Dynamic Programming

Dp

Dosawas 2024-05-04

목차

차례

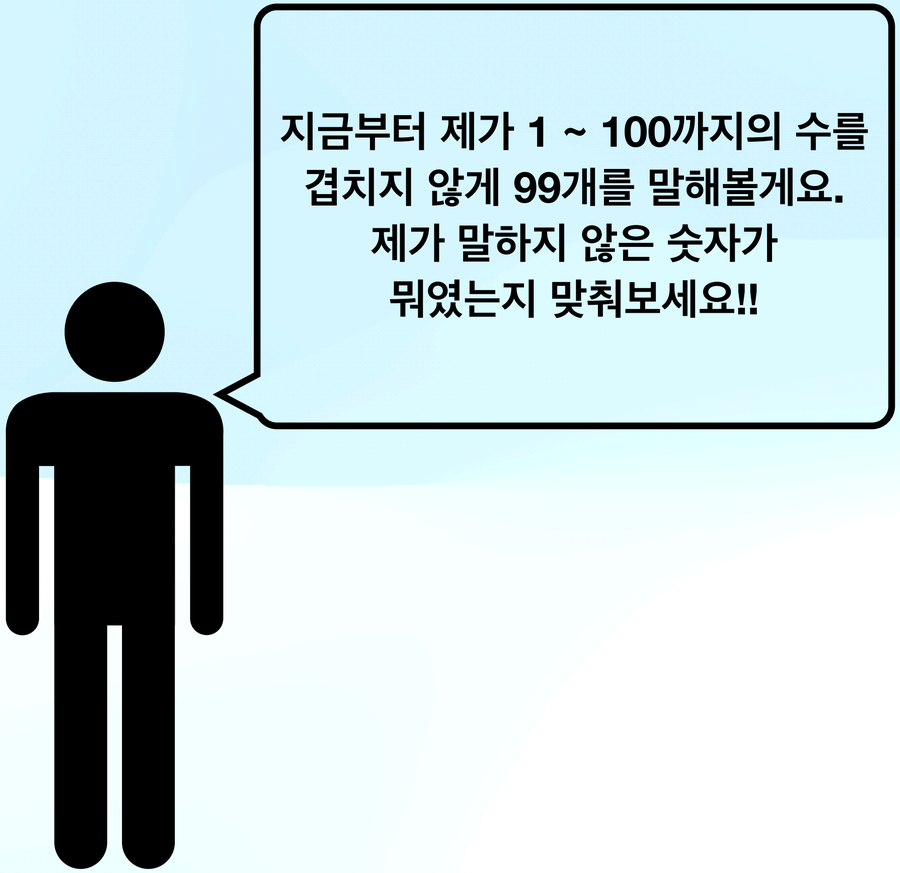
- 4회차 리뷰
- 일반적인 DP : 점화식 세우기
  - Top-down과 Bottom up
- Knapsack technique
  - 배낭 문제
- 대표적인 2차원 DP
  - LCS
  - 구간 dp

4회차 리뷰

- 이분탐색 - log 시간에 원하는 정보를 정렬된 자료에서 찾을 수 있다.
- 부분 합 - 구간에대한 합, 곱 등의 쿼리가 들어올 때 빠르게 쿼리에 대한 답을 구할 수 있다.
- 슬라이딩 윈도우 - 배열을 왼쪽에서 오른쪽으로 한번만 쭉 훑으며 윈도우 내부의 정보를 잘 관리할 수 있다.

- 여러분들 스스로 투포인터에 대해서도 공부해보세요.

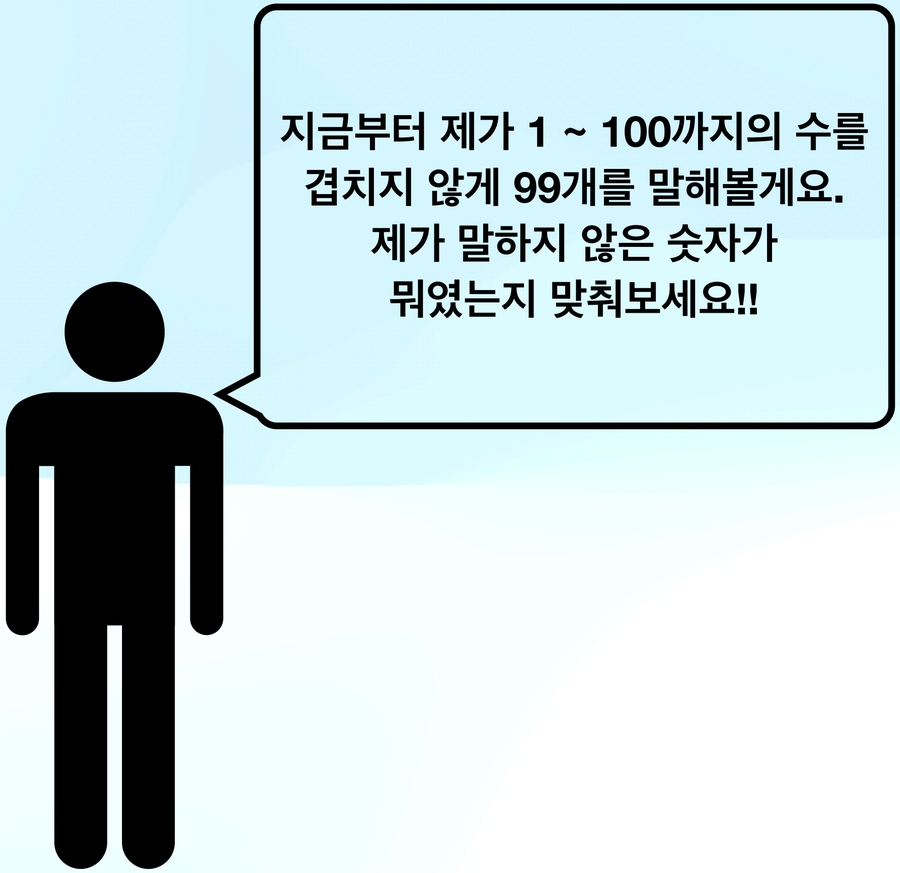
DP = 구해둔 정보를 재사용



1, 30, 10, 23, 4,6, 34, 16,.....

