분할 정복을 이용한 거듭제곱

수학에서 거듭제곱은 큰 수를 만들거나 표현하는 가장 쉽고 대표적인 방법이다. 우리는 학교 수학 수업시간에서 미적분의 지수함수나 확률과 통계의 경우의 수를 구하는 과정 등 알게 모르게 거듭제곱을 사용해왔다.

그 외에도 현재 암호화 알고리즘으로 많이 사용되는 RSA 방식에서 또한 사용되며, 컴퓨터에서 사용하는 IEEE 754 의 부동소수점 방식 또한 일반적으로 2 진수(binary)로 표현된 수 1.x 에 2 의 거듭제곱을 곱한 방식으로 실수를 표현하고 있다. 이에 따라 컴퓨터를 사용한 연산 의존성이 점점 늘어남에 따라 거듭제곱은 다양한 분야로 사용되고 있다.

이번 회지에서는 컴퓨터를 사용해 정수와 행렬의 빠른 거듭제곱을 수행할 수 있는 알고리즘을 제시하고, 이를 적용할 수 있는 예시를 알아본다. 그리고 다항식 계산 방식을 사용해 고속으로 연산할 수 있는 알고리즘을 통해 앞에서 소개한 방식과 얼마나 차이가 나는지 비교해본다.

작성자 **51** 기 오상수

sg05008@naver.com

1. Introduction

프로그래밍으로 어떤 정수의 거듭제곱을 구하는 코드를 작성한다면 일반적으로 지수 값만큼 반복하는 반복문을 통해 값을 구하게 된다. 예를 들어, 아래는 두 정수 a, N 을 입력 받아 a^N 을 구하는 코드이다.

```c
// N번 진행하는 반복문 사용
#include <stdio.h>

int main()
{
  long long N, a, x = 1;
  scanf("%lld%lld", &a, &N);

  for(int i = 1; i <= N; i++)
    x *= a;

  printf("%lld", x);
  return 0;
}
```

Figure 1. 임의의 정수의 거듭제곱을 구하는 C 코드

그런데, N 의 값이 매우 커진다면 정상적인 결과를 출력하지 않게 된다. 64-bit 자료형이 저장할 수 있는 최대 값은 unsigned 기준 $2^{64}-1 \approx 1.8\times10^{19}$ 이므로 오버플로우(overflow) 문제로 인해 불가능하다고 답할 수 있지만, 이는 실수형 변수(float, double 등)를 사용해도 해결할 수 없을 것이다.

따라서, 문제를 조금 바꿔 최종 결과값을 특정 수로 나눈 나머지를 구한다고 하자. 그렇다면 답 자체는 표현할 수 있는 범위에 속하지만, 구하는 과정에서 오버플로우가 발생하면 결국 틀린 답을 출력한다. 설령 이를 해결해도 위의 코드는 $O(N)$의 시간 복잡도를 가지므로 반복문을 탈출할 수 없게 된다. 따라서 이번 회지에서는 이에 대한 해결 방법 및 시간 복잡도를 줄이는 방안을 제시하고자 한다.

2. Background Theory

컴퓨터는 내부적으로 0 과 1 을 의미하는 비트들과 이를 처리하는 여러 비트 연산만으로 모든 것을 계산한다. 여기서 비트, 다시 말해 이진수(binary) 표현의 특징으로는 N 개의 문자를 사용해 0 부터 2^N-1 의 음이 아닌 모든 정수를 표현 가능하다. 이를 바꿔 말하면, 모든 정수는 다음과 같은 2 의 거듭제곱의 합의 형태로 표현할 수 있다. 5 와 3771 을 예시로 들면 다음과 같다.

$$5_{(10)} = 4 + 1 = 2^2 + 2^0 = 101_{(2)}$$
$$3771_{(10)} = 2048 + 1024 + 512 + 128 + 32 + 16 + 8 + 2 + 1$$
$$= 2^{11} + 2^{10} + 2^9 + 2^7 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0 = 111010111011_{(2)}$$

단, 이 방법만으로는 큰 수를 다루는 거듭제곱 연산의 단점으로 계산 과정에서의 오버플로우를 피할 수 없다. 하지만 이는 나머지 연산의 수학적인 성질을 이용하여 해결할 수 있다. 정수 A, B, m 에 대해 다음 성질을 만족한다.

$$(A \times B) \bmod m = \{(A \bmod m) \times (B \bmod m)\} \bmod m$$

즉, 어떤 두 정수를 m 으로 나눈 나머지만을 취해 그것들을 곱하고 다시 나머지를 취해주면, 두 정수의 곱을 m 으로 나눈 나머지와 같다는 것이다. 충분히 큰 두 정수의 곱으로 인해 오버플로우가 일어나기 전에 먼저 나머지 연산을 취해주고 결과값에서 다시 한 번 나머지 연산을 취해 오버플로우 발생을 방지한다는 것이다.

3. Exponentiation Method

위 문단에서 제시한 방법을 사용하여 음이 아닌 두 정수 A, N 에 대해 A^N 를 구하는 방법을 생각해보자. 결과가 매우 클 수 있으므로 m 으로 나눈 나머지 $A^N \bmod m$ 을 구한다고 할 때, N=11 을 예를 들어 지수를 2 의 거듭제곱의 합 형태로 나타내면, $A^{11} = A^{1+2+8} = A^{1+2+8} = A^1 \times A^2 \times A^8$ 가 된다.

