

2020-1 Algorithm


# Quick Sort & Shell Sort Comparison

2016270265 이종헌





실습환경



## Shell Sort : How does a Shell Sort work?



## Quick Sort : Pivot comparison & In-Place Method



## Execution time(Run time) Analysis

# 01. Development Environment

Win10 Pro, i5-6600 , 8GB RAM, Python, Jupyter Notebook



# 01. Development Environment



## Why python and Jupyter Notebook?

1. 프로젝트의 진행을 정확한 그래프를 통해 시각적으로 보여드리고 싶었기 때문입니다. matplotlib을 사용하여 빈도수를 체크하는 코드의 경우 pycharm 혹은 spyder와 같은 인터프리터에서는 결과물밖에 보여지지 않습니다. 쥬피터노트북을 사용함으로 프로젝트의 단계별 진행을 보일 수 있었습니다.
2. 모두 오름차순으로 정렬.



# 02. Shell Sort Algorithm

It is a generalized version of insertion **sort**. In **shell sort**, elements at a specific interval are **sorted**.



## 02. Shell Sort



| 안전 정렬 ( Stable )                            | 불안전 정렬 ( not stable )                         |
|---------------------------------------------|-----------------------------------------------|
| 같은 값(key)의 위치가 정렬 과정에서 바뀌지 않는 것             | 같은 값(key)의 위치가 정렬 과정에서 바뀌는 것                  |
| 내부 정렬 ( Internal sorting )                  | 외부 정렬 ( External sorting )                    |
| 데이터의 크기가 주 기억장소 용량보다 적을 경우<br>기억장소를 활용하여 정렬 | 데이터의 크기가 주 기억장소 용량보다 클 경우<br>외부 기억장치를 사용하여 정렬 |

# 02. Shell Sort



## What is the Shell Sort?

1. Donald L.Shell이 제안한 방법. **삽입정렬을 보완한 알고리즘.**
2. 삽입정렬이 어느 정도 정렬이 된 상태에서는 대단히 빠르다는(Best일 때  $n$ ) 것에서 착안함.
  1. 삽입정렬의 문제점 : 요소들이 삽입될 때, 이웃한 위치로만 이동. 멀리 떨어진곳에 삽입되어야한다면 많은 이동을 해야한다.
3. Insertion Sort와는 다르게 Shell Sort는 전체의 리스트를 한번에 정렬하지 않는다.

*“ It is a generalized version of insertion sort.  
In Shell Sort, elements at a specific interval are sorted.”*

# 02. Shell Sort



## How does a Shell Sort work?

1. 먼저 정렬해야 할 리스트를 일정한 분류 기준에 따라 분류 (gap으로 지정)
2. 연속적이지 않은 여러 개의 부분 리스트를 생성
3. 각 부분 리스트를 삽입 정렬을 이용하여 정렬
4. 모든 부분 리스트가 정렬되면 다시 전체 리스트를 더 좁은 간격을 사용하여 적은 개수의 부분 리스트로 만들어 반복
5. 간격이 1이 될 때 까지 반복하여 마지막에는 insertion Sort로 정렬

*“It is a generalized version of insertion sort.  
In Shell Sort, elements at a specific interval are sorted.”*



# 02. Shell Sort



## How does a Shell Sort work?

1. 정렬해야 할 리스트의 각 gap번째 요소를 추출하여 부분리스트를 만든다. 요소 사이 간격gap 이라고 한다.
  - I. 간격의 초기값은 정렬할 요소들의 개수 / 2
  - II. 생성된 부분 리스트의 개수는 gap의 값과 같다.
2. 반복 할 때 마다 간격을 절반으로 줄이면서 부분 리스트에 속한 값들의 개수는 증가한다.
  - I. 간격은 홀수로 하는 것이 좋다.
  - II. 간격을 절반으로 줄일 때 짝수가 나오면 +1을 해서 홀수로 만든다.
3. 간격 gap이 1이 될 때까지 반복한다.

# 02. Shell Sort



초기상태

|    |   |   |    |   |   |    |   |   |    |    |
|----|---|---|----|---|---|----|---|---|----|----|
| 10 | 8 | 6 | 20 | 4 | 3 | 22 | 1 | 0 | 15 | 16 |
|----|---|---|----|---|---|----|---|---|----|----|

정렬할 값의 수: 10  
간격(gap) k의 초깃값:  $10/2 = 5$

1

간격 k=5 일 때의 부분 리스트들

|    |   |   |    |   |   |    |   |   |  |    |
|----|---|---|----|---|---|----|---|---|--|----|
| 10 |   |   |    |   | 3 |    |   |   |  | 16 |
|    | 8 |   |    |   |   | 22 |   |   |  |    |
|    |   | 6 |    |   |   |    | 1 |   |  |    |
|    |   |   | 20 |   |   |    |   | 0 |  |    |
|    |   |   |    | 4 |   |    |   |   |  | 15 |

