동적 계획법 - Concept

#동적계획법

동적 계획법이란?

1. 분할 정복은 완전 탐색에서 탐색 깊이를 줄이는 방식에서 출발했다. 동적 계획법은 완전 탐색에서 문제를 분할해 나갈 때 부분 문제가 중복되는 걸 줄이려는 노력에서 출발한다.
2. 예를 들어 이항 계수 계산을 단순 재귀로 풀어보자.

```
// 재귀 호출을 통한 이항 계수의 계산
// 이항 계수 : (n r)로 표현하며 이 의미는 n 개의 서로 다른 원소 중에서 r 개의 원소를 순서없이 골라내는 방법의 수를 나타낸 것
// 이항 계수 재귀에서 사용할 점화식 : (n r) = (n-1 r-1) + (n-1 r)
int bino(int n, int r) {
        // 기저 사례 : n == r (모든 원소를 다 고르는 경우), r == 0 (고를 원소가 없는 경우)
        if (r == 0 || n == r) return 1;
        return bino(n-1, r-1) + bino(n-1, r);
}
```

   - 만약 위와 같이 단순 재귀로 문제를 풀면 재귀를 부르면 부를 수록 동일한 내용물을 반복해서 불러오게 될 것이다.
   - 가령 bino(4, 2)를 적용하면 bino(3, 1) + bino(3, 2)로 나뉠 것이고 두 부분문제는 다시 bino(2, 0) + bino(2, 1)과 bino(2, 1) + (bino 2, 2)가 될 것이다. bino(2, 1)이 겹치는 게 보이는가? 여기서 더 아래로 내려가면 bino(1, 1), bino(1, 0) 등. 더욱 겹치는 부분 문제가 생길 것이고 이를 각자 다 한 번 씩 부른다고 치면 꽤 많은 부분 문제들이 중복된다.
   - bino(25, 12)를 계산하려면 1000만 번의 함수 호출이 필요하다. 하지만 공통적으로 겹치는 부분 문제들을 제외하면 181번의 함수 호출만 필요하므로 꽤 연산량을 줄일 수 있다.
3. 이항 계수 계산에 대한 최적화
   - 중복되는 부분 문제를 줄이기 위해 미리 부분 문제의 결과값을 저장해 두었다가 필요할 때마다 저장해 둔 결과값을 불러오는 기법을 기록법(memoization)이라 한다.

```
// 기록법을 통한 이항 계수의 계산
// cache는 별도로 -1로 초기화 해둔다.
int cache[30][30];
int bino(int n, int r) {
        // 기저 사례
        if (r == 0 || n == r) return 1;
        // -1이 아니면 한 번 계산했던 값이니 곧장 반환
        if (cache[n][r] != -1)
                return cache[n][r];
        // 직접 계산한 뒤에 배열에 저장
        return cache[n][r] = bino(n-1, r-1) + bino(n-1, r);
}
```

- cache를 통해서 위와 같이 결과 값을 저장해 놓으면 동일한 부분 문제를 중복해서 부르지 않고 바로 값만 가져올 수 있다.

기록법 구현 패턴

1. 사람 마다 구현 방식이 조금씩 다를 수 있으나, 고전적인 방식은 아래와 같다.
2. 항상 기저사례를 먼저 처리할 것.
3. 함수의 반환값이 항상 0이라는 점을 이용해 cache[]를 모두 -1로 초기화. 만약, 함수의 반환값으로 음수도 들어올 수 있는 환경이라면 별도 방식을 택할 것. 가령, bool 값을 적용한 cache를 하나 더 만들던지.
4. cache에 값을 저장하거나 불러오는 용도로 참조형 변수 하나를 설정하면 편하다. cache[a][b]를 여러번 입력하다가 오타 나면 찾기 힘들 뿐더러 머리만 아프다.
5. memset을 이용하여 cache를 초기화하면 편하다.

```
// 위 디자인 패턴을 통해 구현한 예시
int cache[2500][2500];
int someObscureFunction(int a, int b) {
        // 기저 사례를 먼저 처리
        if (...) return ...;
        // (a, b)에 대한 답을 구한 적 있으면 곧장 반환
        int& ret = cache[a][b];
```

```
        if (ret != -1) return ret;
        // 여기에서 답을 계산
        ...
        return ret;
}
int main(void) {
        // memset을 이용해 cache 배열 초기화
        memset(cache, -1, sizeof(cache));
}
```

