#### 다양한 상황에 따른 정렬 알고리즘 효율성 분석

|  |  |
| --- | --- |
| **학과** | 컴퓨터 공학과 |
| **학번** | 2014154030 |
| **이름** | 이윤복 |

# 목차

* + 1. **개요**
       1. 목적
       2. 측정 배경과 방법
       3. 기대효과
    2. **정렬에 대한 사전 지식**
       1. 삽입 정렬
       2. 병합 정렬
       3. 힙 정렬
       4. 퀵 정렬
       5. 기수 정렬과 Counting sort
    3. **정렬별 수행 시간 분석**
    4. **결론**
    5. **참고 문헌**

# 개요

1. 목적

* 인풋으로 들어오는 데이터의 형태와 양은 항상 다름
* 정렬마다 고유의 특성이 있기에 상황마다 효율적인 정렬은 제각각임
* 특정한 상황에서 어떤 정렬이 효율적인지 알게된다면 성능이 뛰어난 프로그램을 만들 수 있음

1. 측정 배경과 방법

* 동일한 환경 위해 동일한 PC 사용
* C언어를 사용해 구현하며 개발 환경으로는 VISUAL STUDIO를 이용
* 데이터를 저장할 자료 구조는 배열로 통일
* 모든 정렬은 동일하게 두개의 키값을 이용하여 정렬
* 단위를 초로 지정하면 0에 수렴함으로 CLOCK 단위로 측정
* 측정이 완료되면 데이터를 절반씩 줄여가며 재차 측정

1. 기대 효과

* 정렬들을 구현하면서 정렬의 특징과 특성을 자세히 파악할 수 있음
* 직접 상황을 부여하므로 정렬이 어떤 상황에 효율적인지 알 수 있음
* 따라서 후에 상황에 맞게 다양한 정렬을 사용할 수 있을 것

# 0.배경 지식

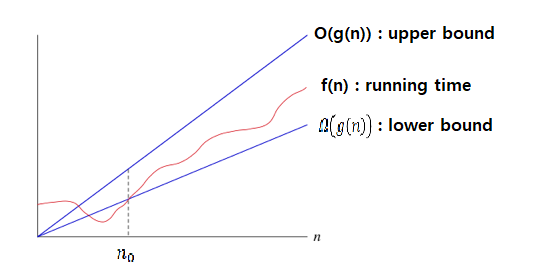
****

그림 0-1 Big-O와 Omega

* 1. **Big-O** 
     1. Big-O의 정의
* 만약에 , 모든 n에 대하여 을 성립하는

양의 상수 c와 이 존재한다면 f(n)은 O(g(n))의 집합의 원소가 될 수 있음

* + 1. Big-O 의 의의
* 그림 0-1에서 볼수 있듯이 Big-O는 f(n)의 상한선임
* 또한 O(g(n))과 f(n) 사이에는 무수한 원소가 존재하며 그 원소들은 대략적으로 모두 똑같은 효율을 나타냄

**0-2 Omega**

0-2-1) Omega의 정의

* 만약에 , 모든 n에 대하여 을 성립하는

양의 상수 c와 이 존재한다면 f(n)은 의 집합의 원소가 될 수 있음

0-2-2) Omega의 의의

* 그림 0-1에서 볼수 있듯이 Omega는 f(n)의 하한선임
* 또한 와 f(n) 사이에는 무수한 원소가 존재함

**0-3 theta**

0-3-1) theta의 정의

* 만약에 , 모든 n에 대하여 을 성립하는 양의 상수 c와 이 존재한다면 f(n)은 의 집합의 원소가 될 수 있음

0-3-2) theta의 의의

* 그림 0-1에서 볼수 있듯이 upper bound인 Big-O와 lower bound 인 Omega 사이에 존재한다면 의 집합의 원소임
* theta를 f(n)의 tight bound라고 함

**0-4 stable sort 와 unstable sort**

* 만약 두개의 키를 가지고 정렬을 할 때 앞에서 첫번째 키에 대해 정렬한 순서가 두번째 키에 대해 정렬할 때도 유지가 된다면 stable sort라고 함
* 만약 정렬한 순서가 흐트러진다면 unstable sort라고 함

