

EUnS's book

EUnS

2020년 2월 2일

차 례

| | |
|--|----|
| 차 례 | 1 |
| 제 1 장 number theory | 4 |
| 1.1 기초 정수론 | 5 |
| 1.2 유클리드 호제법(Euclidean algorithm) | 6 |
| 1.3 확장된 유클리드 알고리즘(Extended Euclidean algorithm) | 8 |
| 1.3.1 베주의 항등식 | 8 |
| 1.3.2 활용 | 8 |
| 1.4 나머지 연산에서 곱셈에 대한 역원 (modular multiplicative inverse) ¹ | 9 |
| 1.5 오일러의 ϕ 함수(Euler's phi (totient) function) | 10 |
| 1.6 오일러 정리(Euler's theorem) ² | 11 |
| 1.7 소수 판별법 | 13 |
| 1.7.1 의사소수 판정(pseudoprime test) | 13 |
| 1.7.2 밀러라빈소수 판별법(Miller-Rabin primality test) | 14 |

¹역원: a 와 연산자에 대해 연산결과가 항등원($= 1$)이 되는 유일한 원소 b 를 a 의 역원이라한다.

²페르마의 소정리는 오일러 정리에서의 특수한 경우이다.

| | |
|---|-----------|
| 1.8 pollard's rho algorithms | 15 |
| 1.9 중국인의 나머지 정리(Chinese Remainder Theorem) | 16 |
| 1.10 Carmichael function | 17 |
| 1.10.1 Solve 31.8-2(CLRS) | 20 |
| 1.11 RSA시스템의 이해 | 23 |
| 1.11.1 RSA 공개 키 암호 시스템 ³ (RSA public-key cryptosystem) | 23 |
| 1.11.2 공개키, 암호키 생성 | 23 |
| 1.11.3 단계 | 23 |
| 1.11.4 복호화 과정 | 23 |
| 1.11.5 이게 과연 안전한가? | 24 |
| 1.11.6 advanced RSA | 24 |
| 제 2 장 Introduction to Quick Sort | 25 |
| 2.1 소개 | 26 |
| 2.2 의사코드 및 동작 | 26 |
| 2.3 여러 기초 지식 | 29 |
| 2.3.1 시간복잡도 | 29 |
| 2.3.2 조화 급수의 상한과 하한 | 29 |
| 2.3.3 확률 | 30 |
| 2.4 대략적인 복잡도 분석 | 31 |
| 2.4.1 최악의 분할 케이스 | 31 |
| 2.4.2 최선의 분할 케이스 | 31 |
| 2.4.3 일반적인 케이스 직관적인 방법 | 32 |
| 2.5 상세 분석 | 34 |
| 2.5.1 최악의 케이스 분석 | 34 |
| 2.5.2 기대 수행 시간 | 35 |
| 2.5.3 Hoare's Partition VS Lomuto's Partition | 37 |
| 2.6 Quick sort의 캐시 히트율 | 40 |
| 2.7 개선 | 42 |
| 2.7.1 재귀함수 제거 | 42 |
| 2.7.2 hybrid sort | 42 |
| 2.7.3 중복값 처리 | 43 |

³로널드 라이베스트(Ron Rivest), 애디 샤미르(Adi Shamir), 레너드 애들먼(Leonard Adleman)이 세명의 이름 앞글자를 따서 지었다.

| | |
|--|-----------|
| 2.7.4 median of three | 46 |
| 2.7.5 Parallelization | 46 |
| 2.8 성능 테스트 | 46 |
| 2.8.1 HOARE VS LOMUTO | 48 |
| 2.8.2 개선 성능 테스트 | 48 |
| 2.8.3 std::sort 성능테스트 VS J. Bentley D. McIlroy | 51 |
| 제 3 장 string matching | 53 |
| 3.1 The naive string-matching algorithm | 54 |
| 3.2 The Rabin-Karp algorithm | 54 |
| 3.3 String matching with finite automata | 56 |
| 3.4 The Knuth-Morris-Pratt algorithm | 59 |
| 제 4 장 FFT | 62 |
| 4.1 개론 | 63 |
| 4.2 점값 표현 | 63 |
| 4.3 계수표현 점값표현 치환에 대한 고찰 | 64 |
| 4.4 DFT와 FFT | 64 |
| 4.4.1 복소수 | 65 |
| 4.4.2 DFT | 65 |
| 4.4.3 FFT | 66 |
| 4.5 DFT^{-1} 역변환 | 67 |
| 4.6 $n = 4$ 일때 수행 예시 | 68 |
| 4.6.1 DFT 수행 | 68 |
| 4.6.2 DFT^{-1} 수행 | 68 |
| 4.7 성능 개선 | 69 |
| 참고 자료 | 72 |
| 참고 문헌 | 74 |

제 1 장

number theory

1.1 기초 정수론

Theorem 1.1.1. d, m, n 이 어떤 정수일 때, 다음이 성립한다.

1. $d \nmid m$ 과 n 의 공약수일 때, $m + n$ 도 d 의 배수이다.
2. $d \nmid m$ 과 n 의 공약수일 때, $m - n$ 도 d 의 배수이다.

Proof. 이에 대한 증명은 간단하다 $m = dq_1, n = dq_2 (q_1, q_2 \in \mathbb{Z})$ 라 하자. $m + n = d(q_1 + q_2), m - n = d(q_1 - q_2)$

□

이 장을 이해하기 위해서 약속 몇 가지를 정의한다.

- $d \nmid n$ 의 약수(인수)일 때 $d \mid n$ 으로 표시한다.
- m 과 n 의 최대공약수는 $\gcd(m, n)$ 이라고 한다.
- $r \circ | a$ 를 b 로 나눈 나머지라면 $r = a \bmod b$ 이다.

이를 써서 위 명제를 다시 적으면 $d \mid n, d \mid m \rightarrow d \mid (m + n), d \mid (m - n)$

Theorem 1.1.2. a, b, z 를 양의 정수라 하면, 다음이 성립한다.

$$ab \bmod z = [(a \bmod z)(b \bmod z)] \bmod z$$

Proof. $w = ab \bmod z$ 라 하자. 다음이 성립하는 q_1 이 존재한다.

$$ab = q_1z + w \iff w = ab - q_1z$$

마찬가지로 $x = a \bmod z, y = b \bmod z$ 라 하면, 다음을 만족시키는 q_2 와 q_3, q 가 존재한다.

$$a = q_2z + x, b = q_3z + y$$

$$\begin{aligned} w &= ab - q_1z = (q_2z + x)(q_3z + y) - q_1z \\ &= (q_2q_3z + q_2y + q_3x - q_1)z + xy \\ &= qz + xy \end{aligned}$$

여기서 $q = q_2q_3z + q_2y + q_3x - q_1$ 이므로

$$xy = -qz + w$$

즉 w 는 xy 를 z 로 나눌 때의 나머지이다. 그러므로 $w = xy \bmod z$ 가 되고 이는 다음과 같이 나타낼 수 있다.

$$ab \bmod z = [(a \bmod z)(b \bmod z)] \bmod z$$

□

이는 큰수를 인수분해해서 작은값으로 나눠서 큰수를 다루는 부담을 덜어주지만 지수승에 대해서도 응용이 가능하다. 이를 이용해서 $a^{29} \bmod z$ 를 계산하는 절차를 예시로 들어보겠다. a^{29} 는 다음과 같은 순서로 계산한다.

$$a, a^5 = a \cdot a^4, a^{13} = a^5 \cdot a^8, a^{29} = a^{13} \cdot a^{16}$$

$a^{29} \bmod z$ 는 다음과 같은 순서로 계산한다.

$$a \bmod z, a^5 \bmod z, a^{13} \bmod z, a^{29} \bmod z$$

$$\begin{aligned} a^2 \bmod z &= [(a \bmod z)(a \bmod z)] \bmod z \\ a^4 \bmod z &= [(a^2 \bmod z)(a^2 \bmod z)] \bmod z \\ a^8 \bmod z &= [(a^4 \bmod z)(a^4 \bmod z)] \bmod z \\ a^{16} \bmod z &= [(a^8 \bmod z)(a^8 \bmod z)] \bmod z \\ a^5 \bmod z &= [(a \bmod z)(a^4 \bmod z)] \bmod z \\ a^{13} \bmod z &= [(a^5 \bmod z)(a^8 \bmod z)] \bmod z \\ a^{29} \bmod z &= [(a^{13} \bmod z)(a^{16} \bmod z)] \bmod z \end{aligned}$$

1.2 유클리드 호제법(Euclidean algorithm)

Theorem 1.2.1. a 가 음이 아닌 정수이고, b 가 양의 정수이며 $r = a \bmod b$ 이면 다음이 성립한다.

$$\gcd(a, b) = \gcd(b, r)$$

수식이 익숙하지 않은 분을 위해 풀어서 설명하자면, a 가 음이 아닌 정수이고, b 가 양의 정수이며, $r \mid a$ 를 b 로 나눈 나머지라면 a 와 b 의 최대공약수는 b 와 r 의 최대공약수와 같다.

Proof. $a = bq + r$ ($0 \leq r < b, q$ 는 어떤 정수)인데, c 를 a 와 b 의 공약수라 하면, c 는 bq 의 약수인 것은 자명하다. a 또한 c 의 약수이므로 c 는 $a - bq (= r)$ 의 약수이다. 따라서 c 는 b 와 r 의 공약수입니다. 반대로 c' 가 b 와 r 의 공약수이면, c' 는 $bq + r (= a)$ 의 약수가 되고 따라서 a 와 b 의 공약수가 된다. 따라서 a 와 b 의 공약수 집합이 b 와 r 의 공약수 집합과 같으므로 $\gcd(a, b) = \gcd(b, r)$ 이 성립한다. \square

유클리드 알고리즘의 의의는 나머지 연산만을 이용해서 빽빽이 돌리면 어떻게 됐든지 간에 최대공약수를 기계적으로 구할수있다는 것에 있다. $\gcd(a, b) = \gcd(b, r)$ 에서 b, r 을 새로운 a, b 로서 값을 넣어서 연속적으로 계산을 하면 언젠가 b 가 $0 \mid$ 되는 순간이 오는데, 이때 a 가 처음 a, b 의 최대 공약수가 되는것이다.

Theorem 1.2.2. $\alpha > \beta$ 일때, 다음이 성립한다.

$$\gcd(\alpha, \beta) = \gcd(\alpha - \beta, \beta)$$

Proof. α, β 의 최대 공약수를 x 라 하자. $\alpha = x \cdot a, \beta = x \cdot b$ (a, b 는 $a > b$ 이며 서로소인 두 정수)이며, $\alpha - \beta = x(a - b)$ 이다 $a - b$ 는 b 와 서로소이며 두 값의 최대공약수는 여전히 x 이다. \square

Corollary 1.2.2.1. $f(n) = 1 + 10 + \cdots + 10^n$ 이라 하자.

$$\gcd(f(x), f(y)) = f(\gcd(x, y))$$
임을 보여라.

Proof. $x > y$ 라 하자.

$$\begin{aligned} f(x) - f(y) &= 10^x + 10^{x-1} + \cdots + 10^{y+1} \\ &= 10^y(10^{x-y} + \cdots + 1) \\ &= f(x - y) \cdot 10^y \\ \gcd(f(x), f(y)) &= \gcd(f(x) - f(y), f(y)) \\ &= \gcd(f(x - y) \cdot 10^y, f(y)) \end{aligned}$$

이때 10^y 와 $f(y)$ 는 항상 서로소이므로 $\gcd(f(x - y), f(y))$ 가 성립한다.

따라서 유클리드 호제법을 전개했을때, $\gcd(f(x), f(y)) = \gcd(f(\gcd(x, y)), 0)$ 이 되고 이는 $f(\gcd(x, y))$ 과 같다. \square

1.3 확장된 유클리드 알고리즘(Extended Euclidean algorithm)

확장된 유클리드 알고리즘은 다음의 방정식에 대해서 s 와 t 를 효율적으로 구하는 방법에 대한 것이다.

a 와 b 가 음이 아니고 동시에 0이 아닌 정수라 하면 다음을 만족시키는 정수 s 와 t 가 존재한다.

$$\gcd(a, b) = s \cdot a + t \cdot b^a$$

^a선형 디오판토스 방정식이라고도 한다.

1.3.1 베주의 항등식

Theorem 1.3.1. $ax + by = \gcd(x, y)$ 인 a, b 가 존재한다.

Proof. 집합 $S = \{m | m = ax + by, x \in \mathbf{Z}, y \in \mathbb{Z}\}$ 를 생각해보면, 이 집합 S 는 $S \subset \mathbf{Z}$, $S \subset \emptyset$ (x, y 를 원소로 가짐을 알 수 있다.)이다. 또한, 자연수의 정렬성으로부터 최소가 되는 원소 d 가 존재한다.

$\alpha \in S \Rightarrow \alpha = qd + r (0 \leq r < d)$ 라 하자.

만약 $d \nmid \alpha$ 일때, $r > 0$, $r = \alpha - qd$, ($\alpha, d \in S$) $\alpha = a_1x + b_1y$, $d = a_2x + b_2y$ 라 하면. $r = (a_1 - a_2q)x + (b_1 - qb_2)y \in S$ $0 < r < d$ 인 r 에 대해 d 가 최소라는 가정이 모순이다.

$\therefore r = 0$, $d \mid \alpha (\forall \alpha \in S)$, $d \mid x, d \mid y \dots d$ 는 x, y 의 공약수, $\gcd(x, y) = k$ 라 할때, $d = akx'' + bky'' = k(ax'' + by'')$ $k \mid d$ 에서 $k = d$ \square

1.3.2 활용

이미 증명되어있는 유클리드 알고리즘의 흐름을 통해서 예시로 이해 해보자.

$a = 273$, $b = 110$ 으로 하는 $\gcd(273, 110)$ 을 구해보자.

$$r = 273 \bmod 110 = 53 \dots 1$$

$a = 110, b = 53$ 으로 지정

$$r = 110 \bmod 53 = 4 \dots 2$$

$a = 53, b = 4$ 로 지정

$$r = 53 \bmod 4 = 1 \cdots 3$$

$a = 4, b = 1$ 로 지정

$$r = 4 \bmod 1 = 0 \cdots 4$$

$r = 0$ 이므로 $\gcd(273, 110)$ 은 최대공약수로 1을 가진다. 여기서 4 식으로 되돌아가면 이는 다음과 같이 쓸 수 있다.

$$1 = 53 - 4 \cdot 13$$

계속 역순으로 뒤집어 올라가자 3

$$4 = 110 - 53 \cdot 2$$

이를 처음의 식에 대입하면

$$1 = 53 - (110 - 53 \cdot 2) \cdot 13 = 27 \cdot 53 - 13 \cdot 110$$

2

$$53 = 273 - 110 \cdot 2$$

이 식을 다시 대입하면

$$1 = 27 \cdot 53 - 13 \cdot 110 = 27 \cdot (273 - 110 \cdot 2) - 13 \cdot 110 = 27 \cdot 273 - 67 \cdot 110$$

따라서 $s = 27, t = -67$ 로서 성립하는 값을 찾았다.

1.4 나머지 연산에서 곱셈에 대한 역원 (modular multiplicative inverse)¹

정의 1.4.1 (Inverse) $\gcd(n, \phi) = 1$ 인 두 정수 $n > 0, \phi > 1$ 가 있다고 하자.^a $n \cdot s \bmod \phi = 1$ 을 만족시키는 s 를 $n \bmod \phi$ 의 역원(inverse)이라고 한다.

^a한 마디로 n 과 ϕ 는 서로소이다.

$\gcd(n, \phi) = 1$ 임을 이용해, 확장된 유clidean 알고리즘을 이용하여 $s' \cdot n + t \cdot \phi = 1$ 이 되는 s' 과 t' 을 구할 수 있다. $n \cdot s' = -t' \phi + 1$ 이 되고 $\phi > 1$ 이므로 1이 나머지가 된다. $n \cdot s' \bmod \phi = 1$ 에서 $s = s' \bmod \phi$ 라 하면 $0 \leq s < \phi$ 가 되며 또한 $s \neq 0$

¹역원: a 와 연산자에 대해 연산결과가 항등원($= 1$)이 되는 유일한 원소 b 를 a 의 역원이라한다.

이다.

위 식을 변형하면, $s' = q \cdot \phi + s$ 가되며 이를 만족하는 정수 q 가 존재한다.
따라서

$$n \cdot s = ns' - \phi nq = -t'\phi + 1 - \phi nq = \phi(-t' - nq) + 1$$

따라서 $n \cdot s \bmod \phi = 1$ 이 된다.