오케이.. 1나왔고,
30나왔고,
...
어?? 까먹었다..

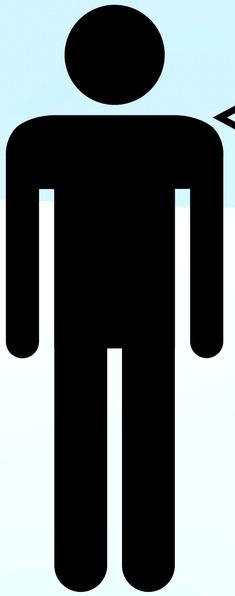
DP = 구해둔 정보를 재사용



1, 30, 10, 23, 4,6, 34, 16,.....

1부터 100까지 다 더하면 5050.
즉,아직 말 안했을 땐 5050
하나 말하고 나선 5049
2개 말하고 나선 5019
3개 말하고 나선 5009
...

DP = 구해둔 정보를 재사용



지금부터 제가 1 ~ 100까지의 수를
겹치지 않게 99개를 말해볼게요.
제가 말하지 않은 숫자가
뭐였는지 맞춰보세요!!

1, 30, 10, 23, 4,6, 34, 16,.....

1부터 100까지 다 더하면 5050.
즉,아직 말 안했을 땐 5050
하나 말하고 나선 5049
2개 말하고 나선 5019
3개 말하고 나선 5009
...

dp[i] : i번째 숫자까지 말했을 때 남은 수들을 다 더한 값
dp[i] = dp[i - 1] - 지금 말한 값
바로 이전에 저장해둔 값을 이용(재사용)!!
정답 : dp[99] : 99개의 숫자를 말했을 때 남아있는 값

아까 핑크머리 친구는 지금 까지 나온 값 모두를 기억해야하지만
파란머리 친구는 딱 1개(dp[i-1])만 기억해두면 된다.

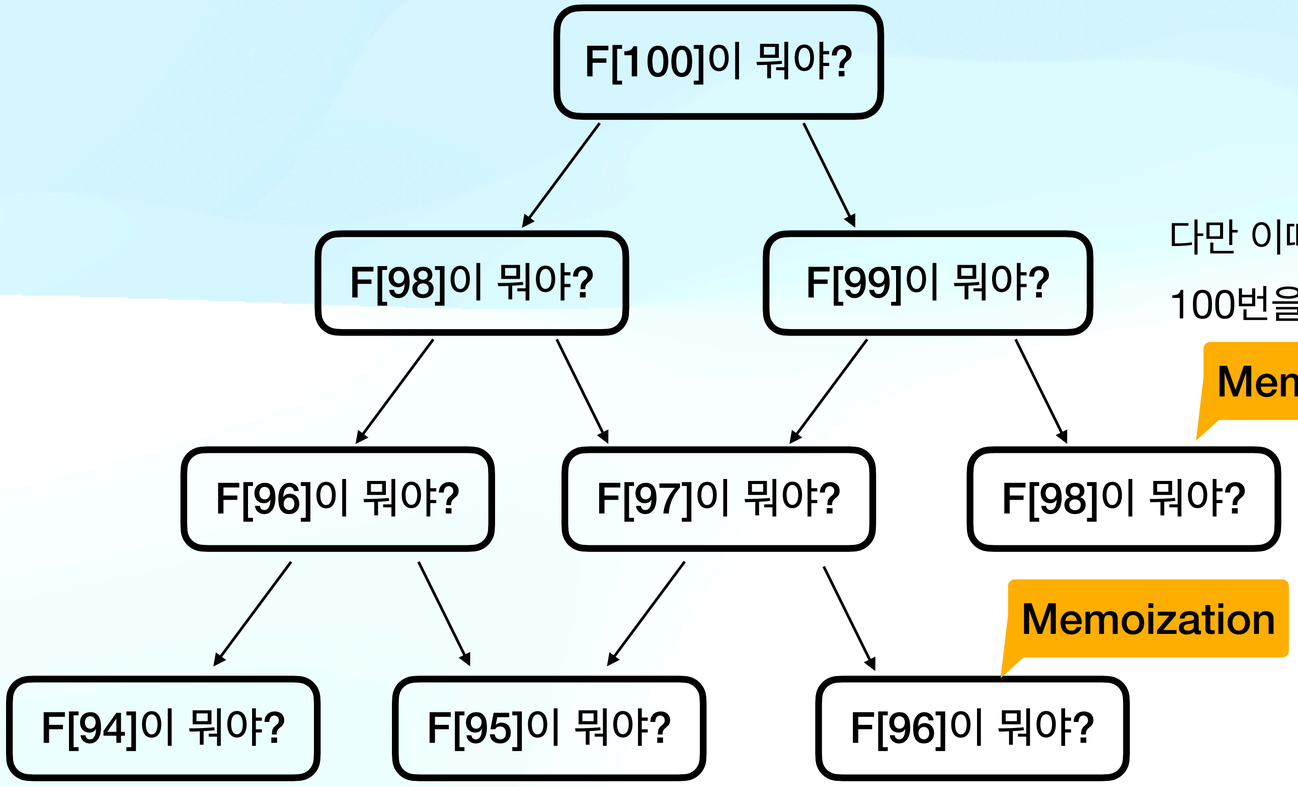
DP = 구해둔 정보 재사용

피보나치 수열

- DP(동적계획법)가 뭐냐? 라고 찾아보면
  - “큰 문제를 작은 문제로 나누어서 푸는 문제” -> 바로 와닿지 않음..
- 피보나치 수열은 1, 1, 2, 3, 5, 8, 13, 21 .. 이렇게 쭉 가는 수열이죠.
  - 규칙성을 보면 바로 이전항 두 개를 더한다는 것입니다.
  - F[i] : i번째 피보나치 수 라고 정의하면 F[i] = F[i-1] + F[i-2] 가 됩니다.
  - 큰 문제 (100번째 피보나치 수 구하기) 를 작은문제(99와 98번째 피보나치 수 구하기)로 나누어서 푼다는 뜻입니다.
- 저번시간에 했던 누적합도 dp의 일종이라고도 볼 수 있겠죠