# 삽입 정렬

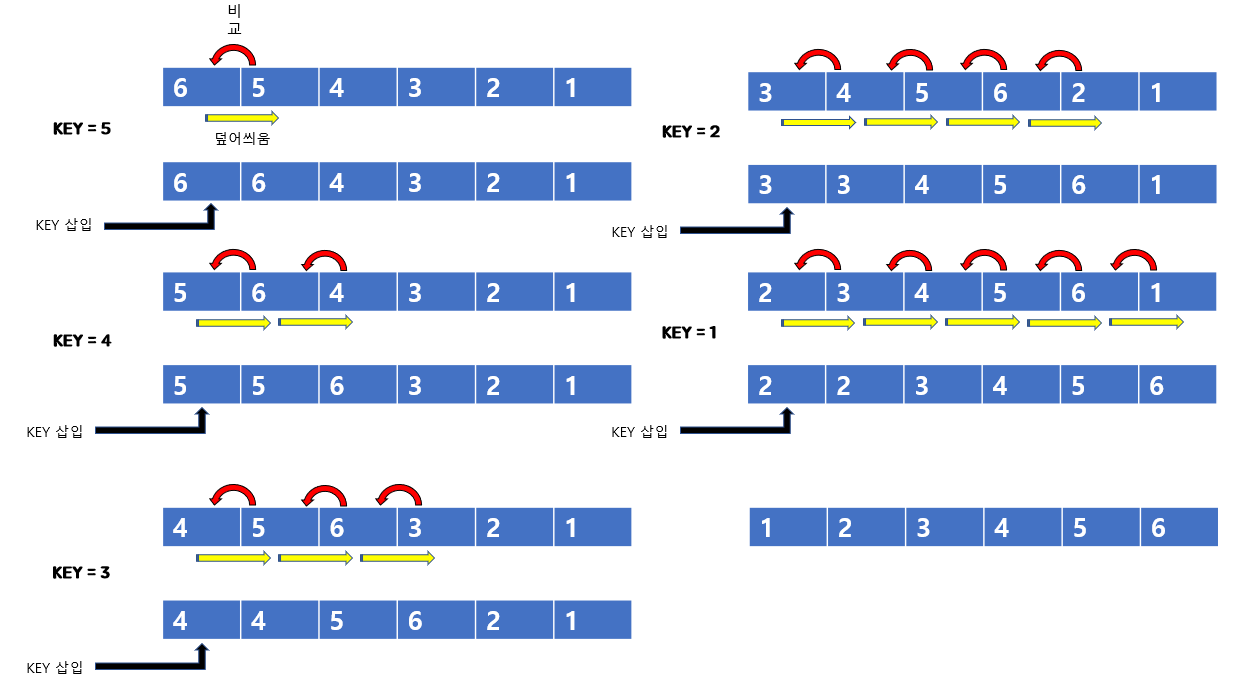


그림 1-1 삽입 정렬 수행도

* 1. **삽입 정렬의 개념**
     1. 삽입 정렬의 특성
* 손안의 카드를 정렬하는 방법과 유사
* 새로운 카드를 기존의 정렬된 카드 사이의 올바른 자리를 찾아 삽입
* 새로 삽입될 카드의 수만큼 반복하게 되면 전체 카드가 정렬
  + 1. 삽입 정렬의 동작 방식
* 1번째 데이터부터 차례대로 이미 정렬된 앞 데이터들과 비교 하여, 자신의 위치를 찾아 삽입함으로써 정렬을 완성하는 알고리즘
* 즉, 세 번째 데이터가 자신의 위치를 찾아야하는 key가 된다면 두 번째 데이터와 첫 번째 데이터와 비교함
* 만약 비교를 통해 key가 삽입될 위치를 찾았다면 그 위치에 key를 삽입하기 위해 모든 데이터를 한 칸씩 뒤로 이동시킴
  1. **삽입 정렬의 장단점**

1-2-1) 장점

* stable sort 방법이기 때문에, 두 개의 키값을 가지고 정렬을 시도할 수 있음
* 데이터의 수가 적을 경우 알고리즘 자체가 매우 간단하므로 다른 복잡한 정렬 방법보다 유리할 수 있음
* 대부분의 데이터가 이미 정렬되어 있는 경우 매우 효율적일 수 있음

1-2-2) 단점

* key가 되는 데이터를 올바른 위치에 삽입해야 하기 위해서는 앞의 모든 데이터들이 한칸씩 이동을 해야만함
* 비교적 많은 데이터들의 이동을 포함할 수 밖에 없음
* 따라서 데이터 수가 많고 데이터의 크기가 클 경우에 적합하지 않음
  1. **삽입 정렬의 best case와 worst case**

1-3-1) best case

* best case의 경우는 이미 정렬된 n개의 데이터가 인풋으로 들어오는 경우
* 모든 요소에 대하여 한번의 비교만 이루어지므로 비교는 n번 이루어짐
* 따라서 best case의 경우 시간 복잡도는 O(n)

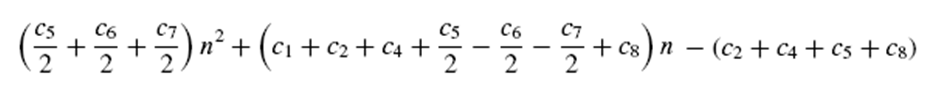
1-3-2) worst case

* worst case의 경우는 반대로 역순인 n개의 데이터가 인풋으로 들어오는 경우
* 비교가 각각 데이터에 대하여 0,1,2,......,n-1번 이루어짐
* 총 비교수를 얻어 내기 위해 각각 데이터들의 비교 횟수를 모두 더하면 총 (n^2 – n) / 2 비교가 이루어짐
* 따라서 worst case의 경우 시간 복잡도는 O(n^2)
  1. **삽입 정렬의 알고리즘과 Big-O(참고 8)**

1-4-1) 삽입 정렬의 알고리즘



1-4-2) Big-O

* 열 별로 Time과 Cost를 곱해 더하면 시간함수를 구할 수 있음
* 시간 복잡도 T(n) = 
* 따라서 삽입 정렬의 Big-O는 O(n^2)임

# 2.병합 정렬

**2-1 병합 정렬의 배경 지식**

2-1-1) Recurrence관계

* 수열에서 일반항이 그보다 앞선 항에 의해 규칙적으로 구해지면 recurrence 관계라고 함
* 즉 앞의 결과가 현재 결과에 영향을 미치는 것을 말함

2-1-2) Divide and Conquer

* 문제를 작은 2개의 문제로 분리하고 각각을 해결한 다음, 결과를 모아서 원래의 문제를 해결하는 전략
* Divide and Conquer 방식은 대개 순환 호출을 이용하여 구현

2-1-3) Divide and Conquer의 세가지 단계

* Divide : 입력 배열을 같은 크기의 2개의 부분 배열로 분할
* Conquer : 분할되어진 부분 배열을 정렬, 부분 배열의 크기가 1이 아니라면 순환 호출을 이용하여 다시 분할 정복 방법을 적용
* Combine : 정렬된 부분 배열들을 하나의 배열로 병합

**2-2 병합 정렬의 개념**

2-2-1) 병합 정렬의 특성

* 병합 정렬은 하나의 리스트를 두 개의 균등한 크기로 분할하고 분할된 부분 리스트들을 정렬 하고 합하는 방식
* 병합 정렬을 구현하기 위해서는 분할된 리스트를 담기 위한 리스트가 추가적으로 필요함
* 각 부분 배열을 정렬하는 방법은 병합 정렬을 순환적으로 호출하여 정렬
* 병합 정렬에서 실제로 정렬이 이루어지는 시점은 분할되어진 리스트를 합병하는 단계

2-2-2) 병합 정렬의 동작 방식

* 정렬되지 않은 리스트를 절반으로 잘라 비슷한 크기의 두 부분 리스트로 나눔
* 그림 2.1 Divide 과정에서 볼수 있듯이 리스트의 길이가 0 또는 1이 될때까지 재귀 호출함
* 두개의 리스트끼리 값들을 처음부터 하나씩 비교하여 두 개의 리스트의 값 중에서 더 작은 값을 새로운 리스트로 옮기며 이 과정을 하나의 리스트가 끝날때 까지 반복
* 만약 둘 중에서 하나의 리스트가 완료되었다면 나머지 리스트의 값들을 전부 새로운 리스트로 복사
* 새로운 리스트를 원래의 리스트로 옮김
* 그림 2.1 Merging 단계에서 볼수 있듯이 이 과정을 재귀적으로 반복

