Dynamic Programming

동적 계획법

2020 Winter 초급

20201546 강효규

Dynamic programming

✓ 주어진 문제를 부분 문제로 나누어서 푼 다음, 그 결과들로 주어진 문제를 푸는 방법

✓ 부분 문제의 정보를 저장해 놓음으로써 시간적, 공간적 효율성 획득

✓ 메모이제이션(memoization) 활용

✓최적 부분 구조를 만족한다.

#2748 피보나치 수 2

문제

피보나치 수는 0과 1로 시작한다. 0번째 피보나치 수는 0이고, 1번째 피보나치 수는 1이다. 그 다음 2번째 부터는 바로 앞 두 피보나치 수의 합이 된다.

이를 식으로 써보면 $F_n = F_{n-1} + F_{n-2}$ ($n \geq 2$)가 된다.

n=17일때 까지 피보나치 수를 써보면 다음과 같다.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597

n이 주어졌을 때, n번째 피보나치 수를 구하는 프로그램을 작성하시오.

Dynamic programming

$$F_n = F_{n-1} + F_{n-2}$$

✓ 피보나치 수열의 정의에 따라

- 전체 문제: n번째 피보나치 수를 구하는 것
- 부분 문제: n-1번째 피보나치수와 n-2번째 피보나치 수를 구하는 것

✓ 그럼 이제 다시 n-1번째 피보나치수와 n-2번째 피보나치 수를 구하는 과정은 최적부분구조를 만족함.

✓ 이렇게 부분문제의 정답을 활용하는 방법 자체가 dp이다.

재귀함수로 naive하게 구현

```
1  //parameter로 n을 전달받으면 n번째 피보나치수를 return 하는 함수
2
3  long long fibo(int n){
4      if(n<2)return n;
5      //f(0)==0,f(1)==1로 정의되어 있고, 재귀함수 탈출 조건
6      return fibo(n-1)+fibo(n-2);
7  }
```

#2748 피보나치 수 2

저 코드 그대로 넣어볼까?

n이 90까지 제한이 있는데 과연 돌아갈까?

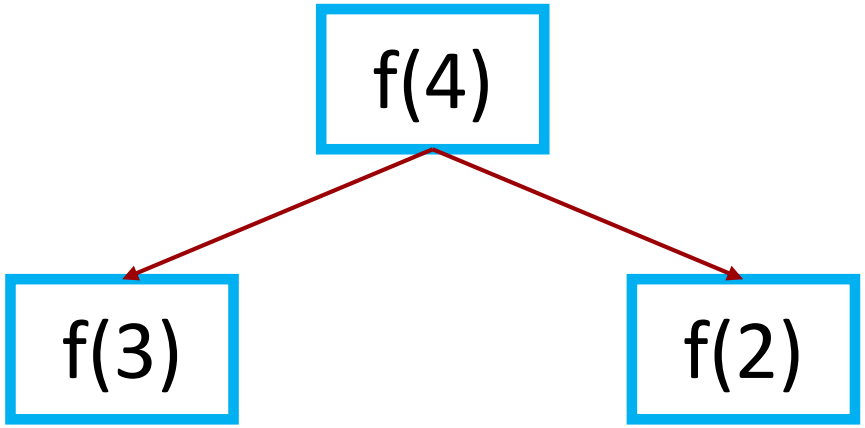
```cpp
#include<iostream>

using namespace std;
using ll = long long;
//피보나치수가 int를 넘어갈 수 있음
ll fibo(int n) {
    if (n < 2)return n;
    return fibo(n - 1) + fibo(n - 2);
}
int n;
int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0), cout.tie(0);
    cin >> n;
    cout << fibo(n);
}

```

TLE(시간초과)를 받습니다.

2748	시간 초과			C++14 / 수정	294 B

Dynamic programming

한번 계산해 놓은 값을 또 다시 계산하는 경우가 많다.



