

Introduction to Quick Sort

EUnS

2019년 11월 14일

차례

차례	1
1 소개	2
2 의사코드 및 동작	2
3 여러 기초 지식	5
3.1 시간복잡도	5
3.2 조화 급수의 상한과 하한	5
3.3 확률	6
4 대략적인 복잡도 분석	8
4.1 최악의 분할 케이스	8
4.2 최선의 분할 케이스	8
4.3 일반적인 케이스 직관적인 방법	9
5 상세 분석	12
5.1 최악의 케이스 분석	12
5.2 기대 수행 시간	12
5.3 Hoare's Partition VS Lomuto's Partition	15
6 Quick sort의 캐시 히트율	18
7 개선	20
7.1 재귀함수 제거	20
7.2 hybrid sort	20
7.3 중복값 처리	21
7.4 median of three	24
7.5 Parallelization	24

8 성능 테스트	25
8.1 HOARE VS LOMUTO	26
8.2 개선 성능 테스트	26
8.3 std::sort 성능테스트 VS J. Bentley D. McIlroy	29
그림 차례	31
표 차례	31
참고 자료	32
참고 문헌	32

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 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.

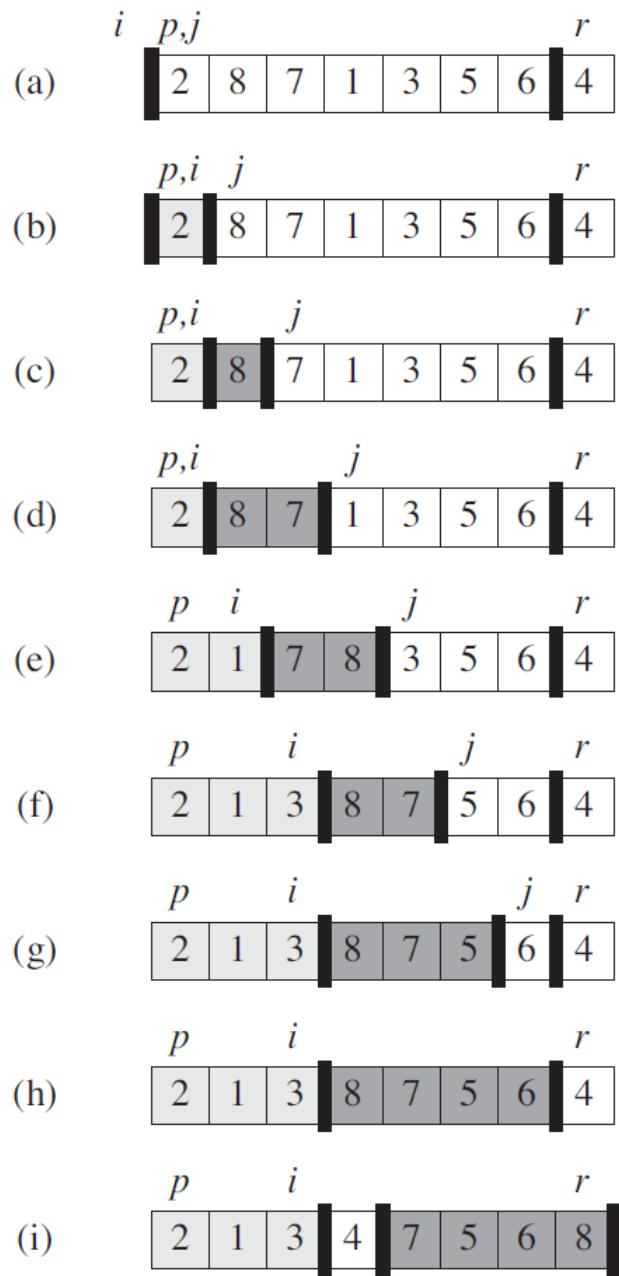


그림 1: quick sort 작동 예시[1]

3 여러 기초 지식

3.1 시간복잡도

정의 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.

