이진탐색 (Binary Search)

- 정렬되어 있는 리스트에서 탐색 범위를 절반씩 좁혀가며 데이터를 탐색하는 방법, 큰 탐색 범위를 가질 시 적용
  시작점, 중간점, 끝점을 이용하여 탐색범위 설정

ex) 이미 정렬된 10개의 데이터 중에서 값이 4인 원소를 찾는 예

Step1. 중간점[4]의 값과 찾을 데이터를 비교했을 때 더 작으므로 중간점 이후의 데이터는 탐색 필요X

0 2 4 6 8 10 12 14 16 18

↑ 시작점[0] ↑ 중간점[4] ↑ 끝점[9]

Step2. 끝점을 중간점의 왼쪽으로 옮긴다. 탐색 범위가 총 4개의 데이터가 되고, 찾을 값이 중간점보다 큰 것을 확인

0 2 4 6 8 10 12 14 16 18

↑ 시작점[0] ↑ 중간점[1] ↑ 끝점[3]

Step3. 시작점 위치를 중간점 오른쪽으로 옮겨준다. 중간점 위치의 값이 찾는 값과 일치하므로 종료한다.

0 2 4 6 8 10 12 14 16 18

↑ 시작점 중간점 [2] ↑ 끝점[3]

1) 이진탐색의 시간복잡도

- 단계마다 탐색 범위를 2로 나누는 것과 동일하므로 연산 횟수는 $\log_2 N$에 비례
- 이진 탐색은 탐색 범위를 절반씩 줄이며, 시간 복잡도는 $O(\log N)$을 보장

2) 소스코드 : 재귀적 표현

```
def binary_search(array, target, start, end):
    if start > end:
        return None
    mid = (start + end) // 2
    # 찾은 경우 중간점 인덱스 반환
    if array[mid] == target:
        return mid
    elif array[mid] > target:
        return binary_search(array, target, start, mid - 1)
    else:
        return binary_search(array, target, mid + 1, end)

# n(원소의 개수)과 target(찾고자 하는 값)을 입력 받기
target = list(map(int, input().split()))
# 전체 원소 입력 받기
array = list(map(int, input().split()))

# 이진 탐색 수행 결과 출력
result = binary_search(array, target, 0, n - 1)
if result == None:
    print("원소가 존재하지 않습니다.")
else:
    print(result + 1)
```

Input

10 7
1 3 5 7 9 11 13 15 17 19

Output

4

Input

10 7
1 3 5 6 9 11 13 15 17 19

Output

원소가 존재하지 않습니다.

3) 소스코드 : 반복문 구현

```
def binary_search(array, target, start, end):
    while start <= end:  시작점이 끝점보다 커지면(찾지 못하면) 종료
        mid = (start + end) // 2
        # 찾은 경우 중간점 인덱스 반환
        if array[mid] == target:
            return mid
        # 중간점의 값보다 찾고자 하는 값이 작은 경우 왼쪽 확인
        elif array[mid] > target:
            end = mid - 1
        # 중간점의 값보다 찾고자 하는 값이 큰 경우 오른쪽 확인
        else:
            start = mid + 1
    return None

# n(원소의 개수)과 target(찾고자 하는 값)을 입력 받기
n, target = list(map(int, input().split()))
# 전체 원소 입력 받기
array = list(map(int, input().split()))

# 이진 탐색 수행 결과 출력
result = binary_search(array, target, 0, n - 1)
if result == None:
    print("원소가 존재하지 않습니다.")
else:
    print(result + 1)
```

input
10 7
1 3 5 7 9 11 13 15 17 19

output
4

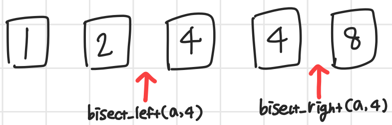
input
10 7
1 3 5 6 9 11 13 15 17 19

output
원소가 존재하지 않습니다.