함수 호출 횟수

f(4):1

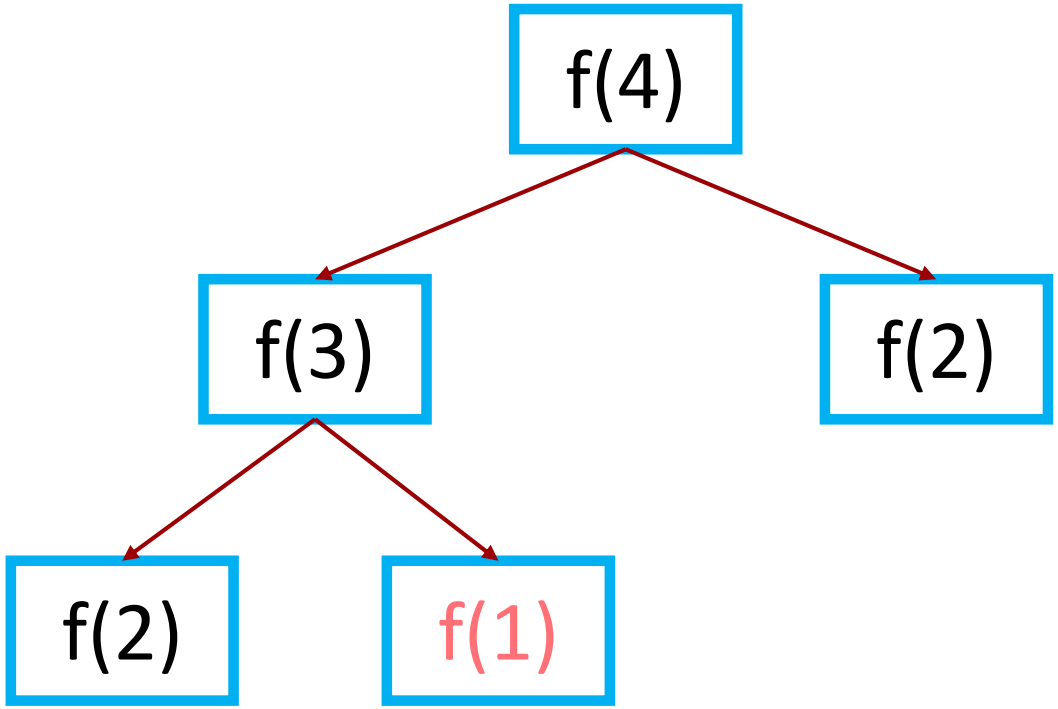
f(3):1

f(2):1

f(1):0

f(0):0

한번 계산해 놓은 값을 또 다시 계산하는 경우가 많다.