여기서 A 의 거듭제곱들은 서로가 거듭제곱 관계에 있음을 볼 수 있다. 자세히 얘기하면, A^8 의 경우 A 를 8 번 곱하여 구하는 것이 아닌 A 를 제곱하여 A^2, A^2 를 제곱하여 A^4, A^4 를 제곱하여 A^8 을 구할 수 있으므로 3 번만 반복하면 구할 수 있다. 이 과정에서 자연스럽게 모든 거듭제곱 수를 구할 수 있으므로 각각의 나머지만을 곱해주면 답을 구할 수 있을 것이다. 즉, 아래의 수식을 계산하는 코드를 작성하면 문제를 해결할 수 있다.

$$A^{11} \bmod m = [\{(A^1 \bmod m) \times (A^2 \bmod m) \bmod m\} \times (A^8 \bmod m)] \bmod m$$

앞의 예시처럼, 일반적으로 이미 구한 거듭제곱 수를 이용하여 계산하면 결과적으로 $O(\log N)$의 시간복잡도로 답을 구할 수 있게 된다. 이를 정리하면 아래와 같다.

1. 먼저 A^1, A^2, A^4, A^8, ...을 순서대로 계산한다. 각 수는 이전에 있는 수를 제곱함으로써 계산할 수 있고, 지수가 X 를 딱 넘지 않을 시점까지만 계산하면 충분할 것이다. X가 64비트 정수의 범위에 있으므로 계산하는 수는 64개보다 작을 것이다.
2. 이제 X 를 이진수로 나타내 보자. 예를 들어 X를 11로 두면, X = 11 = 1 + 2 + 8이다. 그런데 지수법칙에 의해, $A^{11} = A^{1+2+8} = A^1 \times A^2 \times A^8$이 성립한다. 이를 통해 1번 단계에서 미리 계산해 놓았던 수 몇 개만 곱해서 A^X 을 계산할 수 있음을 알 수 있다.

Figure 2. 정수 거듭제곱 방식 정리 (백준 13171 번 문제 설명 중)[1]

N 을 감소시켜도 상관없다면 N 이 홀수일 때만 실제 결과값에 거듭제곱을 곱하고, N 을 절반(소수점 버림)으로 하는 반복문을 구성하면 문제없이 구현된다.

다음으로 거듭제곱을 확장하여 행렬에 적용시켜본다. 이는 단순히 위 문단에서 표기한 A 를 정수가 아닌 행렬로 취급하면 된다. 즉, 행렬 A 를 11 번 거듭제곱하면 아래와 같이 표현된다.

$$A^{11} = A^{1+2+8} = A^{1+2+8} = A^{1} \times A^{2} \times A^{8}$$

따라서, 행렬에서도 곱셈 분할에는 $O(\log N)$의 시간 복잡도를 가지게 된다.

그러나 행렬의 곱은 행과 열의 요소들 각각의 곱들의 합이 하나의 요소로 계산되기 때문에 연산 시간이 행렬 크기에 매우 민감하다.

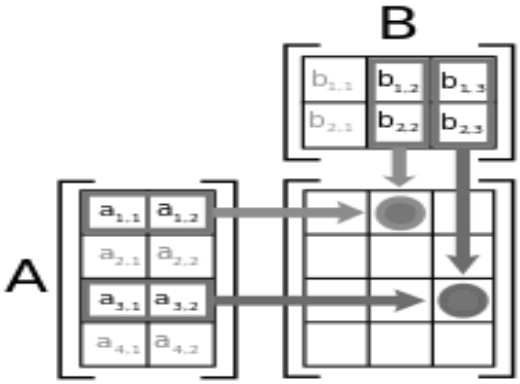

Figure 3. 행렬 곱 모식도 (위키백과 행렬 곱셈 항목 중)[2]

또한, 행렬의 거듭제곱은 정방행렬에서만 정의되므로, 행렬의 크기가 K x K 인 정방행렬의 N 거듭제곱의 시간 복잡도는 아래와 같다.

$$O\left(\left(\text{행렬 곱 시간}\right) \times \left(\text{곱셈 분할}\right)\right) = O(K \times K^2) \times O(\log N) = O(K^3 \log N)$$

즉, 행렬의 크기가 1000 x 1000 정도만 되어도 계산과정에서 나머지 연산이 많이 사용되기 때문에 프로그램의 동작시간은 최소 분 단위로 소요될 수 있다. 따라서, 행렬 곱셈을 사용한 알고리즘의 경우 지수에 비해 크기가 충분히 작은 행렬에 대해서 효과적임을 알 수 있다

4. Examples

분할 정복을 이용한 거듭제곱이 효율적인 것은 알겠으나, 이것만으로는 어디에 어떻게 적용할 수 있을지 와닿지 않을 것이다. 따라서 이를 사용할 수 있는 예시들을 준비해보았다.