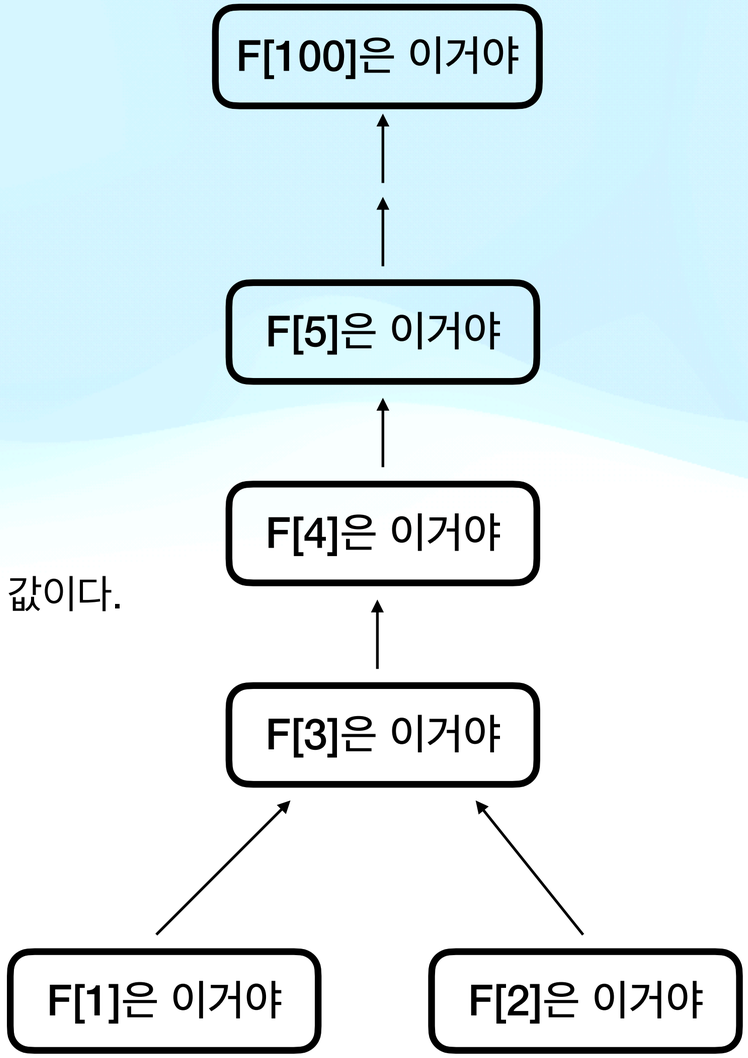
DP = 구해둔 정보 재사용

Top down vs Bottom up



다만 이때 F[98]은

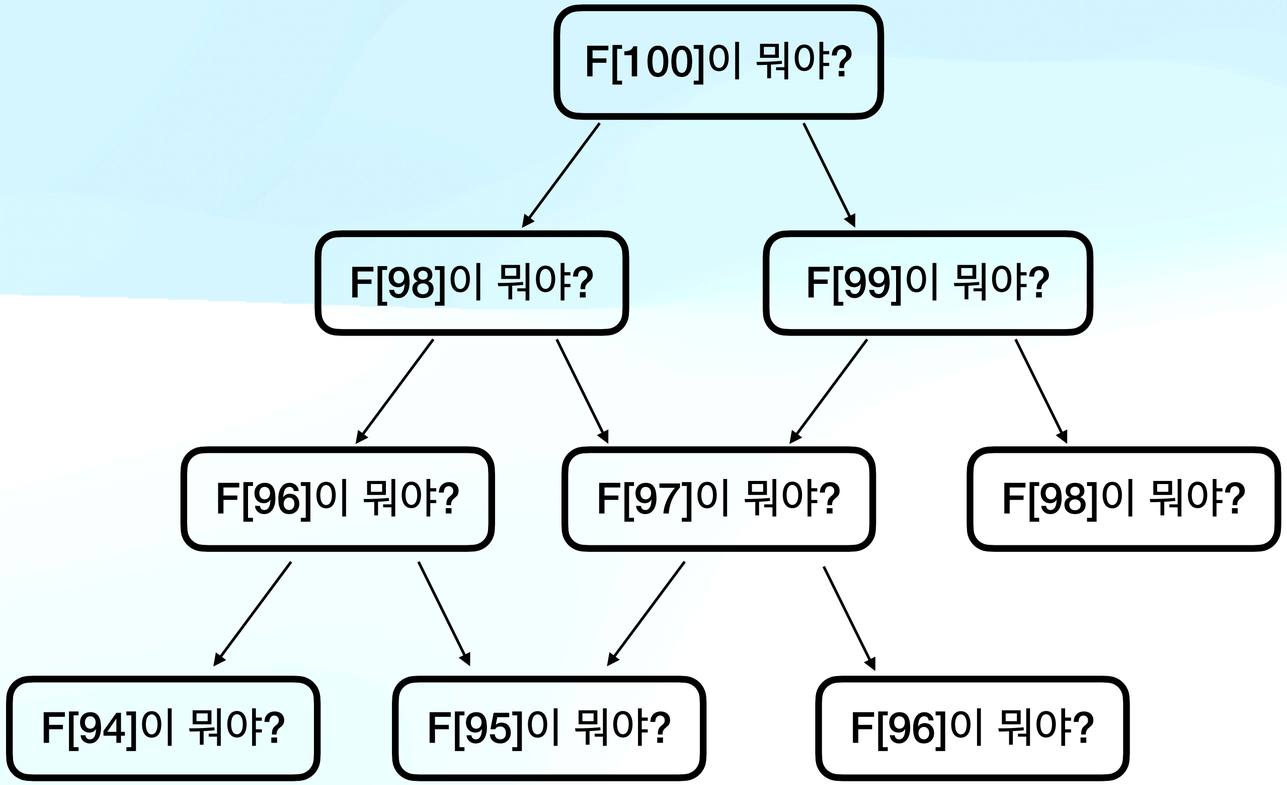
100번을 구해도 같은 값이다.



메모이제이션 하려면 재사용하려던 값이 변하는 값이면 안되겠죠.