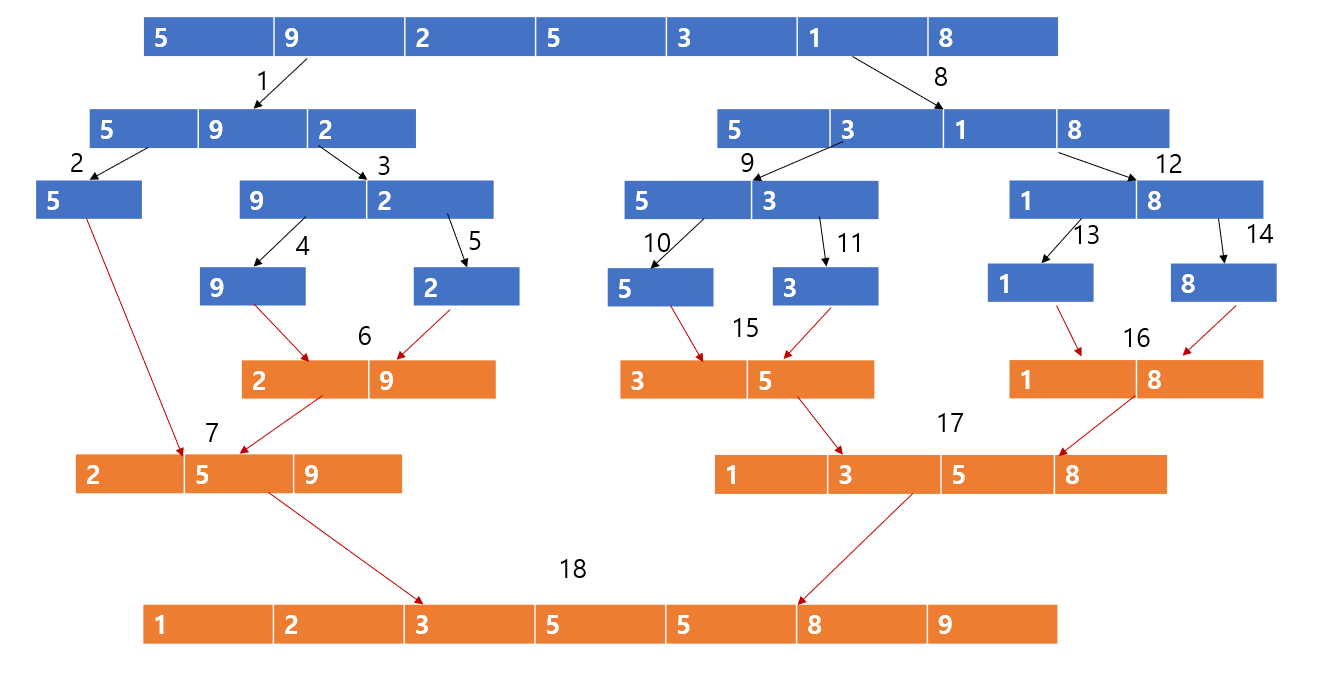


그림 2.1) 병합 정렬 수행도

**2-3 병합 정렬의 장단점**

2-3-1) 단점

* 만약 배열로 구성하게 된다면, 별도의 저장 공간이 필요함
* 임시 배열에 데이터를 반복적으로 옮겨줘야 하므로 이동 횟수가 많음
* 따라서 배열로 구성한 경우 데이터의 크기가 큰 경우에는 많은 시간이 걸림

2-2-2) 장점

* 반면에 데이터들을 연결 리스트로 구성하면, 별도 저장 공간이 필요 없음으로 데이터의 이동은 무시할 수 있을 정도로 작아짐
* 따라서 크기가 큰 데이터를 정렬할 경우에 연결 리스트를 사용한다면, 병합 정렬은 매우 효율적
* stable sort임

**2-4 병합 정렬의 알고리즘과 Big-O**

2-4-1) 병합 정렬의 알고리즘

merge\_sort(low, high)

if(low < high)

mid <- (low + high) / 2

merge\_sort(low,mid)

merge\_sort(mid +1, high)

merging(low,mid,high)

merging(A[],low, mid, high)

for (l1<-low, l2<-mid + 1, i<-low to l1 <= mid and l2<- high) {

if(A[l1] <= A[l2])

mArray[i++] <- A[l1++]

else

mArrray[i++] <- A[l2++]

while(l1 <= mid)

mArray[i++] <- A[l1++]

while(l2 <= high)

mArray[i++] <- A[l2++];

for(i <- low to high)

A[i] <- mArray[i]

2-4-2) 병합 정렬의 Recurrence relation(참고 7)

* divide 과정은 재귀적으로 데이터를 절반으로 나눔으로 T(n/2)임
* merging 과정은 데이터가 n개인 경우 O(n)임
* 따라서 병합 정렬의 Recurrence relation T(n) = T(n/2) + O(n)

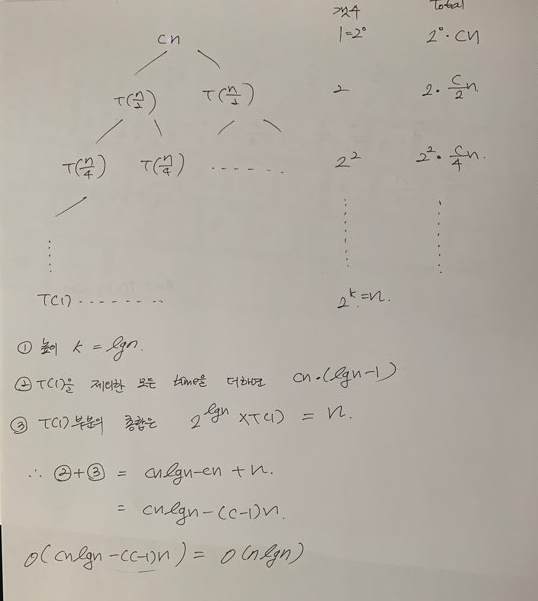


그림 2.2 병합정렬의 Recursion 트리(참고 3)

2-4-3) 병합 정렬의 Big-O(참고3)

* Mathematical Induction의 Guess 과정을 위해 Big-O를 추측해야함
* 병합 정렬은 Recurrence relation으로 나타낼 수 있으므로Recursion 트리를 이용해 추측
* Recursion 트리를 통해 추측한 병합 정렬의 Big-O는 O(nlgn)임
* Big-O의 정의에 의하여 T(n)은 (0<=T(n)<=cnlgn, for all n>n0, c>0) 을 만족해야 병합 정렬의 Big-O는 O(nlgn)이 확정
* Mathematical Induction 방법을 이용하여 증명하면 참임을 알 수 있음
* 따라서 병합 정렬의 Big-O는 O(nlgn)

# 3.힙 정렬

* 1. **배경 지식**

3-1-1) 트리3.

