다이나믹 프로그래밍

- 다이나믹 프로그래밍은 **메모리를 적절히 사용하여 수행 시간 효율성을 비약적으로 향상시키는 방법**입니다.
- 이미 계산된 결과(작은 문제)는 별도의 메모리 영역에 저장하여 다시 계산하지 않도록 합니다.
- 다이나믹 프로그래밍의 구현은 일반적으로 두 가지 방식(탑다운과 보텀업)으로 구성됩니다.

다이나믹 프로그래밍

- 다이나믹 프로그래밍은 **동적 계획법**이라고도 부릅니다.
- 일반적인 프로그래밍 분야에서의 동적(Dynamic)이란 어떤 의미를 가질까요?
  - 자료구조에서 동적 할당(Dynamic Allocation)은 '**프로그램이 실행되는 도중에 실행에 필요한 메모리를 할당하는 기법**'을 의미합니다.
  - 반면에 다이나믹 프로그래밍에서 '다이나믹'은 **별다른 의미 없이 사용된 단어**입니다.

다이나믹 프로그래밍의 조건

- 다이나믹 프로그래밍은 문제가 다음의 조건을 만족할 때 사용할 수 있습니다.
  1. **최적 부분 구조 (Optimal Substructure)**
     - 큰 문제를 작은 문제로 나눌 수 있으며 작은 문제의 답을 모아서 큰 문제를 해결할 수 있습니다.
  2. **중복되는 부분 문제 (Overlapping Subproblem)**
     - 동일한 작은 문제를 반복적으로 해결해야 합니다.

피보나치 수열

- 피보나치 수열 다음과 같은 형태의 수열이며, 다이나믹 프로그래밍으로 효과적으로 계산할 수 있습니다.

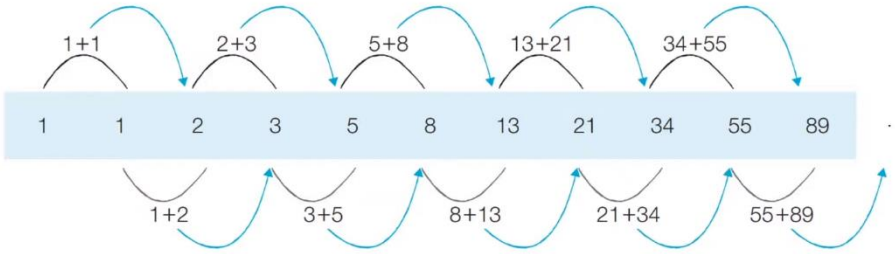
$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

- **점화식**이란 인접한 항들 사이의 관계식을 의미합니다.
- 피보나치 수열을 점화식으로 표현하면 다음과 같습니다.

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

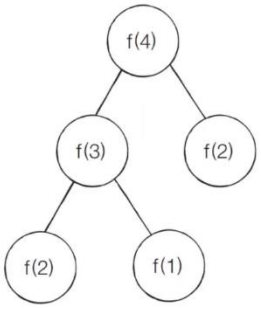
피보나치 수열

- 피보나치 수열이 계산되는 과정은 다음과 같이 표현할 수 있습니다.
  - 프로그래밍에서는 이러한 수열을 배열이나 리스트를 이용해 표현합니다.



피보나치 수열

- 피보나치 수열이 계산되는 과정은 다음과 같이 표현할 수 있습니다.
  - n번째 피보나치 수를 $f(n)$라고 할 때 4번째 피보나치 수 $f(4)$를 구하는 과정은 다음과 같습니다.



피보나치 수열: 단순 재귀 소스코드 (Python)

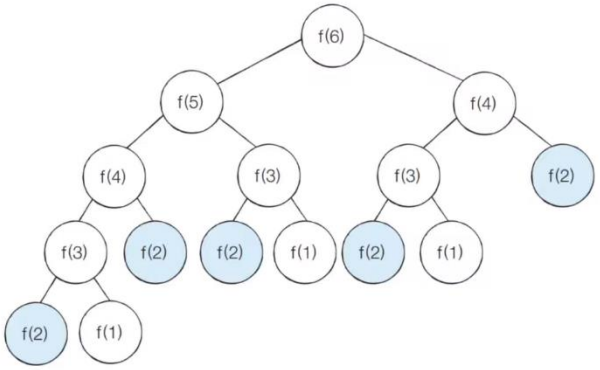
```python
# 피보나치 함수(Fibonacci Function)을 재귀함수로 구현
def fibo(x):
    if x == 1 or x == 2:
        return 1
    return fibo(x - 1) + fibo(x - 2)

print(fibo(4))
```