DP = 구해둔 정보 재사용

Top down



Q0. 알고리즘 고수가 되려면 어떻게 해야해요?

A0. HLD 알고리즘을 알면 된대요!!

Q1. HLD를 알려면 어떻게 해야해요?

A1. LCA를 아셔야해요!

Q2. LCA는 뭔데요?

A2. 아 그거 sparse table부터 배우고 오세요

Q3. Sparse table은 어떻게 알아요?

A3. 일단 배열과 비트마스킹을 공부해보세요

Q4-1. 배열을 공부한다. (기저사례)

Q4-2. 비트마스킹을 공부한다. (기저사례)

궁금한것을 **그때마다** 찾아다니다 보니 알고리즘 고수가 되었다.

DP = 구해둔 정보 재사용

Bottom up

0. 비트마스킹과 배열을 열심히 공부했다.

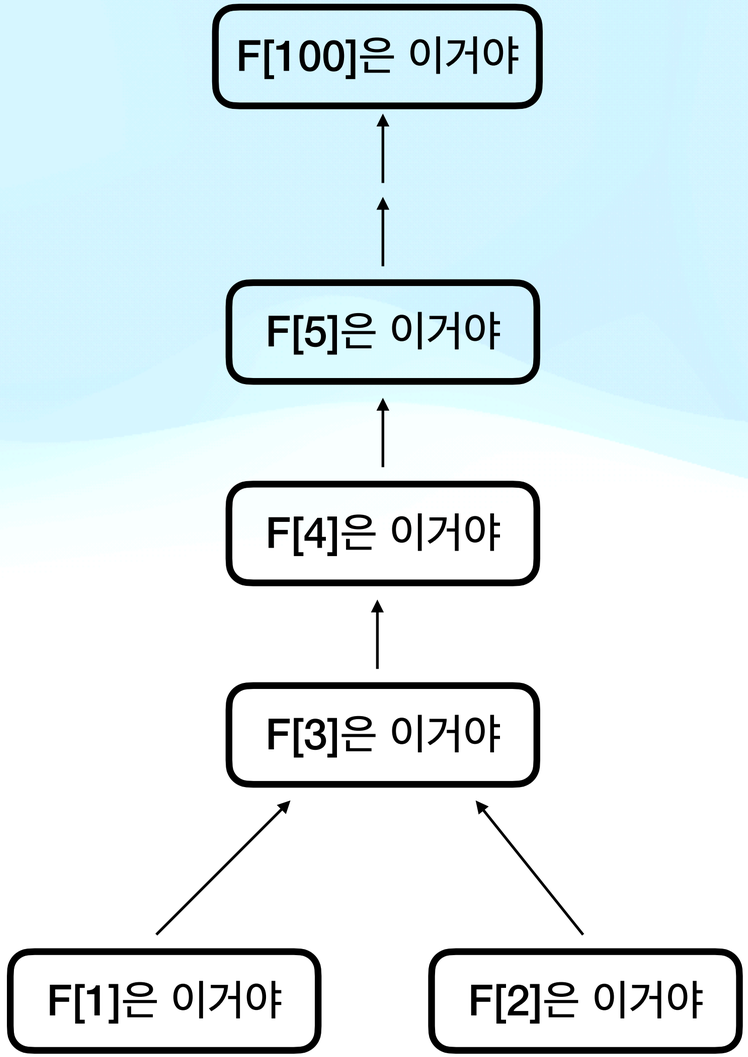
1. 그거 2개 알면 sparse table도 알 수 있다.

2. Sparse table 공부했으면 LCA도 구할 수 있다.

3. LCA를 알면 HLD도 할 수 있다.

4. HLD를 알면 알고리즘 고수가 된다.

아싸 난 **기초**부터 탄탄히해서 알고리즘 고수가 되었다.



DP = 구해둔 정보 재사용

Top down vs Bottom up

Top down	Bottom up
재귀 함수를 쓰기 좋음	반복문(for, while)으로 풀기 좋음
Memoization(이 전에 계산한값 저장)	재귀를 안쓸 수 있어서 빠름, 메모리에 이득이 있음 Tabulation (표채우기) (Ex. 피보나치 계산할 때 이전 항 2개만 저장해두면 됨)
직관적인 코드를 쓸 수 있음 (Ex. 딱보고 -> 아 이 코드는 똥피하기 게임을 만드는 코드구나)	기초부터 쌓아가기 때문에 직관적이지 못함 (Ex. 똥피하기 게임을 만드는데 갑자기 어셈블리어부터 공부함)
어떤 문제는 top-down이 편하고	어떤 문제는 bottom-up이 편합니다.

DP - 점화식 짜기

민균이의 계략 - Longest Increasing Subsequence(LIS) (S2)

시간 제한	메모리 제한	제출	정답	맞힌 사람	정답 비율
1 초	256 MB	4057	2145	1837	54.901%

문제

민균이는 요즘 준민이를 놀리는 일에 재미가 들렸다. 오늘도 그는 준민이를 놀리기 위해 한가지 재미있는 아이디어를 떠올렸다. 그는 하나의 정수가 쓰여 있는 카드 N장을 준비하여 준민이에게 정해진 순서대로 보여줄 것이다. 준민이는 앞의 카드부터 임의의 개수만큼 골라서 민균이에게 제시하게 되는데, 제시한 카드의 수열이 순증가가 아니면 민균이에게 바보라고 놀림받게 된다. 예를 들어 민균이가 보여준 카드가 {4, 9, 10, 9} 일 때 준민이가 {4, 9}를 골랐다면 놀림을 받지 않겠지만, {4, 10, 9}이나 {9, 9}를 제시하면 놀림받게 될 것이다.

생각보다 바보가 아닌 준민이는 한번도 민균이에게 놀림을 받지 않았다. 이에 분노한 민균이는 하나의 조건을 더 추가했다. 이제 준민이가 제시해야하는 수열은 순증가여야 할 뿐만 아니라, 원소의 개수가 제일 많은 수열이여야 한다. 예를 들어 민균이가 보여준 카드가 {8, 9, 1, 2 ,10}일 때 준민이가 {8, 9, 10} 또는 {1, 2, 10}을 골랐다면 놀림을 받지 않겠지만, {8, 9}나 {1, 2}를 제시하면 놀림받게 될 것이다.