* 그래프의 한 종류이며, 사이클이 존재하지 않는 그래프를 트리라고 함
* 그림 3.1을 보면 트리는 하나의 루트 노드를 갖으며, 루트 노드는 0개 이상의 자식노드를 가지고 있음
* 그 자식 노드 또한 0개 이상의 자식 노드를 갖고 있고 결국에 모든 노드가 0개 이상의 자식 노드를 가지고 있음

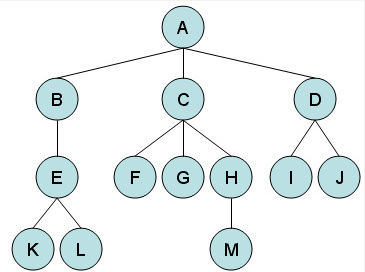


그림 3.1 트리

3-1-2) 완전 이진 트리

* 트리의 한 종류로서, 각 노드가 최대 두 개의 자식을 갖는 트리를 말함
* 그림 3.2를 보면 높이 0을 제외한 나머지 높이에는 노드들이 꽉 차있음
* 새로 들어오는 데이터들은 이진 트리로서의 성질을 유지하며 가장 낮은 레벨의 왼쪽 부터 채워나가야 함

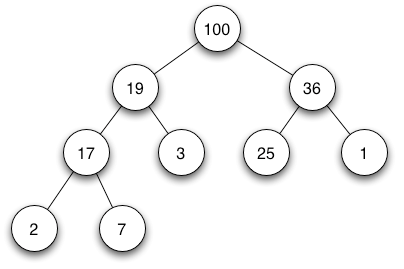


그림 3.2 완전 이진 트리

3-1-3) 힙 자료구조

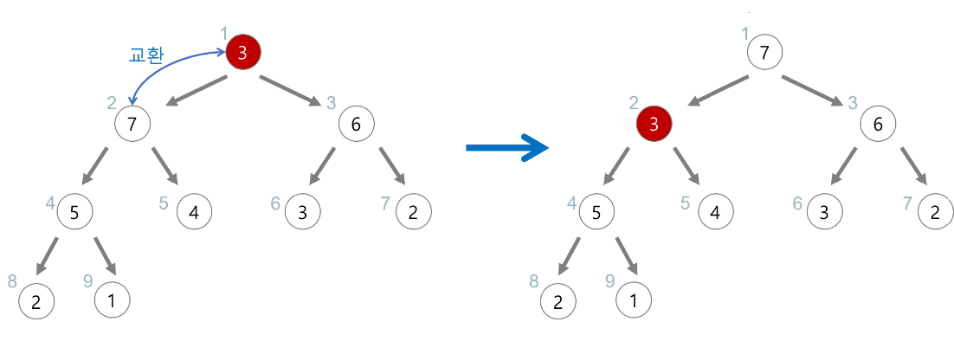
* 힙 자료구조는 그림 3.2와 같이 완전 이진 트리로 볼 수 있는 배열 객체
* 힙에는 최소 힙과 최대 힙 두 종류가 있음
* 최대 힙은 부모 노드의 값이 자식보다 크거나 같은 성질을 만족 해야하며, 최소 힙은 반대로 부모 노드의 값이 자식보다 작거나 같아야 함
* 위의 성질에 의하여 최대 힙의 경우에는 루트노드가 최대값이며 반대로 최소 힙의 경우에는 최소값
* 힙 정렬은 이러한 힙 성질을 이용해 수행하는 것
* 따라서 힙 정렬을 수행하기 전에 힙 구조를 만들어야 함

3-1-4) max heapify

* 최대 힙 특성을 유지하는 데 핵심 역할을 하는 알고리즘
* 모든 노드에 대해 수행하는 것이 아닌, 하나의 노드에 대해 수행되어짐
* 그림 3.3을 보면 이미 힙구조가 유지되어있을 때 사용하며, 특정 노드를 힙구조에 맞게 이동 시킴
* 주로 힙에서 데이터를 제거할 때 max heapify를 수행함

3-1-5) Build a max heap

* 선형 리스트인 데이터가 들어오는 경우, 완전한 힙 구조로 만들어주는 알고리즘
* 결국에는 모든 노드에 대해 Max-heapify 해주는 것임



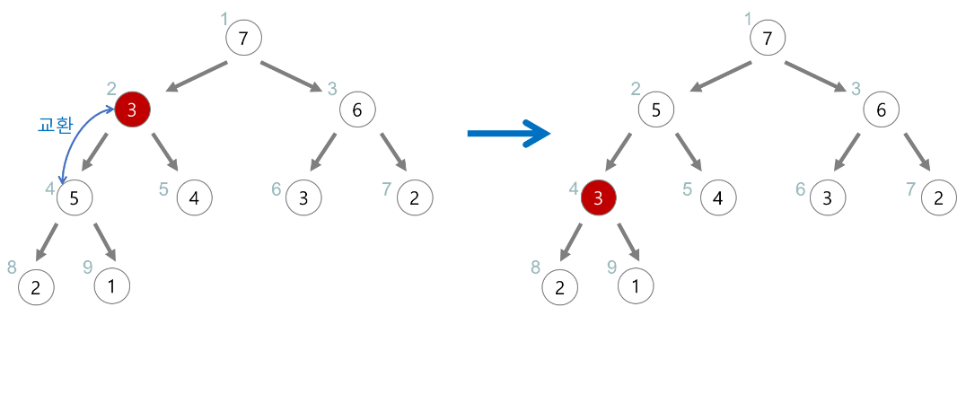


그림 3.3 max-heapify

* 1. **힙 정렬의 개념**

3-2-1) 힙 정렬의 특성

* 힙 정렬은 최대 힙이나 최소 힙을 구성해 정렬하는 방법
* 내림차순 정렬을 위해서는 최대 힙을 사용하고, 오름차순 정렬을 위해서는 최소 힙을 사용

3-2-2) 힙 정렬의 동작 방식

* 선형 리스트로 이루어진 데이터를 받으면 최대 힙 혹은 최소 힙을 구성
* 힙의 루트노드가 최대 또는 최소이므로 힙의 루트노드를 배열에 넣음
* 루트 노드가 제거 됐으므로 힙을 다시 구성한 후 또 다시 힙의 루트노드를 배열에 넣음
* 힙에 데이터가 없을 때 까지 이러한 작업을 반복
  1. **힙 정렬의 장단점**

