#1. 동적 계획법

나정휘 (jhnah917)
https://justiceHui.github.io/

동적 계획법

동적 계획법

- 복잡한 문제를 간단한 문제들로 나누고
- 간단한 문제들을 해결한 뒤
- 간단한 문제들의 답을 이용해 복잡한 문제의 답을 구함

떠오르는 질문 리스트

- 문제의 복잡성이 뭐지?
  - 큰 문제 / 작은 문제로 생각하면 편함
- 항상 가능하지는 않을 텐데...
  - 최적 부분 구조 (Optimal Substructure)
- 이게 효율적일까?
  - 중복되는 부분 문제 (Overlapping Subproblem)

동적 계획법

최적 부분 구조 (Optimal Substructure)

- 큰 문제의 최적해가 작은 문제의 최적해를 포함한다.
- 최적 부분 구조가 아니라면 작은 문제의 답을 이용해서 큰 문제의 답을 구할 수 없음

- ex) 피보나치 수열
  - 큰 문제: fibonacci(N)
  - 작은 문제: fibonacci(0), fibonacci(1)
  - fibonacci(N)은 a*fibonacci(0) + b*fibonacci(1)로 나타낼 수 있음
- ex) 거스름돈 문제
  - 큰 문제: 12560원 만들기
  - 작은 문제: 2560원 만들기
  - 12560원을 만드는 것은 2560원을 만들고 10000원권 지폐 한 장을 추가하면 됨

동적 계획법

중복되는 부분 문제 (Overlapping Subproblem)

- 작은 문제의 답을 여러 번 참조한다.
- 한 번 계산한 답을 저장하면, 다시 참조할 때 저장된 값을 사용하면 되므로 연산량 감소

- ex) 피보나치 수열
  - 큰 문제: fibonacci(N), N > 5
  - 작은 문제: fibonacci(5), fibonacci(4), fibonacci(3), ...
  - fibonacci(N)은 a*fibonacci(5) + b*fibonacci(4)로 나타낼 수 있음
    - 이때 fibonacci(5)와 fibonacci(4)를 각각 a, b번 호출할 텐데
    - 한 번 계산한 다음 결과를 저장하면 실행 시간을 많이 줄일 수 있음
- ex) 거스름돈 문제
  - 큰 문제: 7560원 만들기, 12560원 만들기, 62560원 만들기
  - 작은 문제: 2560원 만들기
  - 2560원을 만드는 방법을 여러 번 참조함

질문) 주어진 문제를 DP로 풀 수 있다는 것을 어떻게 알 수 있나요?

동적 계획법

동적 계획법 문제의 특징

- 큰 문제를 한 개 이상의 작은 문제로 분할할 수 있어야 함
- 큰 문제와 작은 문제를 동일한 방법으로 해결할 수 있어야 함
- 큰 문제의 최적해가 작은 문제들의 최적해들로 구성되어야 함

- 결론: Optimal Substructure, Overlapping Subproblem을 알아야 함

동적 계획법

동적 계획법 문제인지 판단하는 방법

- Optimal Substructure 성질을 만족함을 증명한다.
- 지금까지 문제를 풀어본 경험을 토대로 추측한다.
  - 최소/최대 비용, 트리, 냅색, 경우의 수, 기댓값, 확률, ...
- 왠지 DP일 것 같으니까 일단 믿어본다.

- 증명할 자신이 없으면 그냥 문제를 많이 풀어서 유형을 외우는 것이 편함
- solved.ac 기준 골드까지는 물량으로 밀 수 있음

동적 계획법

동적 계획법 문제인 건 알겠는데...

- 큰 문제와 작은 문제 간의 상관 관계(점화식)을 어떻게 찾지?

- 점화식을 정의한다.
  - 현재 "상태"를 잘 표현할 방법을 생각한다.
    - 배열 인덱스, 선택한 원소의 개수/합/곱(을 M으로 나눈 나머지) 등
  - (최적화) 다른 방식으로 표현할 수 없는지 생각한다.
    - ex) knapsack, 뒤에서 다룸
  - (최적화) 상태의 차원을 줄여도 온전하게 표현할 수 있는지 생각한다.
    - ex) 현재 상태를 (A+B, A, B)로 표현하는 경우, (A+B, A)만 사용해도 온전히 표현할 수 있음

- 점화 관계를 찾는다.
  - 현재 "상태"로 가기 바로 직전 "상태"는 무엇이 있을까?
  - (최적화) 다양한 DP 최적화 기법들(CHT, DnC Opt, Kitamasa, etc.)

동적 계획법 – 예시 1

BOJ 2748 피보나치 수 2

- n ≤ 90 번째 피보나치 수를 구하는 문제
- 재귀 함수의 계산 결과를 저장해서 중복 계산을 피함
  - 한 번도 계산하지 않은 값은 -1로 초기화
  - dp[n] ≠ -1 이면 f(n)을 계산한 적 있다는 뜻

- 시간 복잡도: O(N)

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

ll dp[91];
ll f(int n){
    if(n == 0 || n == 1) return n;
    ll &res = dp[n];
    if(res != -1) return res;
    return res = f(n-1) + f(n-2);
}

int main(){
    int n; cin >> n;
    for(int i=0; i<91; i++) dp[i] = -1;
    cout << f(n);
}
```

동적 계획법 – 예시 1

BOJ 2748 피보나치 수 2

- n ≤ 90 번째 피보나치 수를 구하는 문제
- 재귀 함수의 계산 결과를 저장해서 중복 계산을 피함
  - 한 번도 계산하지 않은 값은 -1로 초기화
  - dp[n] ≠ -1 이면 f(n)을 계산한 적 있다는 뜻

- 시간 복잡도: O(N)

- 반복문을 이용해서 점화식을 계산할 수도 있음
- 두 가지 방법 모두 알아야 함

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

ll dp[91];

int main(){
    int n; cin >> n;
    dp[0] = 0; dp[1] = 1;
    for(int i=2; i<=n; i++) dp[i] = dp[i-1] + dp[i-2];
    cout << dp[n];
}
```

동적 계획법 – 예시 2

BOJ 1463 1로 만들기

- 아래 세 가지 연산을 적절히 사용해서 X를 1로 만드는 최소 연산 횟수
  - X가 3의 배수라면 X ← X/3
  - X가 2의 배수라면 X ← X/2
  - X ← X-1

- f(x) ← f(x-1) + 1
- f(x) ← f(x/2) + 1 if $x \equiv 0 \pmod 2$
- f(x) ← f(x/3) + 1 if $x \equiv 0 \pmod 3$

- 시간 복잡도: O(N)

```cpp
#include <bits/stdc++.h>
using namespace std;

int D[1010101];
int f(int n){
    if(n == 1) return 0;
    int &res = D[n];
    if(res != -1) return res;
    res = f(n - 1) + 1;
    if(n % 2 == 0) res = min(res, f(n / 2) + 1);
    if(n % 3 == 0) res = min(res, f(n / 3) + 1);
    return res;
}

int main(){
    int n; cin >> n;
    for(int i=0; i<=n; i++) D[i] = -1;
    cout << f(n);
}
```

동적 계획법 – 예시 2

BOJ 1463 1로 만들기

- 아래 세 가지 연산을 적절히 사용해서 X를 1로 만드는 최소 연산 횟수
  - X가 3의 배수라면 X ← X/3
  - X가 2의 배수라면 X ← X/2
  - X ← X-1

- f(x) ← f(x-1) + 1
- f(x) ← f(x/2) + 1 if $x \equiv 0 \pmod 2$
- f(x) ← f(x/3) + 1 if $x \equiv 0 \pmod 3$

- 시간 복잡도: O(N)

```cpp
#include <bits/stdc++.h>
using namespace std;

int D[1010101];

int main(){
    int n; cin >> n;
    D[1] = 0;
    for(int i=2; i<=n; i++){
        D[i] = D[i-1] + 1;
        if(i % 2 == 0) D[i] = min(D[i], D[i/2] + 1);
        if(i % 3 == 0) D[i] = min(D[i], D[i/3] + 1);
    }
    cout << D[n];
}
```

동적 계획법 – 예시 3

BOJ 1912 연속합

- 수열이 주어지면 구간 합의 최댓값을 구하는 문제

- D[i] = i번째 수를 마지막 원소로 하는 구간 합의 최댓값
  - i번째 수 하나만 사용하면 D[i] = A[i]
  - i번째 수보다 앞에 있는 수도 사용할 때의 최댓값은 D[i-1] + A[i]
    - D[i-1]은 i-1번째 수로 끝나는 구간 중 합이 가장 크기 때문
  - 따라서 D[i] = max{ A[i], D[i-1] + A[i] }
  - D[i] = max{ 0, D[i-1] } + A[i]

- 시간 복잡도: O(N)

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, A[101010], D[101010];

int main(){
    ios_base::sync_with_stdio(false);cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=1; i<=N; i++) D[i] = max(0,D[i-1]) + A[i];
    int R = D[1];
    for(int i=1; i<=N; i++) R = max(R, D[i]);
    cout << R;
}
```

동적 계획법 – 예시 4

BOJ 11053 가장 긴 증가하는 부분 수열

- 가장 긴 증가하는 부분 수열(LIS)의 길이를 구하는 문제
  - 부분 수열: 수열의 원소 몇 개를 선택해서 순서를 유지한 채로 만든 새로운 수열
  - 증가하는 수열: A[i-1] < A[i]를 만족하는 수열

- D[i] = i번째 수를 마지막 원소로 하는 LIS의 길이
  - i번째 수만 사용하면 D[i] = 1
  - 앞에 있는 수도 사용하면...
  - A[j] < A[i]인 모든 j에 대해 D[i] = max D[j] + 1
    - A[i]보다 작은 수 뒤에 A[i]를 붙이는 방식

- 시간 복잡도: $O(N^2)$

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, A[1010], D[1010];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=1; i<=N; i++){
        D[i] = 1;
        for(int j=1; j<i; j++){
            if(A[j] < A[i]) D[i] = max(D[i], D[j] + 1);
        }
    }
    int R = 0;
    for(int i=1; i<=N; i++) R = max(R, D[i]);
    cout << R;
}
```

동적 계획법 – 예시 4

BOJ 11053 가장 긴 증가하는 부분 수열

- 가장 긴 증가하는 부분 수열(LIS)의 길이를 구하는 문제
  - 부분 수열: 수열의 원소 몇 개를 선택해서 순서를 유지한 채로 만든 새로운 수열
  - 증가하는 수열: A[i-1] < A[i]를 만족하는 수열

- D[i][j] = 1..i번째 수를 적당히 사용했을 때, 마지막 원소의 값이 j인 LIS의 길이
  - j ≠ A[i] 이면 D[i][j] = D[i-1][j]
  - j = A[i] 이면 k < j에 대해 D[i][j] = max D[i-1][k] + 1
    - 1..i-1번째 수를 적당히 사용해서 j보다 작은 수로 끝나는 증가 부분 수열을 만든 다음
    - 그 뒤에 A[i] = j를 붙이는 방식

- 시간 복잡도: O(NX)
- 공간 복잡도: O(NX)

