분할정복 - Concept

#분할정복

분할정복이란?

1. 주어진 문제를 둘 이상의 부분 문제로 나누고 각 부분 문제에 대한 답으로부터 전체 문제의 답을 도출해내는 것
2. 재귀와의 차이점
    - 재귀는 문제를 한 조각 씩 떼내어서 본다
    - 분할정복은 부분 문제를 비슷한 크기로 쪼개어 본다

분할정복의 구성요소

1. 문제를 더 작은 문제로 분할(divide)
2. 각 문제에 대해 구한 답을 원래 문제에 대한 답으로 병합(merge)
3. 더 답을 분할하지 않고 곧장 풀 수 있는 매우 작은 문제(base case)

단순 재귀호출과 분할정복의 차이

1. 단순 재귀호출

```
// 필수 조건 : n >= 1
// 결과 : 1부터 n까지의 합을 반환
int recursiveSum(int n) {
        if (n == 1)
                return 1;
        return n + recursiveSum(n - 1);
}
```

2. 분할정복

```
// 필수 조건 : n은 자연수
// 결과 : 1부터 n까지의 합을 반환
int divideConquerSum(int n) {
        if (n == 1)
                return 1;
        if (n % 2 == 1)
                return fastSum(n - 1) + n;
        return 2 * fastSum(n / 2) + (n / 2) * (n / 2);
}
```

3. 둘의 시간복잡도 분석
    - 두 함수는 전부 반복문이 존재하지 않기에 함수가 호출되는 횟수에 비례한다.
    - recursiveSum()은 n번의 함수 호출이 필요하며 fastSum()은 호출할 때마다 반씩 계산량이 줄어드므로 호출 횟수가 lg n 만큼으로 감소한다
        - 참고사항 : lg는 이진로그의 기호다. 이진로그란 로그의 밑이 2인 걸 의미한다.
    - 위 시간 복잡도를 행렬의 곱셈에 빗대어보자. 행렬의 곱셈은 가장 단순한 알고리즘을 적용하면 시간 복잡도가 O(n^3)이다. 즉, 일일히 하나씩 쪼개서 계산하면 너무 오래걸리지만, 분할 정복을 이용하면 3log n이므로 훨씬 시간이 줄어드는 걸 알 수 있다.

분할정복과 정렬

1. 분할 정복을 이용한 정렬에는 병합 정렬과 퀵 정렬이 있다.
2. 병합정렬은 배열을 쪼개질 수 없을 때까지 절반씩 쪼개고, 다시 합쳐나가면서 부분 배열 원소들 간의 비교로 정렬을 시도한다.
3. 퀵 정렬은 배열 내부의 임의의 기준값(pivot)을 잡고 기준보다 작은 값은 왼쪽, 더 큰 숫자는 오른쪽으로 보내 배열을 쪼갠다. 더는 쪼개질 수 없으면 왼쪽과 오른쪽으로 나뉜 배열들을 정렬하며 합쳐간다.
4. 병합 정렬과 퀵 정렬의 시간복잡도
    - 병합 정렬은 각 단계별 정렬하는 원소의 개수는 n개로 동일하다. 단계는 문제가 밑으로 내려갈 수록 절반으로 줄어드므로 필요한 단계의 수는 lg n이다. 원소 개수 *총 단계* = *n* lg n이므로 시간복잡도는 O(n * lg n)
    - 퀵 정렬은 병합 정렬과 마찬가지로 각 단계별 정렬하는 원소의 개수는 n개이다. 문제는 단계별로 잡은 기준점에 따라 부분 문제가 절반에 가깝게 나뉠지 아니면 한 개만 떼어놓는 일이 될 지 알 수 없다. 그러므로 최악의 경우에는 단순한 재귀 호출과 같은 O(n^2)이며 평균적으

로는 O(n * lg n)이다.