1.5 오일러의 ϕ 함수(Euler's phi (totient) function)

정의 1.5.1 (phi function) 양의 정수 n 에 대해서

$\phi(n)$: 1부터 n 까지의 양의 정수 중에 n 과 서로소인 것의 개수를 나타내는
함수.

$\phi(n)$ 은 다음의 성질이 있다.

Theorem 1.5.1. • 소수 p 에 대해서 $\phi(p) = p - 1$

• m, n 이 서로소인 양의 정수일 때, 다음이 성립한다.

$$\phi(mn) = \phi(m)\phi(n)$$

• 소수 p 와 양의 정수 α 에 대해 다음이 성립한다.

$$\phi(p^\alpha) = p^\alpha \left(1 - \frac{1}{p}\right)$$

Proof. 첫번째 성질은 어찌보면 당연하다 p 는 소수이니 자기 자신을 제외한 모든 수와 서로소이다 (여기서 1도 세야한다.)

두번째 성질은 두수의 곱 mn 은 각각 m 에 대해서 나눠지는 수가 n 개이고 n 에 대해서 나눠지는 수가 m 개이며 mn 으로 나눠지는 수가 한 개이므로 $mn - \frac{mn}{m} - \frac{mn}{n} + \frac{mn}{mn} = mn - m - n + 1 = (m-1)(n-1) = \phi(m)\phi(n)$ 가 된다.
세번째 성질 p^α 보다 같거나 작은 p 의 배수가 되는 것은 다음이 있다.

$$p, 2p, 3p, \dots, p^{\alpha-1}p$$

따라서 총 $p^{\alpha-1}$ 개가 있고 $\phi(p^\alpha) = p^\alpha - p^{\alpha-1} = p^\alpha \left(1 - \frac{1}{p}\right)$

□

Corollary 1.5.1.1. If m_1, m_2, \dots, m_k are k positive integers which are prime each to each, then

$$\phi(m_1 m_2 \dots m_k) = \phi(m_1) \phi(m_2) \dots \phi(m_k).$$

If $m = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_n^{\alpha_n}$ where p_1, p_2, \dots, p_n are different primes and $\alpha_1, \alpha_2, \dots, \alpha_n$ are positive integers, then

$$\phi(m) = m \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \dots \left(1 - \frac{1}{p_n}\right).$$

For,

$$\begin{aligned} \phi(m) &= \phi(p_1^{\alpha_1}) \phi(p_2^{\alpha_2}) \dots \phi(p_n^{\alpha_n}) \\ &= p_1^{\alpha_1} \left(1 - \frac{1}{p_1}\right) p_2^{\alpha_2} \left(1 - \frac{1}{p_2}\right) \dots p_n^{\alpha_n} \left(1 - \frac{1}{p_n}\right) \\ &= m \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \dots \left(1 - \frac{1}{p_n}\right). \end{aligned}$$

1.6 오일러 정리(Euler's theorem)²

Theorem 1.6.1. 임의의 정수 a 와 n 이 서로소일 때, 다음이 성립한다.

$$a^{\phi(n)} \bmod n = 1$$

Proof. 정수 n 에 대해서 1부터 n 까지의 양의 정수 중에 n 과 서로소인 것의 집합을 생각해보자. 그러면 이는 집합

$$A = \{r_1, r_2, r_3, \dots, r_{\phi(n)}\}^3$$

으로 나타낼 수 있다. 이 집합은 A 라하고 이 각 원소에 n 과 서로소인 a 를 곱한 집합을 B 집합이라 하자.

$$B = \{ar_1, ar_2, ar_3, \dots, ar_{\phi(n)}\}$$

²페르마의 소정리는 오일러 정리에서의 특수한 경우이다.

³이러한 집합을 기약잉여계 \mathbb{Z}_n^* 라고 부른다. 또한 집합 A 의 원소의 갯수는 $\phi(n)$ 이다.

확실한건 B 에 있는 모든 원소는 n 과 서로소인 것이다. 그럼 B 집합의 각 원소를 $\text{mod } n$ 에 대해 계산한 것을 생각해보자. 이는 각 원소의 나머지가 a 를 곱하기전 값과 같은지는 모르지만 $\phi(n)$ 개에 대해서 각각 일대일대응이 가능 한다는것을 알수있다. ⁴ 따라서 A 의 모든 원소를 곱한 값에 $\text{mod } n$ 을 한것과 B 의 모든 원소를 곱한 값에 $\text{mod } n$ 을 한 값은 같다.

$$ar_1 \cdot ar_2 \cdot ar_3 \cdots ar_{\phi_n} \equiv r_1 \cdot r_2 \cdot r_3 \cdots r_{\phi_n} \pmod{n}$$

$$a^{\phi(n)} \pmod{n} = 1$$

□

Corollary 1.6.1.1. 페르마의 소정리 : 소수 p 에 대해 다음이 성립한다.

$$a^{p-1} \equiv 1 \pmod{p}$$

Proof. $\phi(p) = p - 1$ 으로 오일러 정리에 의해 성립함을 알 수 있다. □

$$a^{p-1} \equiv 1 \pmod{p}.$$

⁴실제 증명은 귀류법을 통해서 증명할수있다. $ar_i \equiv ar_j \pmod{n}$ 인 $1 \leq i < j \leq \phi(n)$ 의 존재한다고 가정해보자.

1.7 소수 판별법

대표적인 방법

1부터 \sqrt{n} 자연수까지 모두 나눠본후 나눠지는 수가 없을시에 그수는 소수이다. 2일때는 소수라 하고 짝수로 나누는 경우는 없애도 상관없다. $O(\sqrt{n})$

에라토스테네스의 체는 특정 n 이 소수인지 판단하는것과는 무관하므로 제외 한다..

이글의 연장선상인 얘기이다.

하나의 값 n 을 놓고 이값이 소수인지 아닌지 보려면 현재 $O(\sqrt{n})$ 까지 만큼 비교해 보아야 확정적으로 알수있다.

엄청난 크기의 소수를 구하기 위해서 $O(\sqrt{n})$ 만큼의 시간도 길다고 판단해 이보다 좀더 효율적임을 위해서 결국 정확도를 조금 포기하고 확률적으로 소수인지 제대로 판단할 확률이 높은 알고리즘들이 나왔다.

1.7.1 의사소수 판정(pseudoprime test)

n 이 소수일때 성립하는 페르마의 소정리를 판별방식으로 쓴다.

n 과 서로소인 a 에 대해서 a^{n-1} 을 n 으로 나눈 나머지는 무조건 1이 된다. 소수일때는 무조건 성립하니까 이를 판별 방식으로 쓰자는것

따라서 어떤 값 n 에 대해서 a^{n-1} 을 n 으로 나눈 나머지가 1인지 판별해보면 된다. 적당한 a 를 뽑고, a^{n-1} 을 고속 지수승 알고리즘을 통해 $\log n$ 번에 구할수 있다.

```
1 PSEUDOPRIME(n)
2     if MODULAR-EXPONENTIATION(2,n-1,n) != 1
3         return COMPOSITE
4     else return PRIME
```

이때 나오는 오진은 소수가 아닌데 a^{n-1} 을 n 으로 나눈 나머지가 1이 되는 경우이다. 이때 이 값을 **카마이클 수(Carmichael number)**라고 합니다 이 수의 특성도 재밌긴한데(사실 잘모름) 따로다루겠다.

카마이클수를 차례대로 나타내면

561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657, 52633, 62745, 63973

굉장히 띄엄띄엄 있어서 오진율이 낮다.

1.7.2 밀러라빈소수 판별법(Miller-Rabin primality test)

앞의 의사소수 판정을 조금 더 보완하기 위해서 검사를 더 촘촘히 하기로 해보자
다음 증명된 사실을 이용한다.

Theorem 1.7.1. 홀수인 소수 p 와 정수 $1 \leq e$ 에 대해서 $(\text{mod } p^e)$ 에서 $x^2 \equiv n$ 으로 나눈 나머지가 1이 되는 x 의 해는 무조건 $1, -1 (= n - 1)$ 이다.

Proof. $p^e \mid (x+1)(x-1)$ 이 될 때 p^e 는 $(x-1), (x+1)$ 중에 하나만이 될 수가 있다. p^e 가 $(x+1), (x-1)$ 둘다 나눌 수 있을 때에는 p^e 가 2로 나누어지기 때문이다. 만약 p^e 가 $x+1$ 을 나눌 수 있는 경우 $x \equiv -1 \pmod{p^e}$, p^e 가 $x-1$ 을 나눌 수 있는 경우 $x \equiv 1 \pmod{p^e}$ 가 된다. \square

이 정리의 대우에 따라서 1, -1을 근으로 가지지 않은 n 은 합성수라고 판단 할 수 있다.

따라서 이 나머지가 1이 되는데 x 가 $+1$ 인지를 살펴 보면된다.

임의의 a 에 대해서 판단하는 방법은 다음과 같다.

1. $n-1$ 를 2^{t*u} (d 는 홀수)로 나타낸다.
2. a^u 부터 $a^{2^{t*u}}$ 로 점점 제곱하면서 이 사이에 값이 1, $n-1$ 인데 그전의 값이 $+1$ 이 아닌지 판별을 한다.
3. 마지막에 $a^{2^{t*u}}$ 값이 1인지 비교한다.(페르마 소정리)

예를 들어 카마이클 수는 561을 $a = 2$ 로 해서 구해보면 마지막에 $a^{(2^t * d)}$ 이 1이 되지만 제곱하기 전의 값이 1이 아니라서 여기서 걸려지게 된다.

근데 이 판별이 결국 a 에 따라 갈리게 된다. a 값이 n 에 대해서 제곱했을 때 1이 되는 근이 아니어야 판별이 가능하다. 이는 합성수 n 에 대한 다음 판별 방식으로 탐지되는 근이 $1 \sim n-1$ 사이에 최소 $n/2$ 가 존재한다

확실한건 a 를 $2 \sim n/2$ 로 정하면 확실하게 나온다.

a 를 촘촘하게 여러번 선택해서 판별하면 되는데 그러면 또 시간이 오래 걸린다
따라서 탐지율이 a 를 뽑는 횟수에 따라 다르다.

알고리즘 복잡도는 a 를 반복해서 뽑는 k 에 따라서 $O(k \log^3 n)$ 이다. 추가적으로 곱셈을 FFT로 처리했을 때 $O(k \log^2 n)$ 까지 줄일 수 있다.

추가로 난제 중 하나인 리만가설이 맞다면 $2 \log^2 n$ 개의 a 로 검사를 했을 때 소수일 것이라고 판단이 되었을 경우 확정적으로 소수임이 드러나기 때문에 $O(\log^4 n)$ 의 시간복잡도를 가지는 소수판별법이 된다.

1.8 pollard's rho algorithms

의사난수를 발생시켜 구하는 방법이다....

$$g(x) = x^2 - 1 \pmod{n}$$

다음의 식을 사용해 반복적으로 $g(x)$ 를 만들어내고 $y = g(g(x))$ 로 $\gcd(y - x, n)$ 이 1, n 인지 검사하는 방식이다. 그뒤 $x = g(x)$ 로 업데이트하여 반복한다.

1.9 중국인의 나머지 정리(Chinese Remainder Theorem)

$x \equiv a_1 \pmod{m_1}$, $x \equiv a_2 \pmod{m_2}$, \dots ,
 $x \equiv a_n \pmod{m_n}$ ($\forall i, j \text{ gcd}(m_i, m_j) = 1$ ^a) 일 때, $x \not\equiv Z_{m_1 m_2 \dots m_n}$ 에서 유일하게 존재한다.

^a서로소 아이디얼 (pairwise coprime)

Proof. 1. 존재성

$m = m_1 m_2 \dots m_n$, $n_k = \frac{m}{n_k}$ 로 놓자. 그러면 $t_k m_k + s_k n_k = 1$ 인 정수 s_k, t_k 가 존재한다 ($\because \gcd(m_k, n_k) = 1$)⁵ $s_k n_k \equiv 1 \pmod{m_k}$ $x = a_1 n_1 s_1 + \dots + a_n n_n s_n = \sum_{k=1}^n a_k n_k s_k$ $j \neq k \rightarrow m_k \mid n_j \rightarrow x \equiv a_k n_k s_k \equiv a_k \pmod{m_k}$

2. 유일성

귀류법을 사용한다. 서로다른 $x, y \not\equiv Z_{m_1 m_2 \dots m_n}$ 에서 합동식의 해라 하자. $x \equiv y \equiv a_1 \pmod{m_1}$ $x \equiv y \equiv a_2 \pmod{m_2}$ $x \equiv y \equiv a_n \pmod{m_n}$ $x - y \equiv 0 \pmod{m_k}$ ($1 \leq k \leq n$ 인 정수 k) $\text{lcm}(m_1, m_2, \dots, m_n) \mid (x - y) \rightarrow m_1 m_2 \dots m_n (= m) \mid (x - y)$ ($\forall i, j \text{ gcd}(m_i, m_j) = 1$) $\therefore x \equiv y \pmod{\text{lcm}(m_1, m_2, \dots, m_n)}$ 이는 모순이다.

□

⁵비주 항등식

1.10 Carmichael function

<http://www.gutenberg.org/files/13693/13693-pdf.pdf>

Then let us write

$$a^{p-1} = 1 + hp. \quad (1)$$

Raising each member of this equation to the p^{th} power we may write the result in the form

$$a^{p(p-1)} = 1 + h_1 p^2. \quad (2)$$

where h_1 is an integer. Hence

$$a^{p(p-1)} \equiv 1 \pmod{p^2}.$$

By raising each member of (2) to the p^{th} power we can readily show that

$$a^{p^2(p-1)} \equiv 1 \pmod{p^3}.$$

It is now easy to see that we shall have in general

$$a^{p^{\alpha-1}(p-1)} \equiv 1 \pmod{p^\alpha}.$$

where α is a positive integer; that is,

$$a^{\phi(p^\alpha)} \equiv 1 \pmod{p^\alpha}.$$

For the special case when p is 2 this result can be extended. For this case (1) becomes

$$a = 1 + 2h.$$

Squaring we have

$$a^2 = 1 + 4h(h + 1).$$

Hence,

$$a^2 = 1 + 8h_1, \quad (3)$$

where h_1 is an integer. Therefore

$$a^2 \equiv 1 \pmod{2^3}.$$

Squaring (3) we have

$$a^{2^2} = 1 + 2^4h_2;$$

or

$$a^{2^2} \equiv 1 \pmod{2^4}.$$

It is now easy to see that we shall have in general

$$a^{2^{\alpha-2}} \equiv 1 \pmod{2^\alpha}$$

if $\alpha > 2$. That is,

$$a^{\frac{1}{2}\phi(2^\alpha)} \equiv 1 \pmod{2^\alpha} \text{ if } a > 2. \quad (1.1)$$

Now in terms of the ϕ -function let us define a new function $\lambda(m)$ as follows:

$$\begin{aligned} \lambda(2^\alpha) &= \phi(2^\alpha) \text{ if } a = 0, 1, 2; \\ \lambda(2^\alpha) &= \frac{1}{2}\phi(2^\alpha) \text{ if } a > 2; \\ \lambda(p^\alpha) &= \phi(p^\alpha) \text{ if } p \text{ is an odd prime;} \\ \lambda(2^\alpha p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_n^{\alpha_n}) &= \text{lcm}(\lambda(2^\alpha), \lambda(p_1^{\alpha_1}), \lambda(p_2^{\alpha_2}), \dots, \lambda(p_n^{\alpha_n})) \end{aligned}$$

$2, p_1, p_2, \dots, p_n$ being different primes.

Denote by m the number

$$m = 2^\alpha p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_n^{\alpha_n}.$$

Let a be any number prime to m . From our preceding results we have

$$\begin{aligned} a^{\lambda(2^\alpha)} &\equiv 1 \pmod{2^\alpha}, \\ a^{\lambda(p_1^{\alpha_1})} &\equiv 1 \pmod{p_1^{\alpha_1}}, \\ a^{\lambda(p_2^{\alpha_2})} &\equiv 1 \pmod{p_2^{\alpha_2}}, \\ &\dots \\ a^{\lambda(p_n^{\alpha_n})} &\equiv 1 \pmod{p_n^{\alpha_n}}. \end{aligned}$$

Now any one of these congruences remains true if both of its members are raised to the same positive integral power, whatever that power may be. Then let us raise both members of the first congruence to the power $\frac{\lambda(m)}{\lambda(2^\alpha)}$; both members of the second congruence to the power $\frac{\lambda(m)}{\lambda(p_1^{\alpha_1})}$; ...; both members of the last congruence to the power $\frac{\lambda(m)}{\lambda(p_n^{\alpha_n})}$. Then we have

$$\begin{aligned} a^{\lambda(m)} &\equiv 1 \pmod{2^\alpha}, \\ a^{\lambda(m)} &\equiv 1 \pmod{p_1^{\alpha_1}}, \\ &\dots \\ a^{\lambda(m)} &\equiv 1 \pmod{p_n^{\alpha_n}}. \end{aligned}$$

From these congruences we have immediately

$$a^{\lambda(m)} \equiv 1 \pmod{m}.$$

We may state this result in full in the following theorem:

If a and m are any two relatively prime positive integers, the congruence

$$a^{\lambda(m)} \equiv 1 \pmod{m}.$$

is satisfied.