1) 피보나치 수 구하기

피보나치 수는 첫째 및 둘째 항이 1 이며 그 뒤의 모든 항은 바로 앞 두 항의 합인 수열이다.

$$\mathrm{F}_1 = 1, \mathrm{F}_2 = 1, \mathrm{F}_{\mathrm{n}+2} = \mathrm{F}_{\mathrm{n}+1} + \mathrm{F}_n \ (\mathrm{n} \geq 1)$$

점화식 부분과 다음의 항등식을 이용하면 피보나치 수가 2 x 2 정방행렬 형태로 나타날 수 있다.

$$\mathrm{F}_{\mathrm{n}+2} = \mathrm{F}_{\mathrm{n}+1} + \mathrm{F}_n = 1 \times \mathrm{F}_{\mathrm{n}+1} + 1 \times \mathrm{F}_{\mathrm{n}}$$
$$\mathrm{F}_{\mathrm{n}+1} = \mathrm{F}_{\mathrm{n}+1} = 1 \times \mathrm{F}_{\mathrm{n}+1} + 0 \times \mathrm{F}_{\mathrm{n}}$$
$$\Rightarrow \begin{pmatrix} \mathrm{F}_{\mathrm{n}+2} \\ \mathrm{F}_{\mathrm{n}+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \mathrm{F}_{\mathrm{n}+1} \\ \mathrm{F}_{\mathrm{n}} \end{pmatrix}$$

위의 과정을 계속 반복하면, 피보나치 수를 다음과 같이 행렬의 거듭제곱 형태로 표현되는 것을 알 수 있다.

$$\begin{pmatrix} \mathrm{F}_{n+2} \\ \mathrm{F}_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \mathrm{F}_{n+1} \\ \mathrm{F}_{n} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \left(\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \mathrm{F}_{n} \\ \mathrm{F}_{n-1} \end{pmatrix} \right) = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \begin{pmatrix} \mathrm{F}_{n} \\ \mathrm{F}_{n-1} \end{pmatrix} = \cdots = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1} \begin{pmatrix} \mathrm{F}_{1} \\ \mathrm{F}_{0} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$
$$\therefore \begin{pmatrix} \mathrm{F}_{n} \\ \mathrm{F}_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \ (\mathrm{n} \geq 1)$$

즉, 거듭제곱하는 행렬의 크기가 2 x 2 로 고정되므로 $\mathrm{O}(\log \mathrm{N})$으로 N 번째 피보나치 수를(정확히는 특정 수로 나눈 나머지를) 구할 수 있다.

2) 등비수열의 합

음이 아닌 정수를 밑과 지수로 하는 등비수열의 합도 $O(N)$의 반복문을 사용하여 값을 구할 수 있다. 그러나 이번에도 N 이 크다면 어떻게 풀 수 있을까? 가장 먼저 고려해볼 수 있는 방법으로는 등비수열의 합 공식일 것이다. 초항이 a, 등비가 r 인 등비수열의 첫째 항부터 n 항까지의 합 S 는 다음과 같다.

$$S = \frac{a(r^n - 1)}{r - 1}$$

즉, r 의 n 거듭제곱을 앞에서 했던 방법으로 $O(\log N)$의 시간 복잡도로 계산하면 나머지는 단순 사칙연산이므로 $O(\log N)$으로 계산할 수 있을 것이다.

하지만, 등비수열의 합 또한 결과값이 매우 크므로 대부분의 경우 등비수열의 합 S 를 특정 수로 나눈 나머지를 계산하는 문제가 많다. 그렇다면 거듭제곱을 계산하는 과정에서의 수들 또한 나머지 연산을 취하게 되는데, 곱셈에서는 성립하던 식이 나눗셈에서는 성립하지 않는 문제점이 있다.

$$(\mathrm{A} / \mathrm{B}) \bmod \mathrm{m} \neq \{(\mathrm{A} \bmod \mathrm{m})/(\mathrm{B} \bmod \mathrm{m})\} \bmod \mathrm{m}$$
$$\Rightarrow \frac{a\{(r^n \bmod \mathrm{m}) - 1\}}{r - 1} \bmod \mathrm{m} \neq \frac{a(r^n - 1)}{r - 1} \bmod \mathrm{m}$$

따라서, 등비급수의 합 공식 대신 다른 방법을 사용해야 한다. 등비수열의 합은 어떤 점화식으로 표현할 수 있을까? 이번에도 n = 11 일 때 앞에서의 방법과 비슷하게 접근해본다. n 을 절반씩 낮추면서 홀수일 때(이진수 표현 기준 LSB 가 1 일 때)만 특정 연산을 진행하는 반복문을 구현한다면 각 단계에서 아래의 기능을 해야 할 것이다.

단계	n 의 값	설명	수식
0	11 (1011_2)	첫 S 의 값을 0 으로 초기화	$\mathrm{S} = 0$
1	11 ($101\mathbf{1}_2$)	n 이 홀수 → S=S+a	$\mathrm{S_{new}} = \mathrm{S} + \mathrm{a} = \mathrm{a}$
2	5 ($10\mathbf{1}_2$)	n 이 홀수 → $\mathrm{S=S+ar+ar^2}$	$\mathrm{S_{new}} = \mathrm{S} + \mathrm{ar} + \mathrm{ar}^2 = \mathrm{a} + \mathrm{ar} + \mathrm{ar}^2$
3	2 ($1\mathbf{0}_2$)	n 이 짝수 → 기존 값 유지	$\mathrm{S} = \mathrm{a} + \mathrm{ar} + \mathrm{ar}^2$
4	1 ($\mathbf{1}_2$)	n 이 홀수 → $\mathrm{S=S+ar^3 \sim ar^{10}}$	$\mathrm{S_{new}} = \mathrm{S} + \mathrm{ar}^3 + \cdots + \mathrm{ar}^{10} = \mathrm{a} + \mathrm{ar} + \cdots + \mathrm{ar}^9 + \mathrm{ar}^{10}$
5	0	n 이 0 이므로 반복 종료	$\mathrm{S} = \mathrm{a} + \mathrm{ar} + \cdots + \mathrm{ar}^9 + \mathrm{ar}^{10}$