4) 파이썬 이진탐색 라이브러리

from bisect import bisect_left, bisect_right

- bisect_left(a, x): 정렬된 순서를 유지하면서 배열 a에 x를 삽입할 가장 왼쪽 인덱스 반환
- bisect_right(a, x): 정렬된 순서를 유지하면서 배열 a에 x를 삽입할 가장 오른쪽 인덱스 반환
- 찾을 값이 배열에 없다면 0을 반환




```
from bisect import bisect_left, bisect_right

a = [1, 2, 4, 4, 8]
x = 4

print(bisect_left(a, x))
print(bisect_right(a, x))
```

output
2
4

4-1) 값이 특정 범위에 속하는 데이터 수 구하기

```
from bisect import bisect_left, bisect_right

# 값이 [left_value, right_value]인 데이터의 개수를 반환하는 함수
def count_by_range(a, left_value, right_value):
    right_index = bisect_right(a, right_value)
    left_index = bisect_left(a, left_value)
    return right_index - left_index

# 배열 선언
a = [1, 2, 3, 3, 3, 3, 4, 4, 8, 9]

# 값이 4인 데이터 개수 출력
print(count_by_range(a, 4, 4))

# 값이 [-1, 3] 범위에 있는 데이터 개수 출력
print(count_by_range(a, -1, 3))
```

output
2
6

5) 파라메트릭 서치 (Parametric Search)

- 최적화 문제를 결정 문제 ('예' 혹은 '아니오')로 바꾸어 해결하는 기법
  ex) 특정한 조건을 만족하는 가장 알맞은 값을 빠르게 찾는 최적화 문제
- 일반적으로 코딩 테스트에서 파라메트릭 서치는 이진탐색으로 해결

ex) 백준 2805번 나무 자르기
  백준 1654번 랜선 자르기

6) 최장 증가 부분 수열 (LIS) 알고리즘

- 예를 들어, [10, 20, 10, 30, 20, 50] 이라는 수열이 있을 때 가장 긴 증가하는 부분 수열은 [10, 20, 10, 30, 20, 50] 이다.
- 증가하는 부분 수열 중 가장 긴 것
- 가장 일반적인 방법은 **DP**를 이용하는 것이지만, $O(N^2)$의 시간복잡도를 가짐
- 이분탐색을 통해 시간복잡도 개선 → $O(N\log N)$

```
memorization = [0]
arr = [0] + 원래 배열

for case in cases:
    if memorization[-1] < case:          # memorization에 저장된 이전 값이 현재 선택된 값보다 작으면
        memorization.append(case)        # memorization에 값 추가
    else:                                # 크면 memorization에서 이분탐색 시작
        left = 0
        right = len(memorization)

        while left <= right:
            mid = (left + right) // 2

            if memorization[mid] < case:   # 만약 중간값이 선택된 값보다 작으면
                left = mid + 1             # 범위를 오른쪽으로 좁힌다
            else:
                right = mid                # 크거나 같으면 right를 중간값으로 설정한다
        memorization[right] = case         # memorization[right]를 case로 설정한다
```

DP (Dynamic Programming, 동적 계획법)

- 메모리를 적절히 사용하여 수행 시간 효율성을 비약적으로 향상시키는 방법
- 이미 계산된 결과(작은 문제)는 별도의 메모리 영역에 저장하여 다시 계산하지 않도록 한다
  - 한 번 해결한 문제는 다시 해결X인 점에서 완전 탐색을 이용했을 때 매우 비효율적인 시간 복잡도를 가지는 문제라 해도 DP를 이용하면 시간 복잡도를 획기적으로 줄일 수O
- 다이나믹 프로그래밍의 구현은 일반적으로 탑다운(top-down)과 보텀업(bottom-up)으로 구성

  ↳ 하향식 ↳ 상향식

1) 다이나믹 프로그래밍의 조건

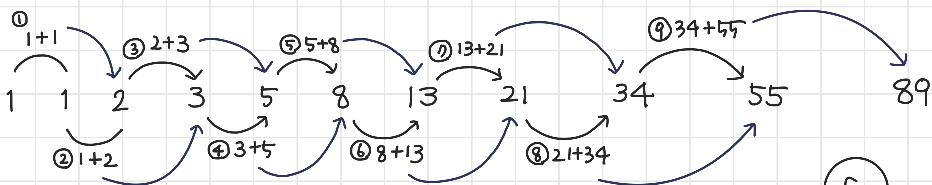
① 최적 부분 구조 (Optimal Substructure)
- 큰 문제를 작은 문제로 나눌 수 있으며 작은 문제의 답을 모아서 큰 문제를 해결할 수 있다.

② 중복되는 부분문제 (Overlapping Subproblem)
- 동일한 작은 문제를 반복적으로 해결해야 한다.

