

알고리즘이란?

문제를 해결하려고 하는데, 어떻게 해결하면 좋을까를 고민
그래서 그 해결과정을 단계적으로, step by step으로 진행하는 방법을 안내.

직관적으로 설명하면, 입력에서 출력까지의 중간 과정을 알고리즘 이라고 한다.
그 중간에 정렬, 재귀, 검색 등의 알고리즘이 있다.
code로는 for문, 비교.

입력 = value, index

출력 = 어디에 있는지, index

어떻게 찾아야 빨리 찾을 수 있을까?

[시간, 비용]을 고려한 알고리즘 비교

비교도 컴퓨터 입장에서 하나의 연산이다.

1. Search

검색 알고리즘 = 내가 원하는 데이터를 찾는 방법

- 종류는?

1. linear Search

선형검색, 순차검색, Sequential Search, 차례대로 검색한다는 뜻

출력 : 위치 정보(index)

2. Binary Search

이진검색

출력 : 위치 정보(index)

3. Hash

출력 : 내가 원하는 데이터를 찾는다.

2. Linear Search

```
def search(array, key):                # list, key = 찾으려는 값

    for i in range(array):              # 리스트에서 for
        if list1[i] == key:             # 찾으려는 값이 리스트 안의 값과 같다면
            return i                   # 그 값의 리스트 index 반환

array1 = ['a', 'b', 'c', 'd']
key = 'c'

print(search(array1, key))
```

실행횟수 몇 번 등이 중요한 게 아니라,
코드에서 무엇이 중요한가 위주로 보자.

위의 코드에서 가장 중요한 부분, 가장 힘을 들인 곳은?

[list1[i] == key:]

이렇게 Sequential Search는 compare, 비교하는 것에 가장 큰 힘을 들이고 있다..

+복잡도

그 결과 값을 구하는데까지

최소로 든 시간 복잡도 = $O(1)$, 한 번에 찾는 것

최악으로 든 시간 복잡도 = $O(n)$, n번째, 맨 마지막에 찾는 것.

즉, linear search의 최악의 경우 시간 복잡도 = $O(n)$

3. Complexity, 복잡도

1. Time Complexity(시간 복잡도) :

실행하는데 필요한 시간 (처리 시간)

얼마나 연산을 적게 하는가. 시간 효율성

2. space complexity(공간 복잡도) :

메모리 (기억 공간)와 파일 공간

컴퓨터 메모리를 얼마나 적게 먹는가.

- 위의 복잡도가 낮은 알고리즘이 좋은 알고리즘이다.
- 어떤 과정을 처리하는데 처리 시간이 짧을수록, 컴퓨터의 메모리를 적게 사용할수록 좋은 알고리즘

- Big-O notation

- 알고리즘의 Time Complexity는 주로 Big-O notation을 사용하여 나타낸다.
- 어떠한 문제를 해결 하기 위한 방법은 다양하기 때문에 **방법(알고리즘) 간에 효율성을 비교**하기 위해 **빅오(big-O) 표기법**을 보통 가장 많이 사용한다.
- 빅-오 표기법 특징

◦ $O(2N) \rightarrow O(N)$

◦ $O(N^2+2N+1) \rightarrow O(N^2)$

= 힘이 가장 썩 애만 남고 나머지는 무시

(참고로, n이 매우 큰 숫자라는 것을 가정한다.)

ex-1.

그러면 이 2가지 경우 중 어떤 게 시간 복잡도가 클까?

빅오표기법으로 보면

$$f(n) = n^2 + 2n + 3$$

$$g(n) = n^2$$

이렇게 둘 다 똑같다.

$$O(n^2)$$

그렇기 때문에 빅오표기법은 가장 큰 차수가 뭔지만 비교하면 된다.

ex-2.

이 두 가지도 동일하다. 가장 큰 차수의 계수도 신경 안 쓴다.