실행 결과

```
3
```

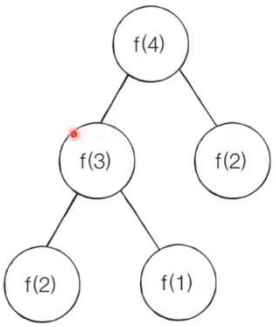

피보나치 수열의 시간 복잡도 분석

- 단순 재귀 함수로 피보나치 수열을 해결하면 지수 시간 복잡도를 가지게 됩니다.
- 다음과 같이 $f(2)$가 여러 번 호출되는 것을 확인할 수 있습니다. (**중복되는 부분 문제**)



피보나치 수열의 효율적인 해법: 다이나믹 프로그래밍

- 다이나믹 프로그래밍의 사용 조건을 만족하는지 확인합니다.
  1. **최적 부분 구조**: 큰 문제를 작은 문제로 나눌 수 있습니다.
  2. **중복되는 부분 문제**: 동일한 작은 문제를 반복적으로 해결합니다.
- 피보나치 수열은 다이나믹 프로그래밍의 사용 조건을 만족합니다.



메모이제이션 (Memoization)

- 메모이제이션은 다이나믹 프로그래밍을 구현하는 방법 중 하나입니다.
- 한 번 계산한 결과를 메모리 공간에 메모하는 기법입니다.
  - 같은 문제를 다시 호출하면 메모했던 결과를 그대로 가져옵니다.
  - 값을 기록해 놓는다는 점에서 **캐싱(Caching)**이라고도 합니다.

탑다운 VS 보텀업

- 탑다운(메모이제이션) 방식은 **하향식**이라고도 하며 보텀업 방식은 **상향식**이라고도 합니다.
- 다이나믹 프로그래밍의 전형적인 형태는 보텀업 방식입니다.
  - 결과 저장용 리스트는 **DP 테이블**이라고 부릅니다.
- 엄밀히 말하면 메모이제이션은 이전에 계산된 결과를 일시적으로 기록해 놓는 넓은 개념을 의미합니다.
  - 따라서 메모이제이션은 다이나믹 프로그래밍에 국한된 개념은 아닙니다.
  - 한 번 계산된 결과를 담아 놓기만 하고 다이나믹 프로그래밍을 위해 활용하지 않을 수도 있습니다.

피보나치 수열: 탑다운 다이나믹 프로그래밍 소스코드 (Python)

```python
# 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)
    return d[x]

print(fibo(99))
```

실행 결과

```
218922995834555169026
```

피보나치 수열: 보텀업 다이나믹 프로그래밍 소스코드 (Python)

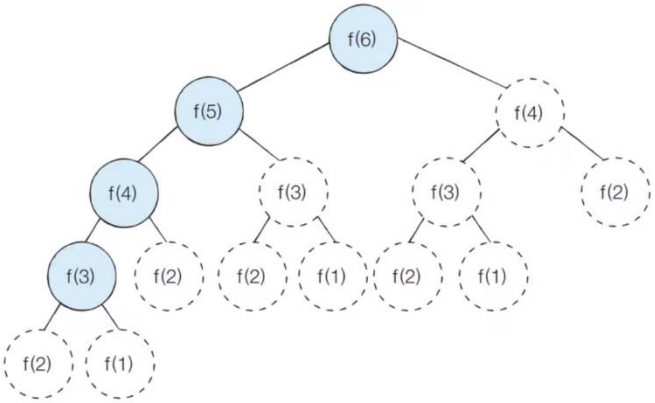
```python
# 앞서 계산된 결과를 저장하기 위한 DP 테이블 초기화
d = [0] * 100

# 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])
```