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

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

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

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

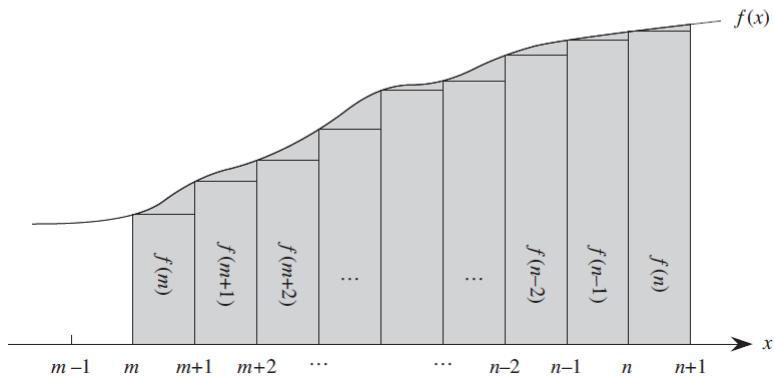
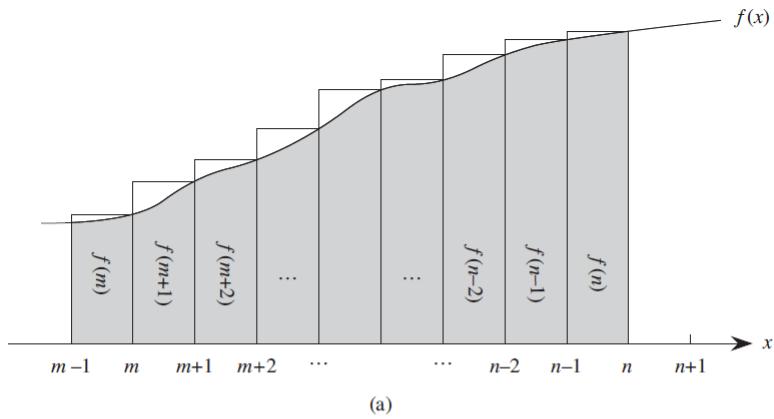


그림 2: 증가함수의 대소비교[1]

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 \Pr 은 A가 일어날 확률이다

4 대략적인 복잡도 분석

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

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.

4.2 최선의 분할 케이스

정확하게 반으로 나누어 쪼을때 최선의 분할 케이스이다.

이때의 비용을 나타낸 재귀함수는

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

이다.

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

³master theory를 사용하여 바로 구해도 된다.

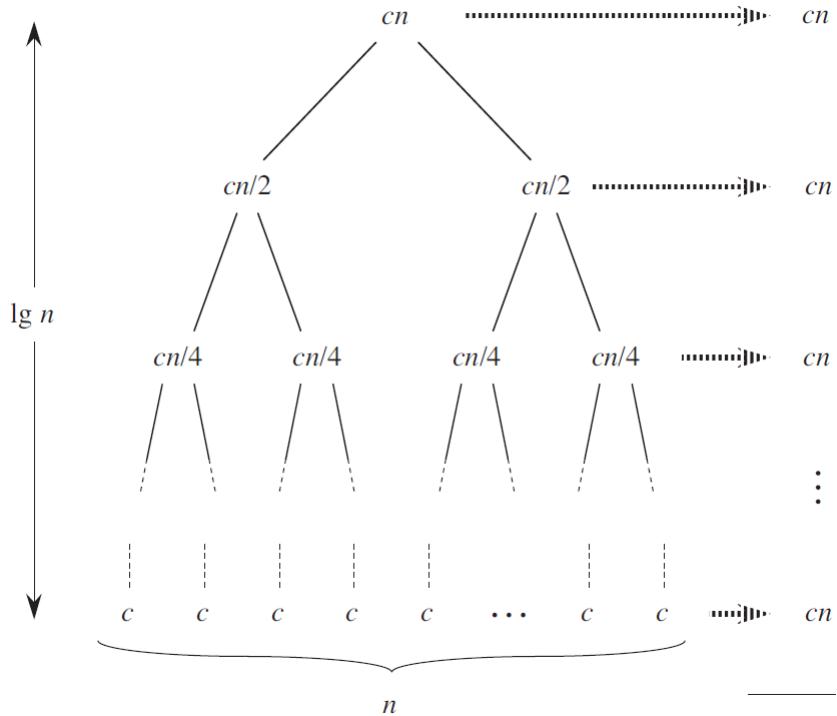


그림 3: quick sort 최선의 분할 케이스 재귀 트리[1]

4.3 일반적인 케이스 직관적인 방법

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

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

항상 9:1로 분할하는 경우

다음의 경우 재귀 함수는 다음이 성립한다.