함수 호출 횟수

f(4):1

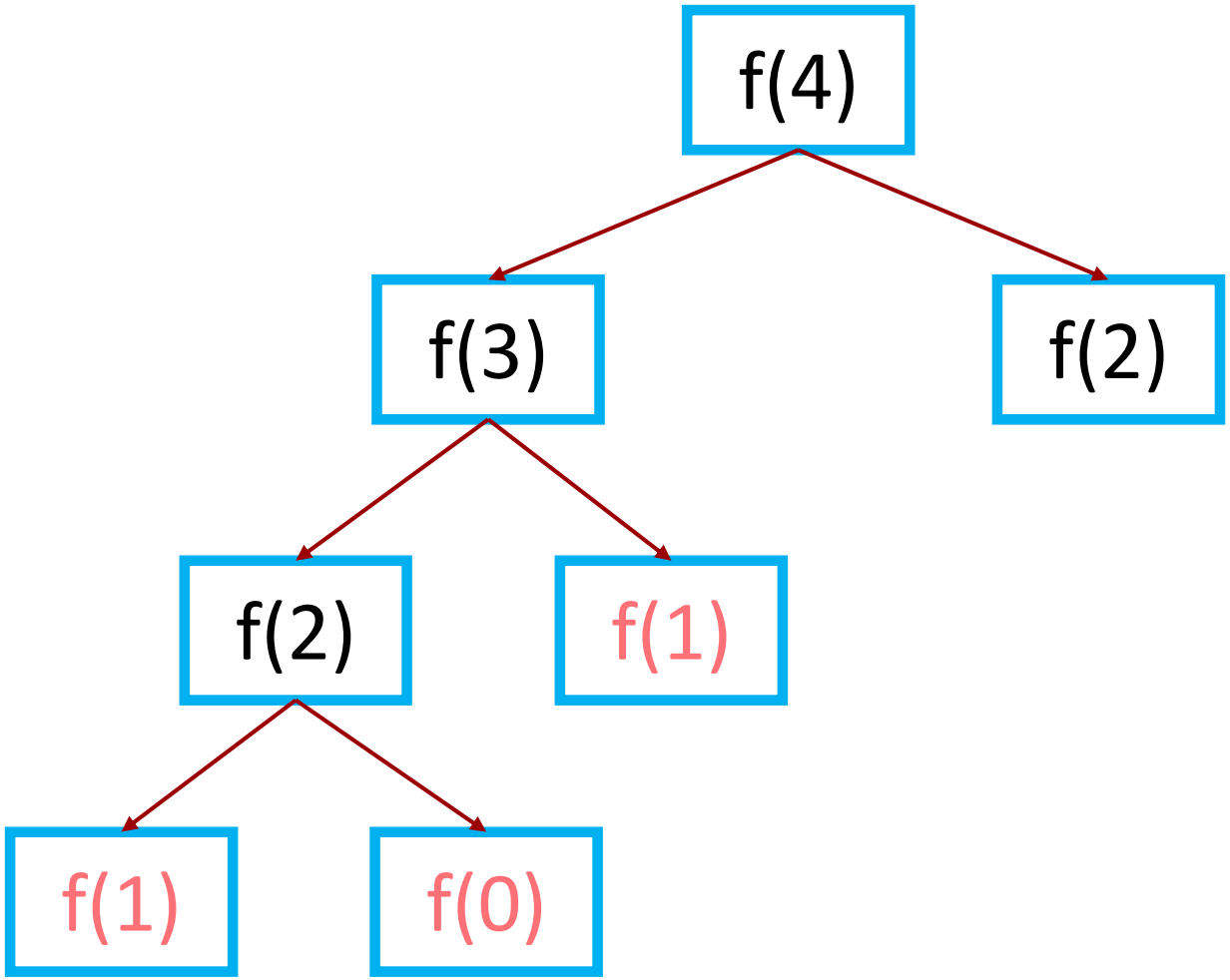
f(3):1

f(2):2

f(1):1

f(0):0

한번 계산해 놓은 값을 또 다시 계산하는 경우가 많다.