← 하나의 부분 리스트

간격 k=5 일 때의 부분 리스트를  
각각 삽입 정렬로 정렬

|   |   |   |   |   |    |    |   |    |  |    |
|---|---|---|---|---|----|----|---|----|--|----|
| 3 |   |   |   |   | 10 |    |   |    |  | 16 |
|   | 8 |   |   |   |    | 22 |   |    |  |    |
|   |   | 1 |   |   |    |    | 6 |    |  |    |
|   |   |   | 0 |   |    |    |   | 20 |  |    |
|   |   |   |   | 4 |    |    |   |    |  | 15 |

1회전 결과

|   |   |   |   |   |    |    |   |    |    |    |
|---|---|---|---|---|----|----|---|----|----|----|
| 3 | 8 | 1 | 0 | 4 | 10 | 22 | 6 | 20 | 15 | 16 |
|---|---|---|---|---|----|----|---|----|----|----|

다음 k의 값:  $(5/2)+1 = 3$

1회전 결과

|   |   |   |   |   |    |    |   |    |    |    |
|---|---|---|---|---|----|----|---|----|----|----|
| 3 | 8 | 1 | 0 | 4 | 10 | 22 | 6 | 20 | 15 | 16 |
|---|---|---|---|---|----|----|---|----|----|----|

다음 k의 값:  $(5/2)+1 = 3$

2

간격 k=3 일 때의 부분 리스트들

|   |   |   |   |   |    |    |   |    |    |    |
|---|---|---|---|---|----|----|---|----|----|----|
| 3 |   |   | 0 |   |    | 22 |   |    | 15 |    |
|   | 8 |   |   | 4 |    |    | 6 |    |    | 16 |
|   |   | 1 |   |   | 10 |    |   | 20 |    |    |

간격 k=3 일 때의 부분 리스트를  
각각 삽입 정렬로 정렬

|   |   |   |   |   |    |    |   |    |    |    |
|---|---|---|---|---|----|----|---|----|----|----|
| 0 |   |   | 3 |   |    | 15 |   |    | 22 |    |
|   | 4 |   |   | 6 |    |    | 8 |    |    | 16 |
|   |   | 1 |   |   | 10 |    |   | 20 |    |    |

2회전 결과

|   |   |   |   |   |    |    |   |    |    |    |
|---|---|---|---|---|----|----|---|----|----|----|
| 0 | 4 | 1 | 3 | 6 | 10 | 15 | 8 | 20 | 22 | 16 |
|---|---|---|---|---|----|----|---|----|----|----|

다음 k의 값:  $3/2 = 1$

3

간격 k=1 일 때의 부분 리스트들

|   |   |   |   |   |    |    |   |    |    |    |
|---|---|---|---|---|----|----|---|----|----|----|
| 0 | 4 | 1 | 3 | 6 | 10 | 15 | 8 | 20 | 22 | 16 |
|---|---|---|---|---|----|----|---|----|----|----|

간격 k=1 일 때의 부분 리스트를  
각각 삽입 정렬로 정렬

|   |   |   |   |   |   |    |    |    |    |    |
|---|---|---|---|---|---|----|----|----|----|----|
| 0 | 1 | 3 | 4 | 6 | 8 | 10 | 15 | 16 | 20 | 22 |
|---|---|---|---|---|---|----|----|----|----|----|

3회전 결과

|   |   |   |   |   |   |    |    |    |    |    |
|---|---|---|---|---|---|----|----|----|----|----|
| 0 | 1 | 3 | 4 | 6 | 8 | 10 | 15 | 16 | 20 | 22 |
|---|---|---|---|---|---|----|----|----|----|----|

오름차순  
완성상태

|   |   |   |   |   |   |    |    |    |    |    |
|---|---|---|---|---|---|----|----|----|----|----|
| 0 | 1 | 3 | 4 | 6 | 8 | 10 | 15 | 16 | 20 | 22 |
|---|---|---|---|---|---|----|----|----|----|----|

# 02. Shell Sort



파이썬으로 구현한 Shell Sort 함수

```
1 def gapInsertionSort(arr, start, gap):
2     for target in range(start+gap, len(arr), gap):
3         val = arr[target]
4         i = target
5         while i > start:
6             if arr[i-gap] > val:
7                 arr[i] = arr[i-gap]
8             else:
9                 break
10            i -= gap
11        arr[i] = val
12
13 def shellSort(arr):
14     gap = len(arr) // 2
15     while gap > 0:
16         for start in range(gap):
17             gapInsertionSort(arr, start, gap)
18         gap = gap // 2
```

```
1 def gapInsertionSort(arr, start, gap):
2     for target in range(start+gap, len(arr), gap):
3         val = arr[target]
4         i = target
5         while i > start:
6             if arr[i-gap] > val:
7                 arr[i] = arr[i-gap]
8             else:
9                 break
10            i -= gap
11        arr[i] = val
12
13 def shellSort2(arr):
14     gap = 1
15     while gap < len(arr):
16         gap = 3*gap + 1
17
18     while gap > 0:
19         for start in range(gap):
20             gapInsertionSort(arr, start, gap)
21         gap = gap // 3
```