실행 결과

```
218922995834555169026
```

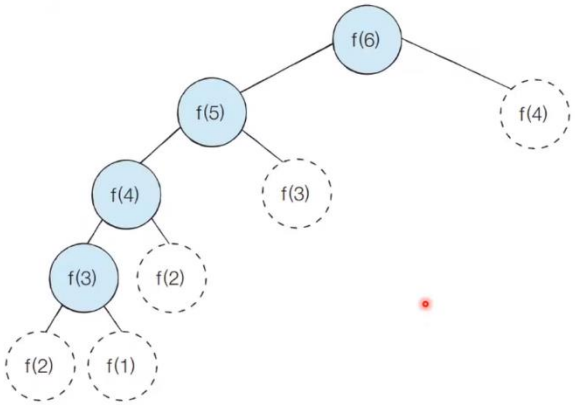

피보나치 수열: 메모이제이션 동작 분석

- 이미 계산된 결과를 메모리에 저장하면 다음과 같이 색칠된 노드만 처리할 것을 기대할 수 있습니다.



피보나치 수열: 메모이제이션 동작 분석

- 실제로 호출되는 함수에 대해서만 확인해 보면 다음과 같이 방문합니다.



피보나치 수열: 메모이제이션 동작 분석

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

```
d = [0] * 100

def fibo(x):
    print('f(' + str(x) + ')', end=' ')
    if x == 1 or x == 2:
        return 1
    if d[x] != 0:
        return d[x]
    d[x] = fibo(x - 1) + fibo(x - 2)
    return d[x]

fibo(6)
```

실행 결과

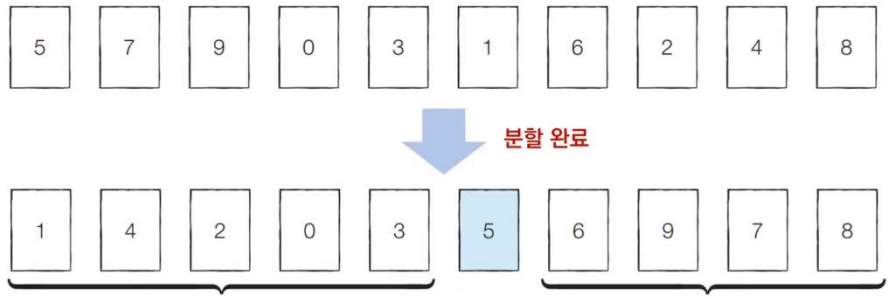
```
f(6) f(5) f(4) f(3) f(2) f(1) f(2) f(3) f(4)
```

다이나믹 프로그래밍 VS 분할 정복

- 다이나믹 프로그래밍과 분할 정복은 모두 **최적 부분 구조**를 가질 때 사용할 수 있습니다.
  - 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**입니다.
  - 다이나믹 프로그래밍 문제에서는 각 부분 문제들이 서로 영향을 미치며 부분 문제가 중복됩니다.
  - 분할 정복 문제에서는 동일한 부분 문제가 반복적으로 계산되지 않습니다.

다이나믹 프로그래밍 VS 분할 정복

- **분할 정복**의 대표적인 예시인 퀵 정렬을 살펴봅시다.
  - 한 번 기준 원소(Pivot)가 자리를 변경해서 자리를 잡으면 그 기준 원소의 위치는 바뀌지 않습니다.
  - 분할 이후에 해당 피벗을 다시 처리하는 부분 문제는 호출하지 않습니다.



다이나믹 프로그래밍 문제에 접근하는 방법

- 주어진 문제가 **다이나믹 프로그래밍 유형임을 파악**하는 것이 중요합니다.
- 가장 먼저 그리디, 구현, 완전 탐색 등의 아이디어로 문제를 해결할 수 있는지 검토할 수 있습니다.
  - 다른 알고리즘으로 풀이 방법이 떠오르지 않으면 다이나믹 프로그래밍을 고려해 봅시다.
- 일단 재귀 함수로 비효율적인 완전 탐색 프로그램을 작성한 뒤에 (탑다운) 작은 문제에서 구한 답이 큰 문제에서 그대로 사용될 수 있으면, 코드를 개선하는 방법을 사용할 수 있습니다.
- 일반적인 코딩 테스트 수준에서는 기본 유형의 다이나믹 프로그래밍 문제가 출제되는 경우가 많습니다.