함수 호출 횟수

f(4):1

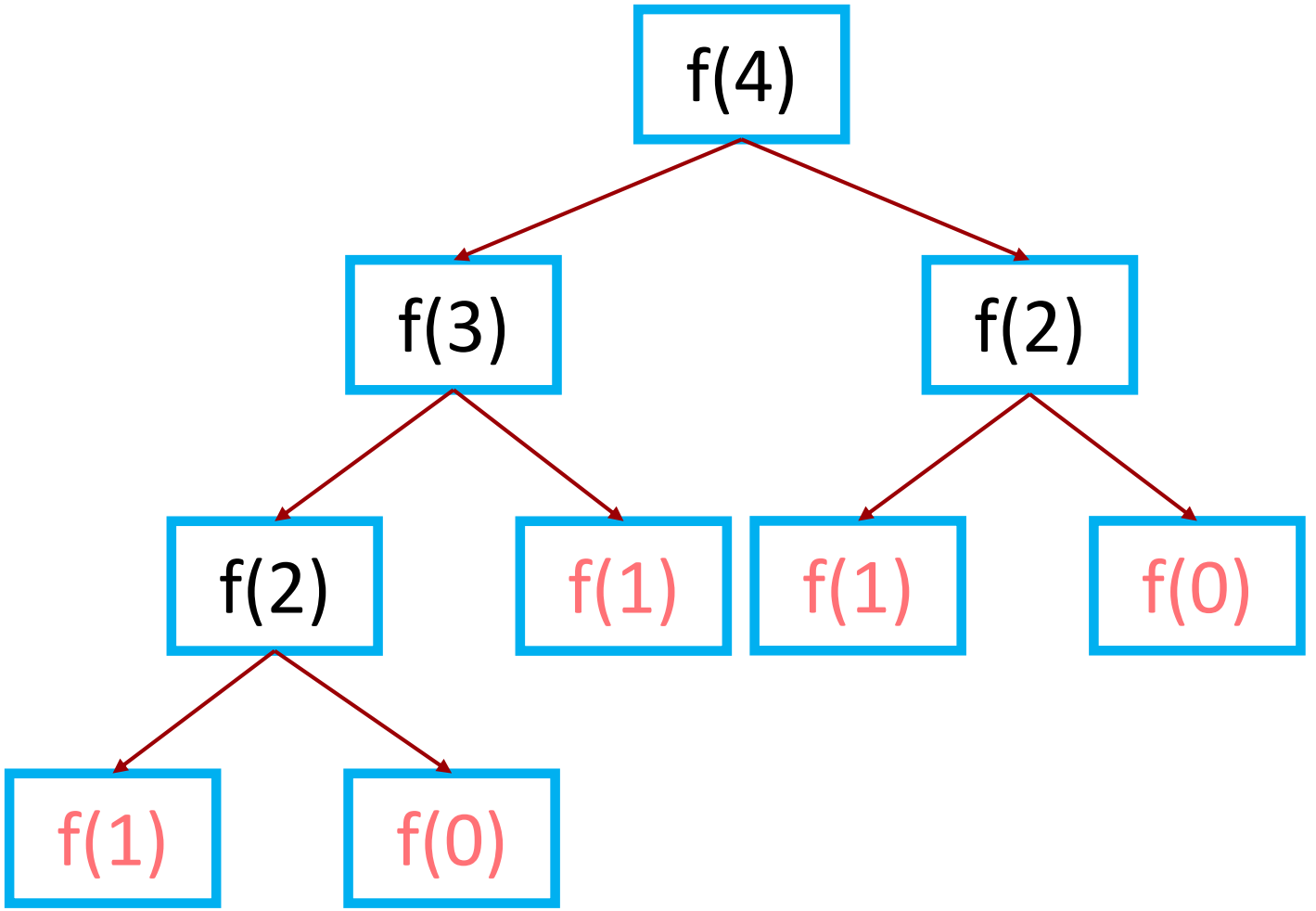
f(3):1

f(2):2

f(1):2

f(0):1

한번 계산해 놓은 값을 또 다시 계산하는 경우가 많다.



함수 호출 횟수

f(4):1

f(3):1

f(2):2

f(1):3

f(0):2

Dynamic programming

n이 4가 아니라 만약 n이 커지면 엄청나게 많은 호출이 이루어지게 된다.

```
f(0) : 4181회 호출
f(1) : 6765회 호출
f(2) : 4181회 호출
f(3) : 2584회 호출
f(4) : 1597회 호출
f(5) : 987회 호출
f(6) : 610회 호출
f(7) : 377회 호출
f(8) : 233회 호출
f(9) : 144회 호출
f(10) : 89회 호출
f(11) : 55회 호출
f(12) : 34회 호출
f(13) : 21회 호출
f(14) : 13회 호출
f(15) : 8회 호출
f(16) : 5회 호출
f(17) : 3회 호출
f(18) : 2회 호출
f(19) : 1회 호출
f(20) : 1회 호출
```

ex) n=20

즉 호출횟수가 지수적으로 커지게 되고, 함수 호출에 사용되는 시간과 메모리가 비효율적으로 사용되게 된다.

이런 문제를 어떻게 해결할까?

이미 구한 정답을 다시 또 구해야 할까?

불필요한 부분이 중복되고 있지는 않을까?

memoization을 활용 → 답을 저장해 놓자!

Memoization

이름부터 알 수 있듯이
우리가 저장해 놓고 싶은 값을 저장(메모)해 놓음으로써
중복 계산을 방지하는 기법!

어떤 부분 문제의 정답을 구해 놓았다면
그 문제의 정답을 배열에 저장해 놓자!