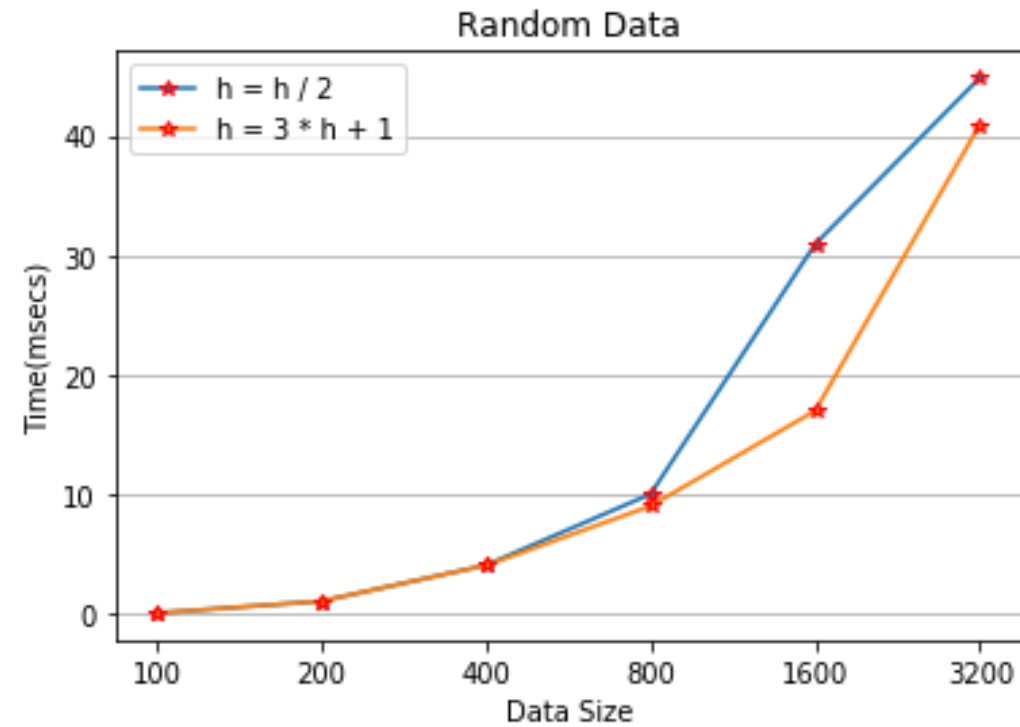
Why gap = 3\*gap + 1?

# 02. Shell Sort



| OEIS    | General term ( $k \geq 1$ )                                                                                                                                                                                                                                                                                  | Concrete gaps                                                                                                                          | Worst-case time complexity                               | Author and year of publication                                  |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------|-----------------------------------------------------------------|
|         | $\left\lfloor \frac{N}{2^k} \right\rfloor$                                                                                                                                                                                                                                                                   | $\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \dots, 1$                                             | $\Theta(N^2)$ [e.g. when $N = 2^p$ ]                     | Shell, 1959 <sup>[4]</sup>                                      |
|         | $2 \left\lfloor \frac{N}{2^{k+1}} \right\rfloor + 1$                                                                                                                                                                                                                                                         | $2 \left\lfloor \frac{N}{4} \right\rfloor + 1, \dots, 3, 1$                                                                            | $\Theta(N^{\frac{3}{2}})$                                | Frank & Lazarus, 1960 <sup>[8]</sup>                            |
| A000225 | $2^k - 1$                                                                                                                                                                                                                                                                                                    | 1, 3, 7, 15, 31, 63, ...                                                                                                               | $\Theta(N^{\frac{3}{2}})$                                | Hibbard, 1963 <sup>[9]</sup>                                    |
| A083318 | $2^k + 1$ , prefixed with 1                                                                                                                                                                                                                                                                                  | 1, 3, 5, 9, 17, 33, 65, ...                                                                                                            | $\Theta(N^{\frac{3}{2}})$                                | Papernov & Stasevich, 1965 <sup>[10]</sup>                      |
| A003586 | Successive numbers of the form $2^p 3^q$ (3-smooth numbers)                                                                                                                                                                                                                                                  | 1, 2, 3, 4, 6, 8, 9, 12, ...                                                                                                           | $\Theta(N \log^2 N)$                                     | Pratt, 1971 <sup>[1]</sup>                                      |
| A003462 | $\frac{3^k - 1}{2}$ , not greater than $\left\lfloor \frac{N}{3} \right\rfloor$                                                                                                                                                                                                                              | 1, 4, 13, 40, 121, ...                                                                                                                 | $\Theta(N^{\frac{3}{2}})$                                | Knuth, 1973, <sup>[3]</sup> based on Pratt, 1971 <sup>[1]</sup> |
| A036569 | $\prod_I a_q, \text{ where}$ $a_q = \min \left\{ n \in \mathbb{N} : n \geq \left( \frac{5}{2} \right)^{q+1}, \forall p: 0 \leq p < q \Rightarrow \gcd(a_p, n) = 1 \right\}$ $I = \left\{ 0 \leq q < r \mid q \neq \frac{1}{2} (r^2 + r) - k \right\}$ $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$ | 1, 3, 7, 21, 48, 112, ...                                                                                                              | $O\left(N^{1 + \sqrt{\frac{8 \ln(5/2)}{\ln(N)}}}\right)$ | Incerpi & Sedgewick, 1985, <sup>[11]</sup> Knuth <sup>[3]</sup> |
| A036562 | $4^k + 3 \cdot 2^{k-1} + 1$ , prefixed with 1                                                                                                                                                                                                                                                                | 1, 8, 23, 77, 281, ...                                                                                                                 | $O(N^{\frac{4}{3}})$                                     | Sedgewick, 1982 <sup>[6]</sup>                                  |
| A033622 | $\begin{cases} 9 \left( 2^k - 2^{\frac{k}{2}} \right) + 1 & k \text{ even,} \\ 8 \cdot 2^k - 6 \cdot 2^{(k+1)/2} + 1 & k \text{ odd} \end{cases}$                                                                                                                                                            | 1, 5, 19, 41, 109, ...                                                                                                                 | $O(N^{\frac{4}{3}})$                                     | Sedgewick, 1986 <sup>[12]</sup>                                 |
|         | $h_k = \max \left\{ \left\lfloor \frac{5h_{k-1}}{11} \right\rfloor, 1 \right\}, h_0 = N$                                                                                                                                                                                                                     | $\left\lfloor \frac{5N}{11} \right\rfloor, \left\lfloor \frac{5}{11} \left\lfloor \frac{5N}{11} \right\rfloor \right\rfloor, \dots, 1$ | Unknown                                                  | Gonnet & Baeza-Yates, 1991 <sup>[13]</sup>                      |
| A108870 | $\left\lceil \frac{1}{5} \left( 9 \cdot \left( \frac{9}{4} \right)^{k-1} - 4 \right) \right\rceil$                                                                                                                                                                                                           | 1, 4, 9, 20, 46, 103, ...                                                                                                              | Unknown                                                  | Tokuda, 1992 <sup>[14]</sup>                                    |
| A102549 | Unknown (experimentally derived)                                                                                                                                                                                                                                                                             | 1, 4, 10, 23, 57, 132, 301, 701                                                                                                        | Unknown                                                  | Ciura, 2001 <sup>[15]</sup>                                     |