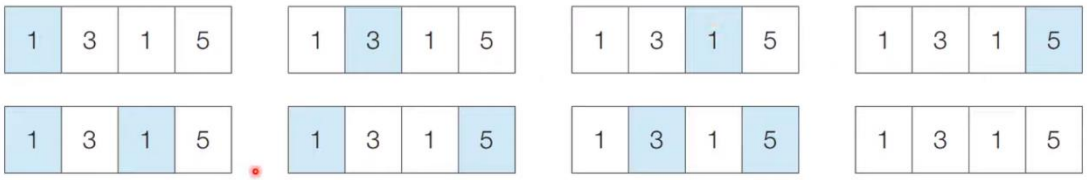
〈문제〉 개미 전사: 문제 설명

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

예시)

〈문제〉 개미 전사: 문제 해결 아이디어

- 예시를 확인해 봅시다. **N = 4**일 때, 다음과 같은 경우들이 존재할 수 있습니다.
  - 식량을 선택할 수 있는 경우의 수는 다음과 같이 8가지입니다.
  - 7번째 경우에서 8만큼의 식량을 얻을 수 있으므로 **최적의 해는 8**입니다.



〈문제〉 개미 전사: 답안 예시 (Python)

```python
# 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] * 100

# 다이나믹 프로그래밍(Dynamic Programming) 진행 (보텀업)
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])
```

〈문제〉 1로 만들기: 문제 설명

- 정수 X가 주어졌을 때, 정수 X에 사용할 수 있는 연산은 다음과 같이 **4가지**입니다.
  1. X가 5로 나누어 떨어지면, 5로 나눕니다.
  2. X가 3으로 나누어 떨어지면, 3으로 나눕니다.
  3. X가 2로 나누어 떨어지면, 2로 나눕니다.
  4. X에서 1을 뺍니다.
- 정수 X가 주어졌을 때, 연산 4개를 적절히 사용해서 값을 1로 만들고자 합니다. 연산을 사용하는 횟수의 최솟값을 출력하세요. 예를 들어 정수가 26이면 다음과 같이 계산해서 3번의 연산이 최솟값입니다.
  - 26 → 25 → 5 → 1

〈문제〉 1로 만들기: 문제 조건

난이도 ●◐○ | **풀이 시간** 20분 | **시간제한** 1초 | **메모리 제한** 128MB

입력 조건 • 첫째 줄에 정수 X가 주어집니다. (1 <= X <= 30,000)

출력 조건 • 첫째 줄에 연산을 하는 횟수의 최솟값을 출력합니다.

입력 예시

```
26
```

출력 예시

```
3
```

〈문제〉 1로 만들기: 문제 해결 아이디어

- $a_i = i$를 1로 만들기 위한 최소 연산 횟수
- 점화식은 다음과 같습니다.

$$a_i = min(a_{i-1}, a_{i/2}, a_{i/3}, a_{i/5}) + 1$$

- 단, 1을 빼는 연산을 제외하고는 해당 수로 나누어떨어질 때에 한해 점화식을 적용할 수 있습니다.

〈문제〉 1로 만들기: 답안 예시 (Python)

```python
# 정수 X를 입력 받기
x = int(input())

# 앞서 계산된 결과를 저장하기 위한 DP 테이블 초기화
d = [0] * 30001

# 다이나믹 프로그래밍(Dynamic Programming) 진행 (보텀업)
for i in range(2, x + 1):
    # 현재의 수에서 1을 빼는 경우
    d[i] = d[i - 1] + 1
    # 현재의 수가 2로 나누어 떨어지는 경우
    if i % 2 == 0:
        d[i] = min(d[i], d[i // 2] + 1)
    # 현재의 수가 3으로 나누어 떨어지는 경우
    if i % 3 == 0:
        d[i] = min(d[i], d[i // 3] + 1)
    # 현재의 수가 5로 나누어 떨어지는 경우
    if i % 5 == 0:
        d[i] = min(d[i], d[i // 5] + 1)

print(d[x])
```

〈문제〉 효율적인 화폐 구성: 문제 설명

- N가지 종류의 화폐가 있습니다. 이 화폐들의 개수를 최소한으로 이용해서 그 가치의 합이 M원이 되도록 하려고 합니다. 이때 각 종류의 화폐는 몇 개라도 사용할 수 있습니다.
- 예를 들어 2원, 3원 단위의 화폐가 있을 때는 15원을 만들기 위해 3원을 5개 사용하는 것이 가장 최소한의 화폐 개수입니다.
- M원을 만들기 위한 최소한의 화폐 개수를 출력하는 프로그램을 작성하세요.

