#study #algorithms

재귀

재귀란?

- 어떤 함수를 실행했을 때, 함수가 종료 전에 자기 자신을 호출하는 행위

```
void fibonacci(int factor) {
    if (factor == 0)
        return 0;
    if (factor == 1)
        return 1;
    return fibonacci(factor - 1) + fibonacci(factor - 2);
}
```

- 위 함수는 피보나치 수열을 구현한 간단한 재귀 알고리즘이다. factor는 피보나치 수열의 몇 번째 항인지를 의미한다.
  - 피보나치 수열은 1, 1, 2, 3, 5, 8 ... 이며 0번째 항을 편의상 0이라 하였을 때 두번째 항부터 현재 항 = (현재 - 1)항 + (현재 - 2)항과 동일한 형태를 보인다.
  - 이처럼 함수에서 자기 자신을 호출하는 경향이 있는 걸 재귀라 한다.

재귀의 특징 장단점

- 특징
  - 종료 조건 : 재귀 함수를 종료하기 위한 조건이다. 재귀 함수는 자기 자신을 호출하기에 별도의 조건이 없으면 끝없이 자기 자신을 호출하다가 스택 메모리가 터져버릴 것이다. 그렇기에 반드시 필요하다.
    - fibonacci 함수를 보면 factor == 0 또는 factor == 1일 때, 자기 자신을 호출하지 않고 변수값을 반환하는 게 보이는 가? 이처럼 더는 함수를 호출하지 않고 끝내는 행위가 필요하다.
  - 깊이 : 첫번째 호출부터 시작해서 종료 조건에 도달할 때까지의 호출 횟수. 함수는 자기 역할을 다 할 때까지 스택 메모리에 적재되어 있으며 자기 자신을 호출하는 재귀 함수는 당연하게도 호출한 함수가 값을 반환할 때까지 메모리에 살아있다. 그렇기에 너무 호출 횟수가 늘어나 버리면 메모리에 재귀 함수가 차지하는 면적도 비대해지다가 결국 스택 오버 플로우가 발생한다.
- 단점
  - 위와 같이 종료 조건을 별도로 생각해야 하는 까다로운 면이 있고 깊이 때문에 섣불리 재귀를 때려박다가 낭패하기 십상이라는 점이 문제다.
  - 반복문보다 연산 횟수에서 비효율적인 경우가 꽤 있다.
- 장점
  - 깔끔한 함수 : 반복문으로 구성하는 것보다 함수 길이가 짧다. 그리고 종료 조건만 잘 구성하면 내부 연산 과정은 스택 간 정보 교환에서 전부 이루어지므로 연산이 복잡할 수록 반복문보다 만

들기 쉽다.
- 반복문 구성 시, 연산이 복잡하면 재귀로 구성할 경우 좀 더 편하게 만들 수 있다.

백트래킹

백트래킹이란?

- 해가 될 가능성이 없는 곳은 다시 돌아오고 해가 될 가능성이 있는 곳을 탐구하는 알고리즘

백트래킹 과정

- 유효한 답의 집합을 정의한다
- 정의한 집합을 그래프로 표현한다
- 유망 함수를 정의한다
  - 유망 함수는 탐색 조건을 정의하는 방식을 의미한다
- 백트래킹 알고리즘을 활용하여 답을 찾는다

백트래킹 활용

- 부분 집합 합 : 1부터 N까지 숫자를 조합했을 때 합이 K가 되는 조합이 몇 개인지 찾기
  - 방법 1 : 완전탐색으로 풀어보자
    - 모든 집합을 구하는 경우의 수 : nCn + nCn-1 + nCn-2 + ... nC1
    - 정말 비효율적이다
  - 방법 2 : 백트래킹으로 풀어보자
    - 유망 함수
      - 현재 조합으로 합이 K가 되면 조합 성공 횟수를 더한다
      - 해당 숫자를 조합하여 합이 K보다 크거나 같으면 더 탐색하지 않는다

```
// N, K라는 전역변수가 이미 존재하는 것으로 가정
bool sum_of_subset(int cur_num, int sum_num, int& set_num) {
    // 유망함수
    if (sum_num == K)
        set_num++;
    if (sum_num >= K)
        return true;

    for (int i = cur_num; i <= N; i++) {
        // 현재 조합의 합이 K보다 크거나 같으면 조합을 추가할 필요가 없으므로 반복문
을 종료한다.
        if (sum_of_subset(i + 1, sum_num + i, set_num))
            break;
    }
    // 이전 스택으로 돌아가서 다음 노드를 탐색한다.
    return false;
}
```

    - 시간복잡도 : K값에 따라 복잡도가 달라지기는 하지만 적어도 완전탐색보다는 효율적인 것을 알 수 있다.