동적 계획법 – 예시 4

BOJ 11053 가장 긴 증가하는 부분 수열

- 가장 긴 증가하는 부분 수열(LIS)의 길이를 구하는 문제
  - 부분 수열: 수열의 원소 몇 개를 선택해서 순서를 유지한 채로 만든 새로운 수열
  - 증가하는 수열: A[i-1] < A[i]를 만족하는 수열

- D[i][j] = 1..i번째 수를 적당히 사용했을 때, 마지막 원소의 값이 j인 LIS의 길이

- i마다 값이 바뀌는 위치는 1개(D[i][A[i]])밖에 없음
  - 굳이 이차원 배열을 모두 들고 다닐 필요 없음

- D[j] = 지금까지 본 수들을 적당히 사용했을 때, 마지막 원소의 값이 j인 LIS의 길이
  - 각 i마다 D[A[i]]를 적당히 갱신하면 됨

- 시간 복잡도: O(NX)
- 공간 복잡도: O(X)

동적 계획법 – 예시 4

BOJ 11053 가장 긴 증가하는 부분 수열

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, A[1010], D[1010];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=1; i<=N; i++){
        for(int j=0; j<A[i]; j++){
            D[A[i]] = max(D[A[i]], D[j] + 1);
        }
    }
    int R = 0;
    for(int i=1; i<=1000; i++) R = max(R, D[i]);
    cout << R;
}
```

동적 계획법 – 예시 5

BOJ 2293 동전 1

- n가지 종류의 동전을 적당히 사용해서 k원을 만드는 경우의 수

- k원을 만드는 방법
  - 1..i-1번째 동전으로 k-cA_i원을 만든 다음
  - A_i원 동전을 c개 사용
- D[i][j] : 1..i번째 동전으로 j원을 만드는 경우의 수
  - D[0][0] = 1
  - D[i][j] = sum D[i-1][j-cA_i]
    - $0 \le j - cA_i \le k, c \ge 0$
    - $0 \le c \le j/A_i$
- 시간 복잡도: O(NK log K)
- 공간 복잡도: O(NK), 메모리 초과

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, K, A[111], D[111][10101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    for(int i=1; i<=N; i++) cin >> A[i];
    D[0][0] = 1;
    for(int i=1; i<=N; i++){
        for(int j=0; j<=K; j++){
            for(int c=0; j-c*A[i]>=0; c++){
                D[i][j] += D[i-1][j-c*A[i]];
            }
        }
    }
    cout << D[N][K];
}
```

동적 계획법 – 예시 5

BOJ 2293 동전 1

- D[i][*]를 계산할 때 D[i-1][*]만 사용함
  - D[i][j] = sum D[i-1][j-cA$_i$] (0 ≤ c ≤ j/A$_i$)
  - D[n][k] 크기의 배열 대신 D[2][k] 만 사용해도 됨
  - D[i][*] 대신 D[i%2][*]를 사용

- 시간 복잡도: O(NK log K)
- 공간 복잡도: O(N+K)

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, K, A[111], D[2][10101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    for(int i=1; i<=N; i++) cin >> A[i];
    D[0][0] = 1;
    for(int i=1; i<=N; i++){
        for(int j=0; j<=K; j++) D[i%2][j] = 0; // init
        for(int j=0; j<=K; j++){
            for(int c=0; c<=j/A[i]; c++){
                D[i%2][j] += D[(i-1)%2][j-c*A[i]];
            }
        }
    }
    cout << D[N%2][K];
}
```

동적 계획법 – 예시 5

BOJ 2293 동전 1

- 매번 c를 전부 탐색하는 것은 비효율적

- 각 상태 전이에서 동전을 1개만 사용하는 방법
  - 1..i번째 동전으로 j-A_i원을 만든 다음 A_i원 동전 사용
  - D[i][j] = D[i-1][j] + D[i][j-A_i]
    - i번째 동전을 사용하는 경우 / 사용하지 않는 경우
    - j-A_i 음수 조심

- 시간 복잡도: O(NK)

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, K, A[111], D[2][10101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    for(int i=1; i<=N; i++) cin >> A[i];
    D[0][0] = 1;
    for(int i=1; i<=N; i++){
        for(int j=0; j<=K; j++){
            D[i%2][j] = D[(i-1)%2][j];
            if(j - A[i] >= 0) D[i%2][j] += D[i%2][j-A[i]];
        }
    }
    cout << D[N%2][K];
}
```

동적 계획법 – 예시 5

BOJ 2293 동전 1

- D[i][j] = D[i-1][j] + D[i][j-Ai]
  - D[i-1][j]를 D[i][j]로 복사한 다음, D[i][j]에 D[i][j-Ai]를 더함

  - 굳이 복사할 필요가 있을까?
  - 그냥 덮어쓰면 됨
    - D[j]를 계산하는 시점에
    - D[j-A_i]에 이미 i번째 동전을 사용한 결과가 반영되어 있음

- 시간 복잡도: O(NK)
- 공간 복잡도: O(N+K)

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, K, A[111], D[10101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    for(int i=1; i<=N; i++) cin >> A[i];
    D[0] = 1;
    for(int i=1; i<=N; i++){
        for(int j=0; j<=K; j++){
            if(j - A[i] >= 0) D[j] += D[j-A[i]];
        }
    }
    cout << D[K];
}
```

동적 계획법 – 예시 6

BOJ 2294 동전 2

- n가지 종류의 동전을 적당히 사용해서 k원을 만들기 위해 필요한 동전의 최소 개수

- D[i][j] : 1..i번째 동전으로 j원을 만들기 위해 필요한 최소 개수
  - D[0][0] = 0
  - D[i][j] = min{ D[i-1][j], D[i][j-A_i] + 1 }

- 동전 1과 동일한 방법으로 해결 가능
  - 시간 복잡도 O(NK)
  - 공간 복잡도 O(N+K)

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, K, A[111], D[10101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=0; i<10101; i++) D[i] = 1e9;
    D[0] = 0;
    for(int i=1; i<=N; i++){
        for(int j=0; j<=K; j++){
            if(j - A[i] >= 0) D[j] = min(D[j], D[j-A[i]]
+ 1);
        }
    }
    cout << (D[K] < 1e9 ? D[K] : -1);
}
```

동적 계획법 – 예시 7

BOJ 12865 평범한 배낭

- 무게가 W_i, 가격이 V_i인 물건 N개
- 최대 K만큼의 무게를 넣을 수 있는 배낭
- 배낭에 넣을 수 있는 물건들의 최대 가치
- 0/1 Knapsack Problem

- D[i][j]
  - 1..i번째 물건을 적당히 선택해서 무게의 합이 j일 때 가능한 가치의 최댓값
  - D[i][j] ← D[i-1][j] (i번째 물건을 선택하지 않는 경우)
  - D[i][j] ← D[i-1][j-W_i] + V_i (i번째 물건을 선택하는 경우)
  - D[i][*]를 계산할 때 D[i-1][*]만 사용하므로 D[N][K] 대신 D[2][K] 사용
- 시간 복잡도: O(NK)
- 공간 복잡도: O(N+K)

동적 계획법 – 예시 7

BOJ 12865 평범한 배낭

- D[i][j] = max(D[i-1][j], D[i-1][j-W_i] + V_i)
  - D[i-1][j]를 가져오는 것은 D[i-1][*]를 D[i][*]로 복사하는 것
  - 동전 1 문제처럼 덮어쓰는 방식으로 할 수 있을까?

- D[j] ← D[j-W_i] + V_i
  - 동전 1은 각 원소를 중복해서 사용할 수 있었지만 이 문제는 불가능
  - D[j]를 계산하는 시점에 D[j-W_i]에 i번째 물건을 반영하지 않은 결과가 있어야 함
  - j를 K부터 0까지 역순으로 보면 됨

  - for(int i=1; i<=N; i++) for(int j=K; j>=W[i]; j--) D[j] = max(D[j], D[j-W[i]] + V[i]);

동적 계획법 – 예시 7

BOJ 12865 평범한 배낭

- 만약 V_i가 작고 K가 크다면? (ex. $V_i \le 10$, $K \le 10^9$)
- D[i][j] = 1..i번째 물건을 적당히 선택해서 가격의 합을 j로 만들 수 있는 무게의 최솟값
- D[i][j] ≤ K를 만족하는 가장 큰 j가 정답

- 2013 KOI 지역 본선 고등부 4번

동적 계획법 – 예시 8

BOJ 9251 LCS

- 두 수열 A, B의 최장 공통 부분 수열의 길이를 구하는 문제
  - ex. ACAYKP, CAPCAK

- D[i][j] = A[1..i]와 B[1..j]의 최장 공통 부분 수열
  - A[i]와 B[j]가 같은 경우
    - A[1..i-1]과 B[1..j-1]의 최장 공통 부분 수열의 맨 뒤에 A[i]를 추가
      - D[i][j] ← D[i-1][j-1] + 1
  - A[i]와 B[j]가 다른 경우
    - A[1..i]와 B[1..j-1]의 최장 공통 부분 수열
      - D[i][j] ← D[i][j-1]
    - A[1..i-1]과 B[1..j]의 최장 공통 부분 수열
      - D[i][j] ← D[i-1][j]

- 시간 복잡도, 공간 복잡도: O(|A||B|)

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, M, D[1010][1010];
string A, B;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> A >> B;
    N = A.size(); M = B.size();
    A = "#" + A; B = "#" + B; // 0-based to 1-based

    for(int i=1; i<=N; i++){
        for(int j=1; j<=M; j++){
            if(A[i] == B[j]) D[i][j] = D[i-1][j-1] + 1;
            else D[i][j] = max(D[i-1][j], D[i][j-1]);
        }
    }
    cout << D[N][M];
}
```

동적 계획법 – 예시 9

BOJ 12852 1로 만들기 2

- 최소 횟수로 1을 만드는 과정을 출력하는 문제

- f(x)가 최소가 되는 바로 직전 상태 P[x]를 저장
  - ex) P[2] = 1, P[4] = 2, P[5] = 4, P[8] = 4
- x가 1이 될 때까지 P[x]를 따라가면 됨
  - 8 → 4 → 2 → 1

- 시간 복잡도: O(N)

```cpp
#include <bits/stdc++.h>
using namespace std;

int D[1010101], P[1010101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int n; cin >> n;
    D[1] = 0; P[1] = -1;
    for(int i=2; i<=n; i++){
        D[i] = D[i-1] + 1;
        P[i] = i - 1;
        if(i % 2 == 0 && D[i] > D[i/2] + 1){
            D[i] = D[i/2] + 1;
            P[i] = i / 2;
        }
        if(i % 3 == 0 && D[i] > D[i/3] + 1){
            D[i] = D[i/3] + 1;
            P[i] = i / 3;
        }
    }
    cout << D[n] << "\n";
    for(int i=n; i!=-1; i=P[i]) cout << i << " ";
}
```

동적 계획법 – 예시 10

BOJ 14002 가장 긴 증가하는 부분 수열 4

- 가장 긴 증가하는 부분 수열(LIS)을 구하는 문제

- D[i] = i번째 수를 마지막 원소로 하는 LIS의 길이
- D[i] = max D[j] + 1
  - 편의상 A[0] = -inf, D[0] = 0이라고 정의하자.

- 각 i마다 D[j]가 최대인 j를 P[i]로 저장
- D[*]가 최대인 지점부터 시작해서 P를 따라가면 됨

- 시간 복잡도: $O(N^2)$

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, A[1010], D[1010], P[1010];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=1; i<=N; i++){
        for(int j=0; j<i; j++){
            if(A[j] < A[i] && D[i] < D[j] + 1){
                D[i] = D[j] + 1;
                P[i] = j;
            }
        }
    }

    int pos = 1;
    for(int i=1; i<=N; i++) if(D[i] > D[pos]) pos = i;
    cout << D[pos] << "\n";

    vector<int> track;
    for(int i=pos; i; i=P[i]) track.push_back(A[i]);
    reverse(track.begin(), track.end());
    for(auto i : track) cout << i << " ";
}
```