3-3-1) 힙 정렬의 장점

* 힙 정렬은 일관된 성능을 나타냄
* 즉, best case, worst case의 경우 구분없이 똑같이 잘 수행됨

3-3-2) 힙 정렬의 단점

* 힙 정렬은 unstable sort임
* 힙 정렬을 사용하려면 데이터를 힙으로 변환 해야하는 추가적인 시간 필요

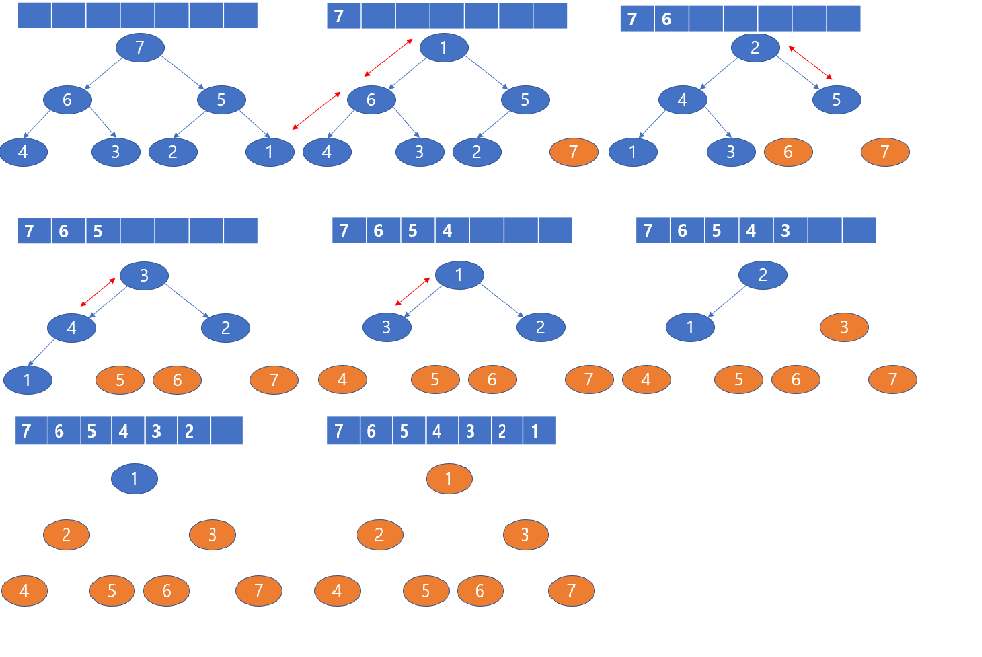


그림 3-4힙 정렬 수행도

* 1. **Max-heapify의 알고리즘과 Big-O**

3-4-1) Max-heapfiy의 worst-case

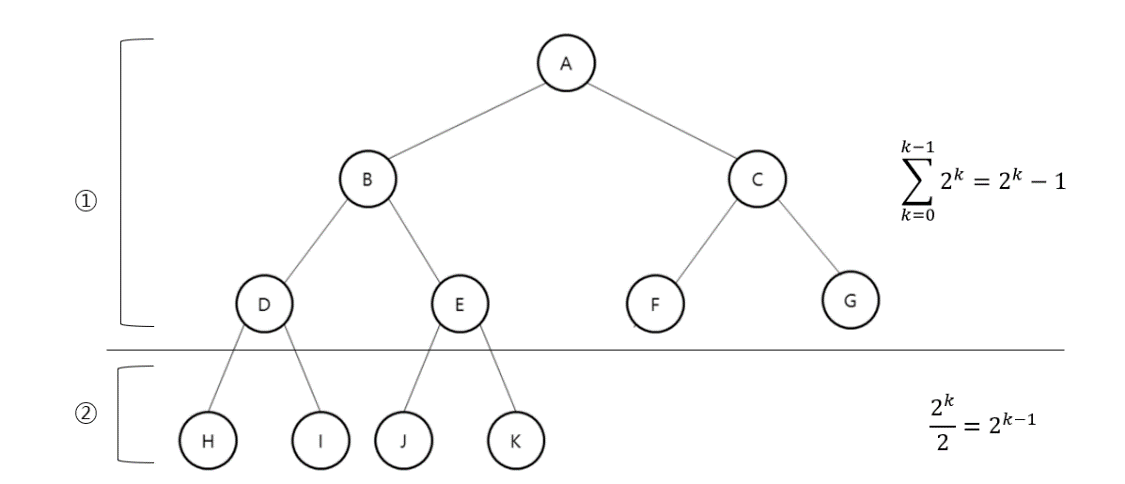


그림 3-5 Max-heapify의 worst-case

* 그림 3-5에서 A의 왼쪽에 있는 서브트리가 포화이진트리면 루트노드에서 왼쪽으로 가기만해도 worst-case임
* 만약 루트노드가 바뀔 경우, 왼쪽 서브트리가 포화이진트리가 아닌 경우에는 루트노드 기준으로 아예 왼쪽으로만 이동하는 경우 worst-case 임

3-4-2) Max-heapfiy의 알고리즘(참고1)

max\_heapify(A,i,heap-size)

l <- LEFT(i)

r <- RIGHT(i)

if l <= heap-size and A[l] > A[i]

then largest <- l

else largest <- i

if r <= heap-size and A[r] > A[largest]

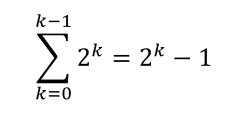
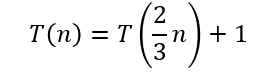
largest <- r

if largest != i

then exchange A[i] <-> A[largest]

MAX-HEAPIFY(A,largest)

3-4-3) Max-heapify 시간 복잡도(참고 1)

* 그림 3-4에서 왼쪽 서브트리의 노드수를 알아야 시간 복잡도를 계산할 수 있음
* k에 대한 전체 노드수는임
* 왼쪽 서브트리의 노드수를 k에 대하여 구해보면  임
* k에 대한 왼쪽 서브트리의 노드수 / k에 대한 전체 노드수를 계산하면 왼쪽 서브트리의 노드수는 임을 알 수 있음
* 따라서 Max-heapify의 시간 복잡도는  임

3-4-4) Max-heapify Big-O

* 그림 2-2와 같이 Recursion Tree를 이용하여 추측하면 Max-heapify의 Big-O는 임을 알 수 있음
* Big-O의 정의에 의하여 T(n)은 (0<=T(n)<=clgn, for all n>n0, c>0) 을 만족해야 Max-heapify의 Big-O는 이 확정
* Mathematical Induction 방법을 이용하여 증명하면 참임을 알 수 있음
* 따라서 병합 정렬의 Big-O는 
  1. **Build a max heap**

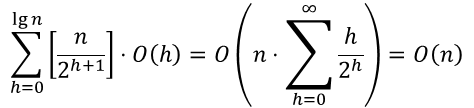
3-5-1) Build a max heap의 알고리즘(참고1)

build\_max\_heap(A,heapsize)

for (i <- heapsize/2 down to 1

do Max-heapify(A,i,heapsize)

3-5-2) Build a max heap의 Big-O(참고 1)

* 원래라면 Max-heapfiy를 n번 반복하므로 임
* max-heapify의 수행시간이 높이에 따라 다를 수 밖에 없기 때문에 개선할 여지가 있음
* 따라서 높이에 따라 수행시간을 달리해 계산하면 build a max heap의 Big-O는 임

3-5-3) 노드 수가 인 이유

* 단말 노드의 수는 개임
* 높이를 기준으로 노드의 수를 보면 한 층씩 올라갈 때 마다 노드 수가 씩 줄어드는 것을 알 수 있음
* 이에 대해 일반화를 해보면 임
  1. **Heap sort의 알고리즘과 Big-O**

3-6-1) Heap sort의 알고리즘(참고1)

heap\_sort(heapsize)

build\_max\_heap(A,heapsize)

for(i<-heapsize downto 1)

exchange( A[0],A[j] )

heapsize <- heapsize-1

Max\_heapify(A,i,heapsize)