당황한 준민이는 일단 제시할 수 있는 수열의 원소의 최대 개수를 구해보기로 하였다. 예를 들어 {8, 9, 1, 2, 10}에서의 원소의 최대 개수는 3이 될 것이다. 도저히 못 풀겠어서 고민하던 준민이는 똑똑하기로 소문난 당신을 찾아가 이 문제를 의뢰하였다! 불쌍하고 딱한 준민이를 위해 이 문제를 해결하는 프로그램을 작성해보자.

입력

첫 번째 줄에는 민균이가 제시한 카드의 개수 N (1 ≤ N ≤ 1,000)이 주어진다.

두 번째 줄에는 민균이가 제시한 카드 N개에 들어있는 정수가 공백(빈 칸)으로 구분되어 주어진다.

각 정수는 1 이상 100,000,000 이하의 자연수이다.

1. 고른 부분수열이 증가해야한다.
2. 고른 부분수열이 최장 부분수열이다.

LIS

DP - 점화식 짜기

민균이의 계략 - Longest Increasing Subsequence(LIS) (S2)

<바텀업>

```cpp
const int MAX = 1001;
int n, a[MAX], dp[MAX];
// dp[i] = i번째 원소를 포함하는 최대 LIS의 길이

int main(){
    fast_io
    cin >> n;
    for(int i=0;i<n;i++) cin >> a[i];
    int res = 0;
    for(int i=0;i<n;i++){
        dp[i] = 1; // 자기 자신만 생각해서 최소 1은 된다.
        for(int j=0;j<i;j++){
            if(a[j] < a[i]){
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        res = max(res, dp[i]);
    }
    cout << res;
}
```

메모리	시간
2028 KB	4 ms
2028 KB	0 ms

$O(N^2)$

<탑다운>

```cpp
const int MAX = 1001;
int n, a[MAX], dp[MAX];
// dp[i] = i번째 원소를 포함하는 최대 LIS의 길이

int memo(int idx){
    int &ret = dp[idx];
    if(ret) return ret;
    ret = 1;
    for(int i=0;i<idx;i++){
        if(a[i] < a[idx])ret = max(ret, memo(i) + 1);
    }
    return ret;
}
int main(){
    fast_io
    cin >> n;
    for(int i=0;i<n;i++) cin >> a[i];
    int res = 0;
    for(int i=0;i<n;i++){
        res = max(res, memo(i));
    }
    cout << res;
}
```

DP - Knapsack technique(배낭 테크닉)

최대한 이득을 보고 싶을 때

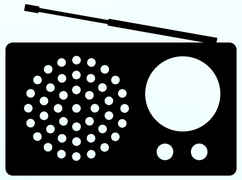
무인도에 가려고 하는데 **가방의 크기가 한정적이네요..**

제일 높은 가치로 담을 수 있는 경우는 가치가 얼마나 될까요??

망치 -
5000원의 가치
700 g의 무게

배낭(Knapsack) -
3000g까지 담을 수 있다.

안전모 -
30000원의 가치
1200g의 무게



라디오 -
40000원의 가치
1800g의 무게

???? -
1000원의 가치
500g의 가치

기타 -
50000원의 가치
2500g의 무게

시계 -
18000원의 가치
800 g의 무게

DP - Knapsack technique(냅색 문제)

평범한 배낭 (G5)

문제

이 문제는 아주 평범한 배낭에 관한 문제이다.

한 달 후면 국가의 부름을 받게 되는 준서는 여행을 가려고 한다. 세상과의 단절을 슬퍼하며 최대한 즐기기 위한 여행이기 때문에, 가지고 다닐 배낭 또한 최대한 가치 있게 싸려고 한다.

준서가 여행에 필요하다고 생각하는 N개의 물건이 있다. 각 물건은 무게 W와 가치 V를 가지는데, 해당 물건을 배낭에 넣어서 가면 준서가 V만큼 즐길 수 있다. 아직 행군을 해본 적이 없는 준서는 최대 K만큼의 무게만을 넣을 수 있는 배낭만 들고 다닐 수 있다. 준서가 최대한 즐거운 여행을 하기 위해 배낭에 넣을 수 있는 물건들의 가치의 최댓값을 알려주자.

입력

첫 줄에 물품의 수 N($1 \le N \le 100$)과 준서가 버틸 수 있는 무게 K($1 \le K \le 100,000$)가 주어진다. 두 번째 줄부터 N개의 줄에 거쳐 각 물건의 무게 W($1 \le W \le 100,000$)와 해당 물건의 가치 V($0 \le V \le 1,000$)가 주어진다.

입력으로 주어지는 모든 수는 정수이다.

DP - Knapsack technique(냅색 문제)

평범한 배낭 (G5)

방법 1 - top-down : dp[i][k] : i번째 물건까지 봤고, 그 때 버틸 수 있는 무게가 k일 때 최대 가치

```cpp
int n, k;
int weight[100001], value[100001];
int dp[100001][101];

int backpack(int capacity, int item)
{ // 가치의 최댓값 리턴하는 함수
    if (item == n) return 0;
    int &ret = dp[capacity][item];
    if (ret != -1) return ret;
    ret = backpack(capacity, item + 1); // 안 고를때 (2가지 경우-고를때 안고를때밖에 없음)
    if (capacity >= weight[item]){
        ret = max(value[item] + backpack(capacity - weight[item], item + 1), ret); // 안고를 때랑 고를 때 비교
    }

    return ret;
}

int main(){
    fast_io
    cin >> n >> k;
    for (int i = 0; i < n; i++) cin >> weight[i] >> value[i];
    memset(dp, -1, sizeof(dp)); // 가치가 0일 수 있으니 -1로 dp초기화
    cout << backpack(k, 0);
}
```

여기서 주목할만한 부분 :

1. 한번 본 물품은 다시 보지 않는다.
2. 무게가 더 작아질 수는 없다!

이를 이용해서 메모리와 시간을 더 줄일 수 있습니다. 바텀업에서 그렇게 풀어보겠습니다.