〈문제〉 효율적인 화폐 구성: 문제 해결 아이디어

- a_i = 금액 i를 만들 수 있는 최소한의 화폐 개수
- k = 각 화폐의 단위
- **점화식**: 각 화폐 단위인 k를 **하나씩 확인하며**
  - a_{i-k}를 만드는 방법이 존재하는 경우, $a_i = \min(a_i, a_{i-k} + 1)$
  - a_{i-k}를 만드는 방법이 존재하지 않는 경우, $a_i = INF$

〈문제〉 효율적인 화폐 구성: 문제 조건

난이도 ●●○ | **풀이 시간** 30분 | **시간 제한** 1초 | **메모리 제한** 128MB

입력 조건 • 첫째 줄에 N, M이 주어진다. ($1 \le N \le 100$, $1 \le M \le 10,000$)
• 이후의 N개의 줄에는 각 화폐의 가치가 주어진다. 화폐의 가치는 10,000보다 작거나 같은 자연수이다.

출력 조건 • 첫째 줄에 최소 화폐 개수를 출력한다. • 불가능할 때는 −1을 출력한다.

입력 예시 1

```
2 15
2
3
```

출력 예시 1

```
5
```

입력 예시 2

```
3 4
3
5
7
```

출력 예시 2

```
-1
```


〈문제〉 효율적인 화폐 구성: 문제 해결 아이디어

- $N = 3$, $M = 7$이고, 각 화폐의 단위가 2, 3, 5인 경우 확인해 봅시다.
- Step 1
  - 첫 번째 화폐 단위인 2를 확인합니다.
  - 점화식에 따라서 다음과 같이 리스트가 갱신됩니다.

인덱스	0	1	2	3	4	5	6	7
값	0	10,001	1	10,001	2	10,001	3	10,001

4원을 만들기 위한 개수는 2개: (2원 + 2원)

7원을 만드는 방법이 없음

〈문제〉 효율적인 화폐 구성: 문제 해결 아이디어

- $N = 3$, $M = 7$이고, 각 화폐의 단위가 2, 3, 5인 경우 확인해 봅시다.
- Step 3
  - 세 번째 화폐 단위인 5를 확인합니다.
  - 점화식에 따라서 다음과 같이 최종적으로 리스트가 갱신됩니다.

인덱스	0	1	2	3	4	5	6	7
값	0	10,001	1	1	2	1	2	2

7원을 만들기 위한 개수는
2개: (2원 + 5원)

〈문제〉 효율적인 화폐 구성: 답안 예시 (Python)

```python
# 정수 N, M을 입력 받기
n, m = map(int, input().split())
# N개의 화폐 단위 정보를 입력받기
array = []
for i in range(n):
    array.append(int(input()))

# 한 번 계산된 결과를 저장하기 위한 DP 테이블 초기화
d = [10001] * (m + 1)

# 다이나믹 프로그래밍(Dynamic Programming) 진행(보텀업)
d[0] = 0
for i in range(n):
    for j in range(array[i], m + 1):
        if d[j - array[i]] != 10001: # (i - k)원을 만드는 방법이 존재하는 경우
            d[j] = min(d[j], d[j - array[i]] + 1)

# 계산된 결과 출력
if d[m] == 10001: # 최종적으로 M원을 만드는 방법이 없는 경우
    print(-1)
else:
    print(d[m])
```

〈문제〉 금광: 문제 설명

- n × m 크기의 금광이 있습니다. 금광은 1 × 1 크기의 칸으로 나누어져 있으며, 각 칸은 특정한 크기의 금이 들어 있습니다.
- 채굴자는 첫 번째 열부터 출발하여 금을 캐기 시작합니다. 맨 처음에는 첫 번째 열의 어느 행에서든 출발할 수 있습니다. 이후에 m - 1번에 걸쳐서 매번 오른쪽 위, 오른쪽, 오른쪽 아래 3가지 중 하나의 위치로 이동해야 합니다. 결과적으로 **채굴자가 얻을 수 있는 금의 최대 크기**를 출력하는 프로그램을 작성하세요.