풀어보자

- N-퀸, https://school.programmers.co.kr/learn/courses/30/lessons/12952
- (난이도 상) 사라지는 발판, https://school.programmers.co.kr/learn/courses/30/lessons/92345

동적 계획법

동적 계획법이란?

- 큰 문제를 작은 문제로 나누었을 때, 작은 문제들의 결과값을 저장하여 동일한 연산을 반복하지 않는 알고리즘
- 재귀에서 언급한 피보나치 알고리즘을 생각해보자
    - fibonacci(7) = fibonacci(5) + fibonacci(6) = fibonacci(3) + fibonacci(4) + fibonacci(4) + fibonacci(5) = ...
    - 자 동일한 factor를 넣은 함수를 불러오는 게 보이는 가? 만약 fibonacci(5)의 연산에서 각 값들을 저장하면 fibonacci(6) 연산에서는 사전에 저장된 fibonacci(4)의 결과값과 fibonacci(5)의 결과값만 불러오면 끝날 것이다.
    - 위와 같이 반복적인 연산을 피하기 위해 결과값을 저장하는 방식이 동적 계획법이다.

동적 계획법의 특징

- 점화식이 필요함
    - 큰 문제를 작은 문제로 쪼갤 수 있는 점화식을 세워야 한다. 그렇게 해야 작은 문제에 해당하는 값을 저장할 수 있다. 수열의 규칙성을 찾아서 점화식을 세우던 게 기억이 나는가? 기억이 나지 않으면 아래를 보자.
    - 예시) 팩토리얼 값을 반환하는 함수 Fact가 있다고 가정하자
        - Fact(N) = N! = N * (N - 1) * (N - 2) * ... * 1
        - 위 식에서 두 가지를 알 수 있다. 하나는 1이라는 상수값이 반드시 있는 것. 둘은 N이 들어가면 N ~ 1까지 곱한다는 것
        - 즉, Fact(N) = Fact(N - 1) * N이다. 그리고 Fact(1)은 1이다.
        - 전자는 재귀에서 자기 자신을 호출하는 방식이고 후자는 종료 조건이다. 이로서 큰 문제를 작은 문제로 쪼개는 재귀 방식의 점화식을 세웠다.
- 재귀 활용
    - 반복문도 쓸 수 있고 반복문이 효율이 좋을 때도 있다. 단, 이번에는 재귀를 통해서 스택 메모리가 쌓이는 과정을 공부하는 중이고 점화식만 짜면 재귀로 만드는 게 훨씬 쉽기에 재귀로 하겠다.
    - 예시) Fact(N) 함수를 구현하자

```
int Fact(int factor) {
    if (factor == 1)
```

```
        return 1;
    return factor * Fact(factor - 1);
}
```

- 메모이제이션
  - 점화식을 구했으니 이제 부분 문제의 값을 저장해서 활용할 차례다

```
// N값은 미리 저장되어 있다고 가정한다.

// 각 부분 문제의 결과값을 저장할 공간을 만든다
int save[N + 1];

int Fact(int factor) {
    // save[facotr]에 0이 아닌 값이 존재하는 경우, 값이 저장된 것
    if (save[factor])
        return save[factor];

    // 종료 조건에 도달하면 값을 넣어주고 아니면 재귀 호출을 통해 결과값을 저장한다
    if (factor == 1)
        save[factor] = 1;
    else
        save[factor] = factor * Fact(factor - 1);

    // 저장된 값을 반환하여 결과값 도출
    return save[factor];
}
```