기록법의 시간복잡도 분석

1. 시간복잡도 : 존재하는 부분 문제의 수 * 한 부분 문제를 풀 때 필요한 반복문의 실행 횟수
2. 이미 계산이 되어 cache에 쌓인 부분 문제는 저장된 값을 불러오면 되므로 상수 시간에 해결이 된다. 즉, 계산되지 않은 부분 문제들에서 이루어지는 재귀 호출 횟수만 따져가면 시간 복잡도를 계산할 수 있다.
3. 이항 계수 계산 식을 예로 들면 bino(n, r)에서 r의 최대치는 n 이므로 부분 문제의 수는 O(n^2)이다. 각 부분문제에서 계산 시, 반복문이 없다. 이 말은 O(1) 즉, 상수 시간이므로 시간 복잡도는 O(n^2 * 1)이 된다.

전통적 최적화 문제

1. 최적화 문제란 여러 개의 가능한 답 중 가능한 가장 좋은 답을 찾아내는 문제를 이른다
2. 최적화 문제를 동적 계획법으로 해결할 시 생각해 볼만 한 것
    - 모든 답을 만들어보고 그 중 최적해의 점수를 반환하는 완전 탐색 알고리즘 설계
    - 전체 답의 점수를 반환하는 것이 아니라, 앞으로 남은 선택들에 해당하는 점수만을 반환하도록 부분 문제 정의를 바꾼다
    - 재귀 호출의 입력에 이전의 선택에 관련된 정보가 있따면 꼭 필요한 것만 남기고 줄인다
    - 입력이 배열이거나 문자열인 경우 가능하다면 적절한 변환을 통해 기록법을 적용한다
    - 위 사항을 모두 만족하고 나서 기록법을 이용한다
3. 예시] 삼각형 문제

```
1
2  4
5  6  7
3  5  6  11
14 19 3  8  13
```

- 위처럼 삼각형으로 숫자가 배치되어 있을 때, 맨 위에서 시작해서 한 번에 한 칸 씩 직하 또는 오른쪽 한 칸 아래로 내려갈 때 지나간 경로의 숫자합이 최대가 되는 값을 구해보자.
- 완전 탐색으로 시작하기
  - path(y, x, sum) : y는 높이, x는 길이. sum은 현재까지의 경로합이다. 이를 이용해 점화식을 구현하면 아래와 같다.
  - path(y, x, sum) = max(path(y + 1, x, sum + triangle[y][x]), path(y + 1, x + 1, sum + triangle[y][x]));
  - 위와 같이 하면 모든 경로를 돌아 보는데 n 개의 가로줄에 2^(n - 1)이 걸린다. n이 20만 초과해도 연산횟수가 1억을 넘어가니 n > 20인 상태에서 1초 주고 문제 풀라하면 터질 것이다.
- 무식하게 기록법 적용해보기

```
// MAX_NUMBER : 한 칸에 들어갈 숫자의 최대치
int n, triangle[100][100];
int cache[100][100][MAX_NUMBER*100+1];
// (x, y) 위치까지 내려오기 전에 만난 숫자들의 합이 sum 일 때
// 맨 아래줄까지 내려가면서 얻을 수 있는 최대 경로를 반환
int path(int y, int x, int sum) {
        // 기저 사례 : 맨 아래 줄까지 도달했을 경우
        if (y == n - 1) return sum + triangle[y][x];
        // 기록법
        int& ret = cache[y][x][sum];
        if (ret != -1) return ret;
        sum += triangle[y][x];
        return ret = max(path(y + 1, x + 1, sum), path1(y + 1, x, sum));
}
```

