Number Theoretic Algorithms

n제곱근, 최대공약수와 진법 변환

April 3, 2019

1 더 강력한 도구

우리가 자주 쓰는 연산들은 보통 가감승제에 한정적이지 않습니다. n제곱근 알고리즘은, 아시다시피 별로 중요하지 않지만 가끔씩 나와서 머릿속을 복잡하게 합니다. n제곱근과 더불어 다양한 함수의 근을 빠르게 찾을 수 있는 Newton's method를 깊이 살펴 보고, 최대공약수 및 진법 변환도 살펴봅니다.

수 M의 자리수를 m으로 쓰겠습니다. $\sqrt[n]{M}$을 이진 탐색(binary search)을 통해서 찾으면 $\mathcal{O}(m \log m \log M) = \mathcal{O}(m^2 \log m)$ 시간이 걸립니다. 최대공약수 알고리즘도 마찬가지입니다: 첫 시간에 자릿수의 선형에 비례하는 횟수의 나눗셈이 이루어진다는 사실을 보았지만, 나눗셈이 $\mathcal{O}(m \log m)$이므로 최대공약수 알고리즘의 전체 시간복잡도는 $\mathcal{O}(m^2 \log m)$이 됩니다. 진법 변환은 $\mathcal{O}(m)$회의 나눗셈을 시행하므로 $\mathcal{O}(m^2 \log m)$입니다.

진법 변환이 왜 필요할까요? 특정 알고리즘의 시간 복잡도는 진법에 의존합니다. 알고리즘이 매우 빠르게 돌아가는 특성이 "진법" 때문인 것들이 있기 때문입니다.[1] 이 경우 어떤 진법으로 쓰인 것을 원하는 진법으로 바꾸고, 다시 그 진법에서 원래 진법으로 돌아오는 등의 연산이 빠르게 시행되어야 할 필요성이 있습니다.

물론 여러분들의 알고리즘 지식과 subquadratic 가감승제만을 이용하여 이 알고리즘들을 구현할 수는 있지만, 우리의 목표는 여전히 subquadratic입니다. 따라서 더 나은 방법을 생각할 필요가 있습니다.

2 Newton's method

나눗셈을 위해서 $f(x) = m - \frac{B^u}{x}$에 대해 Newton's method를 이용했다는 사실을 기억할 것입니다. 나눗셈이 자릿수를 두 배씩 늘려 나가는 성질은 Newton's method의 일반적인 성질입니다. 이것을 보여 보고자 합니다. 증명을 위해서는 Taylor가 필요합니다. $f(x)$의 근을 α라 하면,

$$0 = f(\alpha) = f(x_n) + f'(x_n)(\alpha - x_n) + \frac{1}{2} f''(\xi_n)(\alpha - x_n)^2$$

[1] 이진법의 득을 보는 알고리즘의 좋은 예로 binary gcd algorithm 등이 있습니다. binary search도 이진법이라면 값을 쓰는 시간이 줄어 대단히 빠르게 시행할 수 있습니다.

을 만족하는 ξ_n이 α와 x_n 사이에 존재합니다. $f'(x_n)$이 0이 되지 않는다고 하고, $\varepsilon_n = \alpha - x_n$으로 놓으면,

$$\varepsilon_{n+1} = \alpha - x_{n+1} = \frac{f(x_n)}{f'(x_n)} + \alpha - x_n = -\frac{1}{2}\frac{f''(\xi_n)}{f'(x_n)}\varepsilon_n^2$$

으로부터 우리가 원하는 결과를 얻기는 하는데, 조금 다른 결과를 얻습니다. 우리는 실수를 다루는 게 아니라 정수를 다루므로, ε_n^2는 항상 커집니다. 따라서 $\frac{1}{2}\left|\frac{f''(\xi_n)}{f'(x_n)}\right|$이 충분히 작음을 가정할 수 있어야 Newton's method를 사용할 수 있습니다. 다행히도 나눗셈의 경우에는, 증명했듯이,

$$\frac{1}{2}\left|\frac{f''(\xi_n)}{f'(x_n)}\right| = \left|\frac{\frac{B^u}{\xi_n^2}}{\frac{B^u}{x_n}}\right| = \left|\frac{x_n}{\xi_n^2}\right| \le \left|\frac{1}{x_n}\right|$$

이고, 초기값 x_0를 $\alpha/2$보다 크게 주어서 잘 수렴하게 만들었습니다.