Table 1. 등비수열의 합의 단계별 연산

위의 표처럼 정리한 결과, 더하는 값의 규칙성을 찾기 어려워 보이지만 앞에서의 분할 정복 거듭제곱 방식에서 보았던 형태를 그대로 사용할 수 있어 보인다. 이는 곧, '앞에서 제시한 알고리즘 형태로 등비수열의 합 또한 구현할 수 있다.'는 것을 의미한다. 각각의 상황에서 어떤 연산을 진행해야 위와 같은 결과를 얻을 수 있을까?

표의 설명 부분을 유심히 읽어보면 더하는 항의 개수는 모두 2 의 거듭제곱수인 것을 확인할 수 있다. 더 자세히 얘기하면 자릿수 자체가 항의 개수를 의미하는 것이다. 따라서, 추가로 더하는 항들의 총합을 빠르게 계산할 수 있다면 중간 과정에서는 덧셈과 곱셈으로만 이루어져 있으므로 나머지 연산을 정수 연산에서와 같이 편하게 사용할 수 있을 것이다.

그렇지만 위 방법을 그대로 계산하게 되면 어중간한 부분부터 덧셈이 시작되기 때문에 계산하기 번거롭다는 단점이 있다. 어떻게 해야 덧셈을 더욱 편하게 할 수 있을까? 2 의 거듭제곱수의 개수라는 것을 잘 활용해보면 더하는 값들을 a 부터 ar^{2^k-1}까지 앞으로 당겨서 한다면 계산이 좀 더 편해질 것이다. 아래와 같이 수정된 방법으로 구현할 수 있을 것이다.

단계	n 의 값	설명	수식
0	11 (1011_2)	첫 S 의 값을 0 으로 초기화	$S = 0$
1	11 ($101\mathbf{1}_2$)	n 이 홀수 → $S=r^1S+a$	$S_{new} = S + a = a$
2	5 ($10\mathbf{1}_2$)	n 이 홀수 → $S=r^2S+a+ar$	$S_{new} = r^2S + a + ar = r^2(a) + a + ar = a + ar + ar^2$
3	2 ($1\mathbf{0}_2$)	n 이 짝수 → 기존 값 유지	$S = a + ar + ar^2$
4	1 ($\mathbf{1}_2$)	n 이 홀수 → $S= r^8S+a\sim ar^7$	$S_{new} = r^8S + a + \cdots + ar^7$ $= r^8(a + ar + ar^8) + a + \cdots + ar^7$ $= a + ar + \cdots + ar^9 + ar^{10}$
5	0	n 이 0 이므로 반복 종료	$S = a + ar + \cdots + ar^9 + ar^{10}$

Table 2. 등비수열의 합의 단계별 연산 변형

변형한 방법으로 더욱 편리하게 올바른 등비수열의 합을 구할 수 있는 것을 확인했다. 이제 a 부터 ar^{2^k-1}까지의 합만 빠르게 계산하기만 하면 된다. 더하는 항의 개수가 2 의 거듭제곱 수이므로 분할 정복을 이용한 거듭제곱 형태로 표현할 수 있으며, 이 역시 Table 2 의 방법을 그대로 적용하면 만들어 낼 수 있다. 이 과정을 일반화하여 a 부터 ar^{2^k-1}까지의 합을 S_k 라고 할 때, 아래의 식을 유도할 수 있다.

$$S_0 = a$$
$$S_1 = a + ar = S_0 + rS_0$$
$$S_2 = a + ar + ar^2 + ar^3 = S_1 + r^2 S_1$$
$$S_3 = a + ar + ar^2 + ar^3 + ar^4 + ar^5 + ar^6 + ar^7 = S_2 + r^4 S_2$$
$$\Rightarrow S_k = a + ar + \cdots + ar^{2^k - 1} = S_{k-1} + r^{2^k - 1} S_{k-1} = (1 + r^{2^k - 1}) S_{k-1}$$

이를 코드로 작성하는 것이 어렵다고 느낄 수 있지만 매우 간단한 형태로 구현할 수 있다. 기존 형태에서 분기 조건을 만족하면 지금까지 계산한 가중치 r^k 를 곱한 후 S_k 를 더해주면 되고, S_k 와 가중치를 업데이트 하여 반복문을 진행하는 형태로 코드를 작성하면 앞의 예제들과 같이 $O(\log N)$으로 해결할 수 있다.