동적 계획법 – 예시 11

BOJ 9252 LCS 2

- 두 수열 A, B의 최장 공통 부분 수열을 구하는 문제

- D[i][j] = A[1..i]와 B[1..j]의 최장 공통 부분 수열
- D[i][j]를 구성하는 방법
  - A[i]와 B[j]가 같으면 D[i][j] = D[i-1][j-1] + 1
  - A[i]와 B[j]가 다르면 D[i][j] = max{ D[i][j-1], D[i-1][j] }

- 역추적
  - D[N][M]부터 시작해서, D[i][j]의 값이 왔던 곳으로 따라가면 됨

  - A[i] = B[j]이면 정답에 A[i]를 추가한 다음 D[i-1][j-1]로 이동
  - A[i] ≠ B[j]이면 D[i-1][j]과 D[i][j-1] 중 더 큰 곳으로 이동

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, M, D[1010][1010];
string A, B;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> A >> B;
    N = A.size(); M = B.size();
    A = "#" + A; B = "#" + B; // 0-based to 1-based

    for(int i=1; i<=N; i++){
        for(int j=1; j<=M; j++){
            if(A[i] == B[j]) D[i][j] = D[i-1][j-1] + 1;
            else D[i][j] = max(D[i-1][j], D[i][j-1]);
        }
    }