이제, n제곱근을 위해서 비슷한 함수를 이용합니다. $f_n(x) = 1 - \frac{m}{x^n}$를 이용하면,

$$\begin{aligned} x_{k+1} &= x_k - \left\lfloor \frac{1 - \frac{m}{x_k^n}}{\frac{nm}{x_k^{n+1}}} \right\rfloor \\ &= x_k - \left\lfloor \frac{(x_k^n - m)x_k}{nm} \right\rfloor \end{aligned}$$

이고, 수렴한다면 시간 복잡도는 $\mathcal{O}\left(nm\log nm + m\log\frac{m}{n}\log n\cdot\log m\right) = \mathcal{O}(m\log m(n + \log m\log n))$이며 n이 충분히 작으면 $\mathcal{O}(m\log^2 m)$입니다.

초기값 x_0를 얼마나 가깝게 잡아야 수렴하게 될까요? 이를 위하여 $f''(\xi_k)/2f'(x_k)$의 값을 계산하면,

$$\begin{aligned} \frac{f''(\xi_k)}{2f'(x_k)} &= \frac{-\frac{m}{n(n+1)\xi_k^{n+2}}}{2\frac{m}{nx_k^{n+1}}} \\ &= -\frac{x_k^{n+1}}{2(n+1)\xi_k^{n+2}} \le 0 \end{aligned}$$

은 $\{\varepsilon_k\}$의 부호가 바뀌지 않음을 의미하며,[2] 구하고자 하는 해 α 주위로 진동하지 않음을 의미합니다. 일반성을 잃지 않고 $x_0 < \alpha$를 가정합니다. 실제로 값이 줄어드는지는

$$\left|\frac{f''(\xi_k)}{2f'(x_k)}\right| = \frac{1}{2(n+1)\xi_k}\left|\frac{x_k}{\xi_k}\right|^{n+1} \le \frac{1}{x_k}$$

이고 $\varepsilon_{k+1} \le x_k\left|\frac{\varepsilon_k}{x_k}\right|^2$이므로 우항이 ε_k보다 작게 만들면 됩니다. 따라서 나눗셈과 비슷하게 $x_0 \ge \alpha/2$이라면 수렴하게 됩니다. 실제로는 이보다 훨씬 잘 어림할 수 있으므로 큰 문제는 되지 않습니다.

[2] ε_k에 대한 점화식에는 부등호가 없습니다. 이런 식의 아름다운 논증을 할 수 있게 도와주는 것은 실수에 내재된 아름다운 성질 때문입니다. 더욱 놀라운 것은 실수 이외의 집합에서는 위 논증을 하는 것이 **불가능하다**는 점입니다.

3 자릿수에 대한 분할 정복

분할 정복은 제곱 시간을 제곱미만 시간으로 만드는 테크닉으로 잘 알려져 있습니다. 분할 정복이 이토록 강력한 이유는 naïve한 계산에서 놓치던 중복 계산을 각 부분문제의 관점에서 보도록 **강제하여** 체계적으로 보도록 돕기 때문입니다.