As an excellent example to show the possible difference between the exponent $\lambda(m)$ in this theorem and the exponent $\phi(m)$ in Fermat's general theo-

rem, let us take

$$m = 2^6 \cdot 3^3 \cdot 5 \cdot 7 \cdot 13 \cdot 17 \cdot 19 \cdot 37 \cdot 73.$$

Here

$$\lambda(m) = 2^4 \cdot 3^2, \quad \phi(m) = 2^{31} \cdot 3^{10}.$$

In a later chapter we shall show that there is no exponent ν less than $\lambda(m)$ for which the congruence

$$a^\nu \equiv 1 \pmod{m}$$

is verified for every integer a prime to m .

From our theorem, as stated above, Fermat's general theorem follows as a corollary, since $\lambda(m)$ is obviously a factor of $\phi(m)$,

$$\phi(m) = \phi(2^\alpha) \phi(p_1^{\alpha_1}) \dots \phi(p_n^{\alpha_n}).$$

1.10.1 Solve 31.8-2(CLRS)

It is possible to strengthen Euler's theorem slightly to the form
 $a^{\lambda(n)} \equiv 1 \pmod{n}$ for all $a \in \mathbb{Z}_n^*$,
where $n = p_1^{e_1} \cdots p_r^{e_r}$ and $\lambda(n)$ is defined by

$$\lambda(n) = \text{lcm}(\phi(p_1^{e_1}), \dots, \phi(p_r^{e_r})).$$

Prove that $\lambda(n) \mid \phi(n)$. A composite number n is a Carmichael number if $\lambda(n) \mid n - 1$. The smallest Carmichael number is $561 = 3 \cdot 11 \cdot 17$; here, $\lambda(n) = \text{lcm}(2, 10, 16) = 80$, which divides 560. Prove that Carmichael numbers must be both "square-free" (not divisible by the square of any prime) and the product of at least three primes. (For this reason, they are not very common.)

ENG

1. Prove that $\lambda(n) \mid \phi(n)$.

$$n = p_1^{e_1} \cdots p_r^{e_r}$$

$$\begin{aligned}\phi(n) &= \phi(p_1^{e_1}) * \dots * \phi(p_r^{e_r}) \\ \text{lcm}(\phi(p_1^{e_1}), \dots, \phi(p_r^{e_r})) &| (\phi(p_1^{e_1}) * \dots * \phi(p_r^{e_r})) \\ \lambda(n) &| \phi(n)\end{aligned}$$

2. Prove that Carmichael numbers must be both “square-free” (not divisible by the square of any prime)

let Carmichael number $n = p^\alpha m (\alpha \geq 2, p \nmid m)$ $a^{n-1} \equiv 1 \pmod{n}$ ($\gcd(a, n) = 1$)

set $a = p + 1$ then $(p + 1)^n \equiv p + 1 \pmod{n}$

and $\gcd(p^2, a) = 1$

$$(p + 1)^n \equiv (p + 1)^{p^2 p^{\alpha-2}} \equiv p + 1 \pmod{p^2}$$

but $\gcd(p^2, a) = 1, a \equiv 1 \pmod{p^2}$

$p + 1 \equiv 1 \pmod{p^2}$ This is impossible

<https://math.stackexchange.com/questions/1764812/carmichael-number-square-free>

3. the product of at least three primes.

Assume that $n = pq$, with $p < q$ two distinct primes, is a Carmichael number. Then we have $qa1 \pmod{q`1} \rightarrow n \equiv pq \equiv p \pmod{q`1} \rightarrow n`1 \equiv p`1 \pmod{q`1}$. Here $0 < p`1 < q`1$, so $n`1$ is not divisible by $q`1$.

<https://math.stackexchange.com/questions/432162/carmichael-proof-of-at-least-3-factors>

KR

2. Prove that Carmichael numbers must be both “square-free” (not divisible by the square of any prime)

Proof. 카마이클 수 $n = p^\alpha m (\alpha \geq 2, p \nmid m)$ 라 하자 정의에 따라 다음이 성립한다. $a^n \equiv a \pmod{n}$.

$a \equiv 1 + p \pmod{p^\alpha}$ 라 하자.

$(p + 1)^n \equiv p + 1 \pmod{m}$ 이 되는데. m 의 인자인 p^2 와에 대해서도 다음이 성립한다. $(p + 1)^n \equiv (p + 1)^{p^\alpha m} \equiv (p + 1)^{p^2 p^{\alpha-2} m} \equiv p + 1 \pmod{p^2}$ 그러나 $\gcd(a, n) = 1$ 이라서 $\gcd(a, p^2) = 1$ 이다. $(p + 1)^n \equiv p + 1 \equiv 1 \pmod{p^2}$ 이며 이는 모순이다. \square

3. the product of at least three primes.

Proof. 서로다른 두 소수의 곱이 카마이클 수가 될 수 없음을 보이는것으로 충분하다.

$n = pq$ 라 하자($p < q$ 인 서로다른 소수)

$$q \equiv 1 \pmod{q-1} \rightarrow n \equiv pq \equiv p \pmod{q-1} \rightarrow n-1 \equiv p-1 \pmod{q-1}$$

$0 < p-1 < q-1$, $n-1$ 은 $q-1$ 로 나눠질수 없다. \square

1.11 RSA 시스템의 이해

1.11.1 RSA 공개 키 암호 시스템⁶ (RSA public-key cryptosystem)

이 알고리즘은 보안 기법중 하나로 가장 흔한 예시로서는 공인인증서가 있다.

$$A \longrightarrow B$$

A 가 B 에게 숫자를 하나 보낸다고 생각 해보자. A 에게는 공개키가 필요하며 B 에게는 개인키가 있어야한다. 공개키는 누가 가져도 상관없는 키이며 개인키는 절대로 노출되어서는 안되는 키이다.

A 는 B 에게 a 를 보낼때 공개키를 이용하여 a 를 c 로 암호화 하여 보내며 B 는 c 를 공개키와 개인키를 이용하여 a 로 복호화하여 읽는 방식이다.

1.11.2 공개키, 암호키 생성

두 개의 소수 p, q 를 선택하여 $n = pq$ 를 계산한다.⁷ 그 후 $\phi = (p-1)(q-1)$ 을 계산하고 $\gcd(n, \phi) = 1$ 인 정수 e 을 선택한다. 그후 n 과 e 를 공개한다. $ed \bmod \phi = 1$ 이고 $0 < d < \phi$ 를 만족시키는 d 를 생성하여 d 를 개인키로 사용한다.⁸

1.11.3 단계

A 가 B 에게 정수 $a(0 \leq a \leq z - 1)$ 를 보내기 위해서 A 는 $c = a^e \bmod n$ 를 계산하여 c 를 보낸다.⁹ B 는 $c^d \bmod n$ 를 계산하면 이 값이 a 이다.

1.11.4 복호화 과정

$$\phi(n) = \phi$$

$$ed \bmod \phi = 1 \iff ed = b\phi(n) + 1 (b \text{는 어떤 상수})$$

⁶로널드 라이베스트(Ron Rivest), 애디 샤미르(Adi Shamir), 레너드 애들먼(Leonard Adleman)이 세명의 이름 앞글자를 따서 지었다.

⁷그 후 p, q 는 버린다. 가지고 있어봤자 개인키가 뚫리는 취약점이 될수가 있다.

⁸ d 는 위에서 언급한 나머지 연산에서 곱셈에 대한 역원을 구하는 방법으로 효율적으로 구할수 있다.

⁹ c 를 효율적으로 구하는 방법 또한 위에서 다루었다.

$$\begin{aligned}
c^d \bmod n &= (a^e \bmod n)^d \bmod n \\
&= (a^e)^d \bmod n = a^{ed} \bmod n \\
&= a^{b\phi(n)+1} \bmod n \\
&= (a^{\phi(n)})^b a \bmod n = a
\end{aligned}$$

10

1.11.5 이게 과연 안전한가?

이를 구하기위한 해결방법은 결과적으로 소인수분해와 직결되는데 그냥 n 을 p 와 q 로소인수 분해해버리면 끝난다. 그러나 소인수분해를 다행시간내에하는 알고리즘은 개발되지 않았다.

1.11.6 advanced RSA

오리지널은 위와 같지만 PKCS#1 (en) v2.0에 표준 RSA 알고리즘 방식이 바뀌었다. ϕ 대신 λ 를 사용하는 방법이다. λ 를 사용했을때, ϕ 일때보다 개인키의 범위가 작아지기 때문에 큰값을 가지지 않아도 되기때문이다.

¹⁰오일러정리 사용

제 2 장

Introduction to Quick Sort

2.1 소개

- 일반적으로 가장 많이 사용하는 정렬 알고리즘
- 비교 정렬
- 내부정렬
- 불안정 정렬
- 평균 복잡도: $O(n \lg n)$
- 최악의 복잡도 : $O(n^2)$
- C++ std::sort의 내부구현이 퀵소트로 되어있음¹

2.2 의사코드 및 동작

분할 정복(divide and conquer) 방법을 통해 설계 되었다. 작동의 이해는 당장에 유튜브에 검색만해봐도 동작 설명하는 5분짜리 유튜브가 많으니 그걸 참고하는 게 편하다.

```
1  QUICKSORT(A, p , r )
2      if  p < r
3          q = PARTITION(A, p , r )
4          QUICKSORT(A, p , q-1)
5          QUICKSORT(A, q+1, r )
```

Lomuto's Partition Scheme

```
1  PARTITION(A ,p ,r )
2      x= A[ r ] //pivot
3      i = p-1
4      for j = p to r-1
5          if A[ j]<= x
6              i = i + 1
7              exchange A[ i ] with A[ j ]
8      exchange A[ i+1] with A[ r ]
9      return i + 1
```

¹정확하게는 introsort : quicksort와 heapsort, insertionsort를 셋 다 사용한다.

Hoare, C. A. R.이 1961년 처음으로 Quick sort를 제안했다. PARTITION은 현재 일반적으로 Lomuto가 제안(1999)한 PARTITION이 유명하여 이를 기준으로 설명하며, Hoare가 제안한 Partition의 두 프로시저의 비교는 후에 따로 다룬다.

PARTITION 프로시저의 시간복잡도는 $\Theta(n)$ 이다.

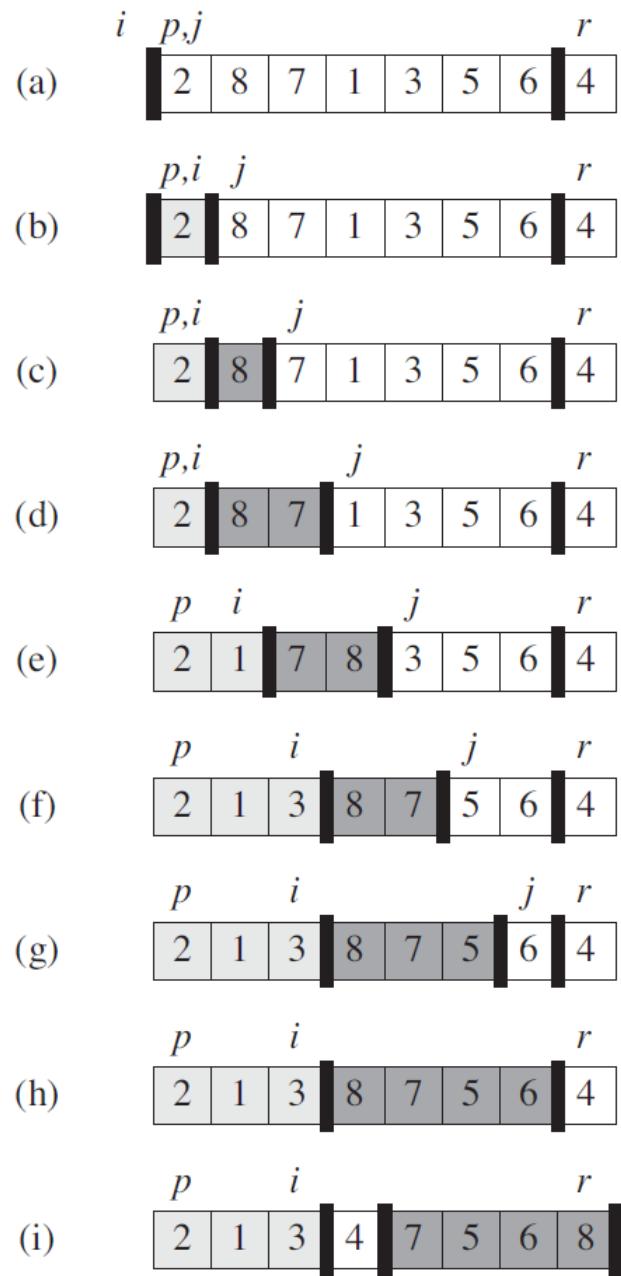


그림 2.1: quick sort 작동 예시[1]

2.3 여러 기초 지식

2.3.1 시간복잡도

정의 2.3.1 (복잡도) 상한, 하한,

- 상한 big O O

함수 $f(n), g(n)$ 에 대해서 $0 \leq f(n) \leq cg(n) (\forall n \leq n_0)$ 을 만족하는 n_0 , 양의 상수 c 가 존재할 때 $f(n) = O(g(n))$ 이라 한다.

- 하한 omega Ω

함수 $f(n), g(n)$ 에 대해서 $0 \leq cg(n) \leq f(n) (\forall n \leq n_0)$ 을 만족하는 n_0 , 양의 상수 c 가 존재할 때 $f(n) = \Omega(g(n))$ 이라 한다.

- Theta Θ

$\Theta(g(n))$ 일 필요충분 조건은 $f(n) = O(g(n))$ 이고 $f(n) = \Omega(g(n))$ 이 성립 할 때 이다.

2.3.2 조화 급수의 상한과 하한

$$\sum_{k=1}^n \frac{1}{k} = \Theta(\lg n)$$

감소 함수 $f(k)$ 에 대해서 다음이 성립한다

$$\int_{m-1}^n f(x)dx \leq \sum_{k=m}^n f(k) \leq \int_m^{n+1} f(x)dx$$

증가함수 $f(k)$ 에 대해 다음이 성립함을 그림 4를 통해서 이해 할 수 있다. 감소함수는 이와 반대로 생각하면 쉽게 해당 부등식을 이해할 수 있다.

$$\int_m^{n+1} f(x)dx \leq \sum_{k=m}^n f(k) \leq \int_{m-1}^n f(x)dx$$

다음 두 가지 방식으로 계산한다

$$\sum_{k=2}^n \frac{1}{k} + 1 \leq \int_2^{n+1} f(x)dx + 1 = \ln(x) + 1 = O(\ln x)$$

$$\int_1^{n+1} f(x)dx = \ln(x+1) = \Omega(\ln x) \leq \sum_{k=1}^n \frac{1}{k}$$

$$\text{따라서 } \sum_{k=1}^n \frac{1}{k} = \Theta(\lg n)$$

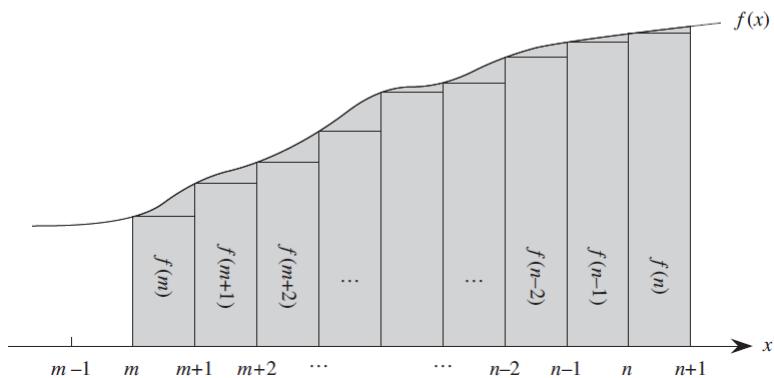
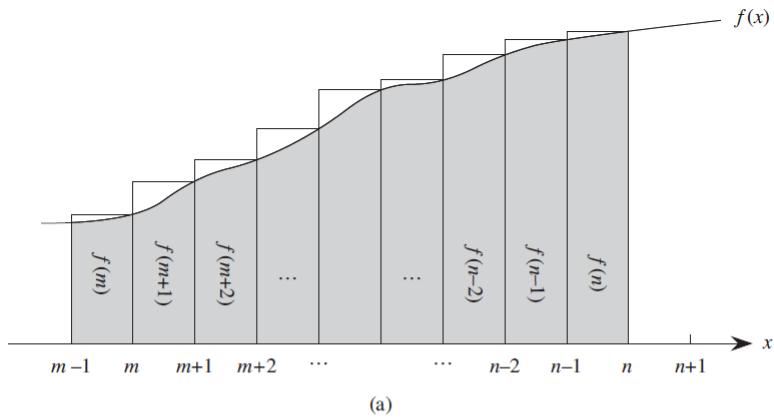


그림 2.2: 증가함수의 대소비교[1]

2.3.3 확률

Indicator random variables

$$I\{A\} = \begin{cases} 1 & (\text{ } H \text{ 발생}) \\ 0 & (\bar{H} \text{ 발생}) \end{cases}$$

$$\begin{aligned}
E[X_A] &= E[I\{A\}] \\
&= 1 \times \Pr(A) + 0 \times \Pr(\bar{A})^2 \\
&= \Pr(A)
\end{aligned}$$

2.4 대략적인 복잡도 분석

해당절과 다음절의 복잡도 분석은 quicksort에 모든 입력값에 중복값이 존재 하지않음을 미리 가정하고있다.