- 위와 같은 무식하게 적용하면 메모리를 차지하는 공간이 많아질 뿐더러, 정작 경로가 y + 1, x + 1이라 해도 위에서 어떤 경로를 거쳐왔는지에 따라 저장되는 값들이 천차만별이니 결국 완전 탐색과 다를 바 없어진다.
- 재귀 함수의 입력을 걸러내야 한다
  - y와 x는 재귀 호출이 풀어야 할 부분 문제를 지정해 줌. 두 입력이 정해져야 경로가 완성되는 것.
  - sum은 어떤 경로로 이 부분 문제에 도달했는 지 알려준다. 지금까지 풀었던 조각들에 대한 정보를 주는 셈.
  - 즉, 경로를 완성하는 데는 sum이 얼마건 상관없다. 단, sum을 입력으로 받지 않으면 이전까지 어떤 숫자를 만났는 지 알 수 없다. 그렇기에 함수의 반환값을 (y, x)에서 시작하는 부분 경로의 최대값으로 변경할 필요가 있다
    - path(y, x) = triangle[y][x] + max(path(y + 1, x), path(y + 1, x + 1))
- 최적 부분 구조를 통한 기록법 적용

```
int n, triangle[100][100];
int cache[100][100];
// (y, x) 위치부터 맨 아래줄까지 내려가면서 얻을 수 있는 최대 경로의 합을 도출한다
int path(int y, int x) {
        // 기저 사례
        if (y == n - 1) return triangle[y][x];
        // 기록법
        int& ret = cache[y][x];
        if (ret != -1) return ret;
        return ret = max(path(y + 1, x), path(y + 1, x + 1)) + triangle[y][x];
}
```

- 부분 경로의 최대합을 구하면서 최종적으로 전체 경로의 최대합을 구해가는 분할 구조이기 때문에 부분 문제의 수는 n^2, 각 부분 문제를 계산하는 데 상수 시간만 소모되므로 시간복잡도는 O(n^2)이 된다.

최적 부분 구조

1. 각 부분 문제의 최적해를 찾는 것으로 전체 구조의 최적해를 찾아낼 수 있을 때, 최적 부분 구조가 성립한다.
2. 가령, 서울 - 평양으로 가는 최단 경로가 판문점을 통과한다 가정하면 (서울 - 판문점), (판문점 - 평양)로 구간을 나누고 두 구간의 최단 경로를 찾아서 둘을 이으면 서울 - 평양로 가는 최단 경로를 찾을 수 있다.
3. 반면, 서울 - 평양에 갈 때, 조건을 하나 더 추가해서 통행료가 3만원을 넘지 않는 선에서 통과하고 싶다 가정하자.

- 이런 경우에는 서울 - 판문점, 판문점 - 평양을 통과하는 각각의 경로와 통행료에 따라 다음 경로의 결정된다. 즉, 서울 - 판문점이 2만원에 1시간 걸리는 통로를 택하면 판문점 - 평양은 무조건 1만원 이하의 경로가 되고, 서울 - 판문점이 1만원에 2시간 걸리는 통로를 택하면 판문점 - 평양은 무조건 2만원 이하의 경로를 택해야 한다.
- 부분 문제의 최적해가 전체 문제의 최적해가 될 지 미지수인 상황이므로 최적 부분 구조는 존재하지 않는다 평한다.

정수 이외의 입력에 대한 기록법

1. stl 중에서 map을 이용하여 쉽게 찾아나간다.
2. 일대일 대응 함수를 작성한다.
   - 입력값을 확인하여 몇 번째 키에 해당하는 지 확인하고 이를 반환해주는 함수를 별도로 만들어 cache의 key 값으로 활용할 수 있다.
3. 입력이 참, 거짓 두 부분으로 나뉘는 부울 배열인 경우에는 비트마스킹을 통해 저장한다.