큰 수의 계산을 subquadratic으로 만들고 싶을 때 구체적인 함수나 계산 방법이 없는 경우 시도해 볼 수 있는 방법이 분할 정복입니다. 최대공약수를 구하는 과정을 더 빠르게 만들기 위해 두 수의 최대공약수를 만들 때 놓치고 있는 정보가 무엇인지 생각해 봅시다. 쉬운 답변은 "중간 과정"인데, 더욱 중요한 것은 각 중간 과정이 어떤 선형 결합인지에 대한 정보입니다:

$a_n = r_{n-2}$	$b_n = r_{n-1}$	$r_n = p_n\mathbf{a} + q_n\mathbf{b}$	(p_n, q_n)
1581	1394	187	(1, -1)
1394	187	85	(-7, 8)
187	85	17	(15, -17)
85	17	0	(-82, 93)

이 선형 결합의 계수는 크게 세 가지 의미가 있습니다.

- 어떤 과정을 거쳐 a와 b의 최대공약수를 구했는지를 알려줍니다. 이 과정은 최대공약수 자체를 구하는 데에는 크게 중요하지 않으나, 예를 들어 $ax + by = k$ 꼴의 방정식을 풀거나 할 때 유용하게 사용될 수 있는 정보입니다.
- 최대공약수를 구할 때 쓴 나눗셈 등 모든 과정을 건너뛸 수 있습니다. (Lehmer)
- a와 b가 포함된 수의 최대공약수에서 이 부분의 digit을 "충분히 없앤다"는 보장이 되어 있는 계수입니다.[3]

우리는 세 번째 성질에 주목하는데, 이를테면 $A = 15810964$, $B = 13949049$에 대해서,

$$U \cdot \begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} 30627 \\ 762509 \end{pmatrix} \quad \text{where} \quad U = \begin{pmatrix} 15 & -17 \\ -82 & 93 \end{pmatrix}$$

입니다. 사실 $\det U = \pm 1$인 어떤 행렬에 대해서도 최대공약수는 같을 것이나, 특히 최대공약수의 계산 결과 계수로 나온 행렬 U는 더 큰 수에 대해서 그 수의 윗부분을 확실히 잘라냄을 보여 주고 있습니다.

우리의 목표는 $\gcd(a, b)$를 구하기 위해 $\gcd(a, b) = \gcd(a', b')$인 크기가 반인 두 수 a', b'과 변환 행렬 U를 주는 함수를 구성하는 것입니다. 이 과정은 **half GCD**라고 부릅니다.

이때 U의 entry의 길이와 a', b'의 길이가 대강 비슷하게 될 것입니다. 따라서 a, b의 길이를 n이라 하면, 먼저 a, b의 위 $n/2$자리를 잘라내 half GCD를 시행하여 (a'_0, b'_0, U)를 얻고, a, b에 U를 apply하면 남은 수의 길이가 $3n/4$쯤 됩니다.[4] 여기서 다시 위 $n/2$를 잘라내 half

[3] 자릿수는 선형적이므로, 각 부분이 (roughly) 독립적으로 작용한다고 생각할 수 있습니다.

[4] 위 예제에서, $-82a + 93b = 0$이어서 leading digit의 반을 없앨 것으로 기대했지만, 실제 apply했을 때는 trailing digit이 차이를 만들어내 수의 크기가 효과적으로 줄지는 않았습니다. 여기서는 trailing digit의 길이가 $n/2$, U의 entry의 길이가 $n/4$이므로 계산 결과가 $3n/4$보다 작음이 보장됩니다.

GCD를 시행하여 (a_1', b_1', U')을 얻고, 비슷한 방식으로 남은 수의 길이가 $n/2$이 되도록 할 수 있습니다. 우리가 돌려주어야 할 행렬 $U'U$의 entry 길이는 역시 $n/2$쯤 되는데, 이것은 U와 U'의 entry 길이가 대강 $n/4$쯤 되기 때문입니다.