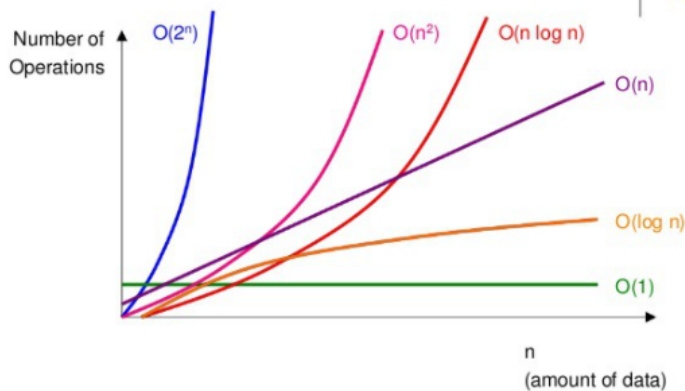
$$f(n) = 2n^2 - > O(n^2)$$

$$g(n) = n^2 - > O(n^2)$$

- 시간복잡도가 크다 = 차수(order)가 크다.
= 비효율적이다.

- 알고리즘의 성능평가 : 최선, 최악, 평균 유형(best, worst, average case)
그 중 **최악의 경우**로 알고리즘 성능을 파악한다.

Comparing Big O Functions



(C) 2010 Thomas J. Cortina, Carnegie Mellon University

- 따라서, 가장 효율 좋은 건 초록색 $O(1)$
- 가장 효율이 안 좋은 건 $O(2^n)$
 - linear search의 시간 복잡도 = $O(n)$
 - binary search의 시간 복잡도 = $O(\log n)$

faster $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$ slower

(상수함수 < 로그함수 < 선형함수 < 다항함수 < 지수함수)

☞ 중요!

이건 sort에서 복잡해질 거기 때문에 잘 알아둬야 함.

1. $O(1)$: 스택에서 Push, Pop
2. $O(\log n)$: 이진트리
3. $O(n)$: for 문
4. $O(n \log n)$: 퀵 정렬(quick sort), 병합정렬(merge sort), 힙 정렬(heap Sort) Sort ★
5. $O(n^2)$: 이중 for 문, 삽입정렬(insertion sort), 거품정렬(bubble sort), 선택정렬(selection sort)
6. $O(2^n)$: 피보나치 수열

4. Binary Search

- linear search의 최악의 시간 복잡도를 대신할 효율적인 알고리즘
- 이진검색, sequence형을 둘로 나눠서 검색하는 알고리즘

- 꼭 정렬(sort)가 되어 있어야 한다.

ex) list2 = [11, 12, 13, 14, 15, **16**, 17, 18, 19, 20, 21]

가운데 index를 계산하는 방법 = (맨 왼쪽의 index + 맨 오른쪽의 index) / 2)

cf. 홀수개일 땐 가운데가 2개. 그 중 왼쪽 index부터 체크.

- 이 때, 최소의 경우 = 1회.
- 최악의 경우는? 4번

$$\log_2 n$$

binary search는 반을 계속 날리는 방식이라 위의 식과 같다.

증명 참조: <https://jwoop.tistory.com/9>

•

```
def BinarySearch(array, key, low, high)
    if low > high:      # 예외 처리
        return false # 맨 끝까지 찾았는데도 못 찾은 경우

    mid = (low+high) / 2 # 나눈 몫.

    if value < array[mid]: # 찾는 값이 가운데 기준 왼쪽
        return binarySearch(array, value, low, mid-1)
    elif array[mid] < value: # 찾는 값이 가운데 기준 오른쪽
        return binarySearch(array, value, mid+1, high)
    else:
        return mid
```

- 이 함수의 특징: 재귀함수,
계속해서 구간을 다르게 하면서 자기 함수를 콜한다.

binary search는 재귀함수로 이루어져 있다.

후에 재귀 알고리즘 나올 거라 기억하자.

- Binary Search의 최악의 경우 Time Complexity
: $O(n) = \log(n)$

5. Compare

Binary vs Linear

- 비교의 입장에서 보면 최소는 동일.
- 그러나 최악의 경우에는 binary가 유리
- 기본적인 차이점은,

Linear search : 정렬이 되든, 안되든 상관이 없다. 한 쪽에서부터 **순차적**으로 같은지만 확인한다.

Binary search : 필수적으로 **정렬**이 되어 있는 상태이어야 한다.

6. Hash

linear & binary search는 index 검색이었다.

여기서 추가로 데이터의 검색 뿐만 아니라, 추가, 삭제 등에도 용이한 알고리즘이 해시법이다.

ex) git commit -m "blahblah"

하고 나서 git log 에 나오는 노란색 글씨들
이게 다 hash다.

- hash table

□□□□□□□□□□□□□□

이 칸 안에 data가 들어가 있는 상태.