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

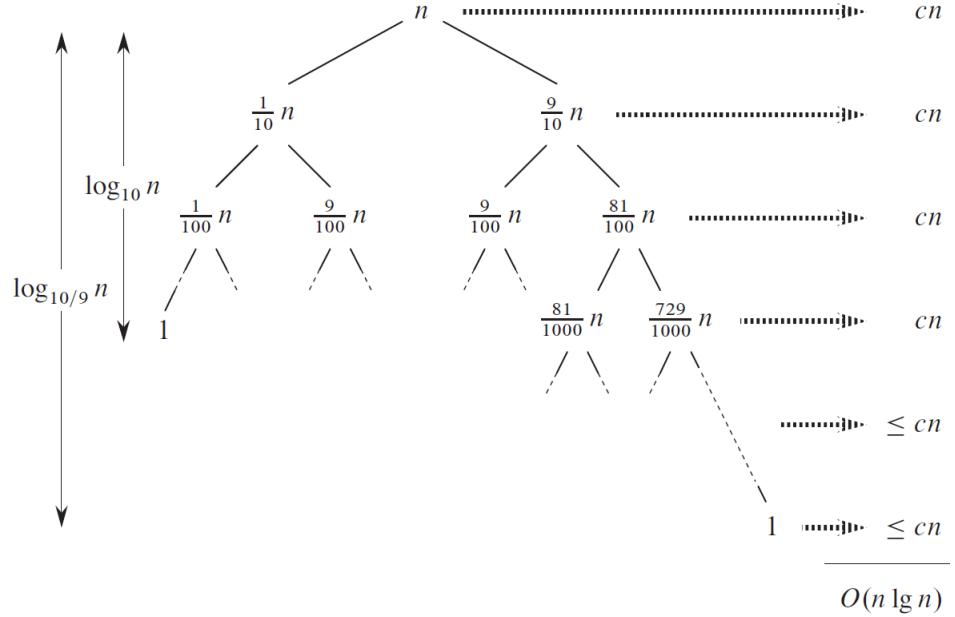


그림 4: 9:1로 분할하는 재귀 트리[1]

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



그림 5: 최악,최선이 번갈아 나타나는 재귀 트리[1]

$T(n)$ 일때의 시간복잡도와 $T(n-1)$ 일때의 시간복잡도는 둘다 $\Theta(n)$ 이다. 따라

서 이 둘의 시간복잡도를 합쳐도 결국 $\Theta(n)$ 이고 이를 합쳐서 보면 결국에 최선의 분할 케이스가 된다 따라서 이때의 시간복잡도는 결국 $\Theta(n \lg n)$ 이다.

5 상세 분석

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 $T(n) \in \Theta(n^2)$ 임을 보이면된다. 치환법을 이용해서 이 점화식을 풀수 있다. $T(n) \leq c_1 n^2$ 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 $T(n) \in c_1 n^2$ 보다 작거나 같음을 보일수있다. 반대로 $c_2 n^2 \leq T(n)$ 임을 가정하고 $c_2 < d$ 인 상수를 잡음으로 $T(n) \in c_2 n^2$ 보다 크거나 같음을 보일수있다. 따라서 $T(n) = \Theta(n^2)$ 를 얻을 수 있다.

5.2 기대 수행 시간

1부터 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 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 바뀌는데 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 $X_{ij} = I\{z_i \text{가 } z_j \text{와 비교한다}\}$