2.4.1 최악의 분할 케이스

최악의 경우 분할 케이스를 생각해보자 이는 왼쪽 오른쪽 분할이 한쪽으로 쏠려 (오름차순,내림차순) 극도로 불균형하게 일어났을때이다. 피봇값에 의한 분할이 아예 일어나지 않을 때 최악의 케이스가 된다. 이때 비용을 나타낸 재귀함수다.

$$T(n) = T(n - 1) + cn$$

$$\begin{aligned}
T(n) &= T(n - 1) + cn \\
&= T(n - 2) + c(n - 1) + cn \\
&= c \sum_{k=1}^n k \\
&= \frac{1}{2}cn^2 \\
&= \Theta(n^2)
\end{aligned}$$

따라서 시간복잡도는 $\Theta(n^2)$ 이다.

2.4.2 최선의 분할 케이스

정확하게 반으로 나누어 졌을때 최선의 분할 케이스이다.

²Pr은 A가 일어날 확률이다

이때의 비용을 나타낸 재귀함수는

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

이다.

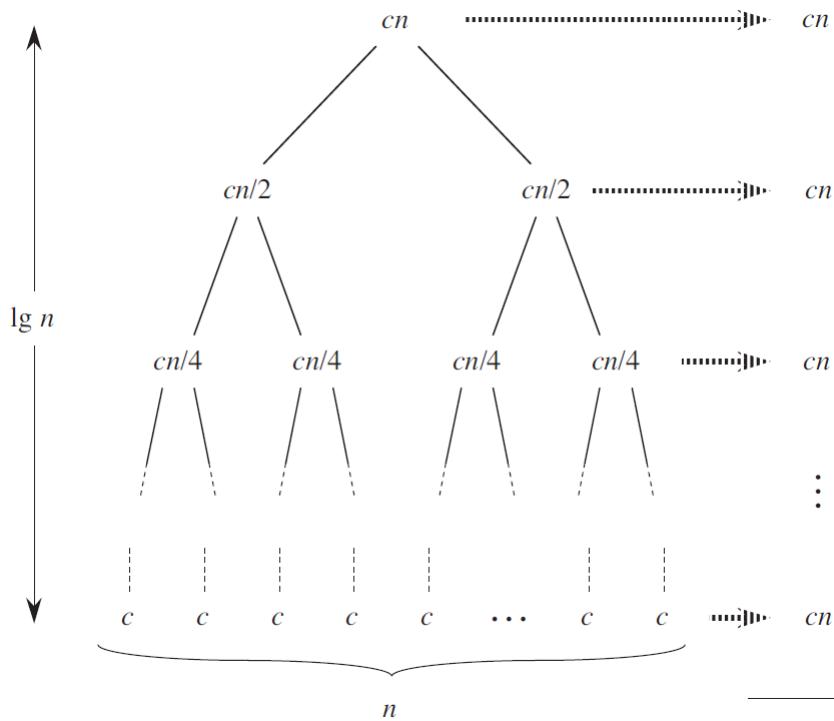


그림 2.3: quick sort 최선의 분할 케이스 재귀 트리[1]

그림 2의 재귀트리를 통해서 전체 비용을 계산하여 시간복잡도를 구하면 $\Theta(n \lg n)$ 이다.³

2.4.3 일반적인 케이스 직관적인 방법

평균적인 경우로 생각해볼수있는 다음 두가지 경우에 대해서 논의를 해 볼 것이다.

³master theory를 사용하여 바로 구해도 된다.

- 항상 9:1로 분할하는 경우
- 최악의 경우와 최선의 경우가 번갈아 나타나는 경우

항상 9:1로 분할하는 경우

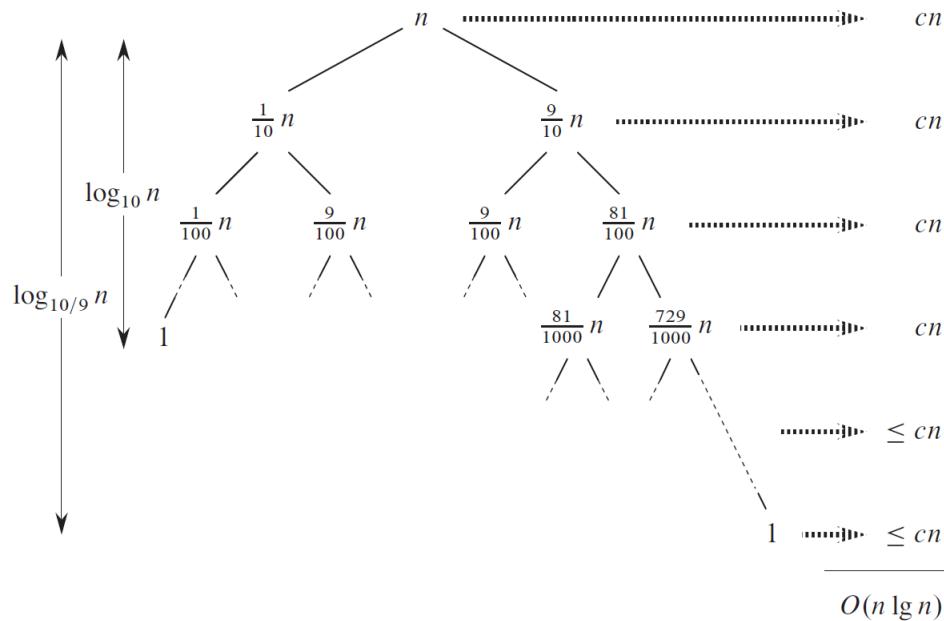


그림 2.4: 9:1로 분할하는 재귀 트리[1]

다음의 경우 재귀 함수는 다음이 성립한다.

$$T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn$$

이다. 이때 재귀트리는 깊이 $\log_{10} n$ 까지 각 깊이의 비용이 cn 이고 그 밑부터는 cn 보다 작은 비용이 든다 따라서 깊이 $\log_{10/9} n$ 에 각 깊이 비용 cn 인것보다 비용이 작으므로 $n \log_{10/9} n = O(n \log n)$ 이 된다. 따라서 $T(n) = \Theta(n \log n)$

최악의 경우와 최선의 경우가 번갈아 나타나는 경우

$T(n)$ 일때의 시간복잡도와 $T(n-1)$ 일때의 시간복잡도는 둘다 $\Theta(n)$ 이다. 따라서 이 둘의 시간복잡도를 합쳐도 결국 $\Theta(n)$ 이고 이를 합쳐서 보면 결국에 최선의

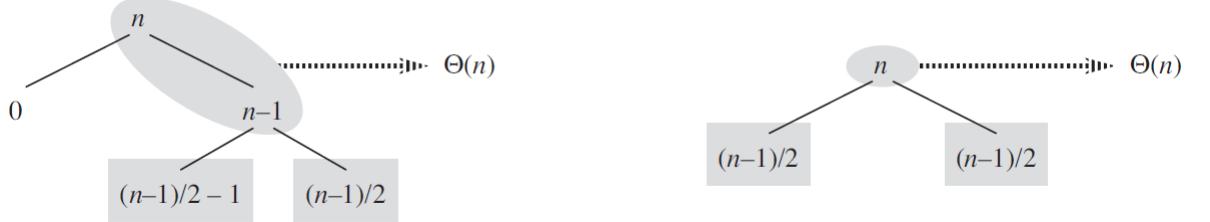


그림 2.5: 최악,최선이 번갈아 나타나는 재귀 트리[1]

분할 케이스가 된다 따라서 이때의 시간복잡도는 결국 $\Theta(n \lg n)$ 이다.

2.5 상세 분석

2.5.1 최악의 케이스 분석

실제 재귀함수를 일반화 식은 다음이 된다.

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

이때 $T(n)$ 이 $\Theta(n^2)$ 임을 보이면된다. 치환법을 이용해서 이 점화식을 풀수있다. $T(n) \leq c_1 n^2$ 이라 가정하자. 그러면

$$T(n) \leq \max_{0 \leq q \leq n-1} (c_1 q^2 + c_1 (n-q-1)^2) + \Theta(n)$$

이 성립한다.

$0 \leq q \leq n-1$ 인 $c_1 q^2 + c_1 (n-q-1)^2$ 에 대해서

$f(q) = c_1 q^2 + c_1 (n-q-1)^2$ 이라하고 $f'(q) = 2c_1 q - 2c_1 (n-q-1)$ 이므로 $q = \frac{n-1}{2}$ 일때 극솟값을 가진다. 이 극솟값은 q 의 존재 범위에 포함되고 따라서 이차함수의 특성에 따라 양 끝점이 최댓값이 될수있는 후보가된다. $q = 0$ 일때와 $q = n-1$ 일때 극댓값을 가지는데 이때의 두 합솟값은 같아서 둘중 어느값을 택해도 최댓값이 된다.

$$\begin{aligned} T(n) &\leq c_1(n-1)^2 + \Theta(n) \\ &\leq c_1 n^2 - c(2n-1) + \Theta(n) \\ &\leq c_1 n^2 \end{aligned}$$

이때 $\Theta(n) = dn$ 에서 $c_1 > d$ 인 상수 c_1 을 가지게 함으로써 결과적으로 $T(n)$ 이 $c_1 n^2$ 보다 작거나 같음을 보일수있다. 반대로 $c_2 n^2 \leq T(n)$ 임을 가정하고 $c_2 < d$ 인 상수를 잡음으로 $T(n)$ 이 $c_2 n^2$ 보다 크거나 같음을 보일수있다. 따라서 $T(n) = \Theta(n^2)$ 를 얻을 수 있다.

2.5.2 기대 수행 시간

1 부터 모든 n 에 대해서 모든 비용의 평균.

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^n X_i\right] \\ &= \sum_{i=1}^n E[X_i] \end{aligned}$$

시간복잡도를 분석하기위해서 다음의 보조정리를 이용한다.

Lemma 2.5.1. X 가 길이가 n 인 배열에서 *QUICKSORT*의 전체 실행에 대해서 *PARTITION*의 4행에서 수행된 비교문의 실행 수라고 가정하면 *QUICKSORT*의 실행시간은 $O(N+X)$ 이다.

Proof. 알고리즘은 *PARTITION*을 최대 n 회 호출한다. 각 호출은 for 루프를 실행하는데, for 루프의 각 반복은 4행 비교문을 실행한다. \square

따라서 모든 호출에 대해서 총 비교수 X 를 구하는 문제로 바꿔는데 각 호출에 대한 비교를 분석하지않고 총 비교수를 계산한다. 그렇게 하기 위해 배열 $A = \{z_1, z_2, \dots, z_n\}$ 의 각 요소가 내림차순으로 정렬되어있다고 생각한다. 또한 집합 $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$ 라 정의한다. 여기서 z_i 와 z_j 는 최대 한번 비교된다. 이유는 *PARTITION*에서 비교를 하는 경우는 하나의 원소가 pivot으로 선택 되었을 때인데, 이후에 이 pivot은 절대로 다른 원소와 비교하지 않는다. 따라서 $X_{ij} = I\{z_i \text{가 } z_j \text{와 비교한다}\}$

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}$$

$$\begin{aligned}
E[x] &= E \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij} \right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^n Pr\{z_i \succ z_j \text{와 비교한다}\}
\end{aligned}$$

$$\begin{aligned}
Pr\{z_i \succ z_j \text{와 비교한다}\} &= Pr\{Z_{ij} \text{에서 } z_i \text{ 또는 } z_j \text{가 첫번째로 선택된다.}\} \\
&= Pr\{Z_{ij} \text{에서 } z_i \text{가 첫번째로 선택된다.}\} + Pr\{Z_{ij} \text{에서 } z_j \text{가 첫번째로 선택된다.}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1}
\end{aligned}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1}$$

이는 $j - i$ 을 k 로 치환해서 상한을 얻을 수 있다.

$$\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&\leq \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k} \\
&\leq \sum_{i=1}^{n-1} c \lg n \\
&\leq cn \lg n
\end{aligned}$$

$$E[X] = O(n \lg n)$$

이한은 다음과 같이 직접구한다.

$$\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&= \sum_{k=1}^{n-1} \frac{2}{k+1} + \sum_{k=1}^{n-2} \frac{2}{k+1} + \cdots + \sum_{k=1}^1 \frac{2}{k+1} \\
&= (n-1) \frac{2}{1+1} + (n-2) \frac{2}{2+1} + \cdots + 1 \times \frac{2}{(n-1)+1} \\
&= \sum_{k=1}^{n-1} \frac{2}{k+1} \times (n-k) \\
&= \sum_{k=1}^{n-1} \left(\frac{2n}{k+1} - \frac{2k}{k+1} \right) \\
&= 2n \sum_{k=1}^{n-1} \frac{1}{k+1} - 2 \sum_{k=1}^{n-1} \frac{k}{k+1} \\
&\geq 2nc \lg n - 2(n-1) \left(\because - \sum_{k=1}^{n-1} \frac{k}{k+1} \geq - \sum_{k=1}^{n-1} \left(\frac{k}{k+1} + \frac{1}{k+1} \right) \right) \\
&\geq \Omega(n \lg n)
\end{aligned}$$

따라서
 $\Theta(n \lg n)$

2.5.3 Hoare's Partition VS Lomuto's Partition

Lomuto's Partition Scheme

```

1 PARTITION(A ,p ,r)
2     x= A[ r ] //pivot
3     i = p-1
4     for j = p to r-1
5         if A[ j]<= x
6             i = i + 1
7             exchange A[ i ] with A[ j ]
8         exchange A[ i+1 ] with A[ r ]
9     return i + 1

```

Hoare's partition scheme

```

1 PARTITION(A ,p ,r)
2     x= A[ r ] //pivot

```

```

3     i = p
4     j = r
5     while TRUE
6         repeat
7             j = j - 1
8             until A[j] <= x
9         repeat
10            i = i + 1
11            until A[i] >= x
12            if i < j
13                exchange A[i] with A[j]
14            else return j

```

다음 사이트의 답변을 번역했습니다 stackexchange

이해하기 편하고 간편한 알고리즘을 따질때 Lomuto's Partition이 간편하다. 그렇기에 우리들이 알고리즘을 이렇게 기억하고 있는것일 것이다. 성능적인 측면만 따지면 다음을 비교해 볼 수 있다.

비교 횟수

모든 요소가 피봇과 비교하기 때문에 둘다 $n-1$ 번 비교한다.