- hashing 종류

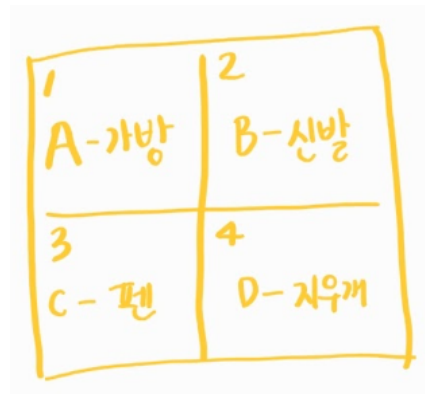
- chaining
- open hashing

- hashing 방법

- 앞에선 value를 주고 key를 찾아왔다.
- 여기선 key를 주고 value를 찾아올 거다.

ex) git switch 48c7... 여기서 [48c7] 이 값이 key가 되는 거다. 이 key가 있어야 사물함을 열 수 있다.

- 가방 갖다줘 : 못 갖다줘, key가 없어서
key(1) 주면서 물건 꺼내줘 : 갖고 올 수 있어.



ex) value : 인증서, key : 열쇠

내 인증서를 갖고 오고 싶은데 개인정보 주면서 갖고 오라고 하는 게 아니라 key 주면서 갖고 오게 하는 것.

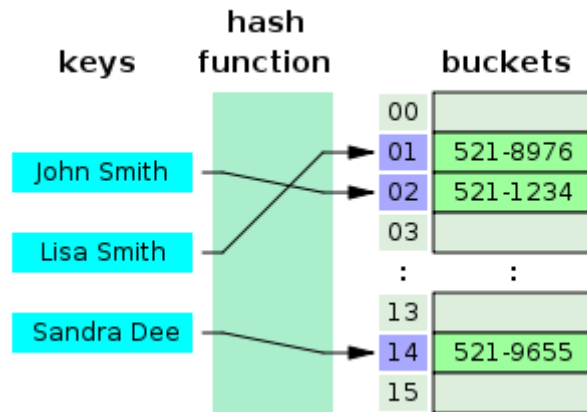
cf. key 값은 겹치지 않음.

git log처럼, 아파트 열쇠 처럼.

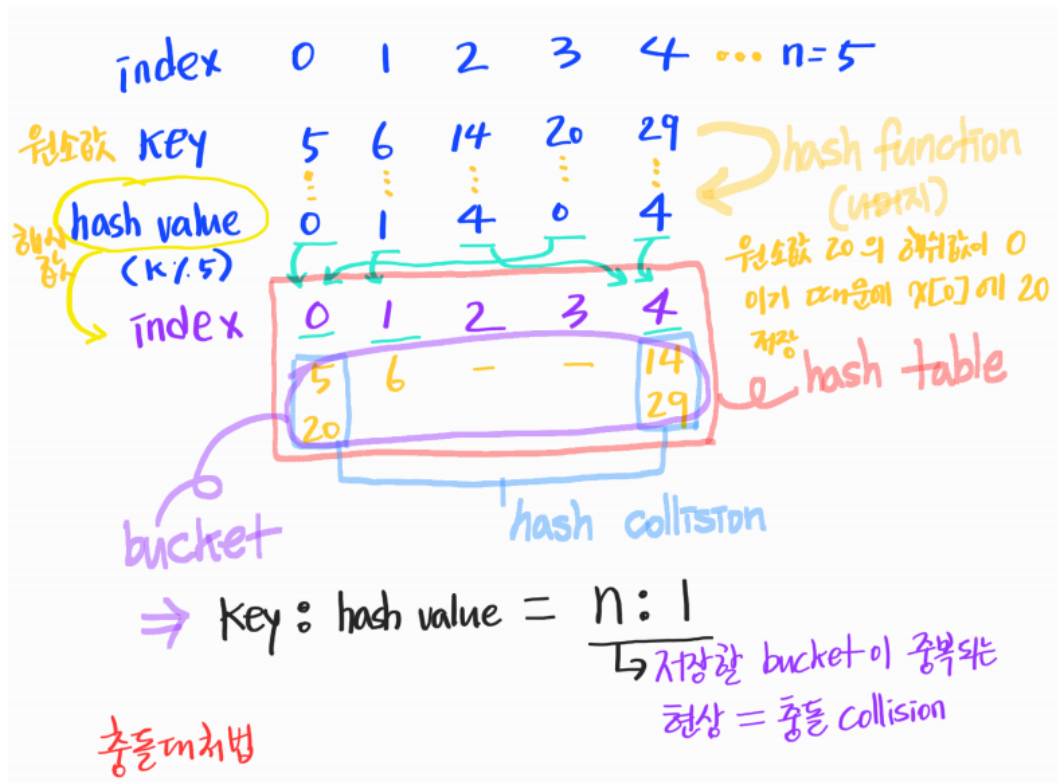
단, hash의 key는 다 int형 해쉬값이다.