지금까지의 내용이 와닿지 않거나 코드 작성이 어렵게 느껴진다면, 행렬의 거듭제곱 방식을 그대로 사용할 수도 있다. 등비수열의 합은 다음의 행렬을 이용해 구할 수 있다.[3]

$$\begin{pmatrix} r & 0 \\ a & 1 \end{pmatrix}$$

이를 거듭제곱 하면 a 의 값이 등비급수의 형태로 전개되는 것을 볼 수 있다.

$$\begin{pmatrix} r & 0 \\ a & 1 \end{pmatrix}^2 = \begin{pmatrix} r & 0 \\ a & 1 \end{pmatrix}\begin{pmatrix} r & 0 \\ a & 1 \end{pmatrix} = \begin{pmatrix} r^2 & 0 \\ a(r+1) & 1 \end{pmatrix}$$
$$\begin{pmatrix} r & 0 \\ a & 1 \end{pmatrix}^3 = \begin{pmatrix} r^2 & 0 \\ a(r+1) & 1 \end{pmatrix}\begin{pmatrix} r & 0 \\ a & 1 \end{pmatrix} = \begin{pmatrix} r^3 & 0 \\ a(r^2+r+1) & 1 \end{pmatrix}$$
$$\begin{pmatrix} r & 0 \\ a & 1 \end{pmatrix}^4 = \begin{pmatrix} r^3 & 0 \\ a(r^2+r+1) & 1 \end{pmatrix}\begin{pmatrix} r & 0 \\ a & 1 \end{pmatrix} = \begin{pmatrix} r^4 & 0 \\ a(r^3+r^2+r+1) & 1 \end{pmatrix}$$
$$\therefore \begin{pmatrix} r & 0 \\ a & 1 \end{pmatrix}^k = \begin{pmatrix} r^k & 0 \\ a(r^k + r^{k-1} + \cdots + r + 1) & 1 \end{pmatrix}$$

앞에서 설명한 내용 또한 위의 수식을 a 의 관점에서 풀어서 설명한 것이며, 궁극적으로 동일하다. 여기까지 봐도 잘 이해가 가지 않는다면 정수 및 행렬의 거듭제곱 부분을 다시 읽고 천천히 코드로 구현해보는 것을 권장한다.

3) 그래프에서 특정 길이를 가진 경로의 '경우의 수' 구하기

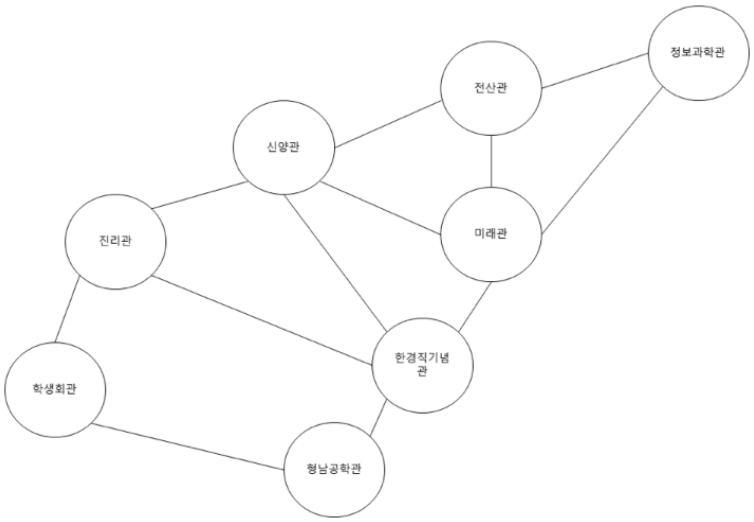

Figure 4. 여러 정점 간 연결이 이루어진 그래프 (백준 12850 번 문제 설명 중)[4]

위와 같은 그래프가 문제에서 주어진다고 하자. 즉, 각각의 정점은 아래와 같이 연결되어있다

- 학생회관 : 진리관, 형남공학관
- 형남공학관 : 학생회관, 한경직기념관
- 진리관 : 학생회관, 한경직기념관, 신양관
- 한경직기념관 : 형남공학관, 진리관, 신양관, 미래관
- ...

위 그래프에서 1 분에 한 간선씩 움직일 수 있을 때, 8 곳 중 임의의 출발지와 도착지 사이에서 T 분의 시간으로 만들 수 있는 경로의 경우의 수는 얼마나 될까? 단, 그 수가 너무 커질 수 있으므로 특정 값으로 나눈 나머지를 구해보자.

먼저 움직이지 않은 상황(t=0)에서는 경우의 수는 직관적으로 출발지와 도착지가 같은 정점이라면 1, 아니라면 0 이라는 것을 알 수 있다. 그렇다면 t 가 0 보다 크다면 어떻게 될까? 1 분이 지난 시점(t=1)에서도 역시 직관적으로 1 분에 한 간선을 움직일 수 있으므로 출발지와 도착지가 서로 연결된 정점이라면 1, 아니라면 0 이다.

그렇지만 2 분이 지난 시점부터는 상황이 복잡해진다. 서로 다른 간선을 건너갈 수도 있고, 동일한 간선을 통해 출발지로 되돌아올 수도 있다. 이 과정에서 출발지와 도착지 사이의 연결에 따라 0, 1 또는 그보다 큰 수일 수도 있다. 여기서부터는 직관으로는 해결하기 어려워진다. 관점을 바꿔서 생각해보자.

출발지, 도착지의 입장이 아니라 움직이는 행위의 입장에서 확인해보면 t=1 에서는 출발지와 연결된 정점의 경우의 수가 전부 1 이 된다. 이외의 정점은 전부 0 이다. 그리고 t=2 는 인접한 t=1 에서 1 이 된 정점의 개수가 경우의 수가 된다.

t=3 이후에는 어떻게 계산을 해야 할까? 개수를 좀 더 확장해서 생각해보면 세는 것을 하나하나 전부 더하는 것으로 받아들일 수 있다. 즉, t=3 시점의 경우 t=2 시점의 인접한 정점들의 값들을 더한 총합을 구하면 된다.