# 02. Shell Sort





# 02. Shell Sort



## Shell Sort Algorithm의 장점

- 연속적이지 않은 부분 리스트에서 자료의 교환이 일어나면 더 큰 거리를 이동한다. 따라서 교환되는 요소들이 삽입 정렬보다는 최종 위치에 있을 가능성이 높아진다. **배열 뒷부분의 작은 숫자를 앞부분으로 '빠르게' 이동시키고, 동시에 앞부분의 큰 숫자는 뒷부분으로 이동시킨다.**
- 부분 리스트는 어느 정도 정렬이 된 상태이기 때문에 부분 리스트의 개수가 1이 되게 되면 셸 정렬은 기본적으로 삽입 정렬을 수행하는 것이지만 삽입 정렬보다 더욱 빠르게 수행된다.
- 알고리즘이 간단하여 프로그램으로 쉽게 구현할 수 있다.

# 02. Shell Sort



## Shell Sort Algorithm의 장점

- 입력 크기가 매우 크지 않은 경우에 매우 좋은 성능을 보인다.
- 임베디드(Embedded) 시스템에서 주로 사용되는 데, Shell Sort의 특징인 간격에 따른 그룹 별 정렬 방식이 H/W 로 정렬 알고리즘을 구현하는데 매우 적합하기 때문이다.

uClibc : 임베디드 리눅스 전용으로 만들어진 소형 C표준 라이브러리.

*“Shell Sort is used in the uClibc library.  
In the past, ShellSort was used in the Linux Kernel”*

## 03. Quick Sort Algorithm

**Quicksort** is a divide-and-conquer **algorithm**.

It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.



# 03. Quick Sort



## What is the Quick Sort?

1. **Divide and Conquer** 방식으로 큰 데이터를 작게 쪼개서 하나씩 해결하는 방법을 씀.  
pivot을 잡아 pivot보다 작은 값은 왼쪽으로, 큰 값은 오른쪽으로 이동을 하게 함.
2. 파이썬의 `list.sort()` 함수나 자바의 `Arrays.sort()`처럼 프로그래밍 언어 차원에서 기본적으로 지원되는 내장 정렬 함수는 대부분은 퀵 정렬을 기본으로 합니다.
3. 일반적으로 원소의 개수가 적어질수록 나쁜 중간값이 선택될 확률이 높아지기 때문에, 원소의 개수에 따라 퀵 정렬에 다른 정렬을 혼합해서 쓰는 경우가 많습니다.