- hash의 특징

- 해싱된 키(Hash Key)를 가지고 배열의 인덱스로 사용하기 때문에 **삽입, 삭제, 검색이 매우 빠르다**.
- 해시 함수(Hash Function)를 사용하는데 **추가적인 연산이 필요하다**.
- 해시 테이블(Hash Table)의 크기가 유한적이고 해시 함수(Hash Function)의 특성상 **해시 충돌(Hash Collision)이 발생할 수 밖에 없다**.
- 충돌이 없거나 적으면 $O(1)$ 의 상수 시간에 가까워지고, 충돌이 발생하면 할수록 성능은 점점 $O(n)$ 에 가까워진다. (극단적 경우)
- int 값들이 들어있는 key의 해시 함수는 **나누기** 함수가 대표적이다.

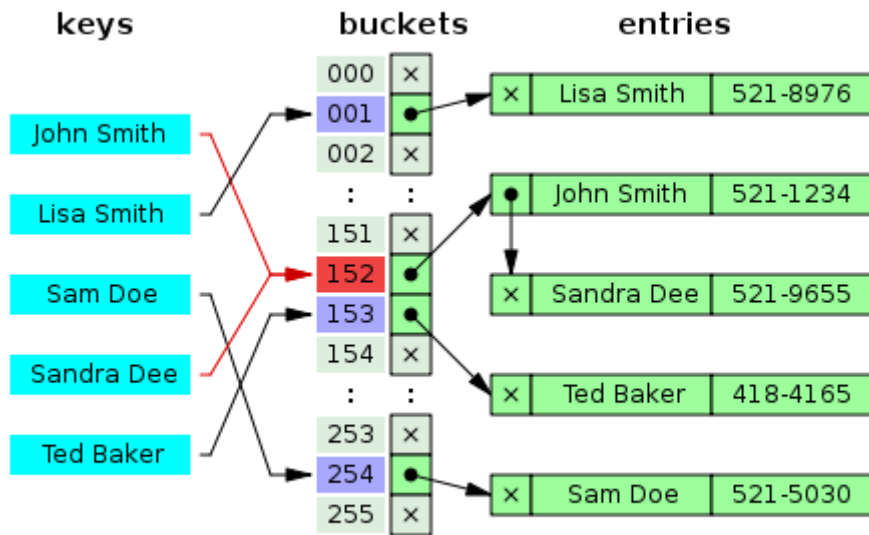


구조로 데이터를 저장하면 Key값으로 데이터를 찾을 때 해시 함수를 1번만 수행하면 되므로 매우 빠르게 데이터를 저장/삭제/조회할 수 있다. 해시테이블의 평균 시간복잡도는 $O(1)$ 이다.



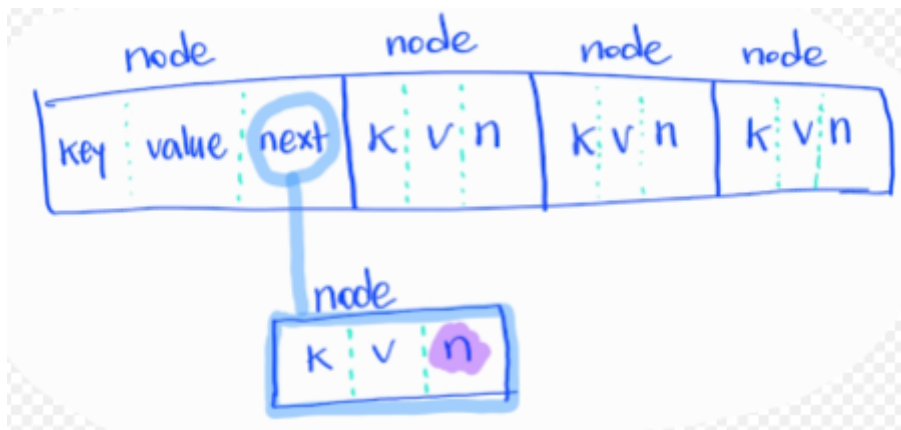
- $n=5$
key 데이터 형태가 int일 때, 대부분 배열의 크기(capacity)로 나눈 나머지를 hash value=new index로 삼는다.
- capacity(용량) = table 크기 = 원소 갯수 = 배열 크기
- 충돌대처법
 1. 체인법
 2. 오픈주소법