이때 $\mathrm{hgcd}(n)$을 길이가 n인 수에 대해 half GCD를 시행하는 데 걸리는 시간이라고 하면, $\mathrm{hgcd}(n) = 2\,\mathrm{hgcd}(n/2) + \mathcal{O}(n \log n)$이므로 half GCD의 소모 시간은 $\mathcal{O}(n \log^2 n)$입니다. half GCD를 $\log n$번 반복하면 GCD를 구할 수 있고, 그 크기가 매 시행마다 반이 되므로 최대공약수를 $\mathcal{O}(n \log^2 n)$에 구할 수 있습니다!

진법 변환을 조금 생각해 봅시다. B진법을 C진법으로 바꾸고자 한다면, 그 변환 함수를 f라 할 때, $f(a \cdot B^n + b) = f(a) \cdot f(B^n) + f(b)$를 계산하면 됩니다. 크기 반인 세 부분문제로 쪼갰고, 곱셈 과정이 $\mathcal{O}(n \log n)$이라 진법 변환이 $\mathcal{O}(n^{\log_2 3} \log n)$이 됩니다.

만약 수를 $a \cdot C^n + b$ 꼴로 나타낼 수 있으면 $f(a \cdot C^n + b) = f(a) \cdot f(C^n) + f(b)$이고, $f(C^n)$은 생각할 필요가 없으므로 부분문제 두 개로 쪼갤 수 있습니다! 즉 나눗셈을 시행하면 a와 b를 구할 수 있습니다. 그러나 이 과정은 $f^{-1}(C^n)$, 즉 B진법으로 나타낸 C^n을 계산해야 합니다. Repeated Squaring 방법을 이용하면 두 방법 모두 본질적인 문제를 해결해, $\mathcal{O}(n \log^2 n)$ 시간에 진법 변환을 시행할 수 있게 해 줍니다.

이렇듯 자릿수에 대한 분할 정복 방법은 시간 복잡도를 줄여야 할 때 가장 먼저 시도해 볼 만한 가치가 있는 테크닉입니다.

4 문제

1. Lambert W 함수는 $w := W(z)$로 쓰며, $z = we^w$를 만족하는 w를 얘기합니다. z가 충분히 크면 값을 하나만 가지게 됩니다. 충분히 큰 z가 주어졌을 때 w를 구하는 방법을 생각해 봅시다.

   (a) $f(x) = xe^x - z$로 두고 Newton's method를 적용하여 점화식을 얻어내세요.

   (b) f''/f'의 부호와 절댓값을 이용하여 수렴 양상을 추정하세요. 즉, 구하고자 하는 값 w 주위로 진동하는지 혹은 단조수렴하는지, $\{\epsilon_k\}$가 줄어듦이 보장되는지 확인하세요.

   (c) W'을 W에 대해 나타내세요. z가 주어졌을 때 $W(z)$를 계산하지 않고 $W'(z)$의 값을 계산할 수 있습니까? $W'(z) = 1/z$로 근사하면,

   $$W(z) = \int_{ne^n}^{z} W'(z)dz + W(ne^n) \approx \log \frac{z}{ne^n} + n = \log \frac{z}{n} \approx \log z - \log \log z$$

   으로 근사할 수 있습니다. 이 근사가 $W(z)$를 Newton's method로 계산하기에 충분함을 보이세요.

   (d) $n = 224$를 사용하여 $W(10^{100})$을 구하세요. iteration을 100번 이상 돌려 보고, 10번 돌린 값과의 차이를 계산하세요.[5]

[5] 이 차이는 제 계산의 경우엔 2.205×10^{-2790}이었습니다. 시간이 많지 않으면 계산하지 않는 것을 추천합니다.

