동적계획법
(Dynamic Programming)

- 개요
- 피보나치 수열
- 이항계수

동적계획법 개요

- 분할과 정복(Divide and Conquer) 방법은 Top-Down Design으로서 문제를 부분문제(subproblem)들로 나누고 이들 부분문제(subproblem)들의 해를 recursive하게 구하여 이들 해들로부터 원래의 문제의 해를 구한다.
  - 작은 문제들은 독립적이다

- 동적계획법(Dynamic Programming)도 주어진 문제의 해를 부분문제(subproblem)들의 해를 이용하여 구한다.
  - 분할과 정복과는 달리, 주어진 문제의 해를 구하는데 필요한 부분문제들은 서로 독립적이지 않다 (부분문제들이 overlap하는 경우)

동적계획법 개요

- 두가지 접근방법
  - Memoization (Top-down)

    Can be formulated both via recursion and saving results in a table (*memoization*). After computing the solution to a subproblem, store in a table. Subsequent calls just do a table lookup.

  - Bottom-Up

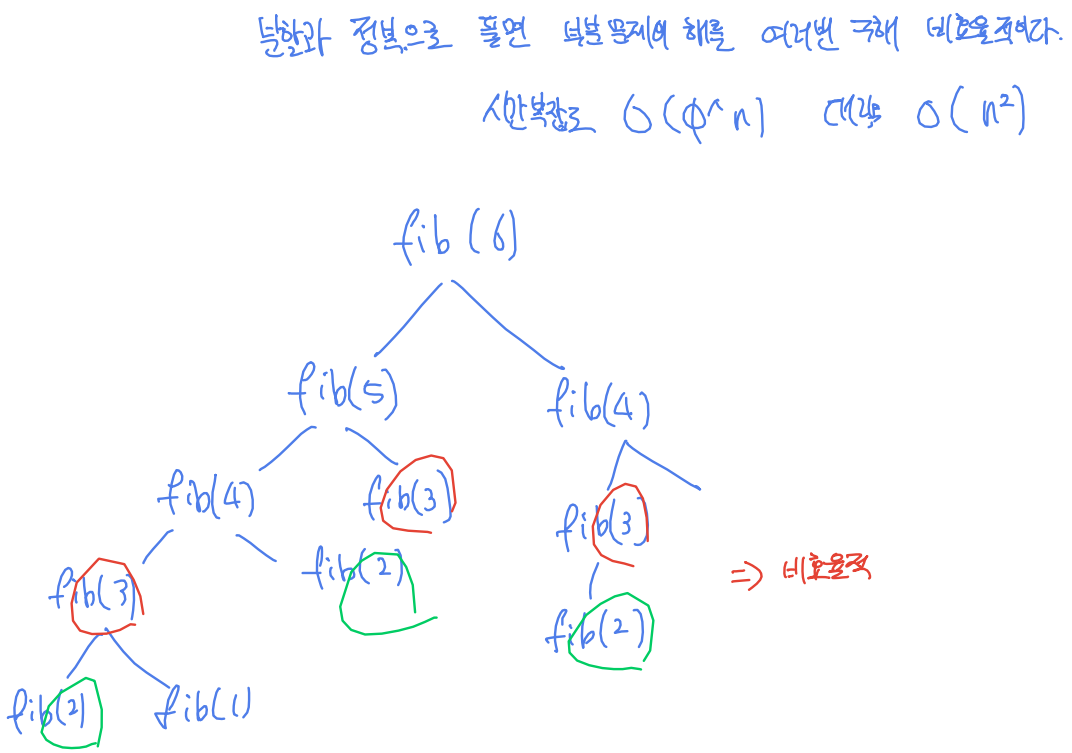
    작은 부분문제들부터 시작하여 큰 부분문제들까지 해를 차례대로 구한다. 부분문제들에 대한 해를 구하면 이를 테이블에 저장한다. 부분 문제에 대한 해를 구할 때 필요한 작은 부분 문제들의 해는 다시 계산하지 않고 테이블을 참조하면 된다.