조합 게임

1. 조합 게임은 바둑, 체스, 장기처럼 두 사람의 참가자가 하는 게임을 가리킨다. 이 때, 게임의 상태가 주어졌을 때 완벽한 한 수를 찾아내는 알고리즘을 구성하는 것.
2. 어떻게 알고리즘을 구성할 수 있는가?
   - 반복문을 통해 불러오면 직관적이지만, 경우의 수도 많고 중간에 게임이 종료될 수도 있으므로 재귀를 통해 구현한다.
   - winner(state, player) : 게임의 현재 상태를 state이고 player가 수를 둘 차례일 때 어느 쪽이 최종적으로 이길 것인가?
     - 어느 쪽이 둘 차례인지 player를 통해 확인한다. 근데 굳이 player가 필요한가? 전달 받은 반환값을 통해 내 차례인지 아니면 상대방 차례인지 구분할 수 있지 않겠는가?
   - canWin(state) : 해당 상태에서 둘 수 있는 최적의 수를 뽑고 true, false로 구분한다. 즉, true면 내가 이긴 거고 false면 상대방이 이긴 것.
     - 위와 같이 바꾸면 굳이 player를 넣지 않아도 내가 이기는 지 상대방이 이기는 지 알 수 있다.
     - 여기서 기록법을 활용하여 동일 state에서 canWin()을 여러번 호출하지 않도록 방지한다.
3. 예시] 틱택토
   - 틱택토는 3 x 3 크기의 게임판에서 하는 3목 게임이다. 흑과 백의 말 대신에 O와 X로 자신의 말을 표시한다.
   - 틱택토는 양쪽 3목 줄을 만들지 못하고 게임판을 꽉 채우는 경우에는 비기는 걸로 간주한다.

```
// num이 한 줄을 만들었는 지를 반환
bool isFinished(const vector<string>& board, char turn);
// 틱택토 게임판이 주어질 때 [0, 19682) 범위의 정수로 반환한다
int bijection(const vector<string>& board) {
        int ret = 0;
        for (int y = 0; y < 3; y++)
                for (int x = 0; x < 3; x++) {
                        ret = ret * 3;
                        if (board[x][y] == 'o') ++ret;
                        else if (board[y][x] == 'x') ret += 2;
                }
        return ret;
}
// cache는 -2로 초기화
int canWin(vector<string>& board, char turn) {
        // 기저 사례 : 마지막에 상대가 둬서 한 줄이 만들어진 경우
        if (isFinished(board, 'o' + 'x' - turn)) return -1;
        int& ret = cache[bijection(board)];
        if (ret != 2) return ret;
        // 모든 반환 값의 min을 취하자.
        int minValue = 2;
        for (int y = 0; y < 3; ++y)
                for (int x = 0; x < 3; ++x)
                        if (board[y][x] == '.') {
                                board[y][x] = turn;
                                minValue = min(minValue, canWin(board, 'o' + 'x' - turn));
                                board[y][x] = '.';
                        }
        // 플레이할 수 없거나, 어떻게 해도 비기는 것이 최선인 경우
        if (minValue == 2 || minValue == 0) return ret = 0;
        // 최선이 상대가 이기는 거라면 난 무조건 지고, 상대가 지는 거라면 난 이긴다
        return ret = -minValue;
}
```

4. 감이 빠른 분들은 눈치챘을 수도 있겠지만, 이러한 조합 게임 문제는 인공지능 쪽에서 깊게 다루고 있다. 이 쪽 분야에 관심이 있는 분들은 경험적 알고리즘, 미니맥스 알고리즘 등을 검색하여 공부하기를 바란다.