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

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

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

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
 $\Theta(n \lg n)$

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

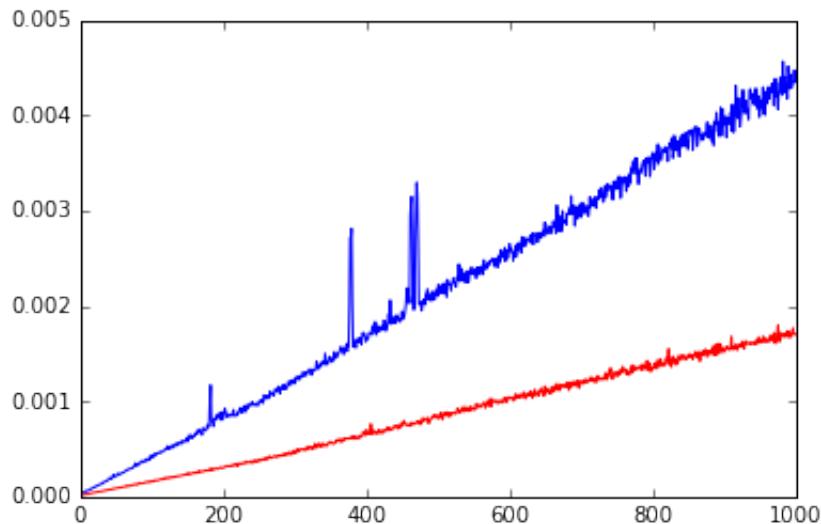
정렬이 이미 되어있는 경우

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

모든 배열이 같은 값으로 설정 되어있을때

Hoare's Partition는 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.

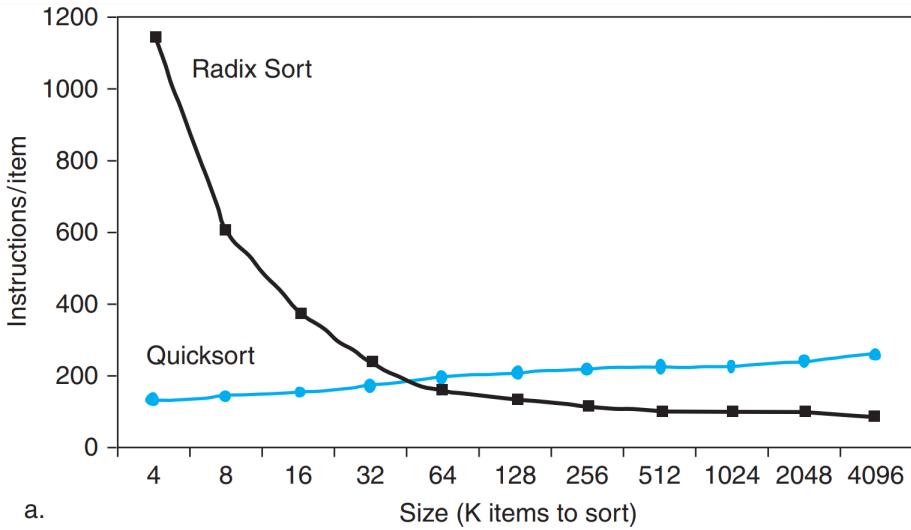
다음의 실제 테스트 결과를 봐도 알 수 있다.⁶



⁵ 참고 : hoare

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

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 생각해 볼 때 값이 정말 커지면 결국에는 시간복잡도에 따라 기수정렬이 더 빠름이 명확할 것으로 예상한다.