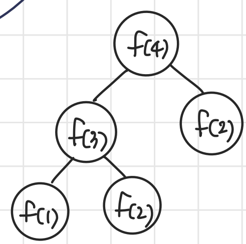
2) 문제 1 - 피보나치 수열

- DP를 이용해 풀 수 있는 대표적인 문제
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ⋯
- 피보나치 수열을 점화식으로 표현하면 * 점화식: 인접한 항들 사이의 관계식

$$a_n = a_{n-1} + a_{n-2},\ a_1 = 1,\ a_2 = 1$$



- 프로그래밍에서는 이러한 수열을 배열이나 리스트를 통해 표현
- n번째 피보나치 수를 $f(n)$라고 할 때 4번째 피보나치 수 $f(4)$를 구하는 과정



2-1) 단순 재귀 소스코드

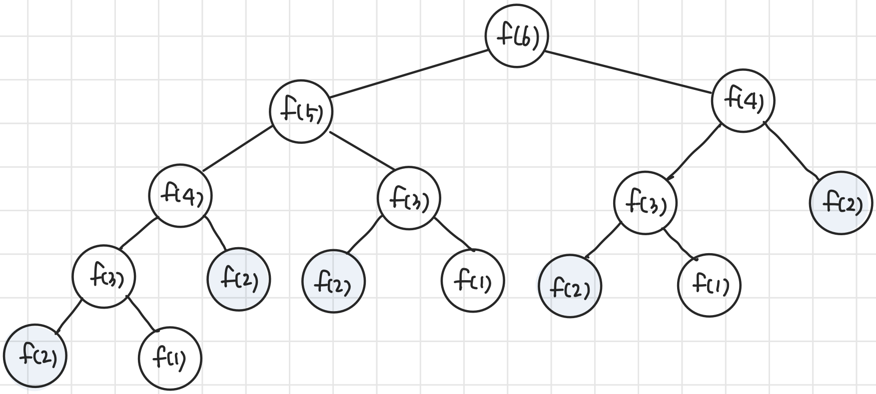
```
def fibo(x):
    if x == 1 or x == 2:
        return 1
    return fibo(x - 1) + fibo(x - 2)

print(fibo(4))
```

Output

3

2-2) 피보나치 수열의 시간 복잡도 분석

- 단순 재귀 함수로 피보나치 수열을 해결하면 지수 시간 복잡도를 가지게 됨



- 다음과 같이 f(2)가 여러 번 호출되며 중복되는 부분 문제 발생
- 피보나치수열의 시간복잡도는
  세타 표기법: $\theta(1.618\cdots^{N})$
  빅오 표기법: $O(2^{N})$
- 빅오 표기법 기준 f(30)을 계산하기 위해 약 10억 가량의 연산을 수행해야 함

2-3) 효율적인 해법: 다이나믹 프로그래밍

- 다이나믹 프로그래밍의 사용 조건을 만족하는지 확인
  ① 최적 부분 구조: 큰 문제를 작은 문제로 나눌 수 있다 ✔
  ② 중복되는 부분 문제: 동일한 작은 문제를 반복적으로 해결 ✔

3) 메모이제이션(Memoization)

- 다이나믹 프로그래밍을 구현하는 방법 중 하나
- 한 번 계산한 결과를 메모리 공간에 메모
  - 같은 문제를 다시 호출하면 메모했던 결과를 그대로 가져옴
  - 값을 기록해놓는다는 점에서 캐싱(Caching)이라고도 함
- 하향식 방법(top-down)에서 사용

4) 탑다운 VS 보텀업

- 탑다운(메모이제이션) 방식은 하향식이라고도 하며 보텀업 방식은 상향식이라고 한다 (반복문 사용)
- 다이나믹 프로그래밍의 전형적 형태는 보텀업
  - 결과 저장용 리스트는 DP 테이블이라고 부름
- 엄밀히 말하면 메모이제이션은 이전에 계산된 결과를 일시적으로 기록해놓는 넓은 개념을 의미
  - 따라서 메모이제이션은 DP에 국한된 개념X
  - 한번 계산된 결과를 담아놓기만 하고 DP를 위해 쓰지 않을 수도