# 03. Quick Sort



## How does the Quick Sort work?

1. 리스트 요소들 중 하나를 피벗으로 선택하고, 피벗 값보다 작은 값, 동일한 값 그리고 큰 값을 담아둘 3개의 리스트를 생성합니다.
2. 반복문을 통해서 각 값을 피벗과 비교 후에 해당하는 리스트에 추가시킵니다.
3. 작은 값과 큰 값을 가지고 있는 배열(리스트)를 대상으로 Quick Sort를 **재귀적으로 호출**합니다.
4. 마지막으로 재귀 호출의 결과를 다시 크기 순으로 합쳐서 정렬된 리스트를 얻을 수 있다.



# 03. Quick Sort



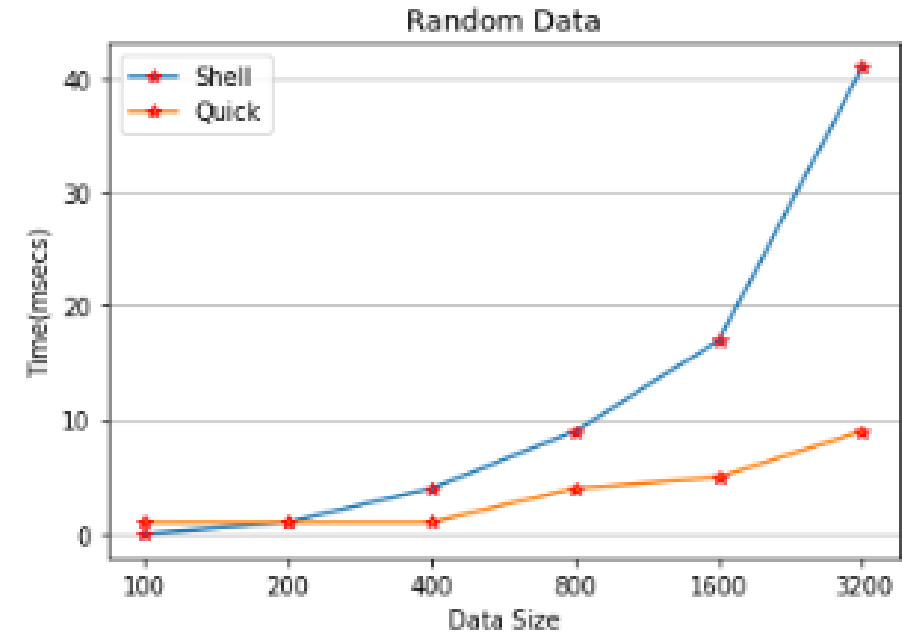
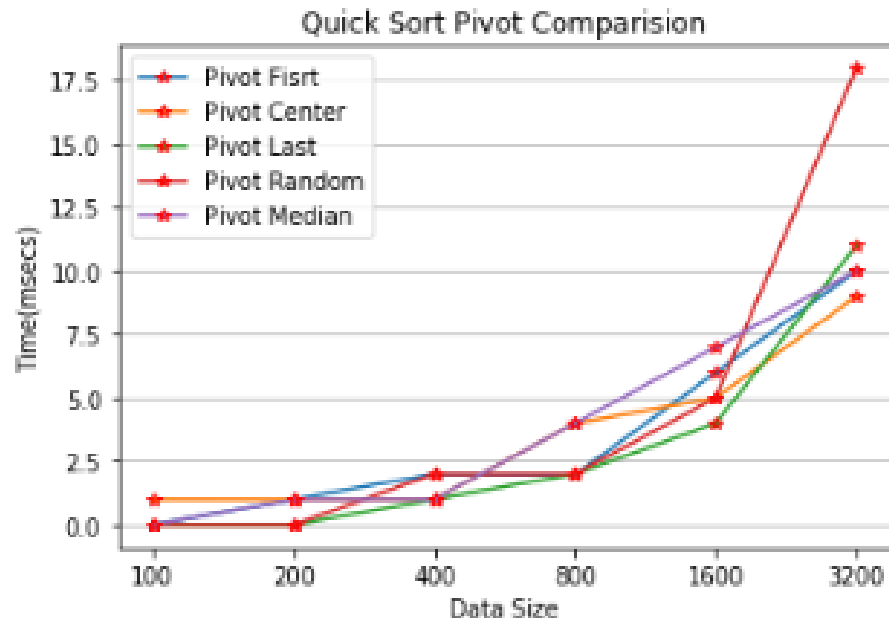
```
1 def quickSort1(arr): #피벗 처음 값
2
3     if len(arr) <= 1:
4         return arr
5     # pivot = arr[len(arr) // 2]
6     pivot = arr[0]
7     lesser_arr, equal_arr, greater_arr = [], [], []
8
9     for num in arr:
10         if num < pivot:
11             lesser_arr.append(num) #작은 리스트로 보내기
12         elif num > pivot:
13             greater_arr.append(num) #큰 리스트로 보내기
14         else:
15             equal_arr.append(num)
16
17     #작은 리스트와 큰 리스트끼리 다시 재귀하여 퀵정렬
18     return quickSort1(lesser_arr) + equal_arr + quickSort1(greater_arr)
19
20 def quickSort2(arr): #피벗 중간 값
21
22     if len(arr) <= 1:
23         return arr
24
25     pivot = arr[len(arr) // 2]
26     lesser_arr, equal_arr, greater_arr = [], [], []
27
28     for num in arr:
29         if num < pivot:
30             lesser_arr.append(num) #작은 리스트로 보내기
31         elif num > pivot:
32             greater_arr.append(num) #큰 리스트로 보내기
33         else:
34             equal_arr.append(num)
35
36     #작은 리스트와 큰 리스트끼리 다시 재귀하여 퀵정렬
37     return quickSort2(lesser_arr) + equal_arr + quickSort2(greater_arr)
```

```
39 def quickSort3(arr): #피벗 마지막 값
40
41     if len(arr) <= 1:
42         return arr
43     pivot = arr[len(arr)-1]
```

```
def quickSort4(arr): #피벗 랜덤 값
    if len(arr) <= 1:
        return arr
    pivot = arr[random.randint(0, len(arr)-1)] #### 랜덤 피벗 ####
    lesser_arr, equal_arr, greater_arr = [], [], []
```

```
def quickSort5(arr): #피벗 처음 중간 마지막의 평균 값
    if len(arr) <= 1:
        return arr
    pivot = arr[(0+(len(arr)//2)+(len(arr)-1))//3]
```