그 부분문제의 정답을 구하기 위해 다시 문제를 풀 필요가 없다는 소리

Fibonacci with Memoization

우선 부분문제들의 정보를 저장할 배열을 만들자!

fib[n]: 피보나치 수열의 n번째 수열을 저장

```
long long fib[100];
//fib[n]--> n번재 피보나치 수를 저장
```

fib을 우선 전부 -1로 초기화 한다.

```
for(int i=0;i<100;i++)fib[i]=-1;
//fib[n]이 안 구해져 있다면 -1의 값으로 초기화 되어 있을 것이다.
```

Fibonacci with Memoization

재귀함수에서 피보나치 수열을 구할 때 배열을 잘 활용해보자!

```
long long fibo(int n){
    if(fib[n]!=-1)return fib[n];
    //만약 fib[n]값이 구해져 있다면 바로 그 정답을 return 한다.
    if(n<2)return fib[n]=n;
    //f[0]=0,f[1]=1이므로

    return fib[n]=fibo(n-1)+fibo(n-2);
    //만약 구해져 있지 않고, 기저 케이스가 아니라면,
    //부분 문제를 호출하여 정답을 구하고, fib배열에 저장하여라.
}
```

#2748 피보나치 수 2

전체 소스 코드

(c언어)

```c
#include<stdio.h>
#include<stdlib.h>

long long fib[100];
long long fibo(int n) {
    if (fib[n] != -1)return fib[n];
    if (n < 2)return fib[n] = n;
    return fib[n] = fibo(n - 1) + fibo(n - 2);
}
int main() {
    int n;
    for(int i=0;i<100;i++)fib[i]=-1;
    scanf("%d", &n);
    printf("%lld", fibo(n));
}
```

Accepted

(c++)

```cpp
#include<iostream>
#include<cstring>

using namespace std;
using ll = long long;
ll fib[100], n;
ll fibo(int n) {
    if (fib[n] != -1)return fib[n];
    if (n < 2)return fib[n] = n;
    return fib[n] = fibo(n - 1) + fibo(n - 2);
}
int main() {
    for(int i=0;i<100;i++)fib[i]=-1;
    cin >> n;
    cout << fibo(n);
}
```

#15988 1,2,3,더하기 3

문제

정수 4를 1, 2, 3의 합으로 나타내는 방법은 총 7가지가 있다. 합을 나타낼 때는 수를 1개 이상 사용해야 한다.

- 1+1+1+1
- 1+1+2
- 1+2+1
- 2+1+1
- 2+2
- 1+3
- 3+1

정수 n이 주어졌을 때, n을 1, 2, 3의 합으로 나타내는 방법의 수를 구하는 프로그램을 작성하시오.

입력

첫째 줄에 테스트 케이스의 개수 T가 주어진다. 각 테스트 케이스는 한 줄로 이루어져 있고, 정수 n이 주어진다. n은 양수이며 1,000,000보다 작거나 같다.

출력

각 테스트 케이스마다, n을 1, 2, 3의 합으로 나타내는 방법의 수를 1,000,000,009로 나눈 나머지를 출력한다.

#15988 1,2,3,더하기 3

정수 4를 1, 2, 3의 합으로 나타내는 방법은 총 7가지가 있다. 합을 나타낼 때는 수를 1개 이상 사용해야 한다.

- 1+1+1+1
- 1+1+2
- 1+2+1
- 2+1+1
- 2+2
- 1+3
- 3+1

결국 우리가 마지막으로 쓰는 연산은 세가지 밖에 없다

+1

+2

+3

```
//dp[n]--->n을 만드는 방법의 수
dp[n]=dp[n-1]+dp[n-2]+dp[n-3];
```

#15988 1,2,3,더하기 3

```c
#include<stdio.h>

long long MOD = 1000000009, dp[1000010], t, n, a, b;
int main() {
    dp[1] = 1, dp[2] = 2, dp[3] = 4;
    //기저 케이스는 미리 구해놓는다.
    for (int i = 4; i <= 1000000; i++) {
        dp[i] += dp[i - 1] + dp[i - 2] + dp[i - 3];
        dp[i] %= MOD;
    }

    scanf("%d", &t);
    while (t--) {
        scanf("%d", &n);
        printf("%d\n", dp[n]);
    }
}
```

```cpp
#include<iostream>

using namespace std;
long long MOD = 1000000009, dp[1000010], t, n, a, b;
int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0), cout.tie(0);
    //많은 출력을 할 때 속도 향상

    dp[1] = 1, dp[2] = 2, dp[3] = 4;
    //기저 케이스는 미리 구해놓는다.
    for (int i = 4; i <= 1000000; i++) {
        dp[i] += dp[i - 1] + dp[i - 2] + dp[i - 3];
        dp[i] %= MOD;
    }
    cin >> t;
    while (t--) {
        cin >> n;
        cout << dp[n] << '\n';
    }
}
```