DP - Knapsack technique(냅색 문제)

평범한 배낭 (G5)

방법 2 - bottom-up : dp[k] : 최대 무게가 k일 때, 최대 가치

```cpp
int dp[100001];
int main(){
    fast_io
    int n, k; cin >> n >> k;
    vector<pii> item(n + 1);
    for (int i = 1; i <= n; i++) cin >> item[i].xx >> item[i].yy;
    int res = 0;
    for (int i = 1; i <= n; i++){
        for (int j = k; j >= 0; j--){ // 무거운 경우부터 체크
            if (j + item[i].xx <= k){ // 한계 무게보다 작으면
                // 이 한계 무게에 대한 최대가치는 j가 한계 무게일 때 최대 가치에 이번 아이템을 넣는 것이 최대일 것이다.
                dp[j + item[i].xx] = max(dp[j + item[i].xx], dp[j] + item[i].yy);
                res = max(res, dp[j + item[i].xx]);
            }
        }
    }
    cout << res;
}
```

가벼운 경우부터 체크한다고 해보자

dp[3] = 1

dp[7] = 3

이 이미 저장되어 있는데 여기에 item이

무게는 4

가치는 3인것이 들어왔다고 해보자.

dp[3 + 4] = max(3, 1 + 3) = 4가 되는데

여기서 dp[11 = 7 + 4] = 4 + 3 으로 같은

item을 두번 넣는 경우가 생길 수 있다.

DP - Knapsack technique(냅색 문제)

예쁜수 (G4)

문제

1보다 크거나 같은 정수 N의 각 자리의 합을 S라고 할 때, S가 N의 약수라면 그 수를 **예쁜수**라고 하자.

지수는 자연수 $M(M \leq 5\,000)$을 예쁜수들의 합으로만 표현하고 싶다.

이때 합이 M인 예쁜수들의 구성이 다른 경우에만 다른 방법이다.

예를 들어 $M = 4$인 경우, $1 + 1 + 2 = 4$과 $2 + 1 + 1 = 4$는 같은 경우다.

지수를 도와 자연수 M을 예쁜수들의 합으로만 표현하는 경우의 수를 구해주자.

경우의 수는 매우 클 수 있으므로 자연수 M을 예쁜수들의 합으로 표현하는 경우의 수를 $K(10^6 \leq K \leq 10^7$, K는 소수)로 나눈 나머지를 구해주자.

입력

첫 번째 줄에 자연수 $M(1 \leq M \leq 5\,000)$과 $K(10^6 \leq K \leq 10^7$, K는 소수)가 공백으로 구분되어 주어진다.

예쁜수를 구하는 것 :
자리수만큼의 시간밖에 안걸림

그리고 한자리수는 모두 예쁜수니까
모든 자연수는 예쁜 수들의
합으로 나타낼 수 있다.

dp[i] : 자연수 i를 예쁜 수들의
합으로 나타내는 경우의 수

예쁜 수들을 item이라고 생각하자

DP - Knapsack technique(냅색 문제)

예쁜수 (G4)

1. 예쁜 수 전처리 :
    1. 아이템인지 아닌지 결정
2. j라는 허용범위 안에 i가 들어가면 다 더해줌
3. 모듈러 연산 %
    1. 더하기나 곱하기 중간중간에 해도 결과 안변함

```cpp
ll m, k;
int pretty[5001];
ll dp[5001];

int main(){
    fast_io
    cin >> m >> k;
    // 예쁜 수 전처리
    for (int i = 1; i < 5001; i++){
        int s = 0, tmp = i;
        while (tmp / 10){
            s += tmp % 10;
            tmp /= 10;
        }
        s += tmp;
        if (!(i % s)) pretty[i] = 1;
    }

    dp[0] = 1; // 아무것도 안쓰는 것도 하나의 경우다.
    for (int i = 1; i <= m; i++){
        if (pretty[i]){ // 예쁜 수라면
            for (int j = i; j <= m; j++){
                // i라는 예쁜 수 없을 때 경우의 수를 더해줌
                dp[j] += dp[j - i];
                dp[j] %= k;
            }
        }
    }

    cout << dp[m];
}
```

2차원 DP - LCS

Longest Common Subsequence

- dosawas와 iwasgosu 가 있습니다.
- 둘 다에 포함되면서
- 가장 긴 글자는 was겠죠

- 한 줄씩 처리한다고 생각하면 됩니다.
-

	D	O	S	A	W	A	S
I	0	0	0	0	0	0	0
W	0	0	0	0	1	0	0
A	0	0	0	1	1	2	2
S	0	0	1	1	1	2	3
G	0	0	1	1	1	2	3
O	0	1	1	1	1	2	3
S	0	1	1	1	1	2	3
U	0	1	1	1	1	2	3

2차원 DP - LCS

Longest Common Subsequence

- dosawas와 i를 비교해봅시다.
-

	D	O	S	A	W	A	S
I	0	0	0	0	0	0	0
W	0	0	0	0	1	0	0
A	0	0	0	1	1	2	2
S	0	0	1	1	1	2	3
G	0	0	1	1	1	2	3
O	0	1	1	1	1	2	3
S	0	1	1	1	1	2	3
U	0	1	1	1	1	2	3

2차원 DP - LCS

Longest Common Subsequence

- dosawas와 iw를 비교해봅시다.
-

	D	O	S	A	W	A	S
I	0	0	0	0	0	0	0
W	0	0	0	0	1	0	0
A	0	0	0	1	1	2	2
S	0	0	1	1	1	2	3
G	0	0	1	1	1	2	3
O	0	1	1	1	1	2	3
S	0	1	1	1	1	2	3
U	0	1	1	1	1	2	3

2차원 DP - LCS

Longest Common Subsequence

- dosawas와 iwa를 비교해봅시다.
- 혼자만 회색으로 표시된 부분을 봅시다.
  - W로 1일 수도 있고
  - A로 1일 수도 있습니다.