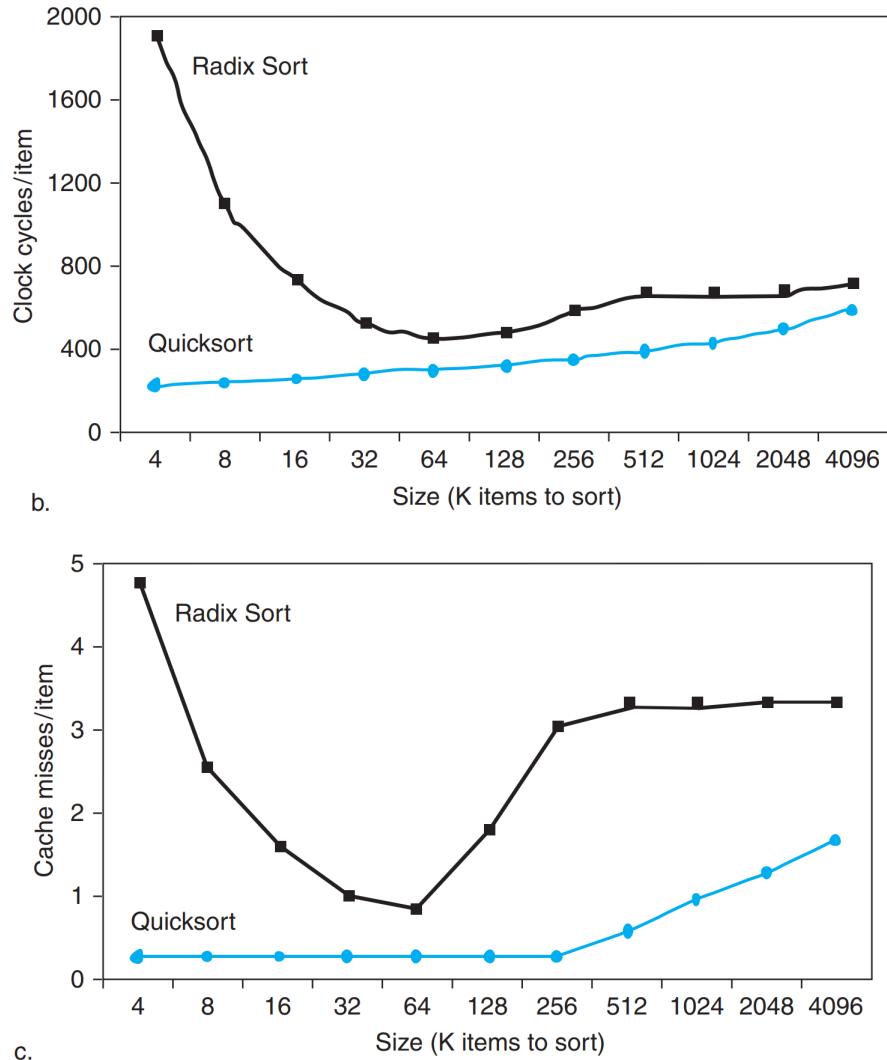


그림 6: Comparing Quicksort and Radix Sort by (a) instructions executed per item sorted (b) time per item sorted, and (c) cache misses per item sorted. This data is from a paper by LaMarca and Ladner [1996]. Due to such results, new versions of Radix Sort have been invented that take memory hierarchy into 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]

7 개선

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 코드가 독성이 떨어진다.

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]
```

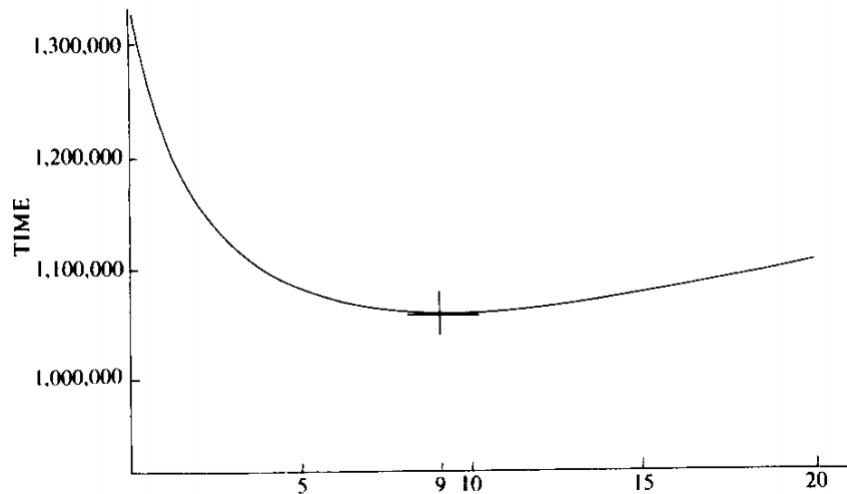


그림 7: 삽입정렬 수행의 분할크기 n 에 따른 쿠정렬 수행시간[3]

```

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

```

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.

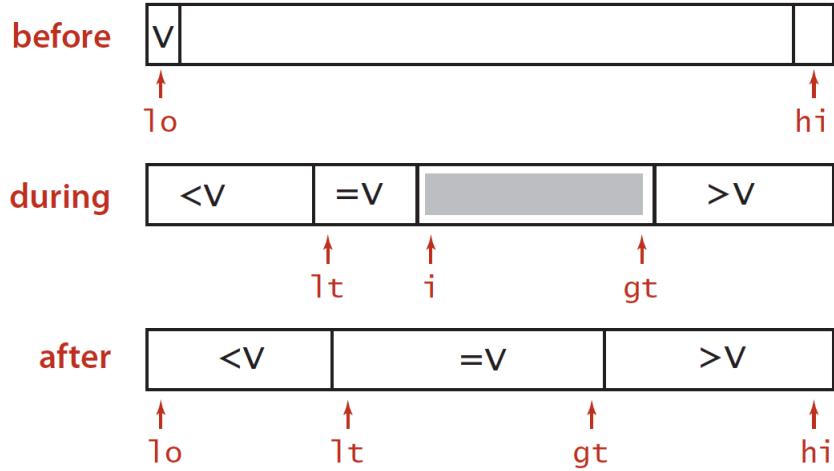


그림 8: 3-way-partitioning 작동 방식 [4]