스왑 횟수

Lomuto's Partition의 경우 $1 \leq x \leq n$ 인 pivot값 x 에 대해서 스왑은 정확히 $x-1$ 번 수행한다.

$$\frac{1}{n} \sum_{x=1}^n (x-1) = \frac{n-1}{2}$$

Hoare's Partition의 경우 i 는 피봇값보다 큰값 j 는 피봇보다 작은 값에 대해서 스왑을 실행한다. 이 스왑을 수행하는 i 의 수와 j 의 수는 언제나 같은데 (당연히 쌍으로 교환하니까) 이 쌍의 수는 결과적으로 초기하 분포⁴를 따른다 따라서 쌍수는 $\frac{(n-x)(x-1)}{n-1}$ 이 된다.

$$\frac{1}{n} \sum_{x=1}^n \frac{(n-x)(x-1)}{n-1} = \frac{n-2}{6}$$

⁴학교에서 통계학을 안들어서 저도 잘 모릅니다 ㅠㅠ

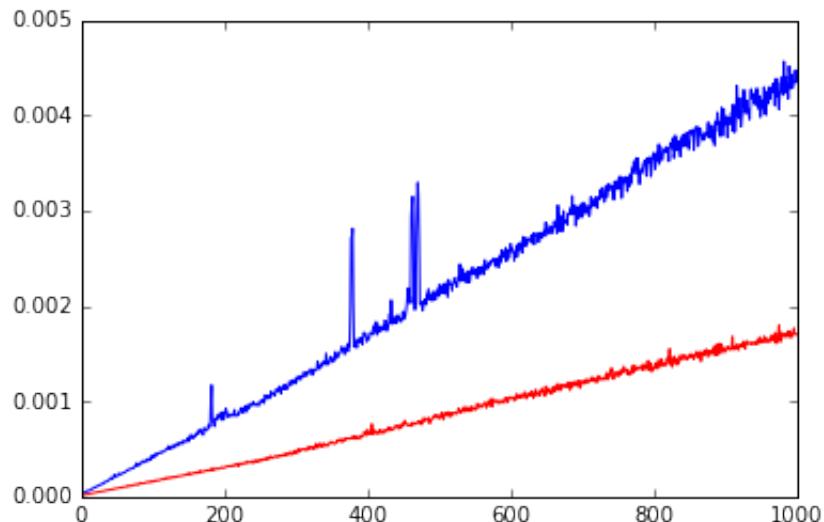
정렬이 이미 되어있는 경우

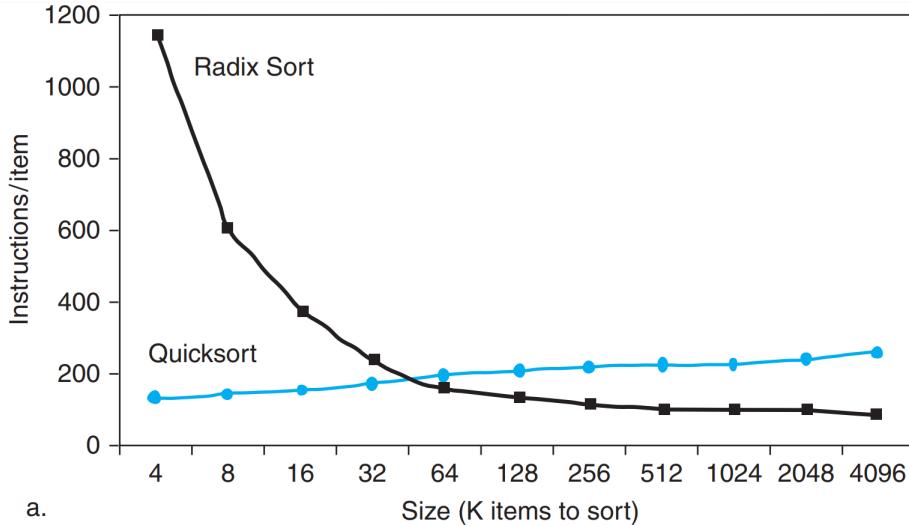
Hoare's Partition은 스왑을 실행하지 않는다 그러나 Lomuto's Partition은 $n/2$ 만큼의 스왑을 수행한다.

모든 배열이 같은 값으로 설정 되어있을때

Hoare's Partition은 무한루프에 빠진다. Lomuto's Partition인 경우 모든 단일 요소에 대해서 스왑을 실행하며 $i = n$ 이 되어 최악의 파티션 또한 가지게되어 수행시간은 $\Theta(n^2)$ 이 된다.

다음의 실제 테스트 결과를 봐도 알 수 있다.⁶





2.6 Quick sort의 캐시 히트율

다음은 Radix sort(기수 정렬)과 Quick sort의 입력 n 에 따른 수행 명령어 수/ n 를 나타낸 것이다. 기수 정렬의 시간복잡도는 $O(n)$ 이나 최고차항의 계수가 커서 초반 입력 n 에 대해서는 Quick sort가 빠른 것을 보여준다.

그러나 실제 수행시간과 캐시 미스율을 비교해 봤을 때, 퀵소트가 높은 캐시 적중률로 인해 기수정렬보다 약간 더 빠름을 볼 수 있다. 이는 알고리즘적인 부분 만으로는 알 수 없는 결과기에 실제 컴퓨터 구조의 캐시 개념을 알아야 한다. 작성자 의견: 해당 그래프에서 n 의 수치가 최대가 5000으로 나와 있는 걸 생각해볼 때 값이 정말 커지면 결국에는 시간복잡도에 따라 기수정렬이 더 빠름이 명확할 것으로 예상한다.

⁵참고 : hoare

⁶사진의 출처인데 Hoare, Lomuto testing 테스트 케이스가 상당히 아쉬움을 알 수 있다.

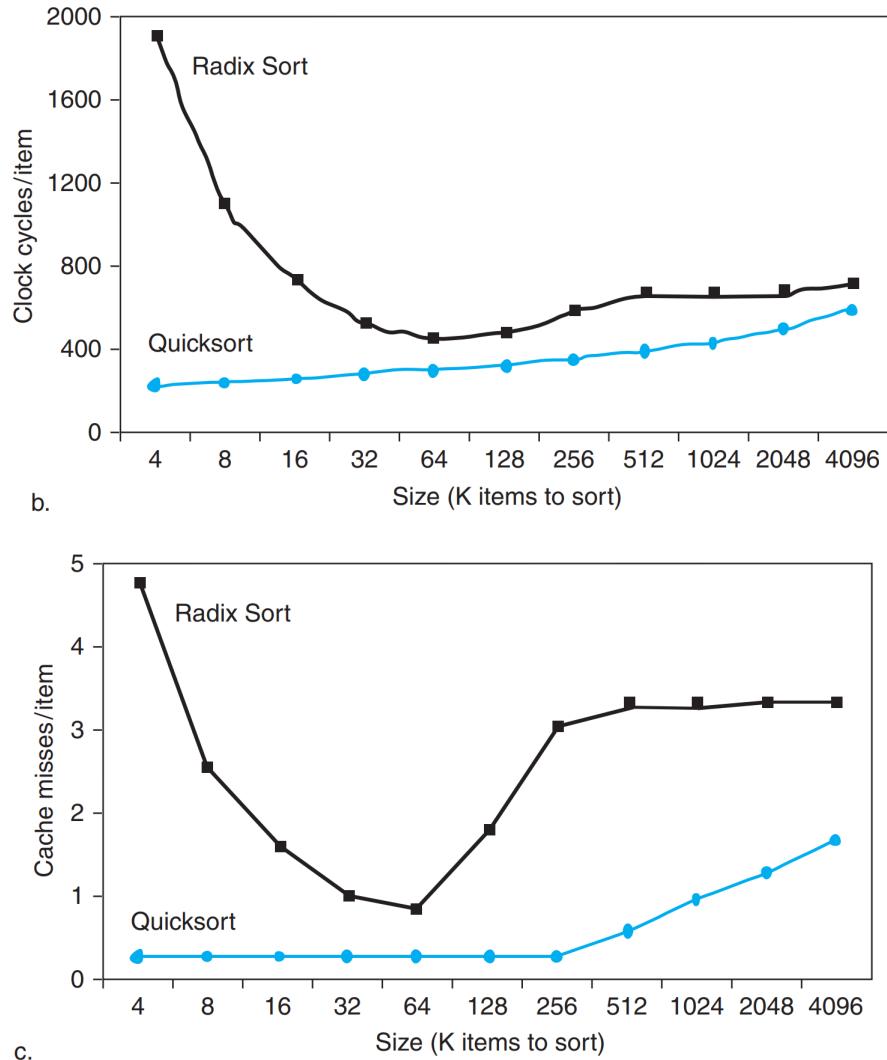


그림 2.6: Comparing Quicksort and Radix Sort by (a) instructions executed per item sorted (b) time per item sorted, and (c) cache misses per item sorted. This data is from a paper by LaMarca and Ladner [1996]. Due to such results, new versions of Radix Sort have been invented that take memory hierarchy into account, to regain its algorithmic advantages. The basic idea of cache optimizations is to use all the data in a block repeatedly before it is replaced on a miss.[2]

2.7 개선

2.7.1 재귀함수 제거

재귀 함수를 사용했을 때에 loop문과 비교했을 때 나타나는 문제점은 다음이 있다.

- 함수스택의 오버헤드
- 스택 오버플로우 위험성
- 메모리 부과

그러나 quick sort의 반복문 사용은 복잡하며 코드가독성이 떨어진다.

2.7.2 hybrid sort

Introsort

Quicksort는 입력값에 대한 의존도가 크기에 일정 깊이로 들어갈 경우 이 밑을 heapsort로 처리하게 한다. heapsort는 Quicksort와 같은 시간복잡도가 $O(n \log n)$ 이지만 최선, 최악에 대해서 비교적 평균적인 수행시간을 보장한다. 일반적으로 한 계 깊이를 $2 \log_2 n$ 으로 설정하고 있다.

Quick insertion sort

삽입 정렬(insertion sort)의 시간 복잡도는 $O(n^2)$ 이지만 베스트 케이스의 경우 (완전히 정렬되었는 경우) $O(n)$ 이다. (탐색만하고 넘어가기 때문) 또한 작은 n 에 대해서는 상대적으로 삽입정렬이 더 빠르게 되어 퀵소트 분할중 분할크기가 일정 n 이하가 되면 삽입정렬으로 처리해 실제 quick sort를 사용했을 때보다 시간적인 이득을 볼 수 있다. 또한 중복처리에 대해서 처리가 매우 빠르기 때문에 이에의 한 성능향상도 생각해 볼 수 있다. 이 n 은 일반적으로 10이며 이 값보다 작을 때 선택정렬을 수행하도록 한다.

다음은 선택정렬을 수행하는 n 에 따른 수행시간이다. 여기서 테스트 케이스의 $N = 10000$ 이다.

```
1 INSERTION_SORT(A)
2     for j = 2 to A.length
3         key = A[j]
4         i = j - 1
5         while i > 0 and A[i] > key
6             A[i+1] = A[i]
```

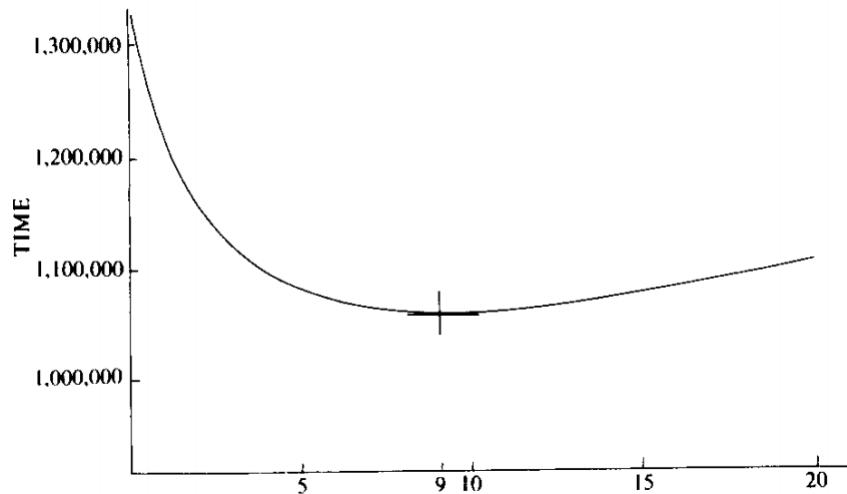


그림 2.7: 삽입정렬 수행의 분할크기 n 에 따른 퀵정렬 수행시간[3]

```

7     i = i - 1
8     A [ i + 1 ] = key

```

2.7.3 중복값 처리

네덜란드 국기 문제(Dutch national flag problem)로 다익스트라가 처음 제시한 이 문제는 quick sort의 중복값 입력에 대한 처리 문제를 다룬다. 배열에 같은 값으로만 들어왔을때를 생각해보자. 같은 값이 들어왔음에도 재귀는 $\lg n$ 까지 깊이 들어간다. 이를 해결하기 위해 분할을 세 가지로 한다. 이를 3 way partitioning이라고 한다. 기존의 왼쪽 오른쪽은 기존의 피봇값보다 크고 작은값이 들어가고 가운데에는 피봇과 같은 값을 모은다 그런후에 재귀의 범위를 왼쪽 오른쪽으로만 한다. 다음은 Dijkstra가 제안한 해답이다. 코드는 c++로 작성되어 있다.

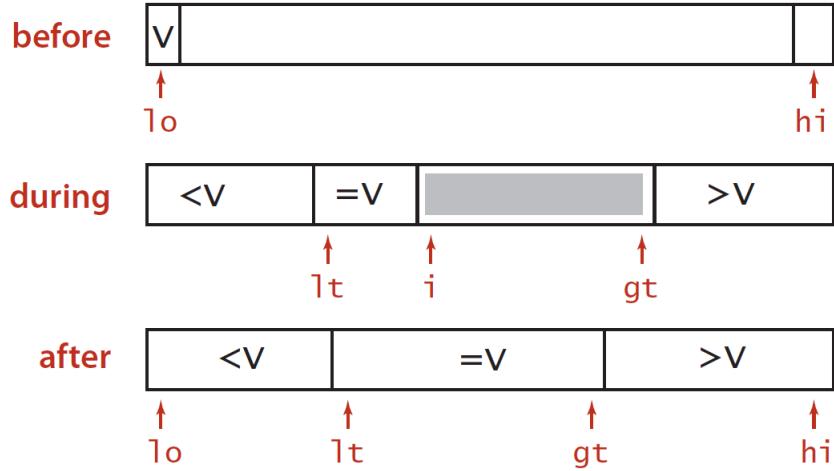


그림 2.8: 3-way-partitioning 작동 방식 [4]

```

1 void Quick3way( int a[], int lo, int hi)
2 {
3     if ( hi <= lo )
4         return ;
5     int lt = lo, i = lo + 1, gt = hi;
6     int v = a[lo];
7     while ( i <= gt )
8     {
9         if ( a[ i ] < v )
10        {
11            std::swap(a[ lt ], a[ i ]);
12            lt++, i++;
13        }
14        else if ( a[ i ] > v )
15        {
16            std::swap(a[ gt ], a[ i ]);
17            gt--;
18        }
19        else // a[ i]==v
20        {
21            i++;
22        }
23    }
24    Quick3way(a, lo , lt - 1);

```

```

25     Quick3way(a, gt + 1, hi);
26 }

```

다음은 J. Bentley과 D. McIlroy 제안한 좀더 빠른 의사코드이다.

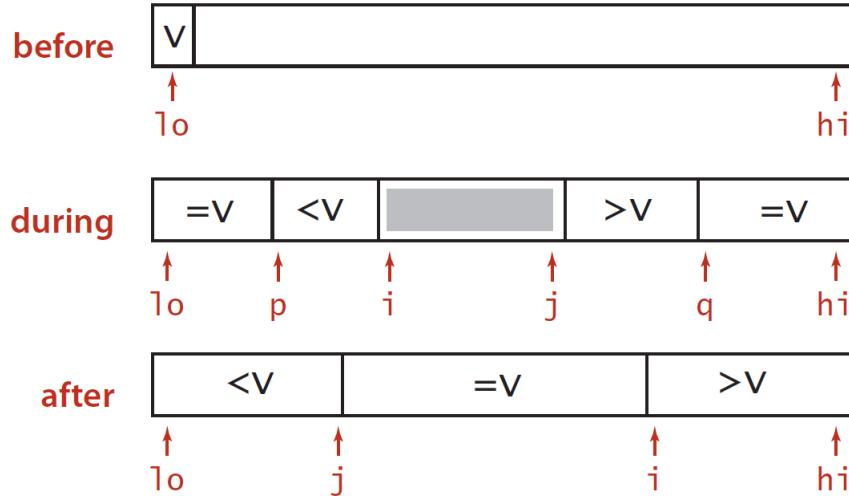


그림 2.9: Fast 3-way partitioning 작동 방식 [4]

```

1 void quicksort(Item a[], int l, int r)
2 {
3     int i = l-1, j = r, p = l-1, q = r; Item v = a[r];
4     if (r <= 1) return;
5     for (;;)
6     {
7         while (a[++i] < v) ;
8         while (v < a[--j])
9             if (j == l) break;
10        if (i >= j)
11            break;
12        exch(a[i], a[j]);
13        if (a[i] == v)
14        {
15            p++;
16            exch(a[p], a[i]);
17        }
18        if (v == a[j])
19        {
20            q--;

```

```

21         exch(a[j], a[q]);
22     }
23 }
24 exch(a[i], a[r]);
25 j = i-1;
26 i = i+1;
27 for (k = l; k < p; k++, j--)
28     exch(a[k], a[j]);
29 for (k = r-1; k > q; k--, i++)
30     exch(a[i], a[k]);
31 quicksort(a, l, j);
32 quicksort(a, i, r);
33 }
34

```

2.7.4 median of three

Sedgewick이 제안했다. 위의 의사코드는 pivot값으로 맨끝의 값을 설정한다. 이는 역순정렬에 의한 최악의 케이스를 생성하기 때문에 이를 입력 p,r의 중점을 피봇값으로 설정하는것만으로도 상수시간에 개선가능하며, 이에 대한 응용으로 값을 랜덤으로 세개 뽑은후 중간값으로 하는 방법도 있다.