	D	O	S	A	W	A	S
I	0	0	0	0	0	0	0
W	0	0	0	0	1	0	0
A	0	0	0	1	1	2	2
S	0	0	1	1	1	2	3
G	0	0	1	1	1	2	3
O	0	1	1	1	1	2	3
S	0	1	1	1	1	2	3
U	0	1	1	1	1	2	3

2차원 DP - LCS

Longest Common Subsequence

- dosawas와 iwas를 비교해봅시다.
- 이번에도 회색을 봅시다.
  - 이미 있었던 WA로 2일 수도 있고
  - AS로 2일 수도 있습니다.
- 이를 통해 dp점화식은 이렇게 됩니다 .
  - $dp[i][j] = max(dp[i-1][j-1]+1, dp[i][j-1], dp[i-1][j])$

	D	O	S	A	W	A	S
I	0	0	0	0	0	0	0
W	0	0	0	0	1	0	0
A	0	0	0	1	1	2	2
S	0	0	1	1	1	2	3
G	0	0	1	1	1	2	3
O	0	1	1	1	1	2	3
S	0	1	1	1	1	2	3
U	0	1	1	1	1	2	3

2차원 DP - LCS

Longest Common Subsequence

```python
A = input()
B = input()

dp = [[0 for _ in range(1001)] for _ in range(1001)]
for i in range(1, len(A)+1) :
    for j in range(1, len(B)+1) :
        if A[i-1] == B[j-1] : dp[i][j] = dp[i-1][j-1] + 1
        dp[i][j] = max(dp[i][j], dp[i-1][j], dp[i][j-1])
print(dp[len(A)][len(B)])

```

간단하게 파이썬으로 짜봤습니다.

1-based로 하는게 가장자리를 처리하기 좋아서 저렇게 구현했습니다.

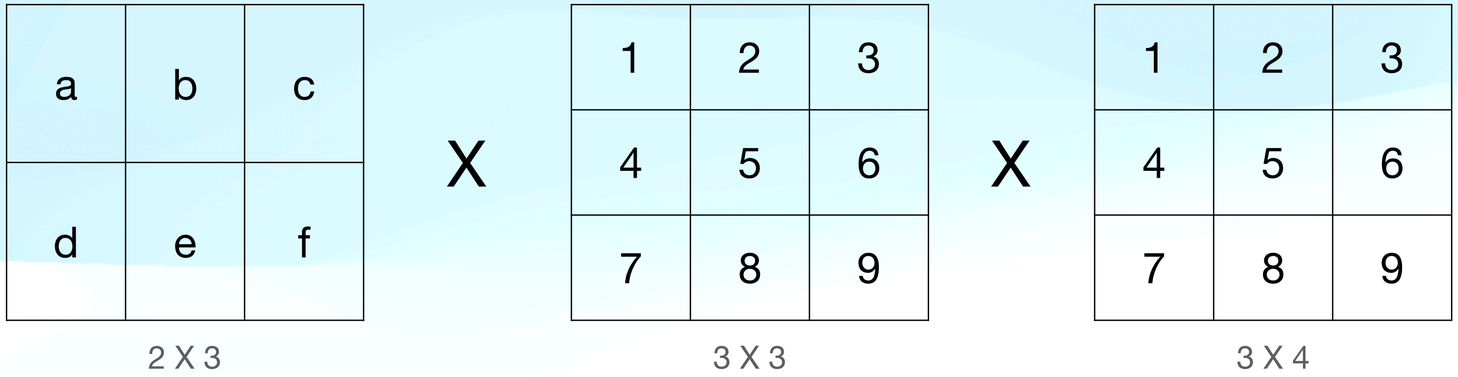
구간 dp - 보통 O(N^2)

구간의 정보를 저장해두는 dp

- 아마 알고리즘 수업을 들으면 무조건 하는 것이 있습니다.
  - 행렬 곱셈 순서에 따른 최적의 행렬 곱셈 연산 횟수를 구하는 문제입니다.
- 보통 이런식입니다.
  - dp[i][j] : i~j 구간에서 최댓값/최솟값
  - $dp[i][j] = f_k(g(dp[i][k], dp[k][j], i, j, k))$
    - 보통 어떤 한 구간의 값은 한 구간 안에 포함된 더 짧은 구간에 대한 값들로 구성