3-6-2) Heap sort의 Big-O

* for loop이 n번 반복되며, 그 안의 명령어인 exchange인 경우 상수시간 그리고 Max\_heapify는 임
* 따라서 heap\_sort의 Big-O는 

# 4.퀵 정렬

**4-1 퀵 정렬의 개념**

4-1-1) 퀵 정렬의 특성

* 하나의 리스트를 피벗을 기준으로 두 개의 비균등한 크기로 분할하고 분할된 부분 리스트를 정렬한 다음, 두 개의 정렬된 부분 리스트를 합하여 전체가 정렬된 리스트가 되게 하는 방법
* 분할 정복 알고리즘의 하나로, 평균적으로 매우 빠른 수행 속도를 보임

4-1-2) 퀵 정렬의 동작 방식

* 리스트 안에 있는 한 요소를 피벗으로 선택함
* 피벗을 기준으로 피벗보다 작은 요소들은 피벗의 왼쪽으로 피벗보다 큰 요소들은 피벗의 오른쪽으로 옮겨짐
* 피벗을 제외한 왼쪽 리스트와 오른쪽 리스트를 정렬함
* 위의 과정을 재귀 호출을 이용하여 리스트의 크기가 0 또는 1이 될 때까지 반복

**4-2 퀵 정렬의 장단점**

4-2-1) 퀵 정렬의 장점

* 한 번 결정된 피벗들이 추후 연산에서 제외되므로 같은 시간 복잡도를 가지는 정렬중에서 가장 빠름
* 병합 정렬과 공통적인 접근 방식이지만, 병합 정렬과 다르게 추가적인 저장공간을 쓰지 않음

4-2-2) 퀵 정렬의 단점

* 만약 그림 4-1처럼 불균형 분할이 일어날 경우 비효율적
* 특히 이미 정렬된 리스트나 역순인 리스트에 대해서 과도한 수행시간이 걸림

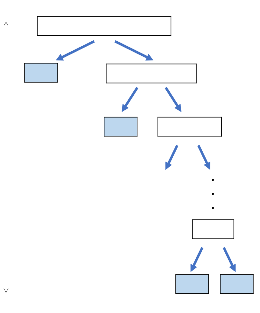


그림 4-1 불균형 분할

**4-3 퀵 정렬의 알고리즘과 Big-O**

4-3-1) 퀵 정렬 알고리즘(참고2)

partition(A,p,r)

x = A[r]

i = p-1

for j<-p to r-1

if A[j] <= x

i <- i+1

exchange A[i] <-> A[j]

exchange A[i+1] <-> A[r]

return i+1

quick\_sort(A,p,r)

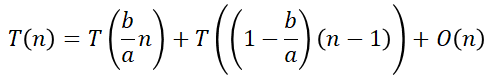
if p<r

q <- partition(A,p,r)

quick\_sort(A,p,q-1)

quick\_sort(A,q+1,r)

4-3-2) average case일 때 Recurrence relation

* 정렬된 데이터나 역순인 데이터가 아닌 무작위 데이터가 들어오는 경우 평균적인 성능을 보여줌
* partition의 경우 p부터 r-1에 대해 for loop가 진행됨으로 O(n)
* pivot의 양쪽으로 나뉘어 재귀 호출이 일어남으로 따라서

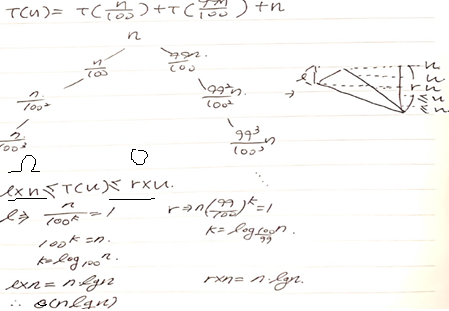
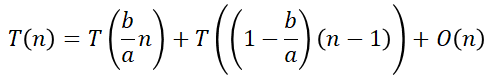


그림 4-2 average case Recursion 트리

4-3-3) average case 일 때 Big-O

*  에 대하여 Recursion 트리를 그리게 되면 b/a가 어떤 값이 되든 같은 Big-O가 나올 수 밖에 없음
*  로그 정의에 의해서

 이므로

임

따라서 average case 일 때 그림 4-2에서 볼수 있듯이 Big-O는 

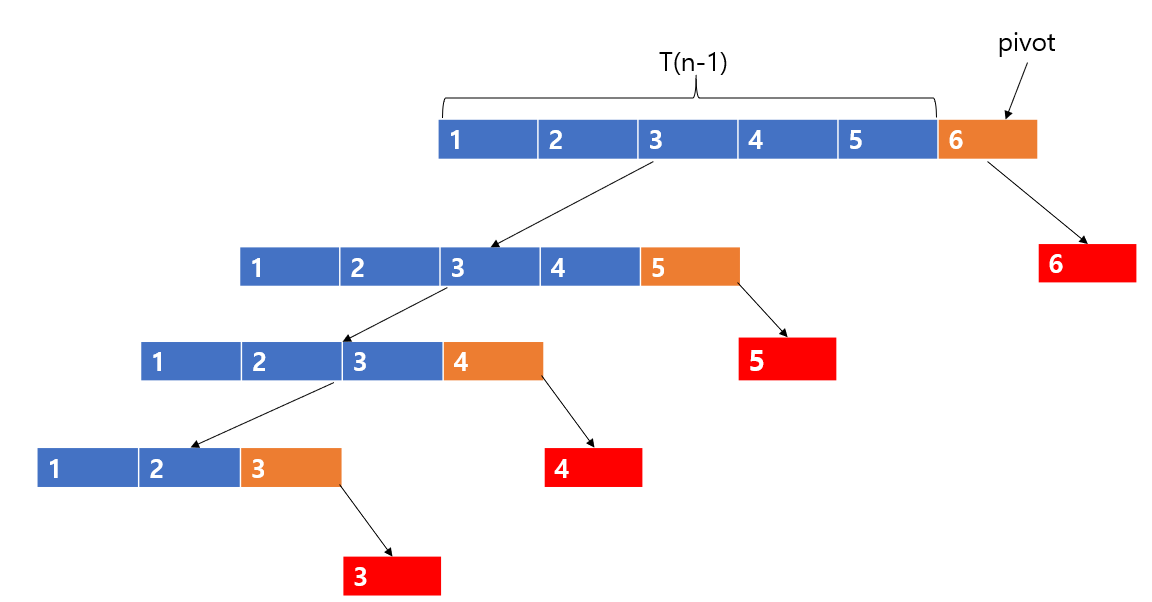


그림 4-3 worst case인 경우

4-3-4) worst case일 때 Recurrence relation

* 그림 4-5처럼 데이터가 이미 정렬된 채로 들어오거나, 아예 역순으로 들어오는 경우 최악의 성능을 보여줌
* 즉, pivot을 첫번째 혹은 마지막 값으로 고를 경우 worst case
* 맨뒤를 pivot으로 고를 경우, 앞부분은 T(n-1)이며 pivot은 정렬할 데이터에서 제외됨으로 T(0)
* 따라서 