2.7.5 Parallelization

리커젼으로 나눠지는 두분할은 각각의 메모리 침범을 하지않는것이 명확하기 때문에 쓰레드 분할로 처리해도 문제가 없다. 이때의 가장 이상적인 수행시간은 트리 깊이인 $O(\lg n)$ 이다. 성능에 대해서는 병렬화의 고질적인 문제들도 복합적으로 고려해야한다.

and ... 멀티피봇에 대한 논의도 있으나 생략한다⁷. quicksort에 대한 연구는 (특히 pivot설정) 매우 많이 진행되었고 진행되고있다.⁸

2.8 성능 테스트

환경

⁷다음이 좋은 참고 자료가 될것이다.Yukun Yao.(2019) A Detailed Analysis of Quicksort Algorithms with Experimental Mathematics

⁸원래 첫 제목은 All about Quicksort였으나 Quicksort에 대한 방대한 연구를 담아내기엔 상상이상으로 엄청난 논문이 줄을 이었다.

- msvc 14.2
- x86 Release모드
- C++
- Intel i7 7700
- RAM 16 GB

2.8.1 HOARE VS LOMUTO

| 횟수 | Hoare's partition | lomuto's partition |
|-----|-------------------|--------------------|
| 1회 | 0.012 | 0.015 |
| 2회 | 0.029 | 0.031 |
| 3회 | 0.033 | 0.036 |
| 4회 | 0.02 | 0.021 |
| 5회 | 0.031 | 0.036 |
| 6회 | 0.009 | 0.012 |
| 7회 | 0.021 | 0.023 |
| 8회 | 0.031 | 0.034 |
| 9회 | 0.017 | 0.018 |
| 10회 | 0.019 | 0.02 |

표 2.1: $n = 10000000$ 랜덤 순열 partition 수행비교

| 횟수 | Hoare's quick sort | lomuto's quick sort |
|-----|--------------------|---------------------|
| 1회 | 0.719 | 0.759 |
| 2회 | 0.704 | 0.77 |
| 3회 | 0.712 | 0.764 |
| 4회 | 0.69 | 0.761 |
| 5회 | 0.698 | 0.758 |
| 6회 | 0.696 | 0.759 |
| 7회 | 0.696 | 0.769 |
| 8회 | 0.692 | 0.761 |
| 9회 | 0.697 | 0.765 |
| 10회 | 0.695 | 0.755 |

표 2.2: $n = 10000000$ 랜덤 순열 quicksort 수행비교

앞서 살펴본 평균적인 시간복잡도와는 다르게 partition의 실제 차이가 세배가 나지 않았다.

2.8.2 개선 성능 테스트

- PARALLELIZATION
- insertion sort 삽입
- 3way partition

| 횟수 | Hoare's quick sort | lomuto's quick sort |
|-----|--------------------|---------------------|
| 1회 | 0.025 | 0.044 |
| 2회 | 0.013 | 0.04 |
| 3회 | 0.013 | 0.038 |
| 4회 | 0.014 | 0.041 |
| 5회 | 0.014 | 0.042 |
| 6회 | 0.013 | 0.044 |
| 7회 | 0.015 | 0.044 |
| 8회 | 0.014 | 0.041 |
| 9회 | 0.014 | 0.041 |
| 10회 | 0.016 | 0.042 |

표 2.3: $n = 10000$ 역정렬된 순열

| 횟수 | Hoare's quick sort | lomuto's quick sort |
|-----|--------------------|---------------------|
| 1회 | 0.023 | 0.035 |
| 2회 | 0.015 | 0.032 |
| 3회 | 0.019 | 0.037 |
| 4회 | 0.017 | 0.033 |
| 5회 | 0.012 | 0.031 |
| 6회 | 0.015 | 0.036 |
| 7회 | 0.016 | 0.036 |
| 8회 | 0.014 | 0.035 |
| 9회 | 0.015 | 0.034 |
| 10회 | 0.02 | 0.036 |

표 2.4: $n = 10000$ 정렬된 순열

insertion sort와 PARALLELIZATION의 파티션 분할에선 lomuto's partition을 사용했다.

| 횟수 | PARALLELIZATION quick sort | quick sort + insertion sort(n _j =10) |
|-----|----------------------------|---|
| 1회 | 0.511 | 0.717 |
| 2회 | 0.605 | 0.714 |
| 3회 | 0.541 | 0.715 |
| 4회 | 0.456 | 0.711 |
| 5회 | 0.727 | 0.715 |
| 6회 | 0.771 | 0.719 |
| 7회 | 0.744 | 0.705 |
| 8회 | 0.643 | 0.711 |
| 9회 | 0.649 | 0.715 |
| 10회 | 0.721 | 0.712 |

표 2.5: n = 10000000 랜덤한 임의 순열

| 횟수 | lumoto's quick sort | Dijkstra's |
|-----|---------------------|------------|
| 1회 | 0.778 | 0.772 |
| 2회 | 0.774 | 0.778 |
| 3회 | 0.759 | 0.777 |
| 4회 | 0.774 | 0.77 |
| 5회 | 0.761 | 0.766 |
| 6회 | 0.773 | 0.778 |
| 7회 | 0.772 | 0.782 |
| 8회 | 0.767 | 0.774 |
| 9회 | 0.777 | 0.784 |
| 10회 | 0.767 | 0.781 |

표 2.6: n = 10000000 랜덤한 비중복 임의 순열

| 횟수 | lumoto's quick sort | Dijkstra's |
|-----|---------------------|------------|
| 1회 | 0.41 | 0.033 |
| 2회 | 0.39 | 0.038 |
| 3회 | 0.366 | 0.032 |
| 4회 | 0.367 | 0.03 |
| 5회 | 0.36 | 0.032 |
| 6회 | 0.357 | 0.032 |
| 7회 | 0.361 | 0.031 |
| 8회 | 0.358 | 0.032 |
| 9회 | 0.356 | 0.031 |
| 10회 | 0.365 | 0.032 |

표 2.7: n = 10000000 0 ~ 1000 범위의 중복포함한 랜덤 순열

| 횟수 | Dijkstra's | J. Bentley D. McIlroy |
|-----|------------|-----------------------|
| 1회 | 0.036 | 0.032 |
| 2회 | 0.037 | 0.034 |
| 3회 | 0.036 | 0.031 |
| 4회 | 0.038 | 0.036 |
| 5회 | 0.04 | 0.035 |
| 6회 | 0.036 | 0.034 |
| 7회 | 0.043 | 0.031 |
| 8회 | 0.036 | 0.033 |
| 9회 | 0.038 | 0.036 |
| 10회 | 0.036 | 0.031 |

표 2.8: $n = 10000000$ 0 ~ 1000 범위의 중복포함한 랜덤 순열

2.8.3 std::sort 성능테스트 VS J. Bentley D. McIlroy

std::sort는 J. Bentley D. McIlroy의 3 way partition과 수행시간이 상당히 유사하다. 3 way partition에 insertion sort를 삽입했었는데 오버헤드 때문에 전체 수행시간이 오히려 안좋아졌다.

| 횟수 | 비중복 임의 순열 | 0 ~ 1000 범위의 중복을 포함한 랜덤 순열 | 모든값이 0 |
|-----|-----------|----------------------------|--------|
| 1회 | 0.964 | 0.38 | 0.007 |
| 2회 | 0.927 | 0.374 | 0.007 |
| 3회 | 0.959 | 0.373 | 0.007 |
| 4회 | 0.95 | 0.386 | 0.006 |
| 5회 | 0.946 | 0.381 | 0.006 |
| 6회 | 0.947 | 0.383 | 0.006 |
| 7회 | 0.947 | 0.392 | 0.006 |
| 8회 | 0.929 | 0.383 | 0.006 |
| 9회 | 0.932 | 0.385 | 0.006 |
| 10회 | 0.957 | 0.383 | 0.007 |

표 2.9: $n = 10000000$ std::sort의 성능 분석

| 횟수 | 비중복 임의 순열 | 0 ~ 1000 범위의 중복을 포함한 랜덤 순열 | 모든값이 0 |
|-----|-----------|----------------------------|--------|
| 1회 | 0.985 | 0.337 | 0.008 |
| 2회 | 1.011 | 0.332 | 0.008 |
| 3회 | 0.979 | 0.338 | 0.008 |
| 4회 | 0.956 | 0.336 | 0.008 |
| 5회 | 1.019 | 0.338 | 0.007 |
| 6회 | 0.974 | 0.333 | 0.007 |
| 7회 | 0.979 | 0.341 | 0.008 |
| 8회 | 0.973 | 0.341 | 0.009 |
| 9회 | 0.98 | 0.326 | 0.009 |
| 10회 | 0.98 | 0.332 | 0.008 |

표 2.10: $n = 10000000$ J. Bentley D. McIlroy의 3 way partition의 성능 분석

제 3 장

string matching

텍스트 $T[1..n]$

패턴 $P[1..m]$

Σ : 문자열에 사용되는 알파벳 집합

3.1 The naive string-matching algorithm

- 전처리 0
- 매칭시간 $O((n - m + 1)m)$

가장간단하면서 일반적으로 생각할수있는 방법이다. 각 k 마다 $T[k]$ 부터 $T[k+m-1]$ 까지 하나하나 P 와 맞는지 확인하는것이다.

```
1 NAIVE-STRING-MATCHER (T,P)
2     n = T.length
3     m = P.length
4     for s = 0 to n-m
5         if (P[1..m] == T[s+1..s+m])
6             print "Pattern occurs with shift s"
```

3.2 The Rabin-Karp algorithm

- 전처리 $\Theta(m)$
- 매칭시간 $O((n - m + 1)m)$

수행시간을 개선하는 첫번째 아이디어는 문자를 숫자로 바꾸는것이다. 111과 121이란 문자열을 비교하려면 첫번째방법인 각 자리별로 3번 비교해야하지만 숫자로 비교하는건 한번만 비교하면된다. 따라서 문자열을 숫자로 바꾸기위한 전처리 시간이 든다. 전처리는 P 패턴만을 $\Theta(m)$ 동안 바꾸고 문자열 T 는 비교와 동시에 처리한다.

그다음 문자열을 어떻게 숫자로 나타낼까를 고민해야하는데 이때 문자가 나타내는 언어의 총갯수에대한 진법에서 10진법으로 변환한다. 따라서 실제 알고리즘을 실행할때 언어의 총갯수 $d (|\Sigma|)$ 를 인자로 넣어주어야한다. 이때 숫자로 바꿀때 너무 긴 문자열을 숫자로 바꿀시에 나타나는 오버플로우를 감안해, 적당한 값으로 나눌 q 를 채용한다. 그래서 그 나타낸 숫자가 매칭이 이루어졌을때 실제로 같은 문자열인지 검사하는 부분이 필요하다. 이 q 는 일반적으로 dq 가 컴퓨터 한워드에 들어가는 소수로 채용한다.

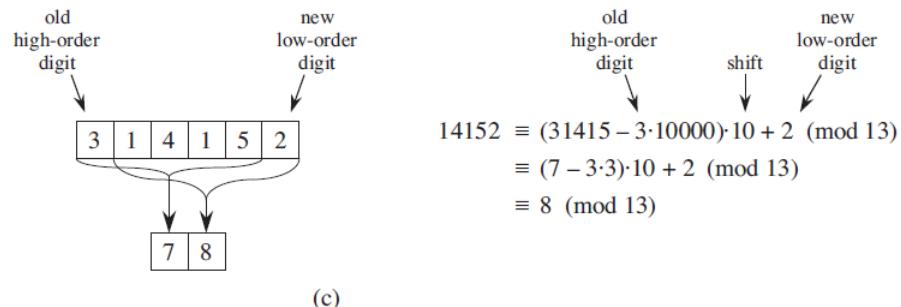
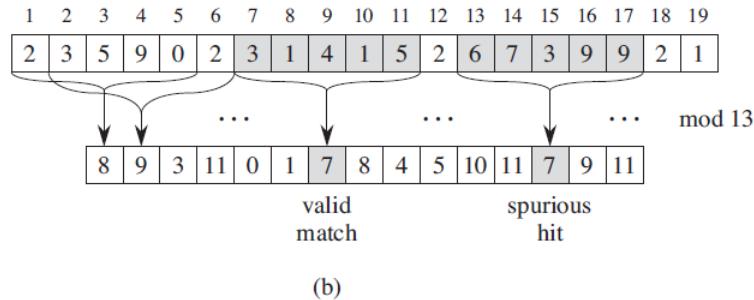
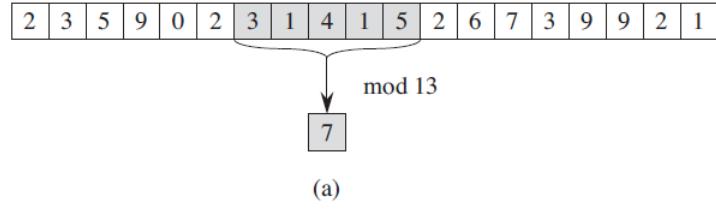


그림 3.1: $d = 10$, $q = 13$ 에 대한 라빈카프 알고리즘에 대한 수행을 나타낸다.
(a)는 “31415”에 대한 t 가 7인모습 (b)는 숫자가 겹치지만 서로 다른 문자열에 대한것 (c)는 숫자비교를 끝낸후 다음 문자열에대한 변환을 나타낸 것이다.

호너의 법칙(Horner's law)을 사용해 다음의 수식으로 $\Theta(m)$ 시간에 패턴 P 에대한 숫자 p 를 전처리한다.

$$p = (dp + P[i]) \bmod q$$

문자열 $T[s+1..s+m]$ 에 대한 숫자 t_s 의 처리는 매칭과 직후에 계산 한다.

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$$

$h = d^{m-1} \bmod q$ 이다. 가장 앞의 $T[s+1]$ 을 h 를 곱해 제거하고 왼쪽으로 시프트 후 $T[s+m+1]$ 을 더하는 방식이다.

```

1 RABIN-KARP-MATCHER(T, P, d, q)
2   n = T.length
3   m = P.length
4   h = d^{m-1} mod q
5   p = 0
6   t_0 = 0
7   for( i = 1 to m)
8     p = (dp+P[i]) mod q
9     t_0 = (dt_0 + T[i]) mod q
10  for s = 0 to n-m
11    if p == t_s
12      if P[1..m] == T[s+1..s+m]
13        print "Pattern occurs with shift s"
14      if s < n - m
15        t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) mod q

```

이 방법은 초기보다는 확실하게 개선되었지만 여전히 완벽하게 일치하는지 확인하기 위해 하나하나 비교하는 방법을 사용한다. 이상적인 경우는 $(m-n+a)$ 정도 돌겠지만 따라서 최악의 경우 매칭 시간이 $O((n-m+1)m)$ 이 된다.

3.3 String matching with finite automata

- 전처리 $O(m|\Sigma|)$
- 매칭시간 $\Theta(n)$

정의 3.3.1 (automata) finite automaton은 다음과 같이 5개의 튜플로 구성된다.

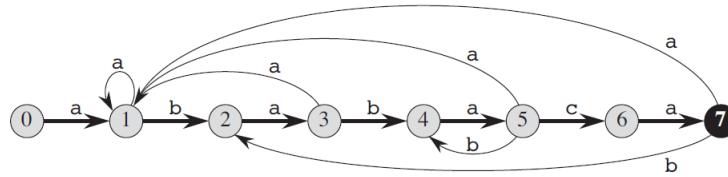
- Q : 유한 상태의 집합
- q_0 : 시작 상태 ($q_0 \in Q$)
- A : 받아들이는 상태의 구분된 상태 ($A \subset Q$)
- Σ : 유한 입력 알파벳
- δ : $Q \times \Sigma$ 에서 Q 로 매핑되는 전이 함수 M

CLRS의 번역본을 그대로 들고왔습니다.