반복적 동적 계획법과 재귀적 동적 계획법

1. 부분 문제 간의 의존성을 쉽게 파악하는 경우, 재귀적 동적 계획법보다 반복적 동적 계획법이 좀 더 쉽다.
2. 예시] 삼각형 위의 최대 경로
   - 전통적 최적화 문제에서 예시로 보았던 삼각형 위의 최대 경로를 생각해 보자
   - 위 문제의 경우, 무조건 다음 수가 아래줄로 내려간 뒤에 이루어진다. 즉, 맨 아랫줄부터 시작해서 한 줄씩 차례차례 올라가면 계산하는 데 필요한 값을 항시 가지기에 재귀 호출을 쓸 필요도 없다

```
int n, triangle[100][100];
int C[100][100];
int iterative() {
        // 기저 사례를 검색한다
        for (int x = 0; x < n; ++x)
                C[n - 1][x] = triangle[n - 1][x];
        // 다른 부분을 계산한다
        for (int y = n - 2; y >= 0; --y)
                for (int x = 0; x < y + 1; ++x)
                        C[y][x] = max(C[y + 1][x], C[y + 1][x + 1]) + triangle[y][x];
        return C[0][0];
}
```

   - 반복적 동적 계획법은 슬라이딩 윈도 기법을 사용할 수 있다. 슬라이딩 윈도 기법은 데이터 전체를 메모리에 유지하는 게 아니라 필요한 부분만을 저장하는 기법이다.

```
int C2[2][10000];
int iterative() {
        // 기저 사례를 계산한다
        for (int x = 0; x < n; ++x)
                C2[(n - 1) % 2][x] = triangle[n - 1][x];
```

```
        // 다른 부분을 계산한다
        for (int y = n - 2; y >= 0; --y)
                for (int x = 0; x < y + 1; ++x)
                        C2[y % 2][x] = max(C2[(y + 1) % 2][x], C2[(y + 1) % 2][x + 1]) + triangle[y]
[x];
        return C2[0][0];
}
```

- 삼각형 문제에서서는 C[] i번째 가로줄 C[i]를 계산하려면 C[i + 1]만 필요할 뿐 나머지 부분은 필요하지 않다. 그래서 기존값을 전부 저장할 필요가 없는 슬라이딩 윈도 기법을 사용할 수 있다.
- 재귀는 어느 부분 문제가 어느 때에 튀어나오는 지 정확히 분간할 수 없으므로 슬라이딩 윈도 기법을 사용할 수 없다.

3. 재귀적 동적 계획법의 장단점
   - 장점
     - 좀 더 직관적인 코드
     - 부분 문제 간의 의존 관계나 계산 순서 고민 필요 X
     - 전체 부분 문제 중 일부의 답만 필요한 경우 더 빨리 동작
   - 단점
     - 슬라이딩 윈도 기법 사용 불가
     - 스택 오버 플로우를 조심해야 함
4. 반복적 동적 계획법의 장단점
   - 장점
     - 구현이 더 짧음
     - 재귀 호출에 필요한 부하가 없어 더 빨리 움직임
     - 슬라이딩 윈도 기법 사용 가능
   - 단점
     - 구현이 좀 더 비직관적
     - 부분 문제 간의 의존 관계를 고려해 계산되는 순서를 고민해야 함

그래서 동적 계획법은 언제 쓰면 좋은데요?

1. 부분 문제가 너무 중복되어서 줄이고 싶다
    - 적절하게 n 분할해서 풀 수 없는 상황에서 완전 탐색으로는 무조건 연산량 초과인 경우에 고려하면 좋다
2. 최적한 문제다.
    - 동적 계획법은 최적화 문제를 해결하고자 나온 알고리즘이다. 그렇기에 최적화 문제가 나오면 동적 계획법을 고려해 보는 게 좋다.
3. 문제가 메모리는 넘쳐나는 데 주어진 시간이 좀 짜다.
    - 보통 메모리를 여유있게 주고 주어진 문제의 연산량에 비해 시간복잡도가 적으면 기록법을 활용하라는 얘기다. 물론, 여기서 더 나아가 미니맥스 알고리즘이나 어디 인공지능에서 나오는 신경망 이론을 접목시켜서 튀어 나오는 알고리즘이 대상일 수 있지만, 그건 인공지능 학과 쪽 대학원을 가서나 생각해보자.