# 03. Quick Sort



# 03. Quick Sort



## In-Place Sorting Method

- 정렬 할 원소들을 위한 추가적인 메모리를 사용하지 않고 현재 메모리에서 정렬을 진행
  - 메모리를 아낄 수 있다.
  - 하지만 코드가 조금 더 복잡하다.
  - 왼쪽과 오른쪽 값을 교환하는 과정에서 중복 값의 위치가 바뀔 수 있으므로 unstable하다.

# 03. Quick Sort



## In-Place Sorting Method

```
1 def quick_sort(arr):
2     def sort(low, high):
3         if high <= low:
4             return
5
6         mid = partition(low, high)
7         sort(low, mid - 1)
8         sort(mid, high)
9
10    def partition(low, high):
11        pivot = arr[(low + high) // 2]
12
13        while low <= high:
14            while arr[low] < pivot:
15                low += 1
16            while arr[high] > pivot:
17                high -= 1
18            if low <= high:
19                arr[low], arr[high] = arr[high], arr[low]
20                low, high = low + 1, high - 1
21        return low
22
23    return sort(0, len(arr) - 1)
```

메인 함수인 quick\_sort()는 크게 **sort()**와 **partition()**로 구성

sort() 함수는 재귀 함수이며 정렬 범위를 시작 인덱스와 끝 인덱스로 인자로 받습니다.

partition() 함수는 정렬 범위를 인자로 받으며 좌우측의 값들을 정렬하고 분할 기준점의 인덱스를 return

이 분할 기준점(mid)는 sort()를 재귀적으로 호출할 때 우측 리스트의 시작 인덱스로 사용됩니다.

- 리스트의 정 가운데 있는 값을 pivot 값을 선택합니다.
- 시작 인덱스(low)는 계속 증가 시키고, 끝 인덱스(high)는 계속 감소, while 루프로 두 인덱스가 서로 교차해서 지나칠 때까지 반복시킵니다.
  - 시작 인덱스(low)가 가리키는 값과 pivot 값을 비교, 더 작은 경우 반복해서 시작 인덱스 값을 증가시킵니다. (**pivot 값보다 큰데 좌측**에 있는 값을 찾기 위해)
  - 끝 인덱스(high)가 가리키는 값과 pivot 값을 비교, 더 작은 경우 반복해서 끝 인덱스 값을 감소시킵니다. (**pivot 값보다 작은데 우측**에 있는 값을 찾기 위해)
  - 두 인덱스가 아직 서로 교차하지 않았다면 시작 인덱스(low)가 가리키는 값과 끝 인덱스(high)가 가리키는 값을 swap 시킵니다.
- 상호 교대 후, 다음 값을 가리키기 위해 두 인덱스를 각자 진행 방향으로 한 칸씩 이동 시킵니다.
- 두 인덱스가 서로 교차해서 지나치게 되어 while 루프를 빠져나왔다면 다음 재귀 호출의 분할 기준점이 될 시작 인덱스를 리턴합니다.

1 12 5 26 7 14 3 7 2

Diagram illustrating the partitioning process in the Quick Sort algorithm. The array contains the values [1, 12, 5, 26, 7, 14, 3, 7, 2]. The pivot value is 7, located at index 4. The current element being compared is 1 at index 0. The pointers i and j are shown at the bottom, indicating the current positions of the pointers.

Diagram illustrating an array with elements: 1, 12, 5, 26, 7, 14, 3, 7, 2. The first element (1) is highlighted in green, and the last element (2) is highlighted in red. Arrows labeled  $i$  and  $j$  point to the second element (12) and the last element (2) respectively.