뭔소린지모르겠죠?

저도 그럼

오토마타 내용 통째로 생략



(a)

| state | input | | | P |
|-------|-------|---|---|------|
| | a | b | c | |
| 0 | 1 | 0 | 0 | a |
| 1 | 1 | 2 | 0 | b |
| 2 | 3 | 0 | 0 | a |
| 3 | 1 | 4 | 0 | b |
| 4 | 5 | 0 | 0 | a |
| 5 | 1 | 4 | 6 | c |
| 6 | 7 | 0 | 0 | a |
| 7 | 1 | 2 | 0 | T[i] |

state $\phi(T_i)$

(b)

(c)

그림 3.2: $|\Sigma| = 3$ “a,b,c”, $m = 7$ 에 대한 스트링 매칭 오토마타 (a) 패턴 P 에 대한 상태 오토마타 (b) 상태표 (c) 상태표에 따른 연산

오토마타를 이용한 스트링 매칭은 전처리에 일치한 앞 문자열 상태에 따른 상태표 δ 를 구하고 이를 이용해 매칭을 하는 것이다. 상태는 $0, \dots, m$ 까지 있다. 초기 시작 상태는 0 이고(매칭되는 알파벳이 하나도 없는 상태) 매칭되는 알파벳갯수에 따라 상태 넘버를 가진다.

매칭시에 문자열 T 는 각 자리마다 상태를 가진다. 각 상태를 가지고 다음에 나오는 문자에 따라 상태를 변화시키고 상태가 m 에 도달하면 매칭이 된것으로 본다.

상태표의 특징은 상태가 변할때 일치한 문자열이 가장 길게 일치하는 상태로 보내는 것이다. 가령 그림 (c)에서 $T[5]$ 에서 $T[6]$ 으로 넘어갈때 “ababa”까지 일치했지만 “b”가 나옴으로써 매칭이 실패하지만 뒤에나온 “b”를 포함해 “abab”까지가 일치하기에 상태가 4로 돌아간것을 볼수있다.

```

1 FINITE-AUTOMATON-MATCHER( $T$ , delta , m)
2 n = T.length
3 q = 0
4 for i = 1 to n
5   q = d(q,T[i])

```

```

6     if q == m
7         print "Pattern occurs with shift i-m"

```

상태표를 구하기 위해서 다음과 같이 진행한다. 각 상태 q 마다 각 알파벳 a 를 뽑는다. 현재 상태 q 의 문자열 + 'a'가 임의의 상태 k 를 끝에서 포함하는지 검사하고 다를시 k 를 계속 낮춰간다.

```

1 COMPUTE-TRANSITION-FUNCTION(P, Sigma)
2 m = P.length
3 for q = 0 to m
4     for a in Sigma
5         k = min(m+1,q+2)
6         repeat
7             k = k - 1
8             until P_k ] P_q-a
9         d(q,a) = k
10 return d

```

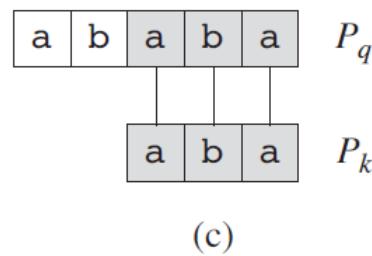
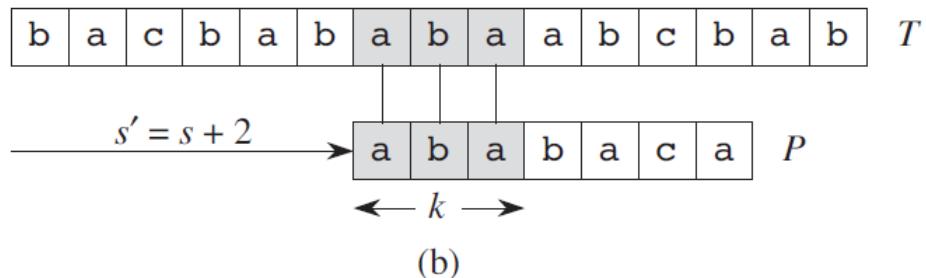
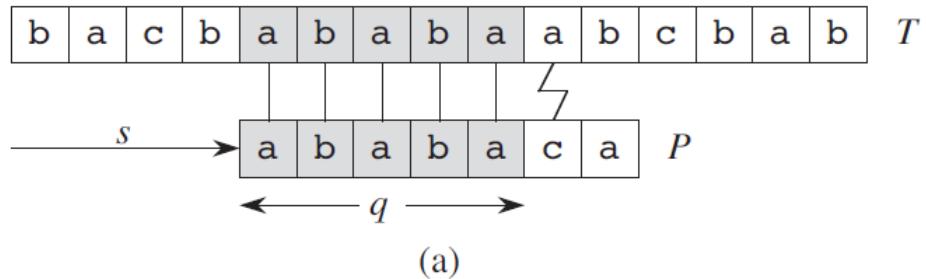
다음의 전처리의 수행시간은 $O(m^3\Sigma)$ 이다. 뒷절의 KMP방식을 차용해서 $O(m\Sigma)$ 로 개선할수있다.

라빈 카프보다 확실히 개선된 방안을 가졌으며 전처리에 문자열 T 의 개입또한 없다. 그러나 위의 전처리 시간은 $O(m^3\Sigma)$ 이며, 상태표의 공간 복잡도의 크기가 $\Theta(m|\Sigma|)$ 만큼 필요하다. 다음 절에서 이를 더 줄여볼것이다.

3.4 The Knuth-Morris-Pratt algorithm

- 전처리 : $O(m)$
- 매칭시간 : $\Theta(n)$

흔히 KMP 알고리즘이라 부르는데 이 알고리즘은 처음의 가장 기본적인 비교방식을 개선했다고 볼 수 있다. 전처리를 통해 계산한 $\pi[1..m]$ 배열을 사용하며, 매칭이 실패했을때, π 배열을 사용한다.



그림과 같이 “ababaca” 패턴을 문자열과 비교한다고 생각해보자. “ababa”까지 맞지만 ‘c’와 문자가 일치하지 않는다. 이때 앞의 문자열은 “ababa”가 패턴이 일치하는 것은 자명하다. 이때 “ababa”에 가장 근접하고 일치하는 수가 적은 P 의 패턴이 “aba”임을 알 수 있는 배열 $\pi[]$ 를 이용해서 다시 비교하지 않아도 “aba”까지 일치함을 알 수 있다.

```

1 KMP-MATCHER(T, P)
2     n = T.length
3     m = P.length
4     PI [] = COMPUTE-PREFIX-FUNCTION(P)

```

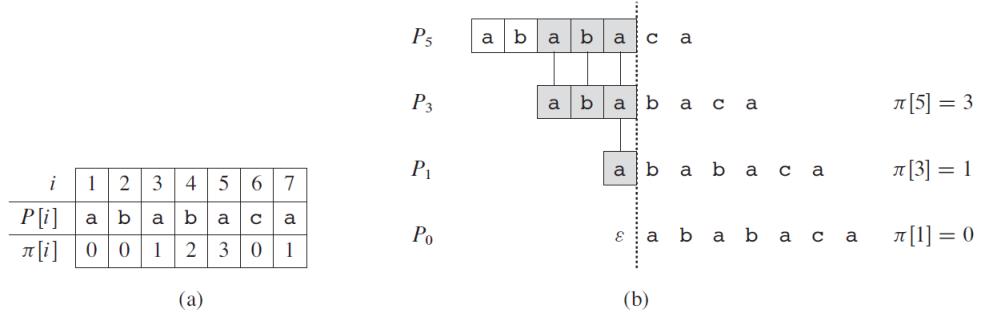


그림 3.3: (a)는 패턴 P 와 전처리한 $\pi[1..7]$ (b)는 $P[5]$ 에서 다음 문자가 매칭이 틀렸을때 그에 따른 π 값과 되돌아가는 순서를 나타낸것이다.

```

5     q = 0 // number of characters matched
6     for i = 1 to n // scan the text from left to right
7         while q > 0 and P[q+1] != T[i]
8             q = PI[q] // next character does not match
9             if P[q+1] == T[i]
10                q = q + 1 // next character matches
11             if q == m // is all of P matched?
12                 print "Pattern occurs with shift" i - m
13                 q = PI[q]

```

```

1 COMPUTE-PREFIX-FUNCTION(P) /
2 m = P.length
3 let PI[1..m] be a new array
4 PI[1] = 0
5 k = 0
6 for q = 2 to m
7     while k > 0 and P[k+1] != P[q]
8         k = P[q]
9         if P[k+1] == P[q]
10            k = k + 1
11        PI[q] = k
12 return PI

```

제 4 장

FFT

4.1 개론

n차 다항식 $A(x)$ 를 예시로 들라고하면 대부분 이렇게 대답할것이다.

$$A(x) = a_0x^0 + a_1x^1 + a_2x^2 + \cdots + a_nx^n = \sum_{k=0}^n a_kx^k$$

이는 컴퓨터상에서 크기가 $n+1$ 인 벡터(배열)에 $\{a_0, a_1, \dots, a_n\}$ 으로 나타내 도 $A(x)$ 를 확정지을 수 있다. 이러한 표현 방식을 **계수 표현(Coefficient representation)**이라고 한다. 우리 일반적으로 사용하는 다항식을 나타내는 방식 이라고 생각하면되는데. 이렇게 나타낸 다항식을 각각 곱하는 경우를 생각해보자. 일반적으로 우리는 종이에서 다음 다항식을 곱할때 이와같이 풀것이다.

$$C(x) = A(x)B(x) = (a_0x^0 + a_1x^1 + a_2x^2 + \cdots + a_nx^n)(b_0x^0 + b_1x^1 + b_2x^2 + \cdots + b_nx^n)$$

2차나 3차의 경우는 한번에 쭉 풀수도있겠지만 n 이 큰 경우일때는 어쩔수없이 $A(x)$ 항하나에 $B(x)$ 를 곱해서 쭉 전개해서 풀것이다. 설명의 편의를 위해서 $A(x)$ 와 $B(x)$ 의 최고차항의 차수가 같다고 가정하고 내용을 진행할 것이고 전체 흐름에 따라 n 의 의미가 중구난방적이다.

이를 직접 컴퓨터에서 계산한다고 생각해보자. 벡터 각요소에 벡터 각요소를 각각 모두 곱하기 때문에 시간복잡도는 $O(n^2)$ 이 될것이다. 이것이 우리가 일반적으로 생각하는 다항식 곱의 시간복잡도이다 이를 $O(n \log n)$ 으로 줄여 보는 것 이 목표이다.

4.2 점값 표현

중,고등학교를 나오면서 다음과 같은 문제를 푼적이 있을거라고 생각한다.

- 2차원 좌표상에서 $(1, 0)$ 과 $(6, 8)$ 을 지나는 직선의 방정식을 구하시오.
- 2차원 좌표상에서 $(1, 0), (6, 8), (-4, 8)$ 을 지나는 다항함수를 구하시오.

간단한 연립방정식처럼 생각하면 최고차항의 차수가 n 에 따라서 어떤 다항식 을 지나는 $n+1$ 개 이상의 점의 좌표를 알고있으면 그 다항식을 특정할 수 있다.

컴퓨터상에서도 그대로 나타낼수 있다. $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$

이를 점값 표현(Point-value representation)이라고 한다. 점값 표현으로 이용할 특징은 같은 좌표의 덧셈과 곱셈은 그냥 단순히 더하고 곱하면되는것이다. $C(x) = A(x)B(x)$ 에 대해서 $A(x_k) * B(x_k) = C(x_k)$ 가 된다. 따라서 점값표현으로 나타나있는 $A(x)$ 와 $B(x)$ 의 다항식 곱을 나타내기위해 $O(n)$ 의 시간만큼 걸린다. 실제로는 $C(n)$ 을 특정하기위해서 $2n - 1$ 만큼의 좌표쌍이 필요하다. 그 만큼 $A(x), B(x)$ 의 좌표쌍 또한 $2x - 1$ 개 만큼 필요하다.

4.3 계수표현 점값표현 치환에 대한 고찰

앞에서 점값표현으로 나타냈을때 다항식의 곱이 간편하다는것을 알았다. 이제 계수표현으로 나타나있는 다항식을 점값표현으로 나타내고 이것의 역변환을 빠르게 할 생각을 하면된다.

1. x 좌표를 임의로 잡고 하나씩 계산하는 경우이다.

그럼 $n + 1$ 개의 계수에 대해서 x 가 x^n, \dots, x 를 구해서 계수를 각각 계산하는것은 $n + \dots + 1 = \frac{1}{2}n^2 + n$ 만큼의 시간이 걸린다. 이때 x 차수를 구하는 부분을 메모이제이션을 사용한다면 $O(2n)$ 만큼 걸린다.

2. 호어의 법칙(Horner's rule) 특정값 x_0 에 대해 점값표현을 가장 빠르게 구할방법은 연쇄적으로 계산을 하는것이다. $A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \dots + x_0(a_{n-2} + x_0(a_{n-1}))) \dots))$ 위 방식에 비해서 추가적인 공간복잡도도 $O(1)$ 이며 메모리 접근도 적기때문에 훨씬 빠르다고 볼수있다. 하지만 n 개의 점값표현을 구하기위해선 결국 $O(n^2)$ 만큼의 시간이 걸리기때문에 다항식을 쌍으로 곱하는것과 다를것이없다.

4.4 DFT와 FFT

변환과 역변환에 $O(n^2)$ 으로 푸는 방법까지는 알았다. x 점으로 복소수 근을 가지게 함으로써 변환에 $O(n \log n)$ 이 걸림을 보일수있다.

전체적인 순서는 이렇게된다.

1. $A(x)$ 와 $B(x)$ 에 $n/2$ 까지의 차수의 계수를 0으로 놓아 벡터를 확장한다.
2. 각각 FFT를 적용하여 점값표현을 구한다.
3. 점별로 곱해 $C(x)$ 를 구한다.

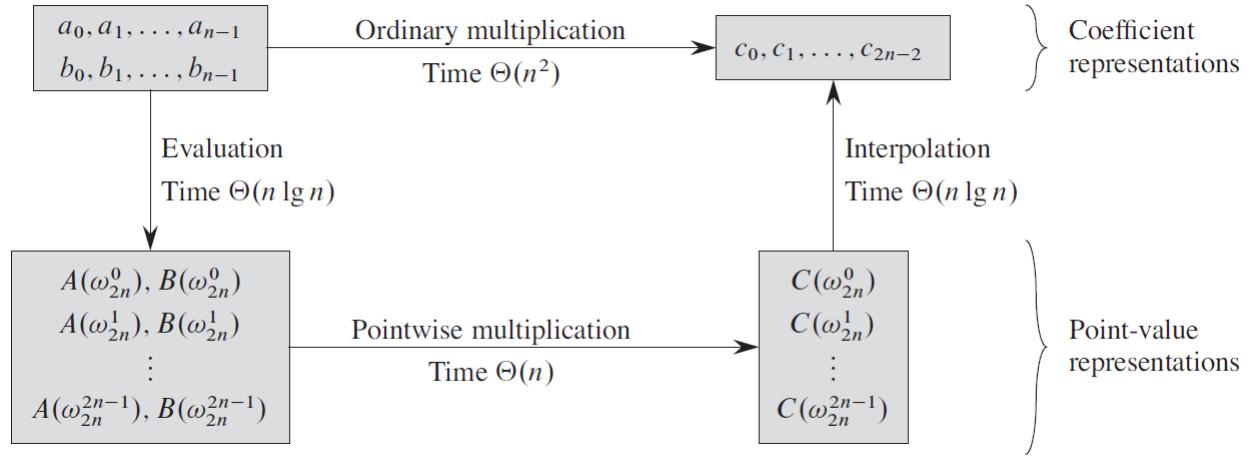


그림 4.1: 다항식곱의 전체 알고리즘 개형도

4. 다시 FFT를 적용하여 $C(x)$ 를 계수형으로 바꾼다.

4.4.1 복소수

복소수는 오일러 방정식으로 부터 시작합니다. $e^{ix} = \cos(x) + i \sin(x)$ $x = \pi$ 를 넣으면 $e^{i\pi} = -1$ 이란 어디서 봤을 수도 있는 식입니다. 여기에 $x = 2\pi$ 를 넣으면 $e^{2\pi i} = 1$ 이란 식이 나옵니다. 이제 ω_n 을 다음과 같이 정의합니다. $\omega_n = e^{\frac{2\pi i}{n}}$

ω_n 을 0부터 $n - 1$ 까지 지수승 한 집합을 나타내자.

$$\{\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}\}$$

이는 순환 구조를 나타낸다 $\omega_n^0 = \omega_n^n = 1$

우리의 목적은 최고차항이 $n - 1$ 인 다항식의 점값표현으로 쓸 x 좌표로 다음의 집합을 채택하는 것이다.