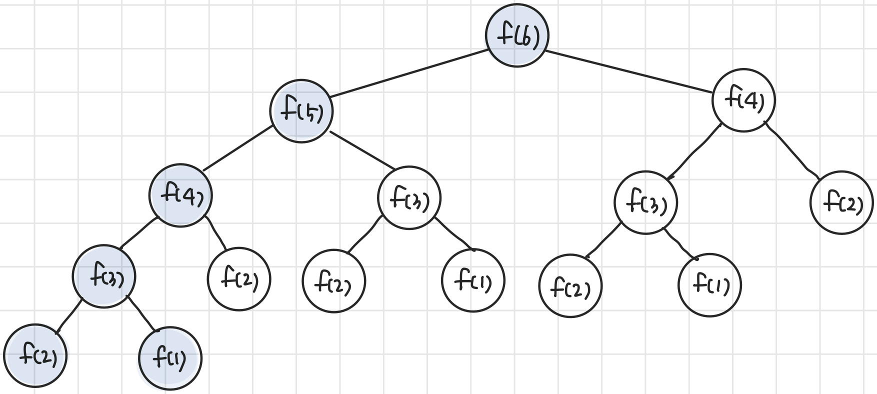
4-1) 피보나치 수열 : 탑다운 다이나믹 프로그래밍 소스코드

```
# 한 번 계산된 결과를 메모이제이션(Memoization)하기 위한 리스트 초기화
d = [0] * 100

# 피보나치 함수(Fibonacci Function)를 재귀함수로 구현(탑다운 다이나믹 프로그래밍)
def fibo(x):
    # 종료 조건(1 혹은 2일 때 1을 반환)
    if x == 1 or x == 2:
        return 1
    # 이미 계산한 적 있는 문제라면 그대로 반환
    if d[x] != 0:
        return d[x]
    # 아직 계산하지 않은 문제라면 점화식에 따라서 피보나치 결과 반환
    d[x] = fibo(x - 1) + fibo(x - 2) 리스트에 기록
    return. d[x]

print(fibo(99))
```

4-2) 피보나치 수열 : 보텀업 다이나믹 프로그래밍 소스코드

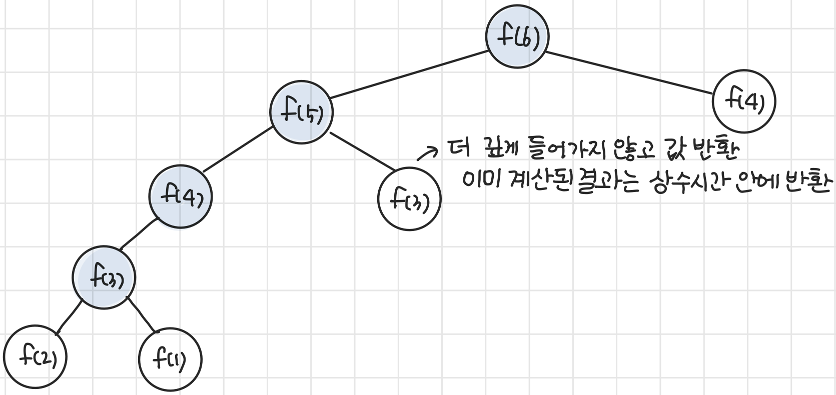
```
# 앞서 계산된 결과를 저장하기 위한 DP 테이블 초기화
d = [0] * 100

# 첫 번째 피보나치 수와 두 번째 피보나치 수는 1
d[1] = 1
d[2] = 1
n = 99

# 피보나치 함수(Fibonacci Function) 반복문으로 구현(보텀업 다이나믹 프로그래밍)
for i in range(3, n + 1):
    d[i] = d[i - 1] + d[i - 2]

print(d[n])
```

4-3) 피보나치 수열 : 메모이제이션 동작 분석



- 이미 계산된결과를 메모리에 저장하면 다음과 같이 색칠된 노드만 처리할 것을 기대

- 실제 호출되는 함수에 대해서만 확인해보면 다음과같이 방문

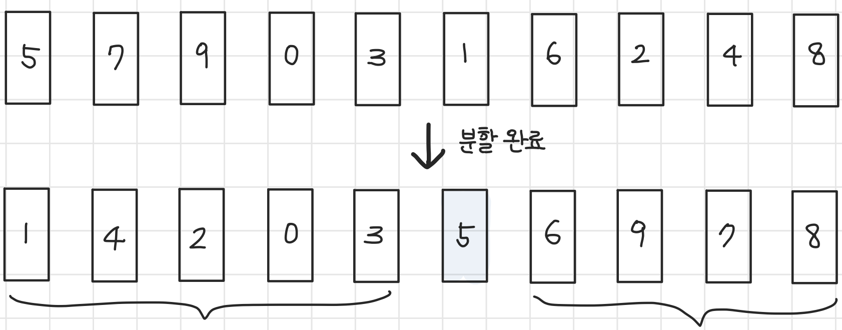


- 메모이제이션을 이용하는 경우 피보나치 수열 함수의 시간 복잡도는 $O(N)$

5) 다이나믹 프로그래밍 VS 분할 정복

- 다이나믹 프로그래밍과 분할 정복은 모두 최적 부분 구조를 가질 때 사용
  - 큰 문제를 작은 문제로 나눌 수 있으며 작은 문제의 답을 모아서 큰 문제를 해결할 수 있는 상황
- 다이나믹 프로그래밍과 분할 정복의 차이점은 부분 문제의 중복
  - 다이나믹 프로그래밍 문제에서는 각 부분문제들이 서로 영향을 미치며 부분 문제가 중복
  - 분할 정복 문제에서는 동일한 부분 문제가 반복적으로 계산X