Chaning



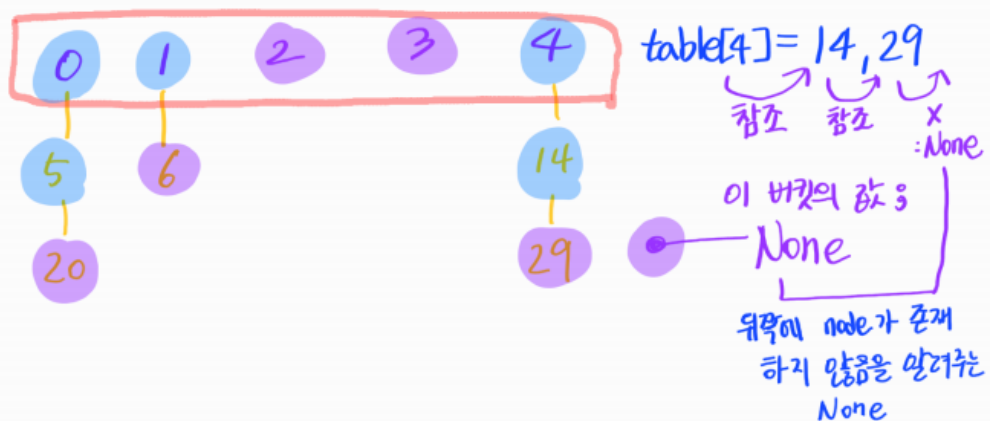
[key-value-next]

- key와 value의 타입은 any
: 숫자, 문자열 다 가능
- next : 뒤쪽 node를 참조 / node가 중요



① Chaining

해시값이 같은 데이터를 chain으로 연결리스트로 연결



- linked list 형태가 된다.

```
def hash_value(self, key: Any) -> int:
    # """해시값을 구함"""
    if isinstance(key, int):
        return key % self.capacity
    return(int(hashlib.sha256(str(key).encode()).hexdigest(), 16) %
self.capacity)
```

```
int(hashlib.sha256(str(key).encode()).hexdigest(), 16)
```

hashlib 통해서 string, int, float 형태가 될 수 있는 원소를 int형인 key 값으로 받겠다. 이 정도로만 이해하고 넘어가도 될 듯하다.

```
# value를 넣을 필요가 없다. 즉,해시에서 데이터를 찾을 때에는 'key'만을 이용해서 찾는다.
def search(self, key: Any) -> Any:
    """키가 key인 원소를 검색하여 값을 반환"""
    hash = self.hash_value_int(key) # 검색하는 키의 해시값을 구한다.
    # hash = self.hash_value(key\
    p = self.table[hash] # 노드를 노드

    while p is not None: # 다음 노드(값)이 있으면 다음 계속 탐색
        if p.key == key: # p의 key와
            return p.value # 검색 성공
        p = p.next # 뒤쪽 노드를 주목

    return None # 검색 실패
```

```
def search(self, key: Any) -> Any:
```

찾을 때는 key로.

return은 value로. 이 때, value는 어떤 형태든 될 수 있으므로 any.