1	3	3	2
2	1	4	1
0	6	4	7

얻을 수 있는 금의 최대 크기: 19

〈문제〉 금광: 문제 조건

난이도 ●◐○ | **풀이 시간** 30분 | **시간 제한** 1초 | **메모리 제한** 128MB | **기출** Flipkart 인터뷰

입력 조건
- 첫째 줄에 테스트 케이스 T가 입력됩니다. (1 ≤ T ≤ 1000)
- 매 테스트 케이스 첫째 줄에 n과 m이 공백으로 구분되어 입력됩니다. (1 ≤ n, m ≤ 20) 둘째 줄에 n × m개의 위치에 매장된 금의 개수가 공백으로 구분되어 입력됩니다. (1 ≤ 각 위치에 매장된 금의 개수 ≤ 100)

출력 조건
- 테스트 케이스마다 채굴자가 얻을 수 있는 금의 최대 크기를 출력합니다. 각 테스트 케이스는 줄 바꿈을 이용해 구분합니다.

입력 예시

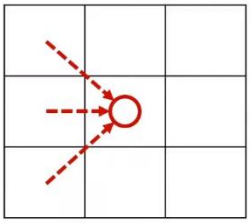
```
2
3 4
1 3 3 2 2 1 4 1 0 6 4 7
4 4
1 3 1 5 2 2 4 1 5 0 2 3 0 6 1 2
```

출력 예시

```
19
16
```

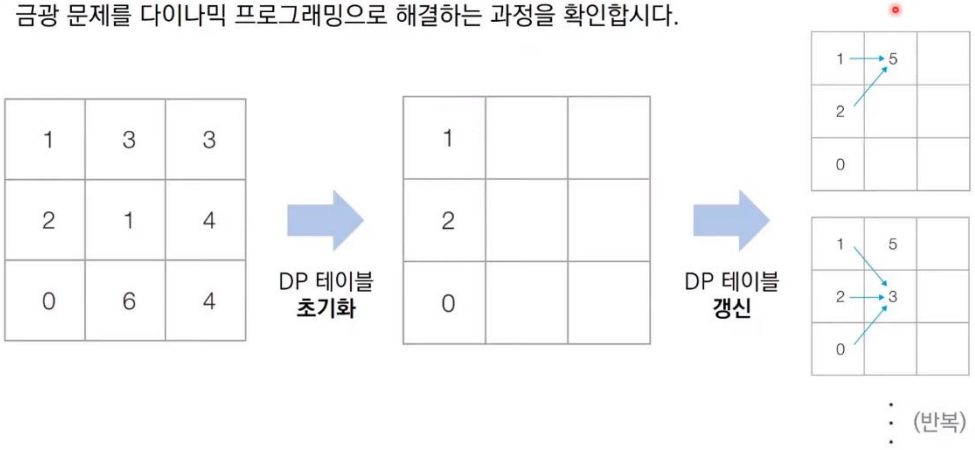
〈문제〉 금광: 문제 해결 아이디어

- 금광의 모든 위치에 대하여 다음의 세 가지만 고려하면 됩니다.
  1. 왼쪽 위에서 오는 경우
  2. 왼쪽 아래에서 오는 경우
  3. 왼쪽에서 오는 경우
- **세 가지** 경우 중에서 가장 많은 금을 가지고 있는 경우를 테이블에 갱신해주어 문제를 해결합니다.



〈문제〉 금광: 문제 해결 아이디어

- 금광 문제를 다이나믹 프로그래밍으로 해결하는 과정을 확인합시다.



〈문제〉 금광: 답안 예시 (Python)

```python
# 테스트 케이스(Test Case) 입력
for tc in range(int(input())):
    # 금광 정보 입력
    n, m = map(int, input().split())
    array = list(map(int, input().split()))
    # 다이나믹 프로그래밍을 위한 2차원 DP 테이블 초기화
    dp = []
    index = 0
    for i in range(n):
        dp.append(array[index:index + m])
        index += m
    # 다이나믹 프로그래밍 진행
    for j in range(1, m):
        for i in range(n):
            # 왼쪽 위에서 오는 경우
            if i == 0: left_up = 0
            else: left_up = dp[i - 1][j - 1]
            # 왼쪽 아래에서 오는 경우
            if i == n - 1: left_down = 0
            else: left_down = dp[i + 1][j - 1]
            # 왼쪽에서 오는 경우
            left = dp[i][j - 1]
            dp[i][j] = dp[i][j] + max(left_up, left_down, left)
    result = 0
    for i in range(n):
        result = max(result, dp[i][m - 1])
    print(result)
```


