PULSE 2023 Fall

10주차 – 분할정복, 동적 계획법

목차

- PS 관련 팁
- Divide & Conquer
- Dynamic Programming

PS 관련 팁

PS tip - 1

- 코드의 수행시간 예측
  - 수행시간에 영향을 미치는 변수는 많지만 big-O 계산법으로 대략 1억번 연산에 1초 정도 걸림
  - Ex) O(N^3)인 코드, N이 최대 500인 경우
    - 500^3 = 125,000,000 -> 대략 1초 정도 걸림
    - 문제의 제한 시간을 비교하면 이러한 알고리즘으로 접근해도 되는지 알 수 있음.

PS tip - 2

- 제출 시 컴파일 에러?

  - 로컬에서는 정상작동하나 제출 시 컴파일 에러가 발생한 경우
  - 컴파일 결과는 컴파일러 또는 버전 차이로 달라질 수 있음(BOJ는 gcc를 사용)
  - 표준 라이브러리에 없는 함수를 쓰는 경우
  - 전역 변수의 변수명과 라이브러리에서 사용하는 변수명이 겹치는 경우 …

PS tip - 3

- 제출 시 런타임 에러?
  - BOJ에서는 보안상 런타임 에러 원인 및 발생 위치를 자세하게 공개하지 않음.
  - 2021년 7월 부터 일부 런타임 에러 원인을 표시함. https://help.acmicpc.net/judge/rte
  - 자주 발생하는 이유
    - 잘못된 메모리 접근 (Iterator나 배열 인덱스 실수)
    - 스택 오버플로우 (무한 재귀)
    - 0으로 나눔 …

PS tip - 4

- 메모리 초과
  - Int 100만개에 4MB 정도 됨.
  - 동적할당 등의 이유로 메모리 사용량이 유동적으로 변할 때는 메모리 사용량이 최대일 때로 계산하여야 함.

Divide & Conquer

분할 정복

- Divide & Conquer
  - 큰 문제를 용이하게 풀 수 있는 단위로 나눈 다음 합쳐서 해결하는 방법
  - 분할 → 부분 문제 해결 → 조합 과정을 거쳐 결론 도출(정복)
    - 분할: 문제를 더 이상 분할할 수 없을 때까지 동일한 유형의 여러 하위 문제로 나눔.
    - 조합: 하위 문제에 대한 결과를 기존 문제에 대한 결과로 조합.
  - 퀵 정렬, 병합 정렬에서 쓰이는 방식.
  - 함수의 재귀적 호출이 필요하기에 스택 오버플로우 발생 가능.

분할 정복

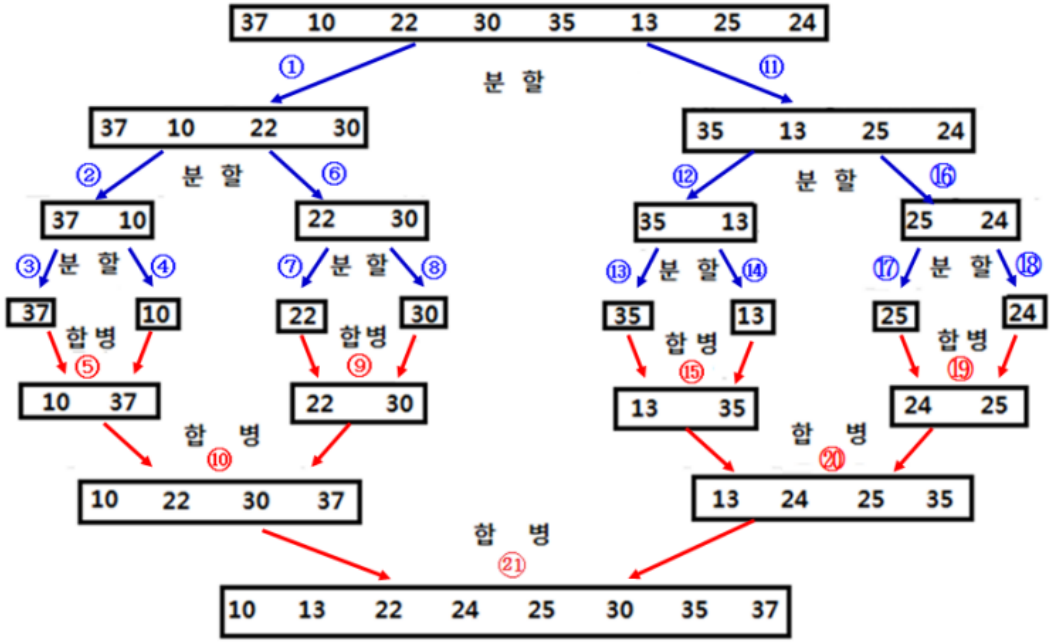
- 문제의 크기가 충분히 작은 경우 직접 해결
- 문제의 크기가 큰 경우 분할하여 해결

```
void DivideAndConquer(InputType in, OutputType out){
    // 문제의 크기가 충분히 작은 경우 직접 해결
    if(in.size() <= Small){
        DirectSolve(in, out);
        return;
    }

    // 문제를 K개의 부분 문제로 분할함
    InputType in_small[K] = Divide(in, K);
    OutputType out_small[K];
    for(int i=0; i<K; i++){
        DivideAndConquer(in_small[i], out_small[i]);
    }
    out = Combine(out_small[0], out_small[1], ... , out_small[k-1]);
}
```