이를 반복해서 생각하면 결국 이 문제도 재귀적으로 설명할 수 있다. 연결된 정점들에 대해 1 분 전의 경우의 수를 모두 더하면 현재 그 정점으로 도착할 수 있는 경우의 수를 구할 수 있는 것이다. 예를 들어, T 분 동안 정보과학관에서 출발해 정보과학관으로 돌아오는 경우의 수를 구한다고 하자.

정보과학관을 0 번으로 반시계 방향으로 정점의 번호를 매기고 t 분 후 k 번 정점에 도착하는 경우의 수를 $n(k,t)$라고 했을 때, 다음의 식이 성립한다.

$$\begin{pmatrix} n(0,t+1) \\ n(1,t+1) \\ n(2,t+1) \\ n(3,t+1) \\ n(4,t+1) \\ n(5,t+1) \\ n(6,t+1) \\ n(7,t+1) \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} n(0,t) \\ n(1,t) \\ n(2,t) \\ n(3,t) \\ n(4,t) \\ n(5,t) \\ n(6,t) \\ n(7,t) \end{pmatrix}$$

위 행렬을 자세히 보면 0 과 1 로 이루어진 boolean 행렬은 그래프의 인접행렬처럼 보인다. 이는 예시로 주어진 그래프가 방향성이 없는 간선으로만 이루어진 그래프이기 때문이고, 일반적인 방향 그래프에서 점화식을 하나씩 나열해서 행렬을 만들어보면 인접행렬의 전치행렬이 만들어지는 것을 쉽게 확인할 수 있다. (정확히는 각 정점끼리 연결된 '개수'로 이루어진 행렬이 된다. 즉, 동일한 정점들끼리의 간선이 여러 개인 그래프라면 불 행렬이 아니게 된다.)

따라서, 앞에서의 예제들과 비슷하게 위의 행렬을 거듭제곱하면 T 분 동안 정보과학관에서 출발하여 정보과학관으로 돌아오는 경로의 경우의 수를 구할 수 있다. 아래의 위 첨자 T 는 전치(Transpose)가 아닌 행렬의 T 번 거듭제곱을 의미한다.

$$\begin{pmatrix} n(0,t+1) \\ n(1,t+1) \\ n(2,t+1) \\ n(3,t+1) \\ n(4,t+1) \\ n(5,t+1) \\ n(6,t+1) \\ n(7,t+1) \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}^{T} \times \begin{pmatrix} n(0,t) \\ n(1,t) \\ n(2,t) \\ n(3,t) \\ n(4,t) \\ n(5,t) \\ n(6,t) \\ n(7,t) \end{pmatrix}$$

결론적으로 피보나치 수 및 그래프 예제를 정리하면 다음과 같은 결론을 이끌어 낼 수 있다.

문제의 조건이 최대/최소 인덱스의 차가 K 인 선형 점화식으로 표현이 가능하다면,
이를 행렬로 표현하여 분할 정복을 이용한 거듭제곱을 이용해 $O(K^3 \log N)$으로 해결할 수 있다.

5. Kitamasa method

앞에서 설명하였듯이 문제를 적절히 인접한 항들의 선형 점화식으로 표현할 수 있다면, 이를 행렬로 표현하여 분할 정복을 이용한 거듭제곱을 이용해 $O(K^3 \log N)$으로 해결할 수 있음을 보였다. 그런데, 그래프의 예제에서는 각 행이 유의미한 정보를 담고 있지만, 피보나치 수열처럼 주어진 선형 점화식을 그대로 사용하는 문제는 그렇지 않다는 것을 감이 좋다면 눈치를 챘을 것이다.

즉, 아래의 피보나치 수의 행렬 표현에서 2 번째 행은 $F_{n+1} = F_{n+1}$ 이므로 정보가 없는 부분이다. 다시 말해, 행렬의 거듭제곱을 위한 더미 데이터이며, 이로 인해 알고리즘의 성능이 낮아지는 것이다.

$$\begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$$

실제로 점화식으로서 정보가 있는 것은 첫째 항이고, 이것만 사용하여 적절히 연산을 하여 행렬을 이용한 거듭제곱과 동일한 결과를 만들어내는 방법이 없을까? 답은 Yes 이다. 이러한 문제는 행렬의 거듭제곱이 아닌 '키타마사법'이라는 알고리즘을 이용해 $O(K^2 \log N)$, 더욱 빠르게 하면 $O(K \log K \log N)$에 계산할 수 있다. 이번 회지에서는 비교적 단순한 $O(K^2 \log N)$ 방식만 설명하도록 한다.[5]

키타마사법이라는 알고리즘은 선형 점화식을 변형하여 점화식을 초항들로만 이루어지게 하는 다음의 형태를 찾는 것이다

$$A_N = c_1 A_{N-1} + c_2 A_{N-2} + \cdots + c_{K-1} A_{N-(K-1)} + c_K A_{N-K} = \sum_{i=1}^{K} c_i A_{N-i}$$

$$\Rightarrow A_N = d_0 A_0 + d_1 A_1 + \cdots + d_{K-2} A_{K-2} + d_{K-1} A_{K-1} = \sum_{i=0}^{K-1} d_i A_i$$

이는 수열의 항을 하나씩 낮추면서 직접 찾을 수 있다. 피보나치 수열처럼 $c_1=c_2=1$ 이고 N=5 일 때를 예시로 수열 d_i 를 구해보면 다음과 같다.