동적 계획법 활용

- 최장 증가 부분 수열(LIS, Longest Increasing Subsequence)
  - 부분 수열 : 주어진 수열 중에서 일부를 뽑아 만든 수열. 수열을 만들 때 원본 수열의 순서는 보장해서 만든다.
  - 최장 증가 부분 수열 : 부분 수열의 원소가 오름차순을 유지하면서 길이가 가장 긴 수열
  - 부분 문제를 어떻게 찾을 것인가?
    - LIS는 당연하게도 주어진 수열의 범위 내에서 뽑을 수 밖에 없다. 그렇기에 주어진 수열의 길이가 1이면 당연하게도 LIS의 길이는 1이 될 수 밖에 없다. (1)
    - 부분 수열은 원본 수열의 순서를 보장하며 최장 증가 부분 수열은 원리에 따라 오름차순을 유지해야 한다.
      - 수열에서 어떤 원소 하나를 뽑았을 때, 가장 긴 부분 수열의 길이는 해당 원소보다 앞에 있는 원소들의 부분 수열 중에서 가장 긴 부분 수열의 길이 + 1일 것이다.
      - LIS는 오름차순을 보장하므로 위 조건을 적용하면 LIS는 해당 원소보다 앞에 있는 원소들의 부분 수열 중에서 부분 수열의 가장 마지막 원소가 현재 원소보다 값이 작고 오름차순이 보장되는 부분 수열의 길이 + 1일 것이다. (2)
  - 점화식

        - dp[1] = 1 (1)
        - dp[N1] = max(dp[K]) + 1 (단, seq[K] < seq[N]) (2)
    - 예시 코드

```
// 수열 길이 N과 수열 배열 seq는 사전에 정의된 걸로 가정한다

// 부분 문제의 결과값을 저장할 배열을 선언한다
int dp[N + 1];

int LIS() {
    // 첫번째 원소만 있는 부분 수열에서는 LIS가 1일 수 밖에 없다
    dp[1] = 1;

    // 현재 원소보다 앞에 위치한 원소들로 이루어진 부분 수열에서 현재 원소와 비교하여 LIS
를 찾는다
    for (int i = 2; i <= N; i++) {
        dp[i] = searchPartialSequenceLIS(i - 1) + 1;
    }

    // 전체 수열에서 LIS 길이를 반환한다.
    return max(dp);
}

int searchPartialSequenceLIS(int factor) {
    // LIS 초기값 설정
    int LIS = 0;

    // 기준인 현재 원소보다 작은 값의 원소가 마지막 원소인 부분 수열 중 LIS를 찾는다
    for (int i = factor; i >= 0; i--) {
        if (seq[i] <= seq[factor] && dp[i] > LIS)
            LIS = dp[i];
    }

    // 찾은 LIS를 반환
    return LIS;
}
```

- 최장 공통 부분 수열(LCS, Longest Common Subsequence)
    - 두 수열이 어떤 기준에 따라 양쪽에서 공통으로 발견할 수 있는 LIS를 의미한다. 즉, 아래의 조건을 모두 만족한다.
        - 오름차순을 유지하는 부분 수열
        - 두 수열이 똑같이 가지고 있는 부분 수열
        - 가장 긴 부분 수열
    - 부분 문제를 어떻게 찾을 것인가
        - 일단 세 가지 조건을 고려할 필요가 있다
            - 공통 : 두 수열에 동일하게 존재하는 문자만 고려한다

        - 부분 수열 : 순서 보장을 위해 현재 찾은 공통 문자가 이전 공통 문자보다 뒤에 있어야 한다.
        - LIS : 순서대로 찾은 공통 문자 중에서 오름차순을 유지하며 가장 긴 부분 수열을 찾아야 한다.
    - 두 수열의 공통 문자 위치가 어디인 지 알아야 한다. 즉, 메모이제이션을 할 dp 배열은 2차원으로 구성한다.
    - 한 수열을 기준으로 LIS를 시작한다. 단, LIS와 달리 이전 부분 수열과 오름차순을 맞추었더라도 다른 수열과 동일 문자의 위치가 같은 곳부터 dp 배열의 횟수를 더한다.
- 점화식
    - dp(0, 0) = 0
    - seq_x[i] == seq_y[i] 이면 dp(i - 1, j - 1) + 1
    - seq_x[i] != seq_y[i] 이면 max(dp(i - 1, j), dp(i, j - 1))

```
int LCS(array seq_x, array seq_y) {
    // dp의 초기값은 전부 0으로 초기화 된 걸로 가정한다
    int dp[seq_x.size()][seq_y.size()];

    for (int i = 1; i < seq_x.size(); i++) {
        for (int j = 1; j < seq_y.size(); j++) {
            if (seq_x[i] == seq_y[j])
                dp[i][j] = dp[i - 1][j - 1] + 1;
            else
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
        }
    }
    return dp[]
}
```

풀어보자

- 2 * n 타일링, https://school.programmers.co.kr/learn/courses/30/lessons/12900
- 가장 큰 정사각형 찾기, https://school.programmers.co.kr/learn/courses/30/lessons/12905
- (난이도 상) 도둑질, https://school.programmers.co.kr/learn/courses/30/lessons/42897