카라츠바의 빠른 곱셈 알고리즘

1. 두 개의 정수를 곱하는 알고리즘. 주로, 수백자리나 수만자리 이상의 큰 수를 다룰 때 사용한다. 매개변수로 받은 두 정수의 자릿수를 각각 절반으로 쪼개고 세 개의 조각으로 나누어서 계산을 시도한다.

```
// a와 b는 정수이며 각각 자릿수가 256자리이다.
a = a1 * 10^128 + a0 // a1은 a의 0 ~ 127번째 자릿수, a0는 a의 128 ~ 255번째 자릿수
b = b1 * 10^128 + b0 // b1은 b의 0 ~ 127번째 자릿수, b0는 b의 128 ~ 255번째 자릿수
a * b = a1 * b1 * 10^256 + (a1 * b0 + a0 * b1) * 10^128 + a0 * b0
(a0 + a1) * (b0 + b1) = a0 * b0 + a1 * b0 + a0 * b1 + a1 * b1
// 위 식을 이용하여 아래와 같이 세 조각으로 구분한다
z2 = a1 * b1
z0 = a0 * b0
z1 = (a0 + a1) * (b0 + b1) - z0 - z2
```

2. 해당 수식을 이용하여 구현한 알고리즘

```
// a += b * (10^k)를 구현한다
void addTo(vector<int>& a, const vector<int>& b, int k);
// a -= b를 구현한다. a >= b를 가정한다
void subForm(vector<int>& a, const vector<int>& b);
// 두 긴 정수의 곱을 반환한다
vector<int> karatsuba(const vector<int>& a, const vector<int>& b) {
        int an = a.size(), bn = b.size();
        // a가 b보다 짧을 경우 둘을 바꾼다
        if (an < bn)
                return karatsuba(b, a);
        // 기저사례 : a나 b가 비어 있는 경우
        if (an == 0 || bn == 0)
                return vector<int>();
        // 기저사례 : a가 비교적 짧은 경우 O(n^2) 곱셈으로 변경한다
```

```
        if (an <= 50)
                return multiply(a, b);
        int half = an / 2;
        // a와 b를 밑에서 half 자리와 나머지로 분리한다
        vector<int> a0(a.begin(), a.begin() + half);
        vector<int> a1(a.begin() + half, a.end());
        vector<int> b0(b.begin(), b.begin() + min<int>(b.size(), half));
        vector<int> b1(b.begin() + min<int>(b.size(), half), b.end());
        // z2 = a1 * b1
        vector<int> z2 = karatusba(a1, b1);
        // z0 = a0 * b0
        vector<int> z0 = karatsuba(a0, b0);
        // a0 = a0 + a1; b0 = b0 + b1
        addTo(a0, a1, 0); addTo(b0, b1, 0);
        // z1 = (a0 * b0) - z0 - z2
        vector<int> z1 = karatsuba(a0, b0);
        subFrom(z1, z0);
        subFrom(z1, z2);
        // ret = z0 + z1 * 10^half + z2 * 10^(half * 2)
        vector<int> ret;
        addTo(ret, z0, 0);
        addTo(ret, z1, half);
        addTo(ret, z2, half + half);
        return ret;
}
```

3. 카라츠바의 빠른 곱셈 알고리즘의 시간복잡도
    - 곱셈의 경우, 자릿수 n이 2의 거듭제곱 2^k라고 가정하면 재귀 호출의 깊이가 k다
    - 재귀를 불러서 한 번 쪼갤 때마다 3분할이 이루어지므로 마지막 단계까지 계산이 이루어지면 부분 문제의 개수는 3^k 개다
    - 분할 정복의 특성 상, 재귀 호출은 로그 증가를 보이므로 시간 복잡도는 O(3^(lg n))이 된다

그래서 분할정복은 언제 쓰면 좋은데요?

1. 문제를 나눌 수 있는가?
    - 문제를 분할할 수 없으면 당연히 분할 정복은 사용할 수 없다. 문제를 분할해서 푸는 게 가능한 경우에는 분할정복을 사용하여 lg n으로 최대한 줄일 수 있으므로 사용해 봐도 좋을 것이다
2. 1번을 만족하는 동시에 완전 탐색으로는 시간 복잡도가 초과하는 경우
    - 기실 모든 경우의 수를 탐색하여 답을 내놓는 것이 가장 간단한 방법이다. 하지만 코딩 테스트에서 해당 방법으로 알고리즘을 구성했는데 시간 초과가 나오면 결국 분할 정복을 통해 시간을 줄여보는 게 낫다. lg n으로도 해결이 안되는 수준이라면 분명 다른 방식의 알고리즘을 요구하는 것이다. 그 때, 적절한 알고리즘이 떠오르지 않는다면 부분 점수라도 챙기자