    string R;
    for(int i=N, j=M; i>0 && j>0; ){
        if(A[i] == B[j]) R += A[i], i--, j--;
        else if(D[i-1][j] > D[i][j-1]) i--;
        else j--;
    }
    reverse(R.begin(), R.end());
    cout << R.size() << "\n" << R;
}
```

#2. 오토마타

나정휘 (jhnah917)
https://justiceHui.github.io/

목차

오토마타

상태 (State)

- 동적 계획법은 문제의 상황을 “상태”로 표현한 다음
- 상태들 간의 전이를 효율적으로 계산하는 방법

- 어떤 작업의 중간 과정을 잘 나타내는 상태들과
- 어떤 이벤트가 발생했을 때 상태의 변화를 관찰하는 것

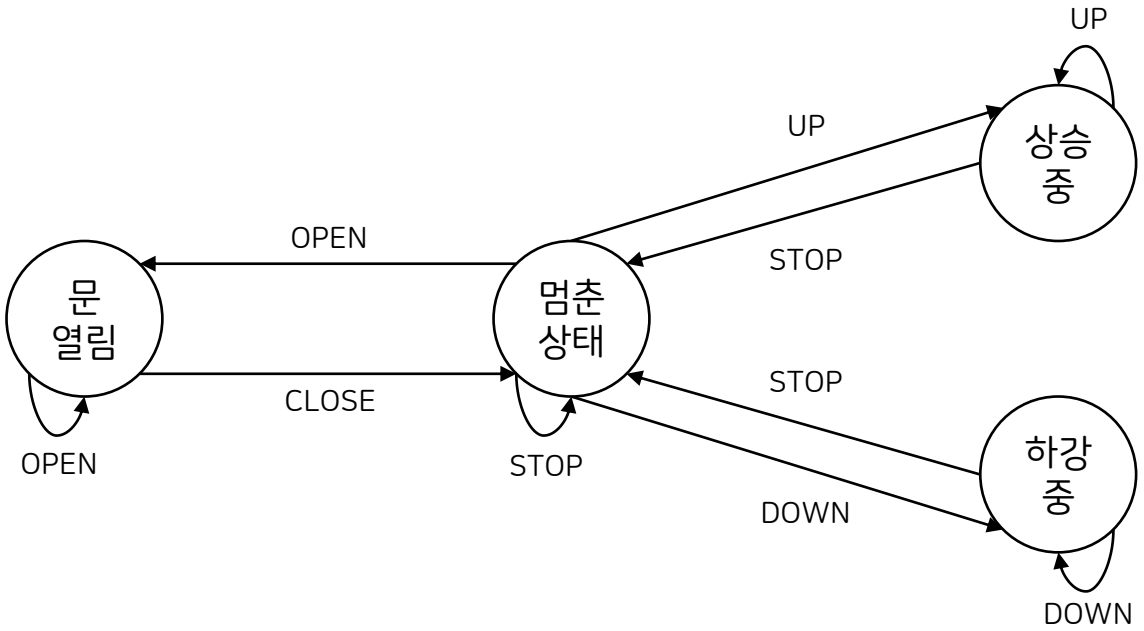
- ex. 0/1 Knapsack Problem
  - (n, k) := 1..n번째 물건을 적당히 선택했을 때 무게의 합이 k인 상태
  - n번째 원소를 선택하면 (n-1, k-W_n) → (n, k)
  - n번째 원소를 선택하지 않으면 (n-1, k) → (n, k)

오토마타

오토마타

- 유한한 개수의 내부 상태를 갖고 있음
- 유한 또는 무한한 크기의 저장 공간을 갖고 있을 수도 있음
- 입력이 주어지면 미리 정해져 있는 방식대로 상태를 전이해서 출력을 내놓는 장치

- ex. 엘리베이터 (상태 4개, 저장 공간 없음)
  - 가능한 입력: OPEN, CLOSE, UP, DOWN, STOP



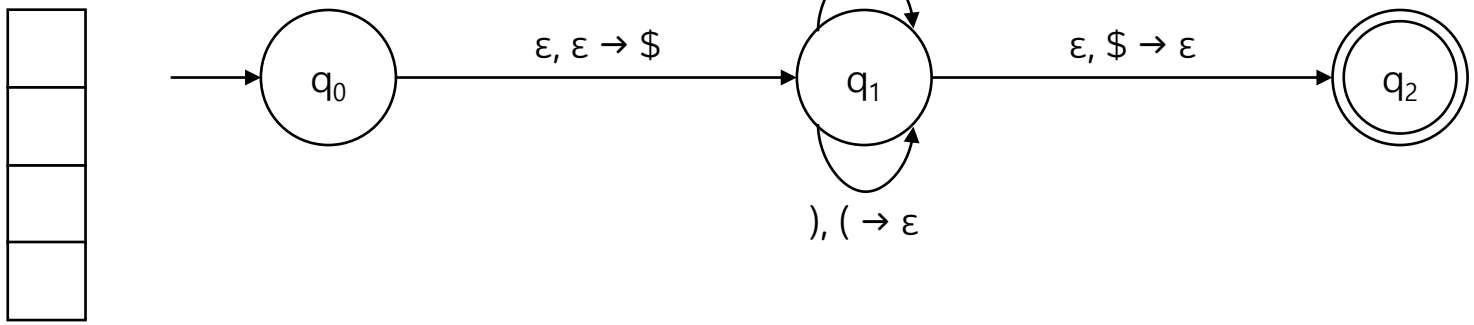
오토마타

오토마타

- ex. 올바른 괄호 문자열 판별 (상태 3개, 스택 형태의 메모리)
  - 가능한 입력: '(', ')'

  - input : ( ) (( ))

  - stack :

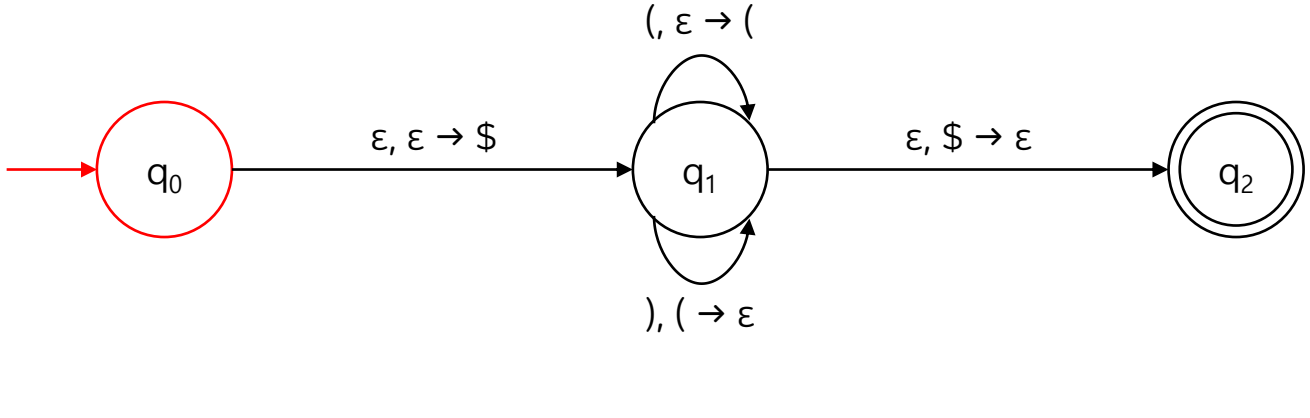


오토마타

오토마타

- ex. 올바른 괄호 문자열 판별 (상태 3개, 스택 형태의 메모리)
  - 가능한 입력: '(', ')'

- input : ( ) (( ))
- stack :

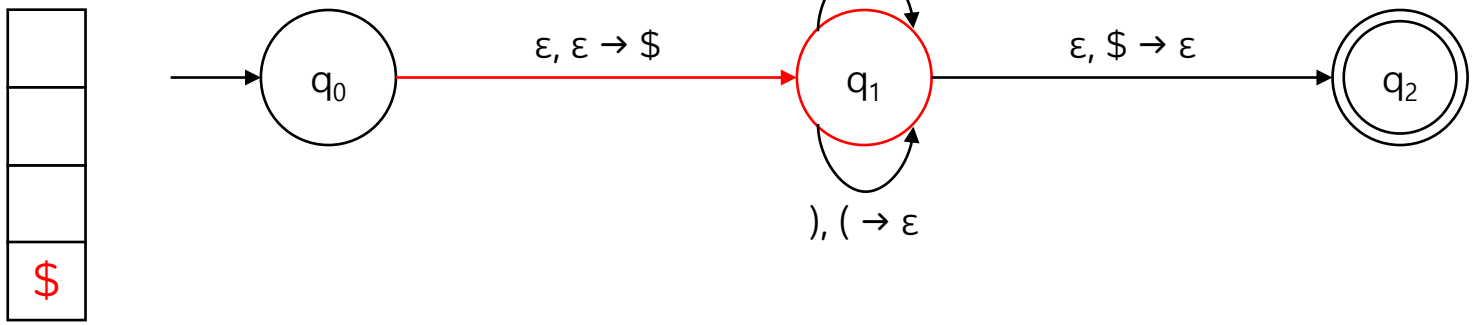


오토마타

오토마타

- ex. 올바른 괄호 문자열 판별 (상태 3개, 스택 형태의 메모리)
  - 가능한 입력: '(', ')'

- input : ( ) (( ))
- stack :

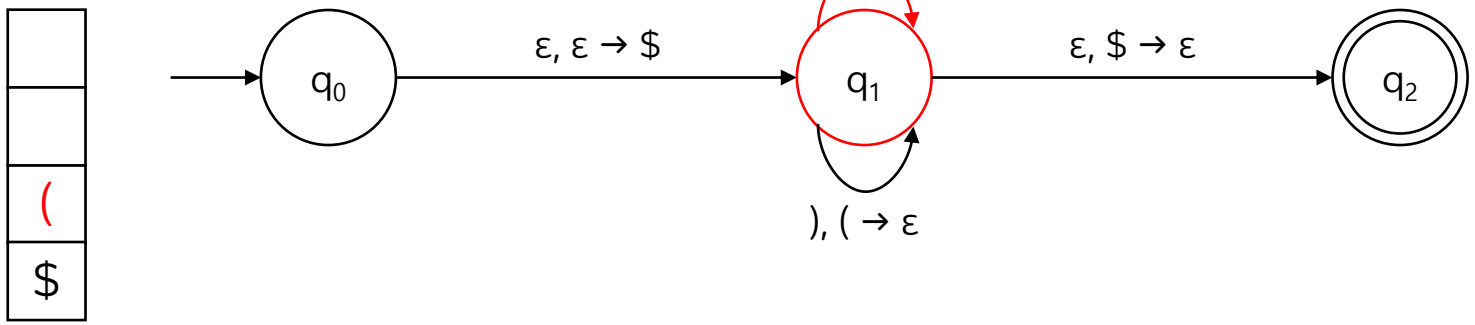


오토마타

오토마타

- ex. 올바른 괄호 문자열 판별 (상태 3개, 스택 형태의 메모리)
  - 가능한 입력: '(', ')'

- input : ( ) (( ))
- stack :

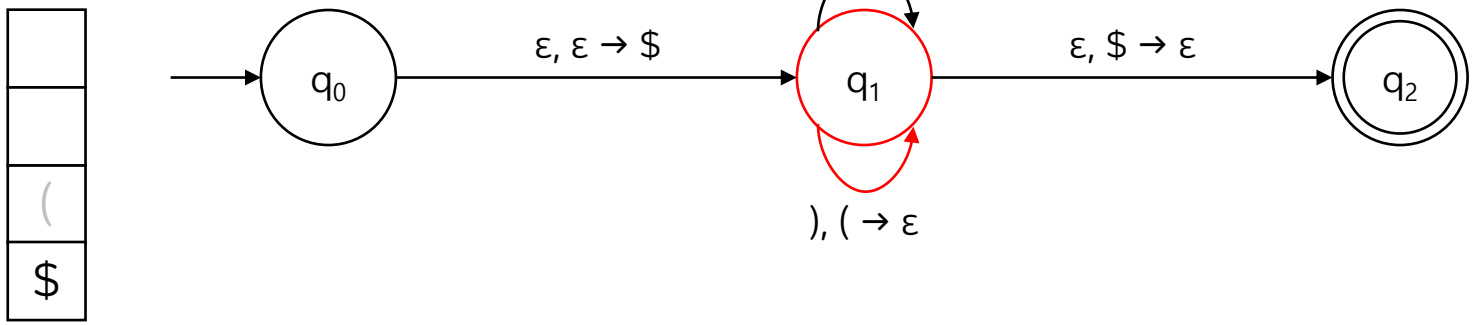


오토마타

오토마타

- ex. 올바른 괄호 문자열 판별 (상태 3개, 스택 형태의 메모리)
  - 가능한 입력: '(', ')'

- input : ( ) (( ))
- stack :

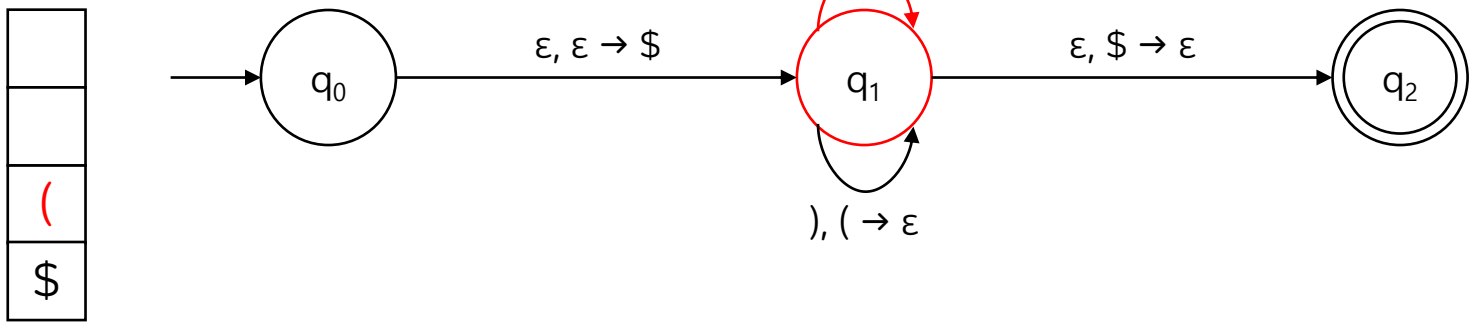


오토마타

오토마타

- ex. 올바른 괄호 문자열 판별 (상태 3개, 스택 형태의 메모리)
  - 가능한 입력: '(', ')'

- input : ( ) (( ))
- stack :

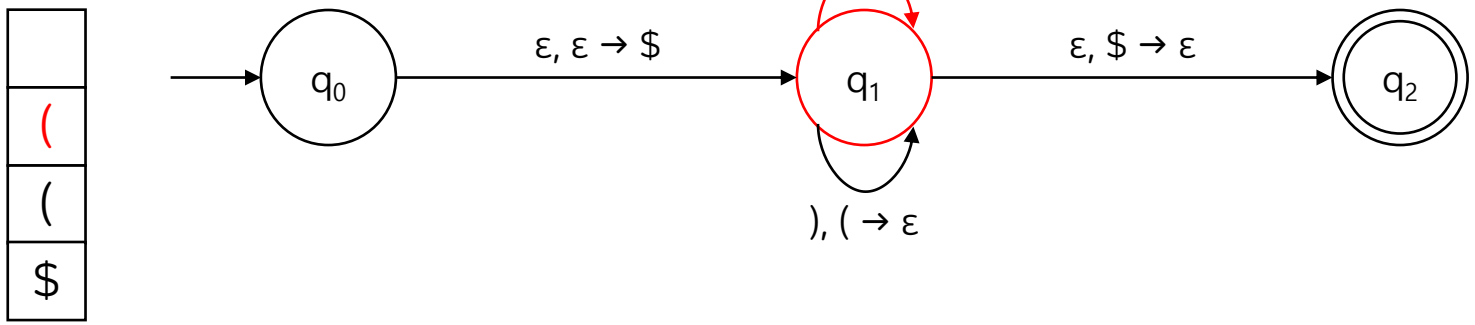


오토마타

오토마타

- ex. 올바른 괄호 문자열 판별 (상태 3개, 스택 형태의 메모리)
  - 가능한 입력: '(', ')'

- input : ( ) (( ))
- stack :

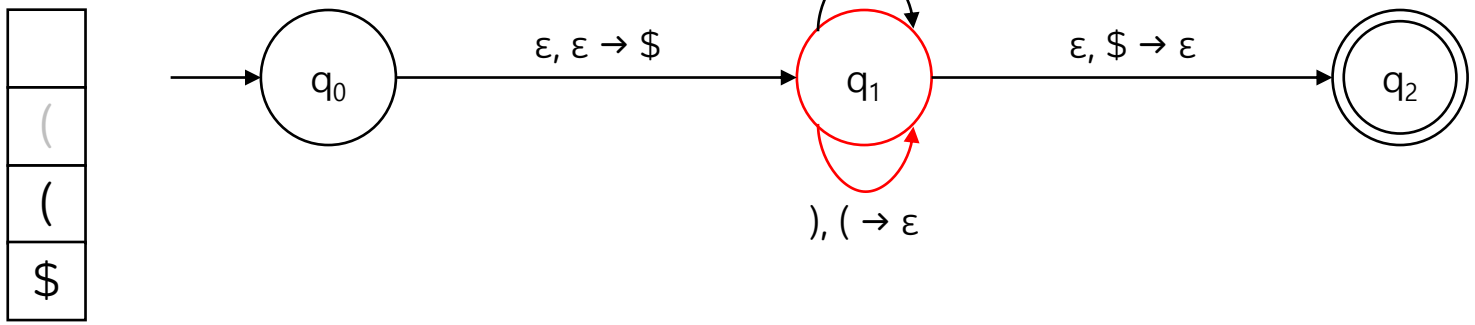


오토마타

오토마타

- ex. 올바른 괄호 문자열 판별 (상태 3개, 스택 형태의 메모리)
  - 가능한 입력: '(', ')'

- input : ( ) (( ))
- stack :

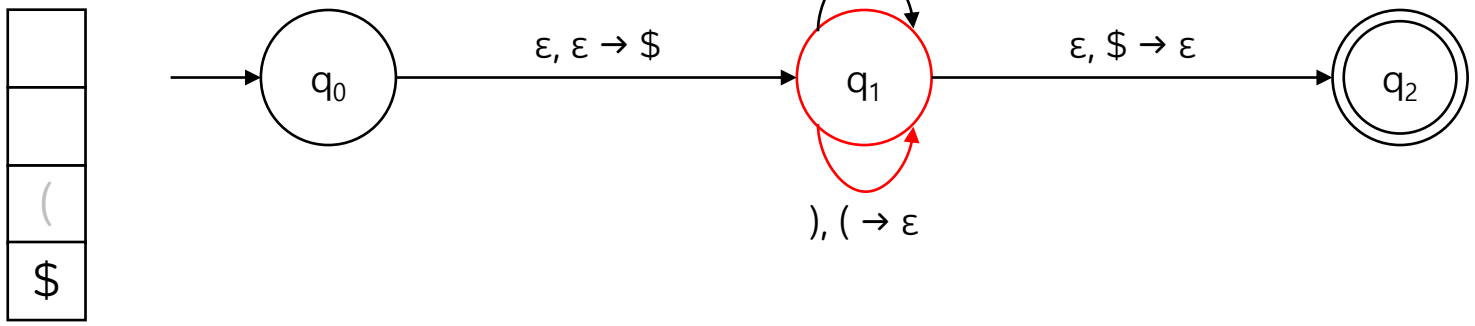


오토마타

오토마타

- ex. 올바른 괄호 문자열 판별 (상태 3개, 스택 형태의 메모리)
  - 가능한 입력: '(', ')'

- input : ( ) (( ))
- stack :

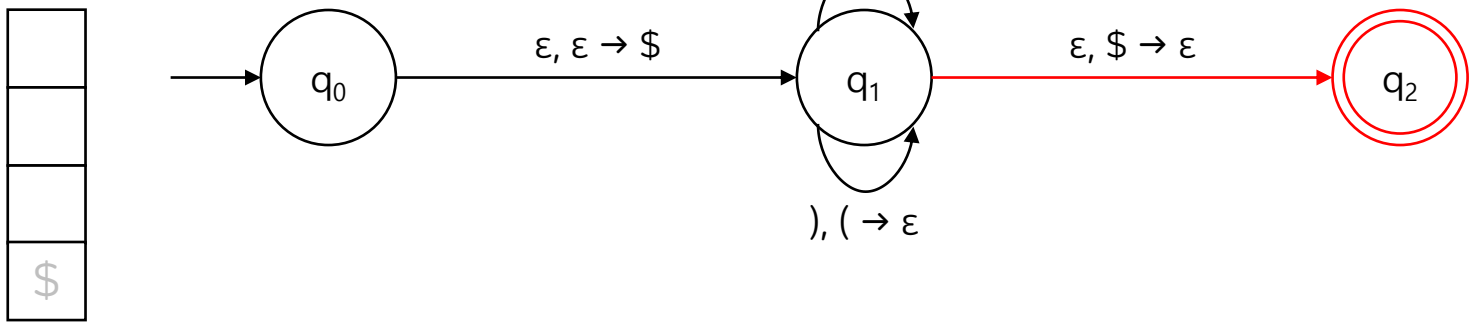


오토마타

오토마타

- ex. 올바른 괄호 문자열 판별 (상태 3개, 스택 형태의 메모리)
  - 가능한 입력: '(', ')'

- input : ( ) (( ))
- stack :

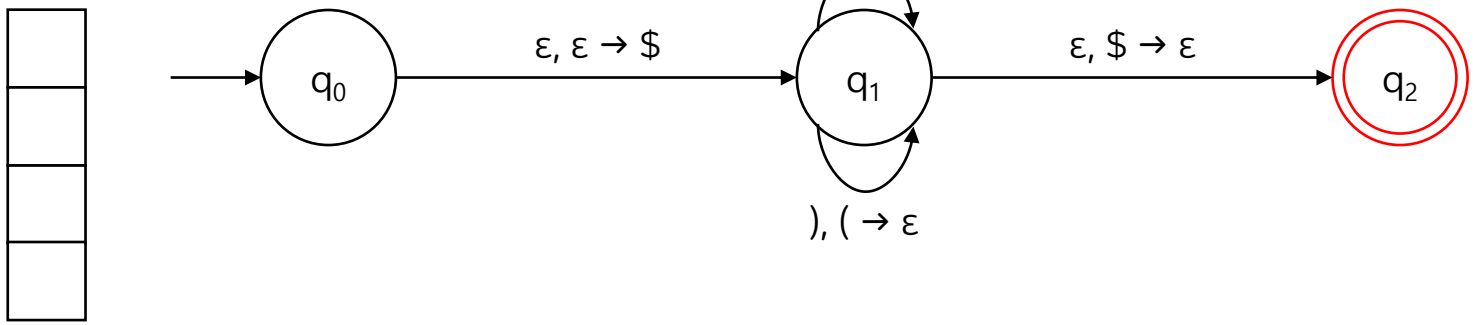


오토마타

오토마타

- ex. 올바른 괄호 문자열 판별 (상태 3개, 스택 형태의 메모리)
  - 가능한 입력: '(', ')'

- input : ( ) (( ))
- stack :



용어 정의

용어 정의