```

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.

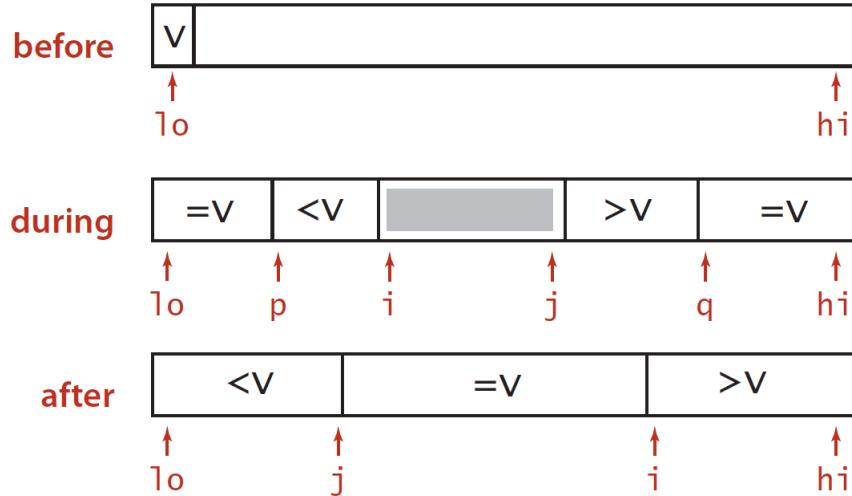


그림 9: Fast 3-way partitioning 작동 방식 [4]

```

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 <= l) 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

```

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.

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

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.

8 성능 테스트

환경

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

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 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: $n = 10000000$ 랜덤 순열 quicksort 수행비교

앞서 살펴본 평균적인 시간복잡도와는 다르게 partition의 실제 차이가 세배
가 나지 않았다.

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 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 4: $n = 10000$ 정렬된 순열

insertion sort와 PARALLELIZATION의 파티션 분할에선 lomuto's partition을 사용했다.

횟수	PARALLELIZATION quick sort	quick sort + insertion sort($n_i=10$)
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 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 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 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 8: $n = 10000000$ 0 ~ 1000 범위의 중복포함한 랜덤 순열

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 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 10: n = 10000000 J. Bentley D. McIlroy의 3 way partition의 성능 분석

그림 차례

1	quick sort 작동 예시[1]	4
2	증가함수의 대소비교[1]	6
3	quick sort 최선의 분할 케이스 재귀 트리[1]	9
4	9:1로 분할하는 재귀 트리[1]	10
5	최악,최선이 번갈아 나타나는 재귀 트리[1]	10
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]	19
7	삽입정렬 수행의 분할크기 n에 따른 쿼드정렬 수행시간[3]	21
8	3-way-partitioning 작동 방식 [4]	22
9	Fast 3-way partitioning 작동 방식 [4]	23

표 차례

1	n = 10000000 랜덤 순열 partition 수행비교	26
2	n = 10000000 랜덤 순열 quicksort 수행비교	26
3	n = 10000 역정렬된 순열	27
4	n = 10000 정렬된 순열	27
5	n = 10000000 랜덤한 임의 순열	28
6	n = 10000000 랜덤한 비중복 임의 순열	28
7	n = 10000000 0 ~ 1000 범위의 중복포함한 랜덤 순열	28
8	n = 10000000 0 ~ 1000 범위의 중복포함한 랜덤 순열	29
9	n = 10000000 std::sort의 성능 분석	29
10	n = 10000000 J. Bentley D. McIlroy의 3 way partition의 성능 분석	30

참고 자료

<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>

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