Accepted

Top-down vs Bottom-up

Top-down	Bottom-up
구하려 하는 문제를 작은 문제로 호출해가며 탐색	이미 알고있는 작은 문제부터 원하는 문제까지 탐색
재귀함수로 구현	반복문으로 구현

```cpp
#include<iostream>
#include<cstring>

using namespace std;
using ll = long long;
ll fib[100], n;
ll fibo(int n) {
    if (fib[n] != -1)return fib[n];
    if (n < 2)return fib[n] = n;
    return fib[n] = fibo(n - 1) + fibo(n - 2);
}
int main() {
    memset(fib, -1, sizeof(fib));
    cin >> n;
    cout << fibo(n);
}
```

```cpp
#include<iostream>

using namespace std;
long long MOD = 1000000009, dp[1000010], t, n, a, b;
int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0), cout.tie(0);
    //많은 출력을 할 때 속도 향상

    dp[1] = 1, dp[2] = 2, dp[3] = 4;
    //기저 케이스는 미리 구해놓는다.
    for (int i = 4; i <= 1000000; i++) {
        dp[i] += dp[i - 1] + dp[i - 2] + dp[i - 3];
        dp[i] %= MOD;
    }
    cin >> t;
    while (t--) {
        cin >> n;
        cout << dp[n] << '\n';
    }
}
```

dp는 어렵다

dp가 어려운 이유는?

1.사람에 따라서 발상적이다

활용 방법에 따라 풀이가 천차만별

즉 배열을 어떻게 활용하고, 부분문제를 무엇으로 설정할지에 따라 유형이 다양하고 어렵다.

ex) 구간 dp, 비트마스킹 dp, 등등...

2. dp 를 최적화하는 다양한 방법이 있다(optimization).

ex)CHT, knuth, 행렬제곱 등등...

#1757 달려달려

어제, 그리고 어제 어제 단체달리기를 두 번이나 하였다. 원장선생님의 이러한 하드 트레이닝으로 월드 학생들의 체력은 거의 박지성 수준이 되었다. 그래서 월드 학생들은 운동장을 도는데 정확히 N분에 완주할 수 있는 시간 안배능력까지 갖추게 되었다.

그래서 N분동안 학생들은 달릴지 아님 쉴지 결정하여야 한다. 그러나 학생들도 인간이기 때문에 계속 달릴 수는 없다. "지침 지수"라는 것이 있어서 1분을 달린다면 "지침 지수"는 1이 올라간다. 반대로 1분을 쉰다면 "지침 지수"는 1이 내려간다. 또한 이 "지침 지수"가 M보다 커지면 학생들은 더 이상 달릴 수가 없다.

아주 특이하게도 학생들은 시간에 따라 달릴 수 있는 거리가 다르다. 만약 i분에 달렸다면 D_i 만큼의 거리를 달릴 수 있다. (i분을 달렸다는 것이 아니라 i분이 되는 때에 달렸다는 뜻임) 또한 학생들이 쉬기 시작하면 지침지수가 0이 되기 전에는 다시 달릴 수가 없다.

물론 이 달리기가 끝나면 학생들은 다시 공부를 해야한다. 그렇기 때문에 달리기가 끝난다음 지침지수가 0이 되지 않는다면 맑은 정신으로 문제를 풀 수가 없기 때문에 달리기가 끝나면 지침지수는 0이 되어야 한다.

월드학생들이 최대한 멀리 갈 수 있는 거리를 구해보자.