- 알파벳(alphabet): symbol들로 구성된 유한 집합
- 문자열(string): 길이가 유한한 symbol들의 나열
  - 알파벳 Σ에서 정의된 모든 문자열을 포함하는 집합을 Σ^*라고 부름
- 언어(language): Σ^*의 부분 집합, 문자열들의 집합

- 예시
  - 알파벳이 $\Sigma_1 = \{a, b, c, \cdots, y, z\}$이면 $\text{abc}, \text{sccc}, \text{jhnah}$와 같은 문자열을 만들 수 있음
  - 알파벳이 $\Sigma_2 = \{0,1\}$이면 $L_2 = \{w \mid w \text{ starts with } 010\}$과 같은 언어를 정의할 수 있음
  - L_2에는 $010, 0101, 010010$과 같은 문자열을 포함함

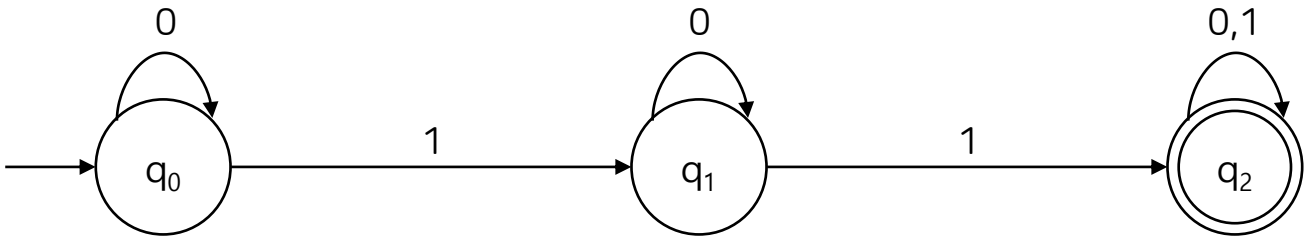
DFA

결정적 유한 오토마타 (Deterministic Finite Automata)

- 문자열을 읽어서 accept 또는 reject 시키는 장치
- 유한한 개수의 내부 상태가 있음
- 문자를 앞에서부터 한 글자씩 읽으면서, 읽은 문자에 따라 정해진 대로 상태를 전이함

- ex. 1이 2개 이상 있는 문자열만 accept하는 DFA
  - 시작점이 없는 화살표는 정확히 1개 존재 → 시작 상태
  - 두 겹으로 그려진 원은 종료 상태, 여러 개 존재할 수 있음

  - 문자열을 모두 읽었을 때 종료 상태 중 한 곳에 도달하면 accept, 그렇지 않으면 reject

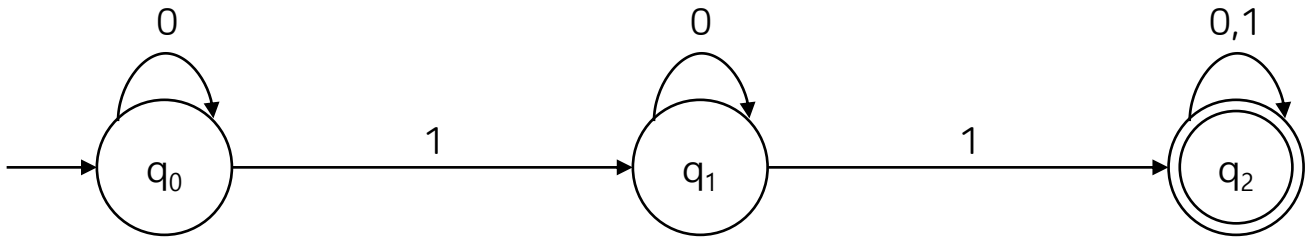


DFA

결정적 유한 오토마타

- DFA는 $M = (Q, \Sigma, \delta, q_0, F)$와 같이 5개의 원소로 이루어진 튜플로 표현함
  - Q: 상태들의 집합
  - Σ: 알파벳
  - $\delta: Q \times \Sigma \to Q$: 전이 함수, $\delta(p, a) \to q$는 현재 상태가 p인 상황에서 a를 읽으면 q로 간다는 뜻
  - $q_0 \in Q$: 시작 상태
  - $F \subseteq Q$: 종료 상태
- M이 accept 시키는 모든 문자열의 집합을 $L(M)$이라고 부름

- 아래의 DFA는 $M_1 = (\{q_0, q_1, q_2\}, \{0,1\}, \delta_1, q_0, \{q_2\})$로 나타낼 수 있음
  - 1이 2개 이상인 문자열만 accept 시키는 DFA
  - $L(M_1) = \{s \mid n_1(s) \geq 2\}$



δ_1	0	1
q_0	q_0	q_1
q_1	q_1	q_2
q_2	q_2	q_2

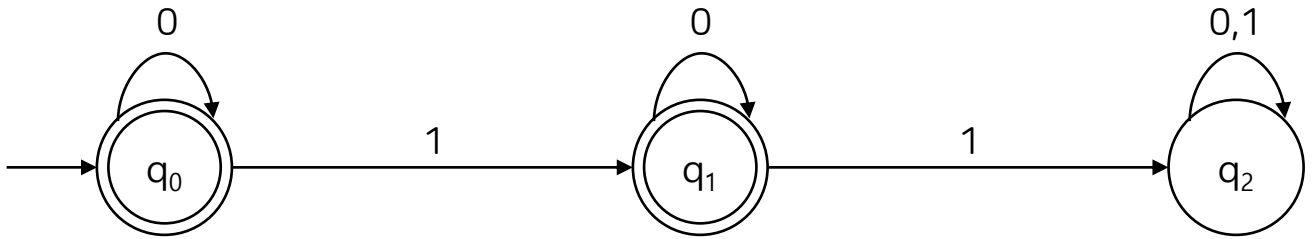
DFA

결정적 유한 오토마타

- DFA는 $M = (Q, \Sigma, \delta, q_0, F)$와 같이 5개의 원소로 이루어진 튜플로 표현함
  - Q: 상태들의 집합
  - Σ: 알파벳
  - $\delta: Q \times \Sigma \to Q$: 전이 함수, $\delta(p, a) \to q$는 현재 상태가 p인 상황에서 a를 읽으면 q로 간다는 뜻
  - $q_0 \in Q$: 시작 상태
  - $F \subseteq Q$: 종료 상태
- M이 accept 시키는 모든 문자열의 집합을 $L(M)$이라고 부름

- 아래의 DFA는 $M_2 = (\{q_0, q_1, q_2\}, \{0,1\}, \delta_2, q_0, \{q_0, q_1\})$로 나타낼 수 있음
  - 1이 2개 미만인 문자열만 accept 시키는 DFA
  - $L(M_2) = \{s \mid n_1(s) < 2\}$

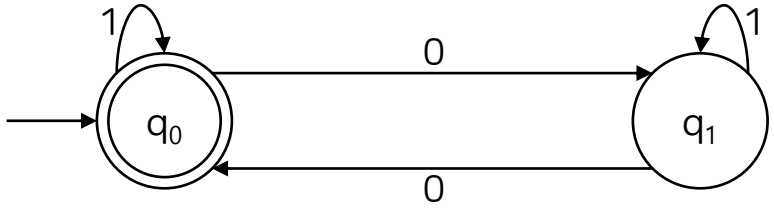
δ_2	0	1
q_0	q_0	q_1
q_1	q_1	q_2
q_2	q_2	q_2



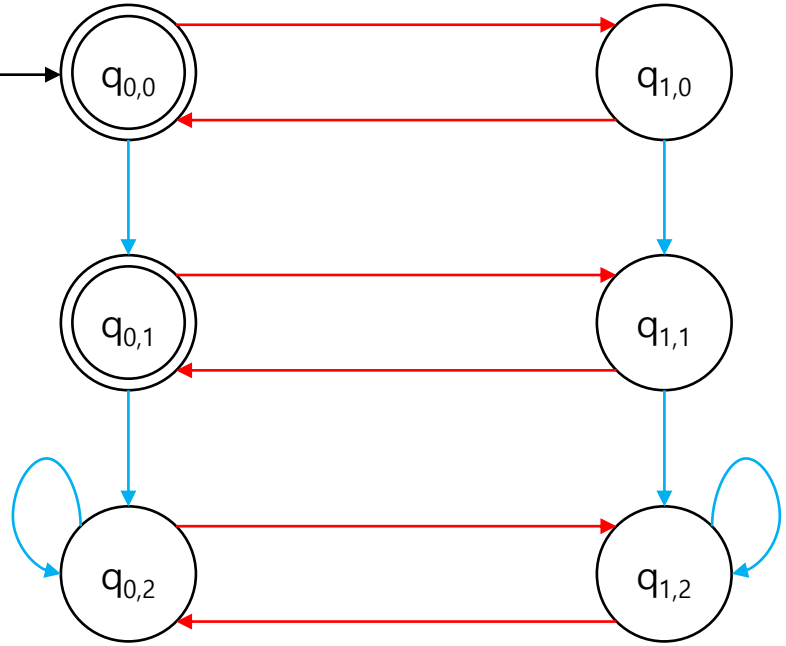
DFA

결정적 유한 오토마타

- 0이 짝수 개 있는 문자열만 accept하는 DFA



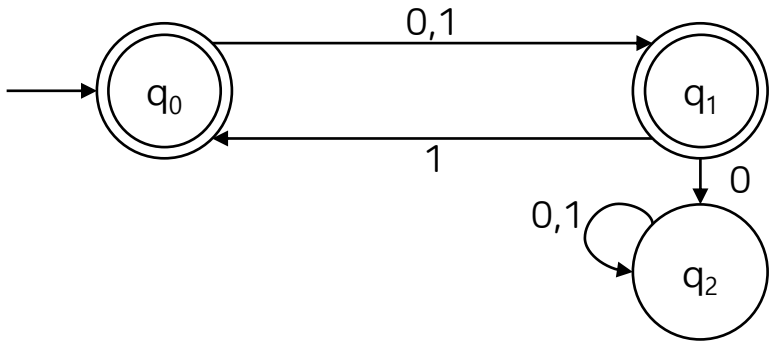
- 0이 짝수 개, 1이 2개 미만 있는 문자열만 accept하는 DFA



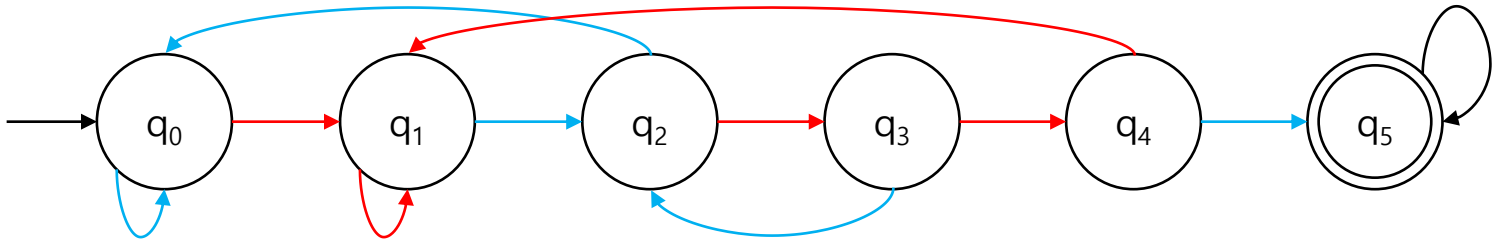
DFA

결정적 유한 오토마타

- 모든 짝수 번째 자리가 1인 문자열만 accept하는 DFA



- 01001을 포함하는 문자열만 accept하는 DFA
  - q_i = 01001과 앞에서부터 i글자 매칭된 상태
  - 이 방법을 잘 활용한 알고리즘이 KMP



DFA

결정적 유한 오토마타

- 십진법으로 해석했을 때 K의 배수인 문자열만 accept하는 DFA
  - q_x = 현재까지 읽은 수를 K로 나눈 나머지가 x인 상태
  - $M = (\{q_0, q_1, \cdots, q_{K-1}\}, \{0,1,2,\cdots,9\}, \delta, q_0, \{q_0\})$
  - $\delta(q_x, y) = q_{(10x+y)\%K}$