4.4.2 DFT

계수형으로 나타나있는 $A(x) = (a_0, a_1, \dots, a_n)$ 를 다음 $y = (y_0, y_1, \dots, y_n)$ 벡터를 이산 푸리에 변환(DFT : Discrete Fourier Transform)이라한다.

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

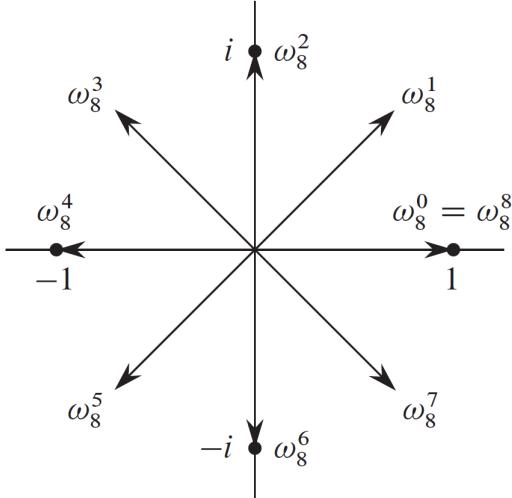


그림 4.2: 단위값의 8번째근

쉽게 나타내면 계수표현으로 나타나 있는 $A(x)$ 를 n 개의 집합이 $\{\omega_n^1, \omega_n^2, \dots, \omega_n^{n-1}\}$ 인 x 좌표에 대한 y 값의 집합입니다.

4.4.3 FFT

DFT를 $O(n \log n)$ 에 구할수있는 알고리즘으로 고속 푸리에 변환(FFT :Fast Fourier Transform)이 있다.

FFT는 분할 정복(divide and conquer)기법을 사용하여 나타낼 수 있다. A_{even} 과 A_{odd} 를 다음과 같이 정의한다. $A_{even}(x) = a_0x^0 + a_2x^1 + a_4x^2 + \dots + a_{n-2}x^{\frac{n}{2}-1}$ $A_{odd}(x) = a_1x^0 + a_3x^1 + a_5x^2 + \dots + a_{n-1}x^{\frac{n}{2}-1}$ $A(x)$ 를 다음과 같이 분할한다.

$$A(x) = A_{even}(x^2) + xA_{odd}(x^2)$$

프로시저는 다음과 같다.

1. A_{even} 과 A_{odd} 로 나눈다.
2. A_{even} 과 A_{odd} 를 각각 재귀로 FFT한다.
3. $A(x) = A_{even}(x^2) + xA_{odd}(x^2)$ 에 따라서 계산한다. 이때 걸리는 복잡도는 $O(n)$

1 RECUSIVE_FFT(a)

2 n = length [a]

```

3 if n == 1
4     return a
5 w_n = e ^{2 pi i / n}
6 w = 1
7 a_even = (a_0, a_2, ..., a_{n-2})
8 a_odd = (a_1, a_3, ..., a_{n-1})
9
10 y_even = RECURSIVE_FFT(a_even)
11 y_odd = RECURSIVE_FFT(a_odd)
12 for( k = 0 to n/2 -1 )
13     y[k] = y_even[k] + wy_odd[k]
14     y[k+(n/2)] = y_even[k] - wy_odd[k]
15     w = w*w_n
16 return y

```

이 알고리즘은 2로 나눈 재귀를 두번부르고 후에 추가적인 O(n)연산이 있다.

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n)$$

4.5 DFT^{-1} 역변환

점값표현에 대해서 다음의 식이 성립이 한다.

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj}$$

DFT 는 다음의 행렬 곱셈 식을 이용했다.

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

역행렬이 존재하며, 역변환에 대해서 다음의 식을 통해 원래의 값이 나온다.

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \frac{1}{n} \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \omega_n^{-3} & \cdots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \omega_n^{-6} & \cdots & \omega_n^{-2(n-1)} \\ 1 & \omega_n^{-3} & \omega_n^{-6} & \omega_n^{-9} & \cdots & \omega_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \omega_n^{-3(n-1)} & \cdots & \omega_n^{-(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

4.6 n = 4 일때 수행 예시

$$e^{i\theta} = \cos \theta + i \sin \theta$$

4.6.1 DFT 수행

$$\omega^j = \begin{cases} 1 & (j = 0, \theta = 0) \\ i & (j = 1, \theta = \frac{1}{2}\pi) \\ -1 & (j = 2, \theta = \pi) \\ -i & (j = 3, \theta = \frac{3}{2}\pi) \end{cases}$$

계수 벡터 $A = \{a_0, a_1, a_2, a_3\}$ 인 경우를 생각해보자. 이를 점값표현으로 변환, 역변환해 본다.

1. 분할: $\{a_0, a_2\}, \{a_1, a_3\}$
2. 분할: $\{a_0\}, \{a_2\}, \{a_1\}, \{a_3\}$
3. 결합: $\{a_0 + a_2, a_0 - a_2\}, \{a_1 + a_3, a_1 - a_3\}$
4. 결합: $\{(a_0 + a_1 + a_2 + a_3), (a_0 + \omega a_1 - a_2 - \omega a_3), (a_0 - a_1 + a_2 - a_3), (a_0 - \omega a_1 - a_2 + \omega a_3)\}$

계수벡터에 $1, \omega, \omega^2, \omega^3$ 을 넣었을때의 결과와 동일하다.

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} a_0 + a_1 + a_2 + a_3 \\ a_0 + \omega a_1 - a_2 - \omega a_3 \\ a_0 - a_1 + a_2 - a_3 \\ a_0 - \omega a_1 - a_2 + \omega a_3 \end{pmatrix}$$

4.6.2 DFT^{-1} 수행

$$\omega^j = \begin{cases} 1 & (j = 0, \theta = 0) \\ -i & (j = -1, \theta = -\frac{1}{2}\pi) \\ -1 & (j = -2, \theta = -\pi) \\ i & (j = -3, \theta = -\frac{3}{2}\pi) \end{cases}$$

$-\omega = \omega^{-1}$ ¹

1. 분할: $\{y_0, y_2\}, \{y_1, y_3\}$

¹참고: cos은 우함수(y축 대칭) sin은 기함수(원점대칭)이다.

2. 분할: $\{y_0\}, \{y_2\}, \{y_1\}, \{y_3\}$
3. 결합: $\{y_0 + y_2, y_0 - y_2\} \{y_1 + y_3, y_1 - y_3\}$
4. 결합: $\{(y_0 + y_1 + y_2 + y_3), (y_0 - \omega y_1 + y_2 + \omega y_3), (y_0 - y_1 + y_2 - y_3), (y_0 + \omega y_1 - y_2 - \omega y_3)\}$

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \frac{1}{n} \begin{pmatrix} y_0 + y_1 + y_2 + y_3 \\ y_0 - \omega y_1 - y_2 + \omega y_3 \\ y_0 - y_1 + y_2 - y_3 \\ y_0 + \omega y_1 - y_2 - \omega y_3 \end{pmatrix}$$

각 y_n 에 곱해지는 계수와 a_n 계수벡터를 풀어써서 행렬에 나타내면

- a_1

$$\begin{pmatrix} 1 & -\omega & -1 & \omega \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & -1 & -\omega \\ 1 & -1 & 1 & -1 \\ 1 & -\omega & -1 & \omega \end{pmatrix}$$

- a_2

$$\begin{pmatrix} 1 & -1 & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & -1 & -\omega \\ 1 & -1 & 1 & -1 \\ 1 & -\omega & -1 & \omega \end{pmatrix}$$

- a_3

$$\begin{pmatrix} 1 & \omega & -1 & -\omega \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & -1 & -\omega \\ 1 & -1 & 1 & -1 \\ 1 & -\omega & -1 & \omega \end{pmatrix}$$

4.7 성능 개선

첫번째로 반복계산되는것을 임시변수로 만들어서 계산을 줄이는 방법이 있다.

```

1 y[k] = y_even[k] + wy_odd[k]
2 y[k+(n/2)] = y_even[k] - wy_odd[k]
```

두번째로 루프풀기가 있다. fft의 루프풀기는 꽤 복잡하다. 분할정복기법에서 분할의 상태를 만들어놓고 반복문으로 conquer를 행한다. conquer시에 임시 배열로 할달되어 나뉘어져있던 짹수 홀수를 기존 하나의 큰 배열에 그대로 사용한다. 분할의 상태를 만들어 놓기위해 임의 위치 이동이 행해진다.

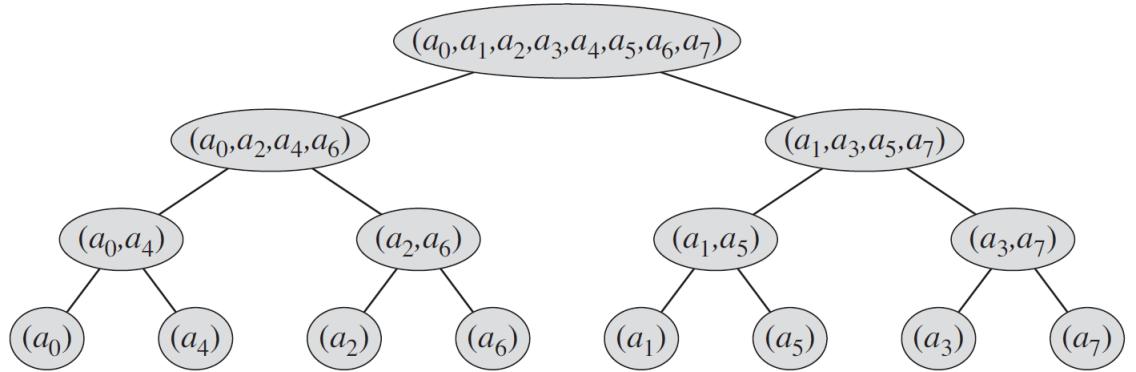
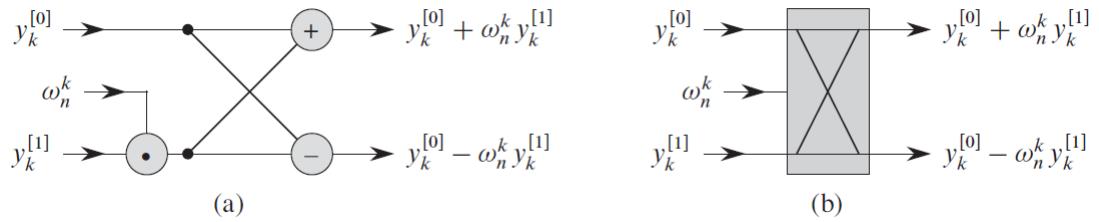


그림 4.3: $n = 8$ 일 때, recursive 시 분할되는 원소들



```

1 ITERATIVE-FFT(a)
2     BIT-REVERSE-COPY(a, A)
3     n = a.length // n is a power of 2
4     for s = 1 to lg n
5         m = 2^s
6         w.m = e^{2 pi i / m}
7         for k = 0 to n-1 by m
8             w = 1
9             for j = 0 to m/2 - 1
10                t = w A[k+j + m / 2]
11                u = A[k+j]
12                A[k+j] = u + t
13                A[k+j + m / 2] = u - t
14                w = w.m
15
16     return A

```

```

1 BIT-REVERSE-COPY(a, A)
2     n = a.length
3     for k = 0 to n - 1

```

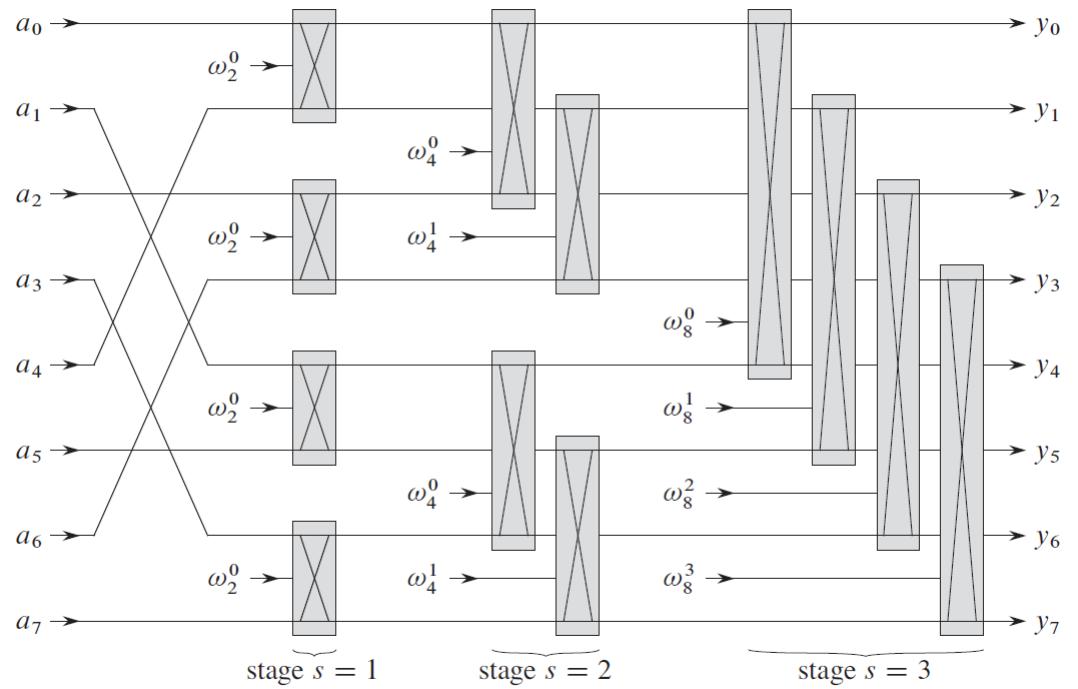


그림 4.4: $n = 8$ 의 iterative-FFT 수행

4

$$A[\text{rev}(k)] = a_k$$

마지막으로 parallel 알고리즘으로 바꾸는것이다.

참고 자료

- [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))
- <https://math.stackexchange.com/questions/1764812/carmichael-number-square-free>
- <https://math.stackexchange.com/questions/432162/carmichael-proof-of-at-least-3-factors>
- <https://crypto.stackexchange.com/questions/26751/rsa-with-lambdan-or-varphin>
- <https://en.wikipedia.org/wiki/Quicksort>
- <https://cs.stackexchange.com/questions/11458/quicksort-partitioning-hoare-vs-lomuto>
- <https://www.acmicpc.net/blog/view/58>
- https://www.youtube.com/watch?v=hq4dpyuX4Uw&list=PL52K_8WQ05oUuH06ML0rah4h05TZ4n381&index=11
- [https://en.wikipedia.org/wiki/Recursion_\(computer_science\)](https://en.wikipedia.org/wiki/Recursion_(computer_science))
- <https://algs4.tistory.com/45>
- <https://arxiv.org/pdf/1905.00118.pdf>
- <http://rohitja.in/lomuto-hoare-partitioning.html>
- <https://www.cs.princeton.edu/~rs/talks/QuicksortIsOptimal.pdf>
- https://en.wikipedia.org/wiki/Dutch_national_flag_problem
- <https://en.wikipedia.org/wiki/Introsort>

- <https://www.geeksforgeeks.org/internal-details-of-stdsort-in-c/>
- <https://stackoverflow.com/questions/44441876/quick-sort-using-stack-in-c>
- <http://fpl.cs.depaul.edu/jriely/ds1/extras/lectures/23Quicksort.pdf>
- <https://blog.naver.com/mindo1103/221234421987>
- <https://blog.naver.com/mindo1103/221234421626>

참고 문헌

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7.
- [2] Patterson, David A./ Hennessy, John L. Computer Organization and Design, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design). 2014. ISBN-13 9788994961897, ISBN-10 8994961895
- [3] Sedgewick, R. (1978). "Implementing Quicksort programs". Comm. ACM. 21 (10): 847–857. doi:10.1145/359619.359631.
- [4] Robert Sedgewick, Kevin Wayne Algorithms, 4th Edition, 2001 ISBN-13: 978-0321573513
- [5] 이산 수학(Discrete Mathematics) (저자:Richard Johnsonbaugh)
- [6] 암호학과 네트워크 보안(cryptography and network security) (저자: Behrouz A. Forouzan)
- [7] Robert D. Carmichael The Theory of Numbers (1914)