, dest))
#       print( "Move a disk from peg {0} to peg  {1}".format(source, dest))

    else:
        hanoiTower(n-1, source, temp, dest)
        print( "Move a disk from peg %d to peg  %d" % (source, dest))
        hanoiTower(n-1, temp, dest, source)
```


하노이 탑(Hanoi Tower) 문제

- 프로그램

```
// C
void hanoiTower(int n, int source, int dest, int temp)
{
    if (n == 1)
        printf( "Move a disk from peg %d to peg  %d₩n", source, dest);
    else {
        hanoiTower(n-1, source, temp, dest);  T(n-1)
        printf( "Move a disk from peg %d to peg  %d₩n", source, dest);
        hanoiTower(n-1, temp, dest, source);  T(n-1)
    }
}
```


하노이탑 알고리즘 수행시간 분석

- **수행시간 분석**

T(n) : n장의 원반을 옮기는데 필요한 move 횟수

n = 1 인 경우 ⇒ T(n) = 1
n > 1 인 경우 ⇒ T(n) = 2*T(n-1) + 1

$T(n) = 2T(n-1) + 1 = 2[2T(n-2) + 1] + 1$
$= 2^2T(n-2) + 2 + 1 = 2^2[2T(n-3) + 1] + 2 + 1$
$= 2^3T(n-3) + 2^2 + 2 + 1$
...
$= 2^kT(n-k) + 2^{k-1} + \dots + 2^2 + 2 + 1 = 2^kT(n-k) + 2^k - 1$
n-k = 1, 즉 k = n-1일때,
$= 2^{n-1}T(1) + 2^{n-1} - 1$
$= 2^n - 1$

- 시간복잡도 :
  T(n) : $O(2^n)$ 혹은 $\Theta(2^n)$
  (수행시간 = 최악의 경우 수행시간 = 평균적인 경우 수행시간)