```cpp
int string_modulo_k(const string &s, int k){
    int res = 0;
    for(auto i : s){
        res = (res * 10 + i - '0') % k;
    }
    return res;
}
```

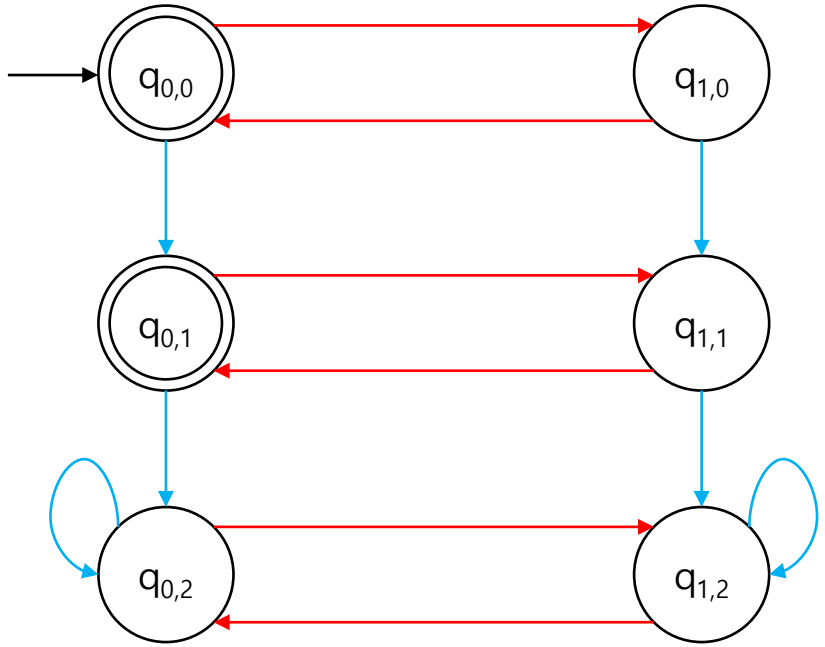
DP

동적 계획법에서의 활용

- 길이가 n이면서 0이 짝수 번, 1이 최대 1번 등장하는 binary string의 개수를 세는 문제
  - D(n, x, y) = 길이가 n이면서 $q_{x,y}$에서 종료하는 binary string의 개수