2. ($\mathcal{O}(N \log N)$ Newton-Raphson Division) 지난 시간에 나눗셈을 얘기할 때 $\mathcal{O}(N \log^2 N)$이라고 얘기했습니다. 이것은 사실이지만, 조금만 잘 생각하면 시간 복잡도에서 로그를 뺄 수 있습니다. 식을 까먹으셨을까봐 들고 왔습니다:

$$x_{k+1} = 2x_k - \left\lfloor \frac{mx_k^2}{B^u} \right\rfloor.$$

   (a) 어떤 정수 n_k와 e_k에 대해 $x_k = n_k B^{u-e_k}$라면, x_{k+1}을 계산하는 과정은 $\mathcal{O}(m \log m)$입니다. 이제 m 대신

$$m_k = \left\lfloor \frac{m}{B^{u-e_k}} \right\rfloor \cdot B^{u-e_k}$$

   을 사용해도, 값을 제대로 산출함을 보이세요. (Hint: $e_k \geq u$일 때까지 "존버.")

   (b) $e_k \geq u$인 최소의 k에 도달했을 때 $\varepsilon_k = \mathcal{O}(\sqrt{m^\star})$임을 보이세요. 이로써, k번 이후로 수렴값을 얻기까지의 iteration이 상수 번입니다.

   (c) i에 대해 바뀐 점화식을 계산하는 데 $\mathcal{O}(e_i \log N)$임을 보이고, $e_{i+1} = 2e_i$임을 보여 전체 과정이 $\mathcal{O}(N \log N)$임을 보이세요.

3. (Karatsuba) $n = n_1 \cdot B^u + n_0$, $m = m_1 \cdot B^u + m_0$이라고 합시다.

   (a) $A_0 = n_0 m_0$, $A_1 = n_1 m_1$, $A_2 = (n_0 + n_1)(m_0 + m_1)$이라 하면,

$$nm = A_1 B^{2u} + (A_2 - A_0 - A_1)\, B^u + A_0$$

   임을 보이세요.

   (b) 이 방법으로 곱셈을 크기 반인 세 개의 부분문제로 쪼개면, 시간복잡도가 $\mathcal{O}(n^{\log_2 3})$임을 보이세요. $\log_2 3 \approx 1.58$이므로, 이 알고리즘도 subquadratic입니다.[6]

   (c) 곱셈 계산 외에도 convolution으로 확장해서 이 idea를 사용할 수 있음을 보이세요. 즉, 알고리즘 전체에서 받아올림/받아내림을 제거할 수 있음을 보이세요.[7]

4. (Major Flaw on Naïve Half GCD and Robust HGCD)

   (a) $n_0 = 35748972$, $m_0 = 23832648$에 대해 HGCD를 시행하기 위해, 앞 4자리에 대해서 quadratic gcd를 수행했습니다:

a_n	b_n	$r_n = p_n a_0 + q_n b_0$	(p_n, q_n)
3574	2383	1191	(1, -1)
2383	1191	1	(-2, 3)
1191	1	0	(2383, -3574)

[6] Master theorem을 사용하지 마세요! 2^k에 대해 증명한 다음, 2^k 꼴로 크기를 늘려 이 과정을 수행할 수 있음을 보이세요. 증명 과정에 gap이 있을 것입니다! 찾아서 메우세요.

[7] 사실 이 방법은, 대단한 것이 아니라, $n(x) = n_1 x + n_0$, $m(x) = m_1 x + m_0$라 놓으면, 전체 계수의 합을 빠르게 계산하기 위해 $(nm)(1) = n(1)m(1)$을 이용한 것입니다. 따라서 우리가 배운 FFT를 이용한 곱셈의 하위호환입니다. FFT의 경우, $(nm)(1)$뿐 아니라 값을 여러 개 이용했기 때문에 더 빨라진 것입니다.

$U = \begin{pmatrix} 1 & -1 \\ -2 & 3 \end{pmatrix}$나 $U = \begin{pmatrix} -2 & 3 \\ 2383 & -3574 \end{pmatrix}$ 모두 두 수의 자릿수를 6 이하로 줄이지 않음을 보이세요. 따라서 HGCD의 대전제가 흔들립니다.