〈문제〉 병사 배치하기: 문제 설명

- N명의 병사가 무작위로 나열되어 있습니다. 각 병사는 특정한 값의 전투력을 보유하고 있습니다.
- 병사를 배치할 때는 **전투력이 높은 병사가 앞쪽에 오도록 내림차순으로 배치**를 하고자 합니다. 다시 말해 앞쪽에 있는 병사의 전투력이 항상 뒤쪽에 있는 병사보다 높아야 합니다.
- 또한 배치 과정에서는 특정한 위치에 있는 병사를 열외시키는 방법을 이용합니다. 그러면서도 남아 있는 병사의 수가 최대가 되도록 하고 싶습니다.

〈문제〉 병사 배치하기: 문제 조건

난이도 ●◐○ | **풀이 시간** 40분 | **시간 제한** 1초 | **메모리 제한** 256MB | **기출** 핵심 유형

입력 조건 • 첫째 줄에 N이 주어집니다. ($1 \leq N \leq 2,000$) 둘째 줄에 각 병사의 전투력이 공백으로 구분되어 차례대로 주어집니다. 각 병사의 전투력은 10,000,000보다 작거나 같은 자연수입니다.

출력 조건 • 첫째 줄에 남아 있는 병사의 수가 최대가 되도록 하기 위해서 열외시켜야 하는 병사의 수를 출력합니다.

입력 예시

```
7
15 11 4 8 5 2 4
```

출력 예시

```
2
```


〈문제〉 병사 배치하기: 문제 해결 아이디어

- 이 문제의 기본 아이디어는 **가장 긴 증가하는 부분 수열(Longest Increasing Subsequence, LIS)**로 알려진 전형적인 다이나믹 프로그래밍 문제의 아이디어와 같습니다.
- 예를 들어 하나의 수열 $array = \{4, 2, 5, 8, 4, 11, 15\}$이 있다고 합시다.
  - 이 수열의 가장 긴 증가하는 부분 수열은 $\{4, 5, 8, 11, 15\}$입니다.
- 본 문제는 가장 긴 감소하는 부분 수열을 찾는 문제로 치환할 수 있으므로, LIS 알고리즘을 조금 수정하여 적용함으로써 정답을 도출할 수 있습니다.

〈문제〉 병사 배치하기: 문제 해결 아이디어

- 가장 긴 증가하는 부분 수열 (LIS) 알고리즘을 확인해 봅시다.
- $D[i]$ = $array[i]$를 마지막 원소로 가지는 부분 수열의 최대 길이
- 점화식은 다음과 같습니다.

모든 $0 \leq j < i$ 에 대하여, `D[i] = max(D[i], D[j] + 1) if array[j] < array[i]`

〈문제〉 병사 배치하기: 문제 해결 아이디어

모든 $0 \leq j < i$ 에 대하여, `D[i] = max(D[i], D[j] + 1) if array[j] < array[i]`

	4	2	5	8	4	11	15
초기 상태: $i = 0$	1	1	1	1	1	1	1
$i = 1$	1	1	1	1	1	1	1
$i = 2$	1	1	2	1	1	1	1
$i = 3$	1	1	2	3	1	1	1
$i = 4$	1	1	2	3	2	1	1
$i = 5$	1	1	2	3	2	4	1
$i = 6$	1	1	2	3	2	4	5

〈문제〉 병사 배치하기: 문제 해결 아이디어

- 가장 먼저 입력 받은 병사 정보의 순서를 뒤집습니다.
- **가장 긴 증가하는 부분 수열 (LIS)** 알고리즘을 수행하여 정답을 도출합니다.

〈문제〉 병사 배치하기: 답안 예시 (Python)

```python
n = int(input())
array = list(map(int, input().split()))
# 순서를 뒤집어 '최장 증가 부분 수열' 문제로 변환
array.reverse()

# 다이나믹 프로그래밍을 위한 1차원 DP 테이블 초기화
dp = [1] * n

# 가장 긴 증가하는 부분 수열(LIS) 알고리즘 수행
for i in range(1, n):
    for j in range(0, i):
        if array[j] < array[i]:
            dp[i] = max(dp[i], dp[j] + 1)

# 열외해야 하는 병사의 최소 수를 출력
print(n - max(dp))
```