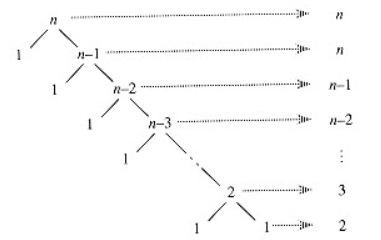
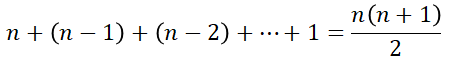
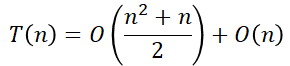


그림 4-4 worst case Recursion 트리(참고 5)

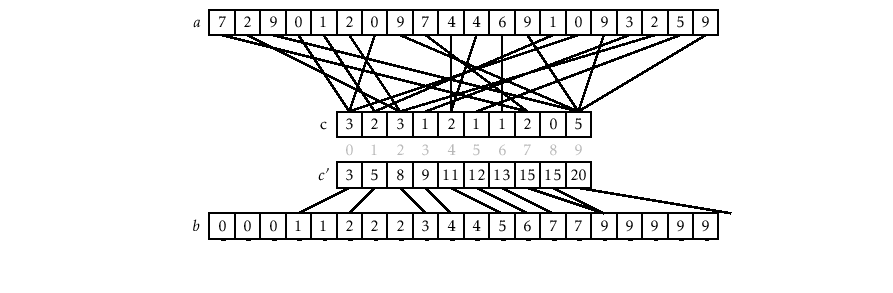
4-3-5) worst case 일 때 Big-O

* 그림 4-4는 위의 Recurrence relation에 대한 Big-O를 추측하기 위한 트리임
* 이므로 

따라서 worst case 일때 Big-O는 

# 5.기수 정렬과 COUNTING SORT

**5-1 Counting sort**

****

5-1 그림 Counting sort(참고 4)

5-1-1) Counting sort의 특성

* 정렬 과정에서 비교가 없는 정렬법 중에 하나
* Counting sort는 데이터의 최댓값을 알고 있어야 사용할 수 있음
* Counting sort의 중요한 특성은 stable sort라는 점
* 따라서 자릿수를 이용하고, stable한 특성이 중요한 기수 정렬에서 주로 사용

5-1-2) Counting sort의 동작 방식

* 그림 5-1에서 보면0부터 요소의 최댓값인 9까지 인덱스를 가지는 배열을 생성
* 각각의 요소들의 갯수를 계산해 해당 인덱스에 저장시킴
* 위의 과정이 완료되면 각각의 요소들을 누적값으로 갱신
* 그 누적값이 해당 요소가 들어갈 index가 됨

5-1-3) Counting sort의 비효율적인 경우

* Counting sort의 경우 0부터 요소의 최댓값까지 인덱스를 가지는 배열을 생성해야함
* 데이터보다 많은 배열의 모든 인덱스에 대해 0으로 초기화 해야함
* 따라서 이러한 경우 매우 비효율적인 정렬

5-1-4) Counting sort의 알고리즘

Counting sort (A,B,K)

1. for i<- 0 to K
2. do C[i] <- 0
3. for j<-1 to length[A]
4. do C[A[j]] <- C[A[j]] + 1
5. for i<-1 to k
6. do C[i] <- C[i] + C[i-1]
7. for j<- length[A] down to 1
8. do B[C[A[j]]] <- A[j]

9. C[A[j]] <- C[A[j]] – 1

5-1-5) Counting sort의 Big-O

* 1열은 k, 즉 배열의 모든 값들을 0으로 초기화 해줌으로 O(k)
* 3열은 N, 즉 각각의 데이터의 갯수를 세는 것이므로 O(n)
* 5열은 마찬가지로 배열의 모든 값들을 누적값으로 갱신해주는 것이므로 O(k)
* 7열은 3열과 마찬가지로 O(n)
* 따라서 Counting sort의 Big-O는 n 또는 k 즉, O(n+k)

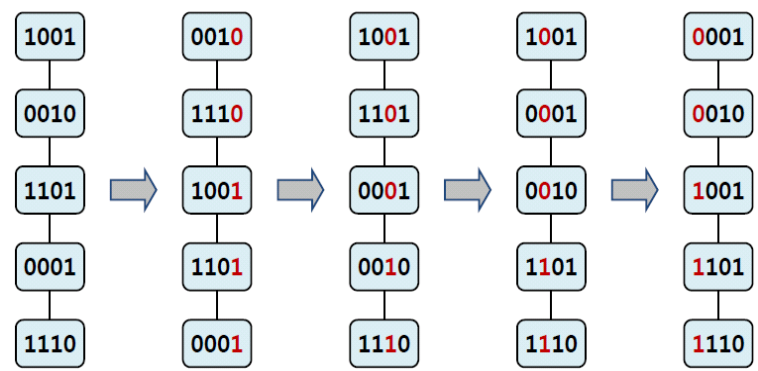
****

그림 5-2 기수 정렬 수행도(참고6)

**5-2 기수 정렬의 개념**

5-2-1) 기수 정렬의 특성

* 순서를 정할 수 있는 데이터가 들어오는 경우 자릿수 별로 정렬
* 기수 정렬은 가장 큰 자릿수부터 시작하는 (MSD)와 가장 작은 자릿수부터 시작하는 (LSD)로 구분되어짐

5-2-2) Counting sort를 주로 사용하는 이유

* 기수 정렬은 자릿수 별로 정렬할 때 다양한 정렬을 사용할 수 있음
* 하지만 자릿수가 k이고 데이터 수가 n이라고 하면 보통 n>k
* 그리고 기수 정렬은 자릿수 별로 정렬한 데이터가 다시 뒤섞이면 안되기 떄문에

꼭 stable한 특성을 가져야함

* 따라서 기수 정렬에서 counting sort가 가장 효율적

5-2-3) 기수 정렬의 동작 방식

* 데이터의 최대 사이즈나 길이를 알고 있지 못한다면 데이터들을 일일히 비교해 최대 길이를 알아냄
* LSD 또는 BSD 방법으로 해당 자릿수에 대해 정렬을 수행시킴
* 하나의 자릿수에 대해 완료되면 다음 자릿수에 대해 또다시 정렬을 수행
* 모든 자릿수에 대해 완료되면 종료

**5-3 기수 정렬의 제한 사항**

5-3-1) 기수 정렬의 정렬 전 조건

* 자릿수를 이용해 정렬하는 방식 때문에 특수한 비교 연산이 필요한 데이터에는 적용할 수 없음
* 데이터의 사이즈나 길이를 알지 못하면 데이터의 사이즈나 길이를 구하기 위한 추가 시간이 소요됨
* 여분의 저장 공간이 더 필요함