1. 피보나치 수열

- f_n: n번째 피보나치 수

  $f_0 = 0, f_1 = 1$

  $f_n = f_{n-1} + f_{n-2}$, $n \geq 2$



피보나치 수열

- f_n: n번째 피보나치 수

  $f_0 = 0, f_1 = 1$

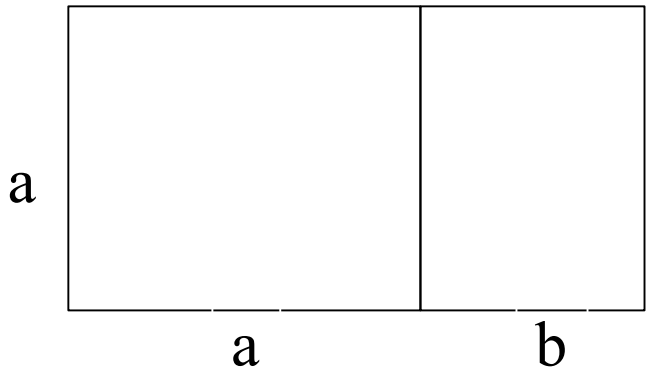
  $f_n = f_{n-1} + f_{n-2}$, $n \geq 2$

  Find the golden ratio when we divide a line into two parts a and b such that

  $(a + b) / a = a / b = \Phi$

Golden ratio $\Phi = (1 + \sqrt{5})/2 = 1.61803398\ldots$

Golden rectangle



a/b
8/5 = 1.6
13/8 = 1.625…
21/13= 1.615…
34/21 = 1.619…
55/34 = 1.617…

For successive Fibonacci numbers a, b , a/b is close to Φ
but not quite it Φ . 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

피보나치 수열

- f_n: n번째 피보나치 수

  $f_0 = 0, f_1 = 1$

  $f_n = f_{n-1} + f_{n-2}$, $n \geq 2$

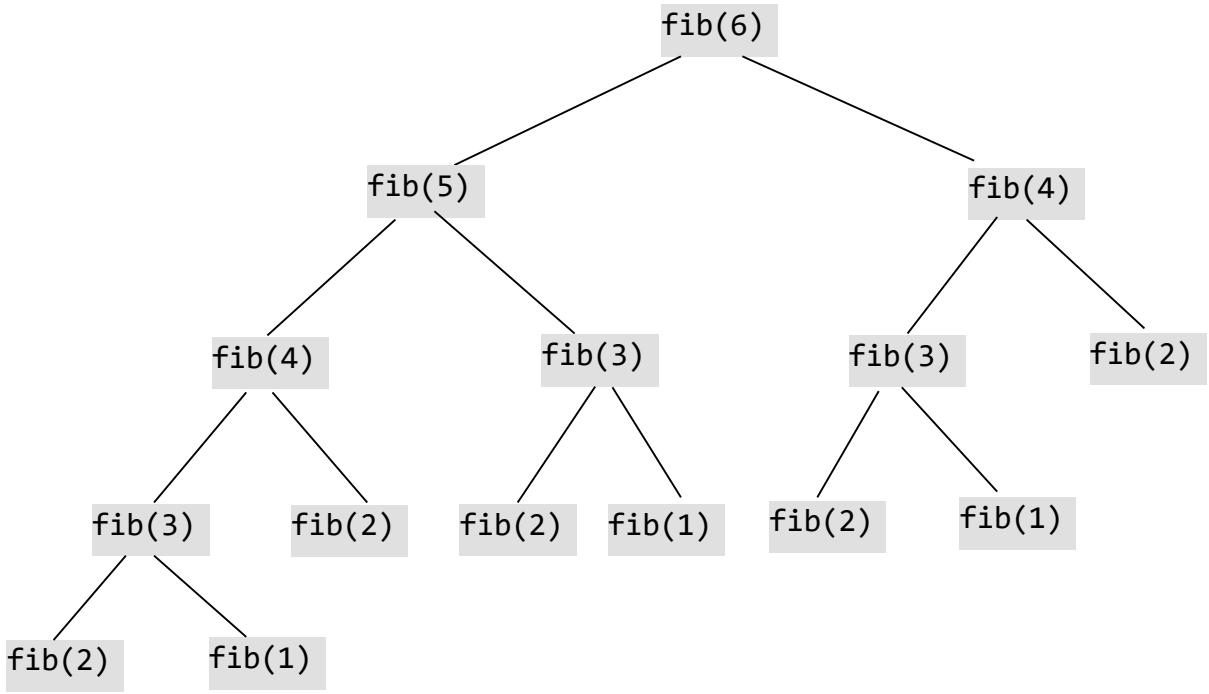
  $\lim_{n \to \infty} f_n / f_{n-1}$ = golden ratio = 1.61803398…