```
int N, D[SZ][2][3];

int main(){
    cin >> N;
    D[0][0][0] = 1;
    for(int i=0; i<N; i++){
        for(int j=0; j<2; j++){
            for(int k=0; k<3; k++){
                D[i+1][(j+1)%2][k] += D[i][j][k];
                D[i+1][j][min(k+1,2)] += D[i][j][k];
            }
        }
    }
    return D[N][0][0] + D[N][0][1];
}
```



DP

동적 계획법에서의 활용

- 길이가 n이고 이진법으로 해석했을 때 9의 배수인 binary string의 개수
  - D(n, m) = 길이가 n이고 9로 나눈 나머지가 m인 binary string의 개수
  - 사실 ((1 << n) - 1) / 9 + 1 출력해도 됨

```
int N, D[101010][9];

int main(){
    cin >> N;
    D[0][0] = 1;
    for(int i=0; i<N; i++){
        for(int j=0; j<9; j++){
            for(int k=0; k<2; k++) D[i+1][(j*2+k)%9] += D[i][j];
        }
    }
    return D[N][0];
}
```

DP

동적 계획법에서의 활용

- 길이가 n인 binary string이 주어졌을 때, 최소 개수로 수정해서 9의 배수가 되도록 수정
  - D(n, m) = 1..n번째 글자를 적당히 수정해서 9로 나눈 나머지가 m이 되도록 만드는 최소 수정 횟수

```
int N, D[101010][9];
string S;