- $A_5 = A_4 + A_3$
- $A_5 = (A_3 + A_2) + A_3 = 2A_3 + A_2$
- $A_5 = 2(A_2 + A_1) + A_2 = 3A_2 + 2A_1$
- $A_5 = 3(A_1 + A_0) + 2A_1 = 5A_1 + 3A_0$
- $d_1 = 5, d_0 = 3$

이 과정을 다른 방식으로 접근하면 양변에 $A_x - c_1 A_{x-1} - c_2 A_{x-2} - \cdots = 0$의 상수배를 빼는 과정으로도 볼 수 있다. 즉, 위의 예시의 경우 $A_x - A_{x-1} - A_{x-2} = 0$의 상수배를 빼는 것이다.

- $A_5 = A_4 + A_3$
- $A_5 = (A_4 + A_3) - (A_4 - A_3 - A_3) = 2A_3 + A_2$
- $A_5 = (2A_3 + A_2) - 2(A_3 - A_2 - A_1) = 3A_2 + 2A_1$
- $A_5 = (3A_2 + 2A_1) - 3(A_2 - A_1 - A_0) = 5A_1 + 3A_0$
- $d_1 = 5, d_0 = 3$

이를 자세히 보면 다항식 나눗셈과 매우 유사한 것 같으며, 실제로 x^N을 아래의 다항식 $f(x)$로 나눈 나머지를 구하는 과정과 동일하다.

$$f(x) = x^K - c_1x^{K-1} - c_2x^{K-2} - \cdots - c_Kx^0 = x^K - \sum_{i=1}^{K} c_ix^{K-i}$$

예시를 대입해보면 x^5를 $x^5 - 2x - 1$로 나눈 나머지를 구하는 과정과 대응하며, 이를 직접 계산해보면 몫의 각 계수들이 위 과정에서의 계수(상수배), 나머지의 각 계수들이 d_i와 일치하게 된다.

즉, 행렬곱 연산이 다항식의 곱셈 및 나머지 연산 $x^N \bmod f(x)$로 바뀌게 되며, 이를 통해 $O(K^3 \log N)$보다 빠르게 계산할 수 있다. 다항식에서도 정수의 거듭제곱에서 사용했던 제곱 및 나머지 성질을 그대로 사용할 수 있으므로 x^N을 $x^1, x^2, 4, \ldots, x^{2^k}$으로 분해하고, 나머지 연산을 취한다.

정리하면, 아래의 식을 계산하는 알고리즘을 통해 수열 d_i, 다시 말해 다항식 각 항의 계수를 구한다.

$$x^N \bmod f(x) = \left\{\left(x^1 \bmod f(x)\right) \times \left(x^2 \bmod f(x)\right) \times \cdots \times \left(x^{2^k} \bmod f(x)\right)\right\} \bmod f(x) = \sum_{i=0}^{K-1} d_ix^i$$

이후, 각 계수들과 주어진 초항들을 곱해 원래 구하고자 했던 값인 $A_N \bmod m$을 구하면 된다.

$$A_N \bmod m = \sum_{i=0}^{K-1} (d_iA_i \bmod m) \bmod m$$

수열 d_i를 구하는 과정에서 $x^1 \bmod f(x), x^2 \bmod f(x), \ldots, x^{2^k} \bmod f(x)$를 각각 구할 때는 다음의 순서를 따르면 된다.

1. x^1부터 시작하는 x^n 다항식, $x^0 = 1$부터 시작하는 결과 다항식을 각각 설정한다. 다항식은 1 차원 배열을 이용한다.
2. x^n이 x^N에 포함되는 경우 x^n과 결과를 곱한다. 이후, 결과 다항식의 차수가 K 미만이 되도록 f(x)로 나눈 나머지를 취한다.
3. x^n을 제곱하고 차수가 K 미만이 되도록 f(x)로 나눈 나머지를 취한다.
4. 2~3 과정을 n 이 N 에 도달할 때까지 반복한다.
5. 반복이 종료되면 배열의 각 요소가 d_i가 된다.

이 과정을 naive 하게 구현하면 다항식 곱셈 및 나눗셈에 $O(K^2)$의 시간이 걸리며, 이러한 곱셈 및 나눗셈이 $O(\log N)$번 발생하므로 키타마사법을 통한 시간복잡도는 다음과 같다.

$$O\left(\left(\text{다항식 곱셈\&나눗셈 시간}\right)\times\left(\text{곱셈 분할}\right)\right)=O(K^2\log N)$$