피보나치 수열

- f_n: n번째 피보나치 수

  $f_0 = 0, f_1 = 1$

  $f_n = f_{n-1} + f_{n-2}$, $n \geq 2$



- 방법 1 (비효율적 방법)

```
Algorithm fib(int n)
    if(n == 0 || n == 1)
        return n
    else
        return fib(n-1) + fib(n-2)
```

✓ 방법 1이 비효율적인 이유: 동일한 부분문제에 대한 해를 여러 번 (반복하여) 구한다.

✓ fib1(n-1)과 fib(n-2)가 overlap한다: 동일한 부분문제를 가진다.

✓ 시간복잡도: 수행시간 T(n) = T(n-1) + T(n-2) + c (상수)

$O(\Phi^n)$

$O(2^n)$

피보나치 수열 – 방법 1

- f_n: n번째 피보나치 수

  $f_0 = 0, f_1 = 1$

  $f_n = f_{n-1} + f_{n-2}$, $n \geq 2$

- 방법 1 (비효율적 방법)

```c
// C
int fib1(int n)
{
    if(n == 0 || n == 1)
        return n;
    else
        return fib1(n-1) + fib1(n-2);
}
```

```python
# Python
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)

def main():
    n = int(input())
    print(fib(n))

if __name__ == '__main__':
    main()
```

피보나치 수열 – 방법 2 (동적계획법: memorization)

- 동적계획법
  방법 **2** - Memoization

```
Algorithm initialize(n, lookupTable)
    for i = 0 to n
        lookupTable[i] = -1 // 초기화

Algorithm fib(n, lookupTable)
    if lookupTable[n] != -1
        return lookupTable[n]
    else
        if n <= 1
            lookupTable[n] = n
        else
            lookupTable[n] = fib(n-1, lookupTable) + fib(n-2, lookupTable)
        return lookupTable[n]
```

```
# Python
def initialize(n):
    lookupTable = [-1 for i in range(n+1)]
    return lookupTable

def fib(n, lookupTable):
    if lookupTable[n] != -1:
        return lookupTable[n]
    else:
        if n <= 1:
            lookupTable[n] = n
        else:
            lookupTable[n] = fib(n-1, lookupTable) + fib(n-2, lookupTable)
        return lookupTable[n]

def main():
    n = int(input())
    lookupTable = initialize(n)
    print(fib(n, lookupTable))

if __name__ == '__main__':
    main()
```

시간복잡도: O(n)

피보나치 수열 – 방법 2 (동적계획법: memorization)

- 동적계획법

  방법 **2** - Memoization

```
// C
#include <stdio.h>
#include <stdlib.h>
void initialize(int n, int *lookupTable)
{
    for (int i = 0; i <= n; i++)
        lookupTable[i] = -1;
}

int fib(int n, int *lookupTable)
{
    if (lookupTable[n] != -1)
        return lookupTable[n];
    else  {
        if (n <= 1)
            return n;
        else {
            lookupTable[n] = fib(n-1, lookupTable) + fib(n-2, lookupTable);
            return lookupTable[n];
      }
    }
}
```

```
int main()
{
    int n;
    int *lookupTable;
    scanf("%d", &n);
    lookupTable = (int*) malloc(sizeof(int)*(n+1));
    initialize(n, lookupTable);

    printf("%d\n", fib(n, lookupTable));
    return 0;
}
```