(b) (a)에서의 문제는 HGCD가 실제로 "자릿수를 반으로 줄이는" 결과가 나오지 않을 수도 있다는 점입니다. 상식적으로, g가 자릿수의 반보다 크다면, 자릿수의 반이 되는 두 수의 최대공약수가 g가 될 수는 없습니다. 그러나 위의 예제처럼 두 수 중 한 개의 수는 자릿수를 반 이하로 줄일 수 있습니다.[8] 이것을 엄밀하게 보이기 위해, 먼저 $|p_n|$과 $|q_n|$이 모두 $\frac{a}{r_{n-1}}$보다 작음을 보이세요. (Hint: $\det U = \pm 1$을 이용하여 $\begin{pmatrix} p_n & r_n \\ p_{n+1} & r_{n+1} \end{pmatrix}$의 determinant가 a 이하임을 보이세요.)

(c) (b)의 결과를 바탕으로 두 수 중 큰 쪽의 원래 자릿수를 k라 하면, HGCD 과정이 0을 반환하지 않는 경우에는 자릿수를 $(\lceil k/2 \rceil + 4)$ 이하로 줄임을 보이세요. 따라서 과정을 $\log_2 k$번 시행하면 길이가 $c \log_2 k + \mathcal{O}(1)$ 이하로 떨어지고, 여기서 quadratic gcd를 사용하면 원래의 시간복잡도가 보장됩니다.

5. (Lehmer's GCD Algorithm)

(a) 임의의 자연수 N, M에 대해 뒤 k자리를 제외하고 (p_n, q_n)을 구할 경우, $p_n N + q_n M$이 $(r_n + p_n)B^k$과 $(r_n + q_n)B^k$ 사이에 있음을 보이세요.

(b) 따라서 만일

$$\left\lfloor \frac{a_n + p_n}{b_n + p_{n+1}} \right\rfloor = \left\lfloor \frac{a_n + q_n}{b_n + q_{n+1}} \right\rfloor$$

이면 앞 자리만으로 p_{n+2}와 q_{n+2}를 유일하게 결정할 수 있음을 보이세요. 따라서 어떤 step의 긴 자리 나눗셈을 없앴습니다.

(c) 이 알고리즘이, B가 충분히 크다면, 긴 자리 나눗셈 횟수의 반 이상을 없앰을 보이세요. 시간 복잡도는 여전히 $\mathcal{O}(n^2 \log n)$이지만, 긴 자리 나눗셈 횟수가 significantly 줄기 때문에 HGCD의 base case gcd algorithm으로 많이 채택됩니다.

6. (Repeated Squaring)

(a) 진법 변환에서, $u = 2^k$으로만 두어도 $\mathcal{O}(n \log^2 n)$ 시간 복잡도를 보장함을 보이세요. (Hint: 3. (b).[9])

(b) $f\left(B^{2^k}\right)$를 필요한 만큼 구하는 데 $\mathcal{O}(n \log n)$임을 보이세요. 따라서 진법 변환을 시행하는 데는 $\mathcal{O}(n \log^2 n)$ 시간이 걸립니다. 만족스럽군요.

(c) $f^{-1}\left(C^{2^k}\right)$을 계산하는 데에도 (a), (b)를 그대로 적용할 수 있음을 보이세요.[10]

[8] 만일 g가 자릿수의 반보다 크다면 그 둘 중 하나의 수가 0이 될 것이고, 아니면 나눗셈을 한 번 시행해 두 수 모두 자릿수를 반 이하로 떨어뜨리면 됩니다.

[9] 이 방법은 온갖 복잡도 증명에서 쓰입니다. Master theorem이라는 도구를 사용하는 것은 알고리즘 시간에 많이 하니, 이 방법에 익숙해지도록 합시다!

[10] 따라서 어느 쪽이든 시간 복잡도는 $\mathcal{O}(n \log^2 n)$입니다. 그러나 특히 계산이 빠른 진법, 예를 들어 2진법이 아니면 나눗셈이 들어가면 느려지기 때문에 보통 전자, 즉 B진법에서 반으로 나눠서 합치는 방법을 씁니다.