int main(){
    cin >> N >> S;
    for(int i=0; i<=N; i++) for(int j=0; j<9; j++) D[i][j] = 1e9;
    D[0][0] = 0;
    for(int i=0; i<N; i++){
        for(int j=0; j<9; j++){
            int c = S[i] - '0';
            D[i+1][(j*2+c)%9] = min(D[i+1][(j*2+c)%9], D[i][j]);
            D[i+1][(j*2+1-c)%9] = min(D[i+1][(j*2+1-c)%9], D[i][j] + 1);
        }
    }
    return D[N][0];
}
```

#3. 경우의 수

나정휘 (jhnah917)
https://justiceHui.github.io/

목차

기본 지식

순열

순열: n개의 원소 중 서로 다른 k개의 원소를 선택해서 나열하는 방법의 수

- $n * (n-1) * \dots * (n-k+1) = n! / (n-k)!$
- 각 상자에 최대 1개의 공이 들어가도록 k개의 공을 n개의 상자에 분배하는 방법의 수
- 고등학생 때는 ${}_nP_k$ 라고 표현했음

중복 순열: n개의 원소 중 k개의 원소를 선택해서 나열하는 방법의 수

- 중복을 허용해서 k개를 선택하고 나열하는 방법
- $n * n * \dots * n = n^k$
- k개의 공을 n개의 상자에 자유롭게 분배하는 방법의 수

조합

조합: n개의 원소 중 서로 다른 k개의 원소를 선택하는 방법의 수

- {n * (n-1) * … * (n-k+1)} / {1 * 2 * … * k} = n! / (n-k)!k!
- 또는 C(n, k) = C(n-1, k-1) + C(n-1, k)
- 2차원 격자에서 (0, 0) → (n, m) 최단 경로의 개수는 (n+m)! / n!m! = C(n+m, n)
  - n+m번의 이동 중 세로 방향으로 이동할 n번의 위치를 정하는 경우의 수

중복 조합: n개의 원소 중 k개의 원소를 선택하는 방법의 수

- k개의 원소가 들어갈 자리를 만든 다음, n-1개의 칸막이를 원소와 원소 사이에 배치
- k+(n-1)개의 자리 중 칸막이가 들어갈 n-1개의 자리를 정하는 경우의 수
- C(n+k-1, n-1) = C(n+k-1, k)

포함 배제의 원리

교집합의 크기를 구할 수 있을 때 합집합의 크기를 구하는 방법

- $|A \cup B| = |A| + |B| - |A \cap B|$
- $|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |B \cap C| - |C \cap A| + |A \cap B \cap C|$
- …

포함 배제의 원리

- 유한집합인 전체집합 U의 부분집합 $A_1, A_2, \cdots, A_n$의 합집합의 원소의 개수

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{\emptyset \neq I \subset [n]} (-1)^{|I|-1} \left|\bigcap_{i \in I} A_i\right|$$

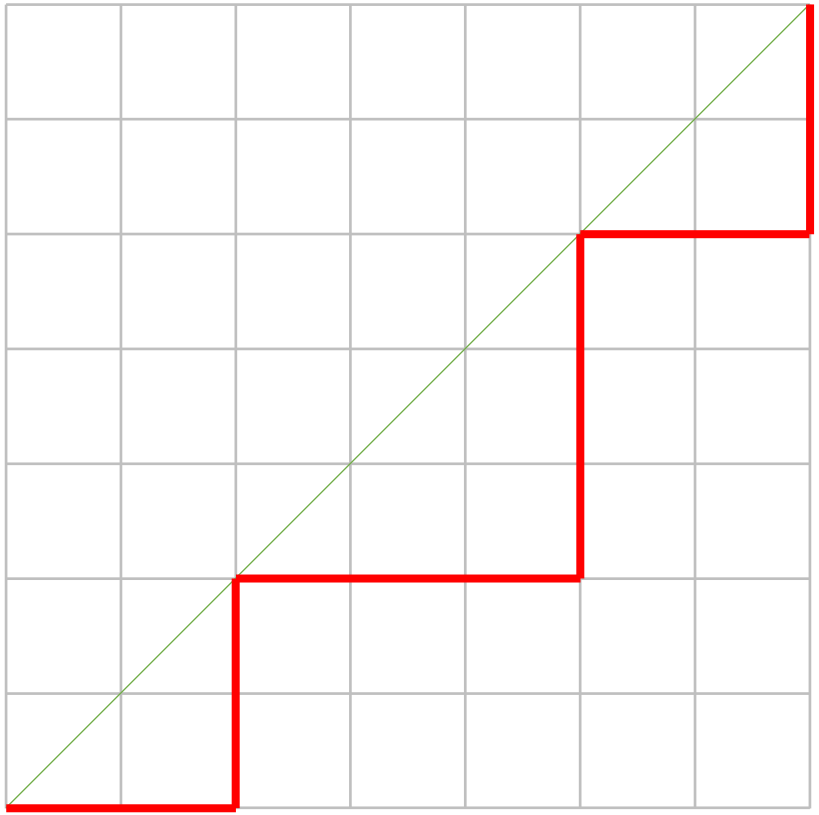
- 집합 홀수 개의 교집합 크기는 더하고, 짝수 개의 교집합 크기는 빼는 방식

카탈란 수

카탈란 수

Dyck Path

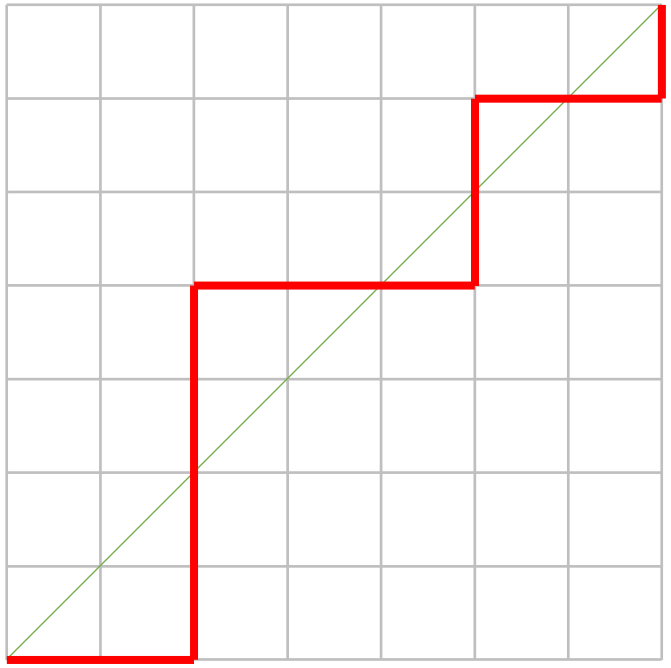
- (0, 0)에서 (n, n)으로 이동할 때, 주대각선(y = x 직선)을 넘어가지 않는 경로
- Dyck Path의 개수 = C(2n, n) – (주대각선을 한 번 이상 통과하는 경로의 개수)

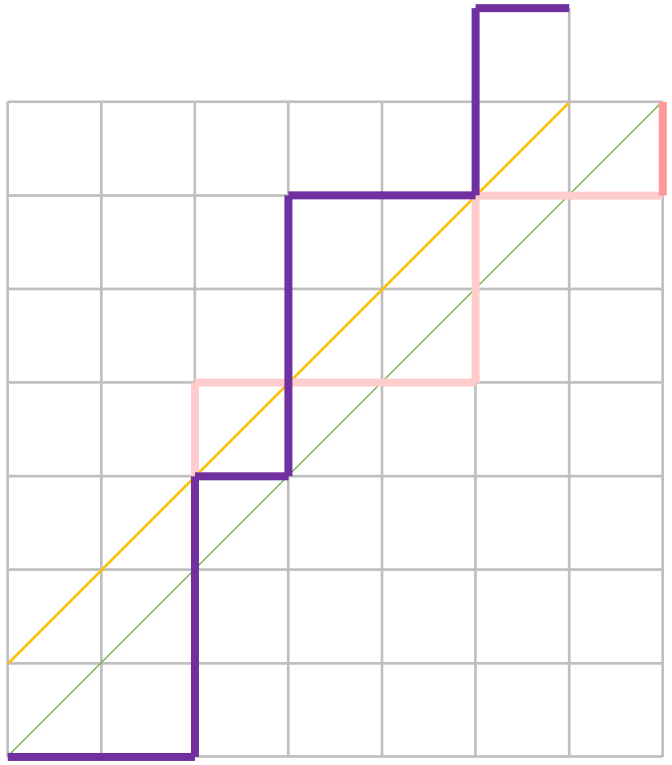


카탈란 수

주대각선을 통과하는 경로의 개수

- 주대각선을 통과하는 직선은 항상 y = x+1 직선 위의 점을 방문함
- 해당 경로가 처음으로 y = x+1 과 만나는 점 이후의 모든 이동을 반전시키면?





카탈란 수

주대각선을 통과하는 경로의 개수

- 주대각선을 통과하는 직선은 항상 y = x+1 직선 위의 점을 방문함
- 해당 경로가 처음으로 y = x+1 과 만나는 점 이후의 모든 이동을 반전시키면?

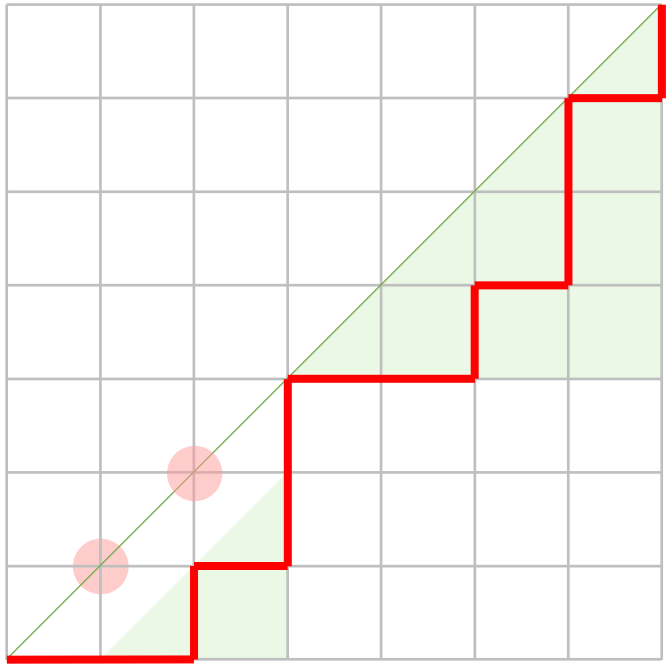
카탈란 수의 일반항

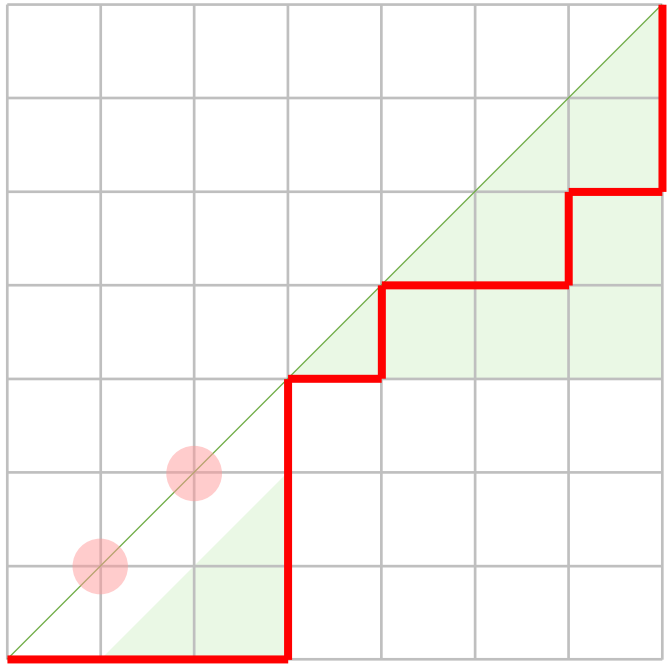
- 모든 Dyck Path는 (0, 0)에서 (n+1, n+1)로 가는 경로와 일대일 대응 가능
- 따라서 Dyck Path의 개수 = $\binom{2n}{n} - \binom{2n}{n+1}$
- 또한, $\binom{2n}{n+1} = \frac{n}{n+1}\binom{2n}{n}$이므로 $\binom{2n}{n} - \binom{2n}{n+1} = \frac{1}{n+1}\binom{2n}{n}$

카탈란 수

카탈란 수의 점화식

- (0, 0)에서 (n, n)으로 이동하는 Dyck Path의 개수를 C_n이라고 하자.
- 원점을 제외하고 y = x 직선과 처음으로 만나는 점을 (k, k)라고 하자.
- (0, 0)에서 (k, k)까지 갈 때는 y = x 직선에 닿으면 안 됨





카탈란 수

카탈란 수의 점화식

- (0, 0)에서 (n, n)으로 이동하는 Dyck Path의 개수를 C_n이라고 하자.
- 원점을 제외하고 y = x 직선과 처음으로 만나는 점을 (k, k)라고 하자.
- (0, 0)에서 (k, k)까지 갈 때는 y = x 직선에 닿으면 안 됨

- (0, 0)에서 (k, k)로 가는 경로의 개수 = C_{k-1}
- (k, k)에서 (n, n)으로 가는 경로의 개수 = C_{n-k}

$$C_n = \sum_{k=1}^{n} C_{k-1}C_{n-k} = \sum_{k=0}^{n-1} C_k C_{n-k-1}$$

- 이 밖에도 $\sum_{n=0}^{\infty} C_n x^n = \frac{1-\sqrt{1-4x}}{2x}$ 같은 생성 함수도 있지만... 생략

카탈란 수

카탈란 수로 해결할 수 있는 문제들

- 여는 괄호 n개와 닫는 괄호 n개로 구성된 올바른 괄호 문자열의 개수
  - 여는 괄호를 ↑ 방향, 닫는 괄호를 → 방향으로 생각하면 Dyck Path와 일대일 대응

- 리프 정점이 n+1개이면서 모든 정점의 자식이 0개 또는 2개인 full binary tree의 개수
  - n+1개의 항으로 구성된 수식에 n쌍의 괄호를 씌워 이항 연산자를 적용하는 경우의 수와 동일
  - 따라서 Dyck Path와 일대일 대응

카탈란 수

카탈란 수로 해결할 수 있는 문제들

- 볼록 n+2각형에 n-1개의 대각선을 그려서 n개의 삼각형으로 분할하는 방법의 수
  - 다각형의 한 번을 고정한 뒤, 그 변을 밑변으로 하는 삼각형을 만들기 위해 대각이 될 꼭짓점을 고정하면?
  - n+2각형에서 한 변 고정했을 때 꼭짓점의 후보는 n가지
  - 그중 k(1 ≤ k ≤ n)번째 점을 선택하면 n+2각형은 다음과 같이 세 조각으로 분할됨
    - 삼각형
    - 후보가 k-1개 있는 k+1각형
    - 후보가 n-k개 있는 n-k+2각형
  - 따라서 $C_n = \sum_{k=1}^{n} C_{k-1}C_{n-k} = \sum_{k=0}^{n-1} C_k C_{n-k-1}$

집합의 분할

제2종 스털링 수

제2종 스털링 수

- n개의 원소로 구성된 집합을 k개의 공집합이 아닌 부분집합으로 분할하는 경우의 수
- ex. n = 3, k = 2면 $\{\{1,2\},\{3\}\},\{\{2,3\},\{1\}\},\{\{3,1\},\{2\}\}$으로 $S(3,2) = 3$

- n번 원소가 들어가는 집합 기준으로 생각하면 점화식을 얻을 수 있음
  - n-1개의 원소를 k-1개의 부분집합으로 분할한 뒤, 새로운 집합을 만들어서 n번 원소 삽입
  - n-1개의 원소를 k개의 부분집합으로 분할한 뒤, n번 원소를 k개의 집합 중 한 곳에 삽입
- $S(n,k) = S(n-1,k-1) + k \cdot S(n-1,k)$

벨 수

벨 수

- n개의 원소로 구성된 집합을 분할하는 경우의 수, $B_n = \sum_{k=0}^{n} S(n,k)$
  - $B_0 = S(0,0) = 1$로 맞추기 위해서 $k = 0$부터 시작

- n번 원소가 속한 부분집합의 크기를 k로 만드는 방법은?
  - n-1개의 원소 중 k-1개를 선택해서 n번 원소와 함께 하나의 집합으로 묶고
  - 남은 n-k를 적절히 분할
- $B_n = \sum_{k=1}^{n} \binom{n-1}{k-1} B_{n-k} = \sum_{k=0}^{n-1} \binom{n-1}{k} B_{n-k-1} = \sum_{k=0}^{n-1} \binom{n-1}{n-k-1} B_{n-k-1} = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k$

- 두 방법 모두 $O(n^2)$ 시간에 $B_0, B_1, \cdots, B_n$ 계산 가능

- $\sum_{n=0}^{\infty} \frac{B_n}{n!} x^n = e^{e^x - 1}$ 같은 생성 함수도 있지만... 생략

제1종 스털링 수

제1종 스털링 수

- n개의 원소를 k개의 방향 있는 사이클로 분할하는 경우의 수

- n번 원소가 들어가는 사이클 기준으로 생각하면 점화식을 얻을 수 있음
  - n-1개의 원소로 k-1개의 사이클을 만든 뒤, n번 원소 하나만 있는 사이클 생성
  - n-1개의 원소로 k개의 사이클을 만든 뒤, n-1개의 간선 중 하나를 골라서 n번 원소 추가
- $S_1(n,k) = S_1(n-1,k-1) + (n-1) \cdot S_1(n-1,k)$

포함 배제의 원리

제2종 스털링 수의 일반항

- n개의 원소로 구성된 집합을 k개의 공집합이 아닌 부분집합으로 분할하는 경우의 수

- n명의 사람을 빈 방을 허용해서 k개의 구분되는 방에 배정하는 것에서 시작
  - 제2종 스털링 수는 공집합을 허용하지 않고, 집합을 구분하지 않음에 주의
  - 여기에서 빈 방이 1개 이상 생기는 모든 배정의 수를 뺀 다음 k!으로 나누면 됨

- n명의 사람을 k개의 방에 집어넣는 방법의 수 k^n
- 빈 방을 r개 이상 만드는 방법의 수 = (빈 방 r개 고정) * (n명을 k-r개의 방에 배정) = $\binom{k}{r}(k-r)^n$

- $k!\,S(n,k) = k^n + \sum_{r=1}^{k}(-1)^r\binom{k}{r}(k-r)^n = \sum_{r=0}^{k}(-1)^r\binom{k}{r}(k-r)^n$
- $\binom{k}{r} = \binom{k}{k-r}$이므로 $k!\,S(n,k) = \sum_{r=0}^{k}(-1)^r\binom{k}{k-r}(k-r)^n = \sum_{r=0}^{k}(-1)^{k-r}\binom{k}{r}r^n$

- 따라서 $S(n,k) = \frac{1}{k!}\sum_{r=0}^{k}(-1)^{k-r}\binom{k}{r}r^n = \frac{(-1)^k}{k!}\sum_{r=0}^{k}(-1)^r\binom{k}{r}r^n$

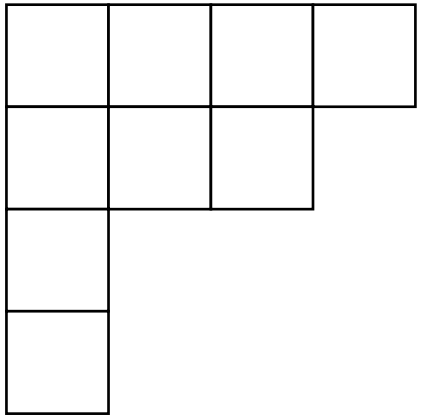
자연수의 분할

분할 수

영 다이어그램과 자연수 분할

- $\lambda = (\lambda_1, \lambda_2, \cdots, \lambda_k)$가 아래 조건을 만족할 때 λ를 N의 자연수 분할이라고 부름
  - λ_i는 양의 정수
  - $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k > 0$
  - $\sum_{i=1}^{k} \lambda_i = N$

- 영 다이어그램: k개의 행으로 구성되어 있고, i번째 행에 λ_i개의 칸이 있는 다이어그램
- ex. $N = 9, \lambda = (4,3,1,1)$



분할 수

분할 수

- 자연수 n을 순서를 고려하지 않고 k개의 자연수의 합으로 나타내는 경우의 수
- 행이 k개인 n칸짜리 영 다이어그램

- 영 다이어그램의 마지막 행의 길이를 기준으로 생각하면
  - 마지막 행의 길이가 1인 경우: 행이 k-1개인 n-1칸짜리 영 다이어그램을 만든 뒤, 밑에 길이가 1인 행 추가
  - 2 이상인 경우: 행이 k개인 n-k칸짜리 영 다이어그램을 만든 뒤, 각 행의 길이를 1씩 증가
- 따라서 $P(n,k) = P(n-1,k-1) + P(n-k,k)$

- $P(n) = \sum_{k=1}^{n} P(n,k)$는 $O(N\sqrt{N})$에 계산할 수 있지만 어려움
  - https://infossm.github.io/blog/2021/05/18/integer-partition/

교란 순열

교란 순열

교란 순열의 점화식

- 교란 순열(완전 순열): 모든 $1 \le i \le n$에 대해 $\pi(i) \neq i$를 만족하는 길이가 n인 순열

- $p(n)$의 값을 기준으로 생각하면 D_n의 점화식을 얻을 수 있음
  - n번째 자리에 $n-1$이 들어간 상황, 즉 $p(n) = n-1$인 상황을 생각해 보자
  - n번 원소는 $p(1), p(2), \cdots, p(n-1)$ 중 원하는 자리에 들어갈 수 있음

  - 만약 n번 원소가 $n-1$번째 자리에 들어가면? $p(n-1) = n$
  - 남은 $1,2,\cdots,n-2$번 원소를 적절히 배치하면 됨
  - 정답에 D_{n-2} 만큼 기여

  - n번 원소가 $n-1$번째가 아닌 다른 자리에 들어간다면? $p(i) = n, 1 \le i < n-1$
  - n번 원소에게 $n-1$이라는 가면을 임시로 씌운 다음 $1,2,\cdots,n-1$을 이용해 교란 순열 생성
    - n번 원소가 $n-1$이라는 가면을 썼기 때문에 $n-1$번 자리로 가지 않음
    - 따라서 $p(1) = n, p(2) = n, \cdots, p(n-2) = n$인 상황을 모두 확인 가능
  - 정답에 D_{n-1} 만큼 기여

- 이를 $\pi(n) = 1,2,\cdots,n-1$ 인 상황에서 모두 확인하면 $D_n = (n-1)(D_{n-1} + D_{n-2})$

교란 순열

교란 순열의 일반항

- $D_n = (n-1)(D_{n-1} + D_{n-2}) = (n-1)D_{n-1} + (n-2)D_{n-2}$
- nD_{n-1}을 왼쪽으로 넘기면 $D_n - nD_{n-1} = -D_{n-1} + (n-1)D_{n-2}$

- $A_n = D_n - nD_{n-1}$이라고 정의하면 $A_n = -A_{n-1}$이므로 A는 공비가 -1인 등비수열
- $A_1 = D_1 - D_0 = 0 - 1 = -1$이므로 $A_n = (-1)^n$

- $A_n = D_n - nD_{n-1} = (-1)^n$, 양변을 $n!$으로 나누면 $\frac{D_n}{n!} - \frac{D_{n-1}}{(n-1)!} = \frac{(-1)^n}{n!}$
- $B_n = \frac{D_n}{n!}$으로 정의하면 $B_n - B_{n-1} = \frac{(-1)^n}{n!}$
- $B_n = B_1 + \sum_{k=2}^{n} \frac{(-1)^k}{k!}, B_1 = \frac{D_1}{1!} = 0$ 이므로 $B_n = \sum_{k=2}^{n} \frac{(-1)^k}{k!}$
- $\frac{(-1)^0}{0!} + \frac{(-1)^1}{1!} = 0$이므로 $B_n = \sum_{k=0}^{n} \frac{(-1)^k}{k!}$

- 따라서 $\mathrm{n!\,B_n} = D_n = n! \sum_{k=1}^{n} \frac{(-1)^k}{k!}$

교란 순열

교란 순열의 일반항

- 포함 배제를 사용해도 같은 점화식을 얻을 수 있음
  - n개의 원소를 나열하는 $n!$가지 방법 중 $\pi(i) = i$인 i의 개수가 k개 이상인 순열의 개수 = $\frac{n!}{k!}$
  - 따라서 $D_n = \sum_{k=0}^{n}(-1)^k \times \frac{n!}{k!} = n! \sum_{k=0}^{n} \frac{(-1)^k}{k!}$

- $\frac{D_n}{n!} = \sum_{k=0}^{n} \frac{(-1)^k}{k!}$ 은 뭔가 익숙한 형태의 식인데…

교란 순열

교란 순열의 일반항

- 포함 배제를 사용해도 같은 점화식을 얻을 수 있음
  - n개의 원소를 나열하는 $n!$가지 방법 중 $\pi(i) = i$인 i의 개수가 k개 이상인 순열의 개수 = $\frac{n!}{k!}$
  - 따라서 $D_n = \sum_{k=0}^{n}(-1)^k \times \frac{n!}{k!} = n! \sum_{k=0}^{n} \frac{(-1)^k}{k!}$

- $\frac{D_n}{n!} = \sum_{k=0}^{n} \frac{(-1)^k}{k!}$ 은 뭔가 익숙한 형태의 식인데…
  - $e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$ 에 $x = -1$을 대입한 형태
  - 따라서 $\lim_{n\to\infty} \frac{D_n}{n!} = \frac{1}{e}$