피보나치 수열 – 방법 3: bottom up

- 동적 계획법
  방법 3 - bottom-up

피보나치 수들을 저장하는 테이블 F을 만들고 F[0], F[1], F[2],..., F[n]을 순차적으로 구한다 (i번째 피보나치 수를 F[i]에 저장)

방법 2 (Python)

```
def fib(n):
  if n <= 1:
      return n
  F = [0 for i in range(n+1)]
  F[0] = 0
  F[1] = 1
  for i in range(2,n+1):
    F[i] = F[i - 1] + F[i - 2]
  return F[n]
```

시간복잡도: O(n)

방법 2 (C)

```
#include <stdlib.h>
int fib1( int n )
{
// int F[MAX];
  int *F = (int *)malloc(sizeof(int)*(n+1));
  F[0] = 0; F[1] = 1;
  for( int i = 2; i <= n; i++)
      F[i] = F[i - 1] + F[i - 2];
  return F[n];
}
```

피보나치 수열 – 방법 4: O(1) 메모리 공간 사용 반복을 이용

- 방법 4
  효율적인 메모리 공간 (추가적인 메모리 공간: O(1))
  반복을 이용

```
// C
int fib(int n)
{  int prev_prev, prev, current;
   int i,
   if (n <= 1)
      return 1;
   prev_prev = prev = 1;
   for(i = 2; i <= n; i++)
   {  current = prev_prev + prev;
      prev_prev = prev;
      prev = current;
   }
   return current;
}
```

prev_prev=0; prev=1;

prev_prev = F[i-2]
prev = F[i-1]
current = F[i]

```
# Python
def fib(n):
    if n == 0 or n == 1:
        return n
    prev_prev = 0
    prev = 1

    for i in range(2,n+1):
        current = prev_prev + prev
        prev_prev = prev
        prev = current
    '''
    prev = 0
    current = 1
    for i in range(2,n+1):
        current, prev= current + prev, current
    '''
    return current

def main():
    n = int(input())
    print(fib(n))

if __name__ == '__main__':
    main()
```

시간복잡도: O(n)

2. 이항계수 (Binomial Coefficient)

- 정의 1: ${}_nC_k$: n개 원소에서 k개를 선택하는 경우의 수

  ${}_nC_k = n! / (k!\,(n-k)!)$

  $\binom{n}{k}$

- 정의 2:

  n개 원소 번호: 1, 2, 3, 4, …, n

  n개 원소에서 k개를 선택하는 경우의 수: ${}_nC_k$, $\binom{n}{k}$

  원소 n을 선택하는 경우들과 선택하지 않는 경우들로 나눌 수 있다.

  예:

  n = 4, k = 3

  {1, 2, 4}, {1, 3, 4}, {2, 3, 4} : 4를 선택하는 경우들

  {1, 2, 3} : 4를 선택하지 않는 경우들

이항계수

- 정의 2:

  n개 원소 번호: 1, 2, 3, 4, …, n

  n개 원소에서 k개를 선택하는 경우의 수: ${}_nC_k$

  원소 n을 선택하는 경우들과 선택하지 않는 경우들로 나눌 수 있다

  예:

  n = 5, k = 3일 때

  n(=5)을 선택하는 경우들: ${}_{n-1}C_{k-1}$

  {1, 2, 5}, {1, 3, 5}, {1, 4, 5}, {2, 3, 5}, {2, 4, 5}, {3, 4, 5} :

  n(=5)를 선택하지 않는 경우들: ${}_{n-1}C_k$

  {1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4}

이항계수

- 정의 2 (재귀적 정의):

$$
{}_nC_k = \begin{cases} {}_{n-1}C_{k-1} + {}_{n-1}C_k & ,0 < k < n \\ 1 & ,k = 0 \text{ 혹은 } k = n \end{cases}
$$