Diagram illustrating an array with 9 elements: [1, 2, 5, 26, 7, 14, 3, 7, 12]. The elements at indices 3 and 7 (values 26 and 7) are highlighted with red boxes. Arrows point from labels  $i$  and  $j$  below to these elements respectively.

The diagram shows an array of 9 elements: 1, 2, 5, 7, 7, 14, 3, 26, 12. The elements are enclosed in boxes. The boxes for the 5th element (7) and the 7th element (3) are highlighted with red borders. Below the array, two upward-pointing arrows are labeled  $i$  and  $j$ , corresponding to the 5th and 7th positions respectively.

The diagram shows an array of 9 elements: 1, 2, 5, 7, 3, 14, 7, 26, 12. Below the array, two red arrows point upwards. The first arrow, labeled 'j', points to the element 3 at index 4. The second arrow, labeled 'i', points to the element 14 at index 5.

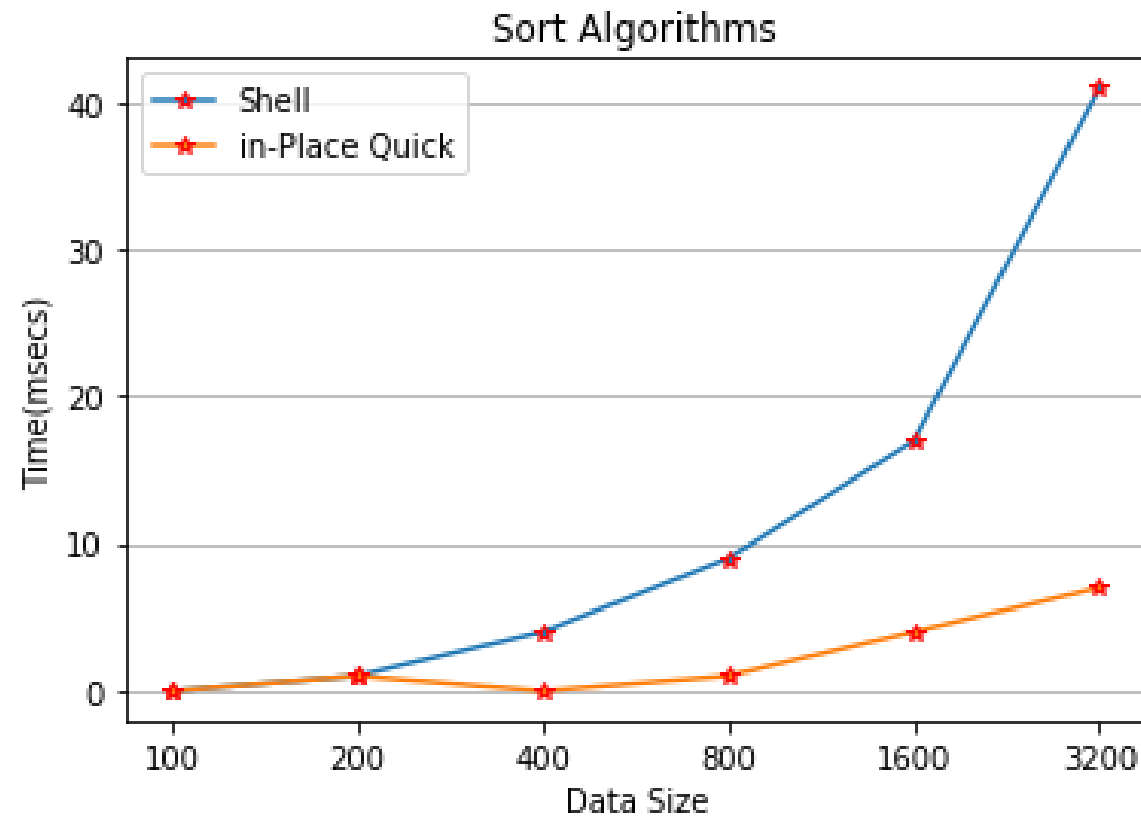
1 2 5 7 3                      14 7 26 12