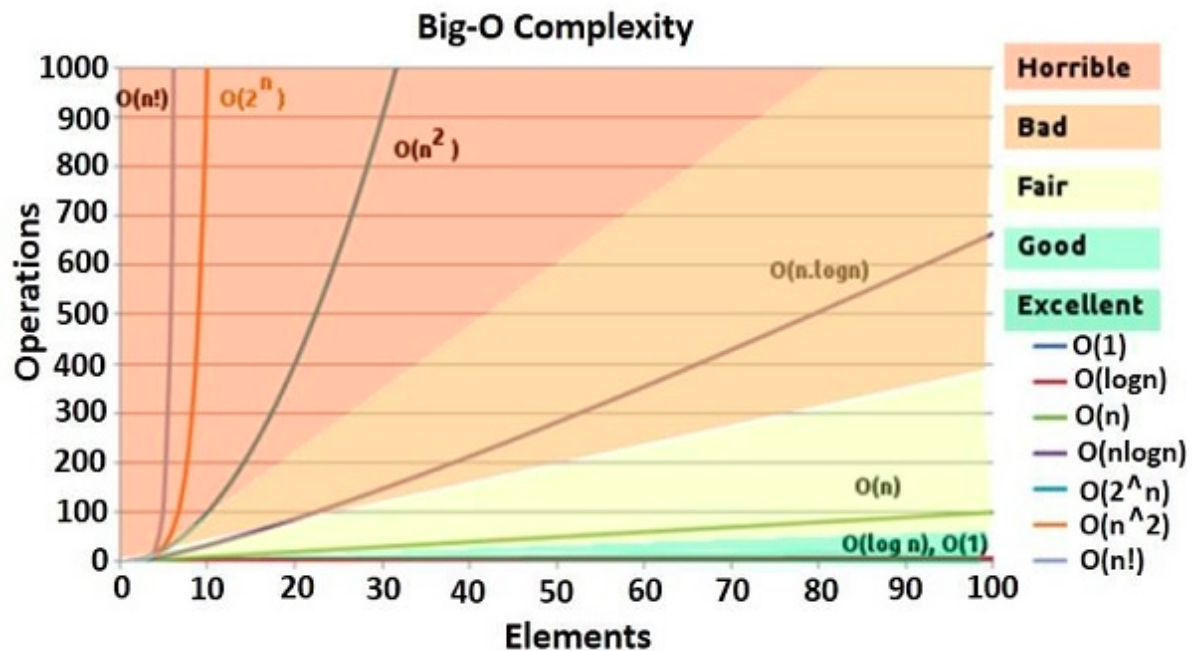
//////////코드이해필요//////////

Open Addressing

- open addressing는 hash collision이 발생 했을 때 rehashing(재해시)를 통해 빈 버킷을 찾아 데이터를 추가하는 방법이다.
- chaining처럼 지정한 메모리 외에 추가적인 저장공간이 필요가 없다.
- 삽입,삭제시 오버헤드가 적다.
 - **오버헤드(overhead)**는 어떤 처리를 하기 위해 들어가는 간접적인 처리 시간 · 메모리 등을 말한다.
- 저장할 데이터가 적을 때 더 유리하다.
 - 저장할 데이터가 많을 경우에는 chained hash가 더 유리할 수 있다.

cf. 참고사항

- 코드를 실제로 구현해보자. 외우는 것이 아니라 이렇게 나오는 구나를 직접 보고 실험해보기
-



마지막 주제는 알고리즘하면 놓칠 수 없는 부분이지! 바로 '시간 복잡도 분석(Analysis of Time Complexity)' 입니다.
그럼 시작해보죠!

간단히 유도를 해보겠습니다. 우선, 이진 탐색을 반복할수록 남아있는 (탐색할) 자료의 개수가 어떻게 될까요?

처음에 입력된 갯수를 N 이라하면,

첫 시행 후에는 반이 버려져서 $\frac{N}{2}$,

두 번째 시행 후에는 또 그 반이 버려져서 $\frac{1}{2} \times \frac{N}{2}$,

세 번째 시행 후에는 또 그 반이 버려져서 $\frac{1}{2} \times \frac{1}{2} \times \frac{N}{2}$,

규칙이 보이시나요? 그렇습니다. 매 시행마다 탐색할 자료의 개수가 점점 반씩 줄어드는 걸 알 수 있죠.

그럼 K 번의 시행 후에는? $\left(\frac{1}{2}\right)^K N$ 개가 남게 되겠죠.

이렇게 계속해서 탐색을 반복하다보면 탐색이 끝나는 시점에는 (최악의 경우 즉, 찾는 데이터가 없는 경우) 남은 자료가 1개가 남게 되겠죠.

따라서, $\left(\frac{1}{2}\right)^K N \approx 1$ 라고 표기할 수 있겠군요.

이 때, 양 변에 2^K 를 곱해주면 $2^K \approx N$

마지막으로 양 변에 2를 밑으로 하는 로그를 취해주면, $K \approx \log_2 N$

출처 및 참고]

1. <https://github.com/GyungBin-Park>
2. <https://jwoop.tistory.com/9>
3. <https://mangkyu.tistory.com/102>

4. <https://noahlogs.tistory.com/27>