**5-4 기수 정렬의 알고리즘과 Big-O**

5-4-1) 기수 정렬의 알고리즘

radix\_sort(A,d)

1. for i<-1 to d
2. do counting\_sort(A) on digit i

5-4-2) 기수 정렬의 Big-O

* 1열은 자릿수 d 만큼 반복됨으로 O(d)
* counting sort의 시간 복잡도는 O(n+k)임
* 따라서 자릿수 d 만큼 counting sort가 반복됨으로 기수 정렬의 Big-O는

O(dn+dk)

# 정렬별 수행시간 분석

1. 삽입 정렬

* 다섯가지 정렬 중에 성능은 가장 최악임
* 하지만 데이터 수에 따른 수행 시간 감소는 다섯가지 정렬중에 압도적으로 높았음
* 결과적으로 데이터가 적으면 어느정도 효율이 있는 정렬이지만 데이터가 일정이상 많아지면 최악의 성능을 보여줌
* stable sort임

1. 병합 정렬

* 비교 정렬 중에서 가장 최적의 성능을 보여줌
* stable sort임

1. 힙 정렬

* 비교 정렬 중에서 평균적인 성능을 보여줌
* 정렬 수행중에 heap을 만들어주는 작업이 필요하기에 추가적인 시간 대략, 4~6clocks 정도의 시간이 소요
* 전체 수행 시간에 비해 매우 작은 수치이므로 큰 영향은 없음
* 평균적인 성능을 보여줬지만 unstable sort이기 때문에 두 개의 키를 이용한 정렬은 성공적이지 못함
* 그림 1에서 볼수 있듯이 나중에 수행한 년도 정렬은 잘 되어있지만 앞에서 수행한 제목의 순서가 무작위로 섞어있음



그림 1 unstable sort

1. 퀵 정렬

* 삽입 정렬을 제외한 비교 정렬 중에서 가장 안좋은 성능을 보여줌
* 하지만 데이터의 수가 줄어들수록 다른 정렬들과 비슷한 성능을 보여주는 것을 알 수 있음
* 힙 정렬과 같이 unstable sort임
* 앞에서 조사한 바에 의하면 비교 정렬 중에 가장 성능이 좋아야할 정렬이 퀵 정렬이지만 결과는 가장 안좋았음
* 앞에서 조사한 바에 의하면 퀵 정렬은 average case와 best case와의 수행 시간이 극과 극임
* 따라서 추가적으로 데이터 타입 별로 정렬 수행시간 비교

1. 기수 정렬

* 데이터에 특수한 언어들이 존재하기 때문에 문자열 데이터인 제목에 대해서는 정렬을 수행하지 못함
* 따라서 기수 정렬은 년도에 대해서만 정렬 후 측정
* 키 값의 가장 긴 자릿수를 알아내야 하기 때문에 어느정도 추가적인 시간이 필요함
* 하지만 기수 정렬의 Big-O는 O(dn+dk) 이므로 긴 자릿수를 알아내기 위한 비교는 O(n) 이기 때문에 큰 차이가 없음

1. 데이터 타입에 따른 정렬 수행시간 비교

* 퀵 정렬을 제외하고 나머지 정렬들은 각각 타이틀과 년도를 정렬할 때 소요된 시간이 큰 차이가 없었음
* 하지만 퀵정렬의 경우 타이틀에 대해 정렬시 정렬중에 가장 빨랐지만 년도에 대해 정렬시 삽입 정렬을 제외한 다른 정렬들에 비해 훨씬 많은 시간이 소요
* 이 결과를 통해 퀵 정렬은 수행속도가 빠른건 사실이지만 데이터에 영향을 많이 받는 것을 알 수 있음

# 결론

□ 정렬 별로 수행시간에는 분명히 어느정도 차이가 존재

* 데이터가 작으면 거의 차이가 나지 않음
* 하지만 데이터가 커지면 커질수록 그 차이가 점점 커지는 것을 확인할 수 있음
* 모든 데이터가 해당 정렬에 무조건 적합하지 않음
* 모든 데이터가 정렬들의 averge case를 따르지는 않음
* 데이터에 따라 같은 정렬의 성능이 상당히 달라질 수 있음
* 따라서 개발자는 어느정도 데이터에 대해 파악을 하고 있어야 적합한 정렬을 사용할 수 있음
* stable sort와 unstable sort
* 삽입 정렬과 병합 정렬과 기수 정렬을 제외하고는 나머지는 모두 unstable sort임
* 측정 상황과 같이 두 가지 키를 이용해 정렬을 시도할 때는 힙 정렬과 퀵 정렬이 성능이 좋다해도 사용할 수 없음
* 따라서 stable한 정렬과 unstable한 정렬을 구분할 줄 알아야하며 상황에 맞게 사용해야함
* 본인 의견
* 당연히 퀵 정렬이 최고의 성능을 보여줄거라 예상했음
* 하지만 예상 외로 병합 정렬이 최고의 성능을 보여줌
* 그러므로 데이터에 따라서 효율적인 정렬을 파악해 사용해야함
* 따라서 지금 사용하는 데이터에 대해서는 병합 정렬이 가장 효율적임

# 참고 문헌

1. **서적**

* 참고 1

Thomas H. Cormen, Charles E, Leiserson, Ronald L.Rivest, Clifford Stein,INTRODUCTION TO ALGORITHMS,2014

P 153~P164 힙 정렬

* 참고 2

Thomas H. Cormen, Charles E, Leiserson, Ronald L.Rivest, Clifford Stein,INTRODUCTION TO ALGORITHMS,2014

P 172~P176 퀵 정렬

* 참고 3

Thomas H. Cormen, Charles E, Leiserson, Ronald L.Rivest, Clifford Stein,INTRODUCTION TO ALGORITHMS,2014

P 89~P94 점화식을 풀기 위한 재귀 트리 방법

1. **웹사이트**

* 참고 4

opendatastructures,<http://opendatastructures.org/odscpp/11_2_Counting_Sort_Radix_So.html> (2018.11.28)

Fiqure 11.7 : Counting Sort 수행도 이미지

* 참고 5

stackoverflow, <https://stackoverflow.com/questions/35416760/worst-case-scenario-quicksort-binary-tree> (2018.11.28)

Quick sort worst case Recursion tree 이미지

* 참고 6

Ji-Dum,http://www.jidum.com/jidums/view.do?jidumId=523(2018,12,2)

Radix sort 수행도 이미지

1. **그 외**

* 참고 7

Networking Research Lab,알고리즘 강의 자료

p 22 Analyzing Divide-And-Conquer Algorithms

* 참고 8
* Networking Research Lab,알고리즘 강의 자료

p 9 ~ p11 Analyzing Insertion sort Algorithm