6. Results

같은 조건에 대해 일반적인 행렬 거듭제곱을 이용한 소스코드 및 결과와 키타마사법을 적용한 소스코드 및 결과에 대해 비교해본다. 두 C 소스 코드는 온라인 채점 테스트 사이트인 "백준"의 23819 번 문제 '묻고 더블로 마셔'에 대한 예제 코드이며, 소스 코드 하단의 것이 각각의 제출 결과이다.

```c
#include <stdio.h>

int n,k,a[100],P,i,j,l;
long long x[100][100],I[100][100],s,t;

void M(long long T[][100])
{
  long long X[k][k];
  for(i=0;i<k;i++)
    for(j=0;j<k;X[i][j++]=t)
      for(t=l=0;l<k;l++)
        t+=x[i][l]*T[l][j],t=t<P?t:t%P;

  for(i=0;i<k;i++)
    for(j=0;j<k;j++)
      T[i][j]=X[i][j];
}

int main()
{
  for(scanf("%d%d",&n,&k);i<k;scanf("%d",a+i++))
    I[i][i]=x[0][i]=x[i][i-(i>0)]=1;

  scanf("%d",&P);
  for(n-=k;n;n>>=1)
  {
    if(n&1)
      M(I);
    M(x);
  }

  for(i=0;i<k;i++,s=s<P?s:s%P)
    s+=I[0][i]*a[k-i-1];
  printf("%lld",s);
  return 0;
}
```

맞았습니다!!	1272 KB	540 ms	C99 / 수정

```c
#include <stdio.h>
typedef long long L;

int n,k,P,a[100],A[100],B[100]={0,1,},i,j;

void M(int T[])
{
  L t[200]={0,},z;
  for(i=0;i<k;i++)
    for(j=0;j<k;t[i+j++]=z<P?z:z%P)
      z=t[i+j]+(L)T[i]*B[j];

  for(i=k+k-1;--i>=k;)
    for(j=i-1;j>=i-k;t[j--]=z<P?z:z-P)
      z=t[j]+t[i];

  for(i=0;i<k;i++)
    T[i]=t[i];
}

L K()
{
  for(n-=k+1;n;n>>=1)
  {
    if(n&1)
      M(A);
    M(B);
  }

  L s;
  for(s=i=0;i<k;i++)
    s+=(L)A[i]*a[i],s=s<P?s:s%P;
  return s;
}

int main()
{
  for(scanf("%d%d",&n,&k);i<k;A[i++]=1)
    scanf("%d",a+i);
  printf("%lld",k>scanf("%d",&P)?K():*a%P);
  return 0;
}
```

맞았습니다!!	1116 KB	4 ms	C99 / 수정

Figure 5-1 ~ 5-4. 행렬의 거듭제곱 및 키타마사법 C 코드 및 결과 비교

앞에서 정리했던 것처럼 행렬의 크기를 최대 100 까지 할당하기 때문에 소요 시간을 확인하면 135 배 정도의 유의미한 차이를 볼 수 있다. 즉, 앞에서 비교했던 행렬 크기만큼의 차이가 나게 된다. 또한, 행렬의 경우 2 차원 배열 형태로 저장해야 하므로, 1 차원 배열만 사용하는 키타마사법보다 메모리도 더 많이 소모함을 확인할 수 있다.

7. Conclusion

이번 회지를 통해 거듭제곱을 빠르게 계산하는 방법을 알아보았다. 이는 큰 수를 여러 개의 작은 수로 나누어 다항 시간이 걸리는 알고리즘을 log scale 로 낮추는 전형적인 분할 정복 방식을 사용한다는 것을 볼 수 있다.

또한, Figure 5 를 보면 실제 알고리즘이 적용되는 부분에 n 의 크기를 절반으로 하면서 홀수일 때만 결과값을 서브루틴의 인자로 보내고 예비 변수(혹은 배열)을 업데이트 하면 되는 생각보다 단순한 방식을 사용함을 볼 수 있다. 이처럼 분할 정복을 사용하면 생각보다 단순하게 지수시간이 걸리는 알고리즘을 다항시간으로, 다항시간이 걸리는 알고리즘을 로그 시간으로 빠르게 처리할 수 있음을 보여준다.

그리고 키타마사법의 예시에서는 다항식의 곱셈 및 나눗셈을 naive 하게 적용하였다. 사전 지식이 있는 독자라면 다항식 곱셈을 더욱 빠르게 처리해주는 고속 푸리에 변환(Fast Fourier Transform; FFT) 알고리즘을 추가로 사용한다면 $O(K \log K \log N)$의 시간복잡도로 해결할 수 있다는 사실도 아마 눈치를 챘을 것이다. FFT 알고리즘을 아는 분들이라면 이 또한 적용해보길 바란다는 말을 마지막으로 글을 마친다.

Reference

[1] Backjoon Online Judge, "A"(13171) 제 4 회 kriiicon P1 번 문제
https://www.acmicpc.net/problem/13171

[2] 위키백과, "행렬 곱셈"
https://ko.wikipedia.org/wiki/%ED%96%89%EB%A0%AC_%EA%B3%B1%EC%85%88

[3] song seong min, "백준 15712 번: 등비수열" 백준 15712 번 등비수열 문제 행렬을 이용한 풀이 방법
https://memoacmicpc.tistory.com/24

[4] Backjoon Online Judge, "본대 산책 2"(12850) 숭실대학교 SCCC 2016 Summer Contest L 번 문제
https://www.acmicpc.net/problem/12850

[5] JusticeHui 가 PS 하는 블로그, "키타마사법(Kitamasa Method, きたまさ法)"
https://justicehui.github.io/hard-algorithm/2021/03/13/kitamasa/