행렬곱셈 순서

- 행렬 곱셈 순서로 생각해봅시다.
-



A B C

((AB)C) 로 계산하기 vs (A(BC)) 로 계산하기

행렬 곱셈 순서

((AB)C) 로 계산하기

2×3 테이블에 3번 곱셈

+

2×4 테이블에 3번 곱셈

18 + 24 = 42

a	b	c
d	e	f

2 X 3

X

1	2	3
4	5	6
7	8	9

3 X 3

X

1	2	3
4	5	6
7	8	9

3 X 4

A B C

ㄱ	ㄴ	ㄷ
ㄹ	ㅁ	ㅂ

2 X 3

X

a_1	a_2	a_3
a_4	a_5	a_6
a_7	a_8	a_9

3 X 4

=

A	B	C	D
E	F	G	H

2 X 4

행렬 곱셈 순서
(A(BC)) 로 계산하기

3×4 테이블에 3번 곱셈

+

2×4 테이블에 3번 곱셈

36 + 24 = 60

a	b	c
d	e	f

2 X 3

X

1	2	3
4	5	6
7	8	9

3 X 3

X

1	2	3
4	5	6
7	8	9

3 X 4

A B C

ㄱ	ㄴ	ㄷ
ㄹ	ㅁ	ㅂ

2 X 3

X

b_1	b_2	b_3	b_4
b_5	b_6	b_7	b_8
b_9	b_10	b_11	b_12

3 X 4

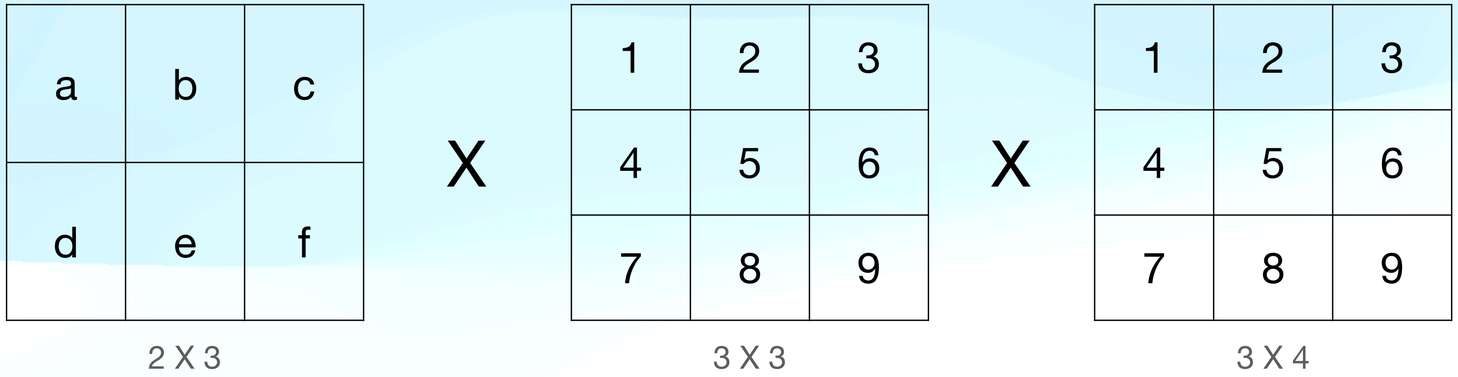
=

A	B	C	D
E	F	G	H

2 X 4

행렬 곱셈 순서

구간의 정보를 저장해두는 dp



그래서 A부터 C까지 최소로 행렬 곱셈 연산 횟수의 최솟값은 = 40이 됩니다.

행렬 곱셈 순서 (G3)

구간의 정보를 저장해두는 dp

문제

크기가 N×M인 행렬 A와 M×K인 B를 곱할 때 필요한 곱셈 연산의 수는 총 N×M×K번이다. 행렬 N개를 곱하는데 필요한 곱셈 연산의 수는 행렬을 곱하는 순서에 따라 달라지게 된다.

예를 들어, A의 크기가 5×3이고, B의 크기가 3×2, C의 크기가 2×6인 경우에 행렬의 곱 ABC를 구하는 경우를 생각해보자.

- AB를 먼저 곱하고 C를 곱하는 경우 (AB)C에 필요한 곱셈 연산의 수는 5×3×2 + 5×2×6 = 30 + 60 = 90번이다.
- BC를 먼저 곱하고 A를 곱하는 경우 A(BC)에 필요한 곱셈 연산의 수는 3×2×6 + 5×3×6 = 36 + 90 = 126번이다.

같은 곱셈이지만, 곱셈을 하는 순서에 따라서 곱셈 연산의 수가 달라진다.

행렬 N개의 크기가 주어졌을 때, 모든 행렬을 곱하는데 필요한 곱셈 연산 횟수의 최솟값을 구하는 프로그램을 작성하시오. 입력으로 주어진 행렬의 순서를 바꾸면 안 된다.

입력

첫째 줄에 행렬의 개수 N($1 \le N \le 500$)이 주어진다.

둘째 줄부터 N개 줄에는 행렬의 크기 r과 c가 주어진다. ($1 \le r, c \le 500$)

항상 순서대로 곱셈을 할 수 있는 크기만 입력으로 주어진다.

행렬 곱셈 순서 (G3)

구간의 정보를 저장해두는 dp

arr[i] : 행렬이 곱셈 가능한 상태니까

겹치는 부분은 합치고 쭉 펼쳐놓은 것

dp[s][e] : s ~ e까지 행렬 곱셈을 수행했을 때

최소 연산 횟수.

= min(dp[s][i] + dp[i][e] + arr[s]arr[i]arr[e])

기저사례 : 구간의 차이가 2가 되면

행렬 곱셈하는 방법이 1가지 밖에 없음

```cpp
const int INF = INT_MAX;
int n;
ll dp[501][501]; // i~j까지 곱셈횟수의 최솟값
ll arr[501];

ll solve(int s, int e){
    if (e - s < 2)
        return 0;
    if (e - s == 2){
        return arr[s] * arr[s + 1] * arr[e];
    }
    ll &ret = dp[s][e];
    if (ret) return ret;
    ret = INF;
    for (int i = s + 1; i < e; i++){
        ret = min(ret, arr[s] * arr[i] * arr[e] + solve(s, i) + solve(i, e));
    }
    return ret;
}

int main(){
    fast_io;
    cin >> n;
    for (int i = 0; i < n; i++) cin >> arr[i] >> arr[i + 1];
    cout << solve(0, n);
}
```

마무리

- dp는 문제가 매우매우 다양하고 많아서 유형별로 외워서 풀 수가 없습니다.
- 그래도 조금의 팁을 드려보자면
  - 정답을 dp의 value로 그리고 주어진 정보들을 idx로 나타낼 방법을 찾아보기
  - 조금씩 넣어보면서 규칙성을 찾아보기 (점화식)
- LCS나 LIS는 응용이 되는 경우가 많음
  - (문제에서 대놓고 “LIS나 LCS입니다”라고 안가르쳐주는 경우가 많음)

- dp는 점화식만 잘 짜면 솔직히 쉬워짐. (문제를 어떻게 풀지 설계하는 것의 중요성)

마무리

추천문제

- 1003 피보나치 함수
- 1699 제곱수의 합
- 1912 연속합 : 다른 문제에 기본적으로 응용될 수 있는 dp
- 9625 BABBA : dp가 꼭 모든 테이블을 채울 필요는 없다.
- 9184 신나는 함수 실행 : 이 문제를 바텀업으로 풀 수 있을까? -> 저는 못하겠더라구요.
- 1633 최고의 팀 만들기 : dp점화식을 잘 짤 수 있을까?
  - Hint : 저는 3차원 dp로 풀었습니다. - 2차원도 가능한걸로 압니다.

마무리

추천문제

- 12865 평범한 배낭 : 더 효율적인 방법으로 풀어보기
- 25958 예쁜수
- 2629 양팔저울 : 좋은 배낭 문제
- 9251 LCS
- 1958 LCS3 : LCS를 3차원으로 잘 바꿔보자
- 11049 행렬 곱셈 순서
- 11066 파일합치기 : 문제를 잘 읽어야함 - 연속적
- 13975 파일합치기 3 : 이 문제는 DP가 아닙니다 -> 위의 문제와 어떤 차이가 있는지 생각하고 풀어보기