입력

첫째 줄에 운동할 시간 N($1 \le N \le 10000$)과 M($1 \le M \le 500$)이 주어진다. 이어서 N개의 줄에 i분에 달릴수 있는 거리 D_i($1 \le D_i \le 1,000$)가 차례차례 주어진다.

출력

첫째 줄에 최대로 멀리 갈 수 있는 거리를 출력하라.

#1757 달려달려

가장 처음 해볼 수 있는 생각

dp를 안쓴다면?

매 초마다 달릴지 말지를 결정해서 2의 n제곱 가지 경우의 수에 대해서
다 따져보면 답은 나오겠다 but(n<=10000)

dp를 쓴다면?

우리가 필요한건 세가지 상태

✓ 몇 초에서?

✓ 지침지수가 몇이지?

✓ 달리고 있나?

#1757 달려달려

dp를 다음과 같이 정의하자

```
int dp[10005][505][2];
//dp[x][y][z]: x초에 y의 지침지수를 가지고 z는 뛰고 있는지를 나타낼 때, 달린 거리의 최댓값
```

그럼이제 x초의 dp 배열을 채워가기 위해서는
X-1초에 저장되어 있는 배열을 잘 참조하면 되지 않을까?

★기억해야할 점
학생들이 쉬기 시작하면 지침지수가 0이 되기 전에는 다시 달릴 수가 없다.
0일때 쉬면 지침지수는 그대로 0이다.

#1757 달려달려

✓ j가 1,0 이 아닐때

```
dp[i][j][1] = dp[i - 1][j - 1][1] + arr[i];
dp[i][j][0] = max(dp[i - 1][j + 1][1], dp[i - 1][j + 1][0]);
```

✓ j가 0일 때는 쉬면,지침지수는 -1이 아니라 그대로 0이다.(최솟값이 0이므로)

```
//j가 0일이면서 달리는 경우는 없다.
dp[i][j][0] = max({ dp[i - 1][j + 1][0], dp[i - 1][j + 1][1],dp[i - 1][j][0] });
```

✓ j가 1일 때는 쉬다가 달리는 것이 가능하다.

```
dp[i][j][1] = max(dp[i - 1][j - 1][1], dp[i - 1][j - 1][0]) + arr[i];
dp[i][j][0] = max(dp[i - 1][j + 1][1], dp[i - 1][j + 1][0]);
```

#1757 달려달려

```cpp
#include<iostream>
#include<algorithm>

using namespace std;
int dp[10000 + 5][500 + 5][2], arr[10001], n, m;
int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0), cout.tie(0);
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        cin >> arr[i];
    for (int i = 1; i <= n; i++)
        for (int j = 0; j <= m; j++) {
            if (j != 1) {
                if (j) {//j가 0이라면 dp[i-1][j-1]에 접근하지 못한다.
                    dp[i][j][1] = dp[i - 1][j - 1][1] + arr[i];
                    dp[i][j][0] = max(dp[i - 1][j + 1][1], dp[i - 1][j + 1][0]);
                }
                else {
                    //j가 0일이면서 달리는 경우는 없다.
                    dp[i][j][0] = max({ dp[i - 1][j + 1][0], dp[i - 1][j + 1][1],dp[i - 1][j][0] });
                    //j가 0일때 쉬면 다시 j가 0인 것을 생각해주자. 지침지수가 0보다 내려갈 수가 없으니까.
                }
            }
            else {
                dp[i][j][1] = max(dp[i - 1][j - 1][1], dp[i - 1][j - 1][0]) + arr[i];
                dp[i][j][0] = max(dp[i - 1][j + 1][1], dp[i - 1][j + 1][0]);
            }
        }
    cout << dp[n][0][0];
}
```

Accepted

#10982 행운쿠키 제작소

데브베이커리에서는 기념일을 맞아 직원들에게 행운쿠키를 선물하기로 하였다. 회사의 간식을 담당하는 철수는 나누어줄 행운 쿠키를 준비하는 역할을 맡게 되었다. 행운쿠키를 만들기 위해서는 N개의 행운반죽을 2개의 오븐을 이용해 구워야 한다. 각각의 행운반죽은 2개의 오븐 중 1개의 오븐에서만 구워져야 하며, 어떤 오븐에서 굽는지에 따라 구워지는데 걸리는 시간이 다르다. 각각의 오븐은 독립적으로 반죽을 구울 수 있으며, 오븐에 반죽을 넣거나 빼는데 걸리는 시간은 없다고 가정하자.