✓방법 1 (비효율적 방법)

```
int C(int n, int k)
{
  if (k == 0 || n == k)
    return 1;
  else
    return C(n-1, k-1) + C(n-1, k);
}
```

```
def C(n,k):
    if k == 0 or k == n:
        return 1
    else:
        return C(n-1,k-1)+C(n-1,k)
```

이항계수 (계속)

✓ 방법 2 (효율적 방법) - **동적 계획법**을 이용한 방법

$_iC_j$를 배열 원소 C[i][j]에 저장

C[i][j] = 1 , j = 0 혹은 j = i

= C[i-1][j-1] + C[i-1] [j] , 0 < j < I

```
// C++
// const int MAX = 10;
int dpBinomial(int n, int k)
{  int i, j;
   int **C = new int*[n+1];  //   int C[MAX][MAX];
   for(i = 0; i <= n; i++)
      C[i] = new int[n+1];
   for(i = 0; i <= n; i++)
      for(j = 0; j <= min(i, k); j++)
         if(j == 0 || j == i)
            C[i][j] = 1;
         else
            C[i][j] = C[i-1][j-1] + C[i-1][j];
   for(i = 0; i <= n; i++)
      delete [] C[i];
   int result = C[n][k];
   delete [] C;
   return rsult];
}
```

```
# Python
def dpBinomial(n,k):
    C = [[0 for j in range(n+1)] for i in range(n+1)]
    for i in range(1,n+1):
        for j in range(min(i,k)+1):
            if j == 0 or j == i:
                C[i][j] = 1
            else:
                C[i][j] = C[i-1][j-1]+C[i-1][j]
    return C[n][k]

def main():
    n, k = input().split()
    n, k = int(n), int(k)
    print(dpBinomial(n,k))

if __name__ == '__main__':
    main()
```

시간복잡도: O(nk)

동적계획법

- 동적계획법은 주로 최적화 (optimization) 문제를 해결하는 데 주로 사용한다.
  최적화 문제: 목적함수 값을 최대 혹은 최소로 하는 해를 구하는 문제

- 동적계획법은 다음의 최적화 원칙이 적용되는 문제의 해를 구하는데 사용된다.

  - 최적화 원칙(Principle of Optimality)

    The principle of optimality applies if the optimal solution to a problem always contains optimal solutions to all subproblems

    문제의 최적해는 모든 부분 문제의 최적해를 포함한다

    혹은

    Whatever the initial state is, the remaining decisions must be optimal with regard the state following from the first decision

    문제의 최적해를 구하는 일련의 선택들을 D_1, D_2, …, D_{n-1}, D_n라 하자. 첫 번째 선택 D_1을 한 후의 상태에서 D_2, …, D_{n-1}, D_n는 최적해가 된다.

동적계획법 (계속)

- 동적계획법은 주로 최적화 (optimization) 문제를 해결하는 데 주로 사용한다. 동적계획법을 이용한 문제 해결은 아래의 4 단계를 거친다.

단계 1. 최적 해의 구조를 파악한다.

단계 2. 최적 해의 (목적함수) 값을 재귀적으로 (recursively) 정의한다.

단계 3. 단계 2의 재귀적 정의로부터 최적 해의 (목적함)수 값을 bottom-up(혹은 memoization)으로 구하면서 테이블에 저장한다.

(bottom-up: 작은 부분문제에서 시작하여 큰 부분문제들까지 최적 해의 목적함수 값을 차례대로 구한다)

단계 4. 단계 3에서 구한 정보를 이용하여 최적 해를 찾는다.

※ 단계 1-3은 동적계획법으로 해를 구할 때 필수적인 과정임

단계 4는 최적 해의 목적함수 값만 구하는 경우는 필요 없고, 최적 해를 찾고자 하는 경우에 필요.