ex) 분할 정복의 대표적 예시인 퀵정렬



- 한 번 기준 원소(Pivot)가 자리를 변경해서 자리를 잡으면 그 기준 원소의 위치 변경X
- 분할 이후 해당 피벗 다시 처리 호출X

6) 다이나믹 프로그래밍 문제 접근 방법

- 주어진 문제가 다이나믹 프로그래밍 유형임을 파악하는 것이 중요
- 가장 먼저 그리디, 구현, 완전 탐색 등의 아이디어로 문제를 해결할 수 있는지 검토
  - 다른 알고리즘 풀이 방법이 떠오르지 x 면 DP 검토
- 일단 재귀 함수로 비효율적인 완전탐색 프로그램을 작성한 뒤에 (탑다운) 최적 부분 구조인지 확인 후 코드 개선
- 일반적인 코딩 테스트에서는 기본 유형의 DP 출제

〈문제〉 개미 전사: 문제 설명

- 개미 전사는 부족한 식량을 충당하고자 메뚜기 마을의 식량 창고를 몰래 공격하려고 합니다. 메뚜기 마을에는 여러 개의 식량 창고가 있는데 식량 창고는 일직선으로 이어져 있습니다.
- 각 식량창고에는 정해진 수의 식량을 저장하고 있으며 개미 전사는 식량 창고를 선택적으로 약탈하여 식량을 빼앗을 예정입니다. 이때 메뚜기 정찰병들은 일직선으로 존재하는 식량 창고 중에서 서로 인접한 식량 창고가 공격받으면 바로 알아챌 수 있습니다.
- 따라서 개미 전사가 정찰병에게 들키지 않고 식량창고를 약탈하기 위해서는 최소한 한 칸 이상 떨어진 식량 창고를 약탈해야 합니다.

ex)

창고0	창고1	창고2	창고3
1	3	1	5

창고 0을 골랐을 때 → 창고 2, 3 중 약탈
창고 1을 골랐을 때 → 창고 3 약탈

- 예를 들어 식량 창고 4개가 다음과 같이 존재한다고 가정했을 때,

  (1, 3, 1, 5)

- 개미 전사는 두 번째 식량 창고와 네 번째 식량 창고를 선택했을 때 최댓값인 총 8개의 식량을 빼앗을 수 있습니다. 개미 전사는 식량 창고가 이렇게 일직선 상일 때 최대한 많은 식량을 얻길 원합니다.
- 개미 전사를 위해 식량창고 N개에 대한 정보가 주어졌을 때 얻을 수 있는 식량의 최댓값을 구하는 프로그램을 작성하세요.

문제 조건

입력조건 1) 첫째 줄에 식량창고의 개수 N이 주어집니다. ($3 <= N <= 100$)
2) 둘째 줄에 공백을 기준으로 각 식량창고에 저장된 식량의 개수 K가 주어집니다. ($0 <= K <= 1{,}000$)

출력조건 1) 첫째 줄에 개미 전사가 얻을 수 있는 식량의 최댓값을 출력하세요.

input

```
4
1 3 1 5
```

output

```
8
```

a_i = i 번째 식량창고까지의 최적의 해 (얻을 수 있는 식량의 최댓값)
k_i = i 번째 식량창고에 있는 식량의 양

$$a_i = max(a_{i-1}, a_{i-2} + k_i)$$

한 칸 이상 떨어진 식량창고는 항상 털 수 있으므로 $(i-3)$ 번째 이하는 고려할 필요X

```
# 정수 N을 입력 받기
n = int(input())
# 모든 식량 정보 입력 받기
array = list(map(int, input().split()))

# 앞서 계산된 결과를 저장하기 위한 DP 테이블 초기화
d[0] = array[0]
d[1] = max(array[0], array[1])
for i in range(2, n):
    d[i] = max(d[i - 1], d[i.- 2] + array[i])

# 계산된 결과 출력
print(d[n - 1])
```