분할 정복

- 합병 정렬(Merge Sort)
  - 정렬 알고리즘 중 하나
  - 크기가 1일 때가지 분할 후 합병 시 정렬하면서 합병
  - 합병 대상 간 비교를 통해 다음 합병 가능
- Worst case:
  - W(n) = W(h) + W(m) + h + m – 1
  - W(n) = W($\lfloor \frac{n}{2} \rfloor$) + W($\lfloor \frac{n}{2} \rfloor$) + n – 1
  - W(n) ∈ O($nlgn$)



Dynamic Programming

동적 계획법

- DP – Dynamic Programming
  - 큰 문제를 작은 문제로 나누어서 풀이하는 방식의 알고리즘
  - 두 가지 속성 – Overlapping Subproblem / Optimal Substructure
- 분할 정복과의 차이
  - 동적 계획법은 나눠진 문제가 중복될 수 있다.
  - 동적 계획법은 답을 한 번만 계산하고 그 결과를 여러 번 사용하여 빠르다.
    - Cache(캐시): 이미 계산한 값을 저장해 두는 메모리
    - Overlapping Subproblems(중복되는 부분 문제): 두 번 이상 계산되는 부분 문제

동적 계획법

- 1. Overlapping Subproblem
  - 특정 문제가 여러 개의 부분 문제로 나눠질 수 있는 문제
  - Base case(기저 사례)를 제외한 모든 문제는 작은 문제로 쪼갤 수 있고, 같은 방법으로 문제 해결 가능

  - Ex) 피보나치 수열
    - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
    - F(0) = 0, F(1) = 1
    - F(n) = F(n-1) + F(n-2) (n >= 2)

  - F(n)은 F(n-1)과 F(n-2)의 답을 이용하여 구할 수 있음

동적 계획법

- 2. Optimal Substructure
  - 부분 문제의 최적의 해결책이 전체 문제의 최적의 해결책이 되는 경우
  - 문제의 크기에 상관없이 하나의 문제 정답은 일정함. → 중복되는 문제!
  - 이 값을 매번 구하는 것은 비효율적... → 메모이제이션(Memoization)
- 메모이제이션(Memoization)
  - 정답을 한 번 구하면 저장
  - Cache에 메모하여 불러옴

```
int memo[100];
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        if (memo[n] > 0) {      // memo의 값이 0이 아니면
            return memo[n];     // 그 값을 그대로 사용
        }
        memo[n] = fibonacci(n-1) + fibonacci(n-2);
        return memo[n];
    }
}
```

동적 계획법

- 동적 계획법 구현 방법
  - Bottom-up(상향식) 접근 – Tabulation
    - 반복문 구현
    - 작은 문제부터 풀이
    - 문제의 크기가 점점 커짐
  - Top-down(하향식) 접근 – Memoization
    - 재귀 호출 사용
    - 큰 문제를 나누고, 나눈 문제를 풀고,
      큰 문제를 푸는 방식

동적 계획법

- Ex) factorial 구현
  - Factorial은 n까지의 수를 모두 곱한 것.
  - Factorial(n) = Factorial(n-1) * n
  - **Bottom-up(상향식) 접근** – Tabulation
    - 1부터 차례대로 팩토리얼 값들을 구하고, 재사용하면서 Factorial(n)에 접근 가능

```
#include <bits/stdc++.h>
#define MAX 101

using namespace std;

int dp[MAX];

void factorial(int n){
    dp[0] = 1;
    for(int i = 1; i <= n; i++){
        dp[i] = dp[i-1] * i;
    }
}

int main(){
    int n;
    cin >> n;
    factorial(n);
    cout << dp[n];
}
```

동적 계획법

- Ex) factorial 구현
  - **Top-down(하향식) 접근** – Memoization
    - 결과값에 먼저 접근 시도

```cpp
#include <bits/stdc++.h>
#define MAX 101

using namespace std;

int dp[MAX];

int factorial(int n){
    if (n <= 1) return 1;
    if(dp[n] > 0) return dp[n];
    dp[n] = factorial(n-1) * n;
    return dp[n];
}

int main(){
    int n;
    cin >> n;
    cout << factorial(n);
}
```