철수는 행운반죽을 모두 구워야 퇴근을 할 수 있다고 한다. 철수가 최대한 빨리 퇴근을 할 수 있도록 행운반죽을 모두 굽는데 걸리는 최소 시간을 구해주자.

입력

첫 번째 줄에 테스트 케이스의 수 T 가 주어진다.

각 케이스의 첫 번째 줄에 철수가 가지고 있는 행운반죽의 개수 N($1 \le N \le 1,000$)이 주어진다.

그 다음 N개의 줄에 각 행운반죽이 오븐 1에서 구워지는데 걸리는 시간 a_i, 오븐 2에서 구워지는데 걸리는 시간 b_i가 주어진다. ($1 \le a_i, b_i \le 100$)

출력

각 테스트 케이스마다 한 줄씩 철수가 행운반죽을 모두 굽는데 걸리는 최소 시간을 출력한다.

#10982 행운쿠키 제작소

naive한 접근

dp를 안쓴다면?

매 쿠키마다 a오븐에 넣을지 b오븐에 넣을지를 결정해서 브루트포스

2의 n제곱가지 경우 but(n<=1000)

dp를 어떻게 쓰지?

이 문제의 핵심 key

모든 쿠키를 다 구워야 한다는 것

즉 a오븐이든 b오븐이든 쿠키를 넣긴 넣어야 한다는 것

#10982 행운쿠키 제작소

dp[x]: a오븐에서 굽는데 사용한 시간이 x일때, b오븐에서 사용한 굽는 시간의 최솟값
모든 쿠키의 입력에 대해서 각각의 입력이 들어올 때마다 모든 dp값을 업데이트

```
while(n--){
    int time_a,time_b;
    //쿠키가 각각의 오븐에서 걸리는 시간
    cin>>time_a>>time_b;
    for(int i=100000;i>=0;i--){
        if(i>=time_a)dp[i]=min(dp[i]+time_b,dp[i-time_a]);
        else dp[i]+=time_b;
    }
}
```

O(N*100000) 이 걸리고 시간제한이 5초라 넉넉하다.

#10982 행운쿠키 제작소

```cpp
#include<iostream>
#include<algorithm>

using namespace std;
using ll = long long;
ll t, n, a, b, c, dp[100500],sum;
ll inf = 1e18;
int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0), cout.tie(0);
    cin >> t;
    while (t--) {
        cin >> n;
        fill(dp, dp + 100500, inf);
        dp[0] = 0;
        while (n--) {
            cin >> a >> b;
            for (int i = 100000; i >= 0; i--) {
                if (i>=a)dp[i] = min(dp[i] + b, dp[i - a]);
                else dp[i] += b;
            }
        }
        c = inf;
        for (ll i = 0; i < 100001; i++)
            c = min(c, max(dp[i], i));
        cout << c << '\n';
    }
}
```

Accepted

#추천문제

필수문제

2748	피보나치 수 2
15988	1, 2, 3 더하기 3
1463	1로 만들기
1757	달려달려
20500	Ezreal 여눈부터 가네 ㅈㅈ

Easy
Normal

2839	설탕 배달
2670	연속부분최대곱
2579	계단 오르기
11726	2×n 타일링
9461	파도반 수열
2407	조합
14494	다이나믹이 뭐예요?
12865	평범한 배낭
9251	LCS
13302	리조트
2698	인접한 비트의 개수
20544	공룡게임
11066	파일 합치기
11062	카드 게임
2629	양팔저울

Hard
Challenging

16117	실버런
14586	도미노 (Small)
19645	햄최몇?
19576	약수
20443	배드민턴 대회
10982	행운쿠키 제작소
12920	평범한 배낭 2
17840	피보나치 음악
2449	전구
10468	숫자뽑기게임
12928	트리와 경로의 길이
13448	SW 역량 테스트
1126	같은 탑
2337	트리 자르기