■ ■ ■

1 2 3 5 7 7 12 14 26



# 03. Quick Sort

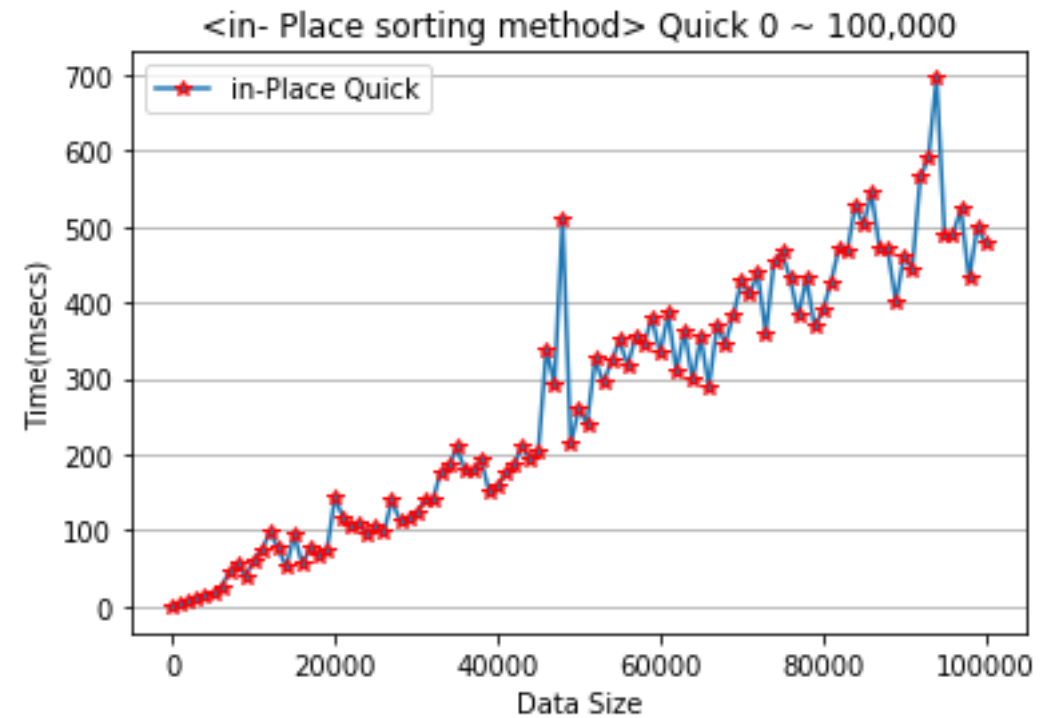
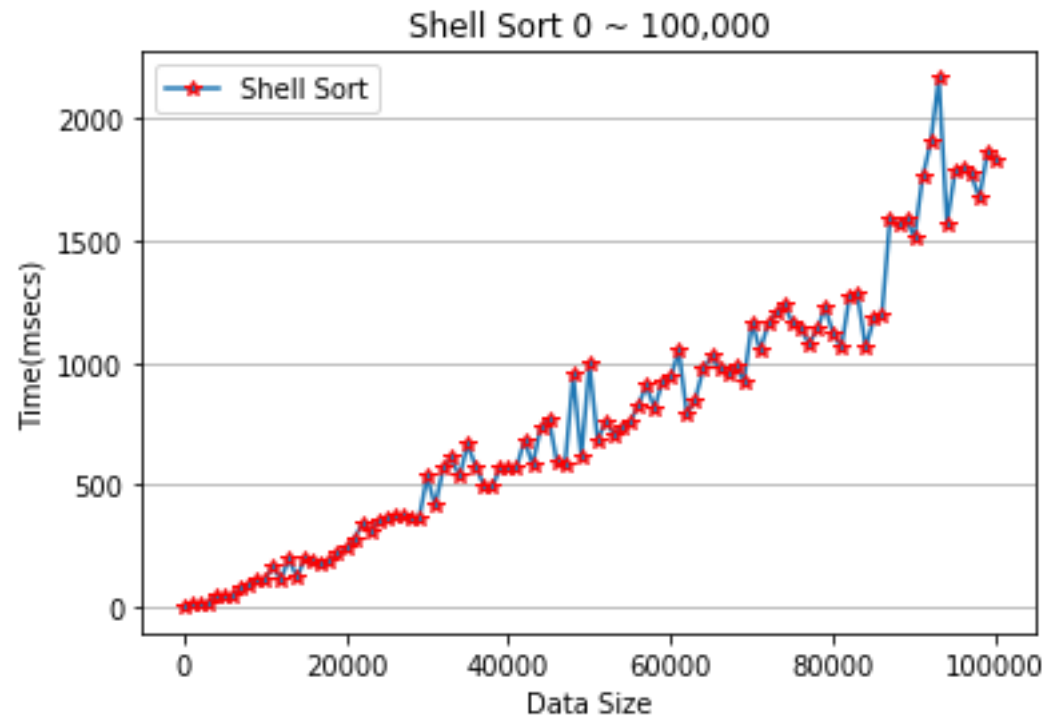


# 04. Execution Time Analysis

Comparing 2 types of algorithm methods



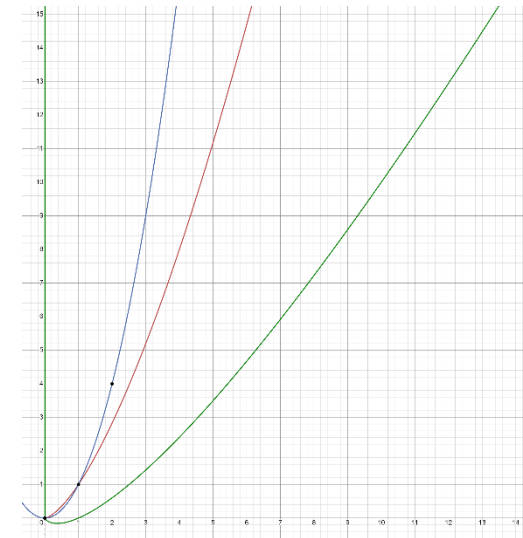
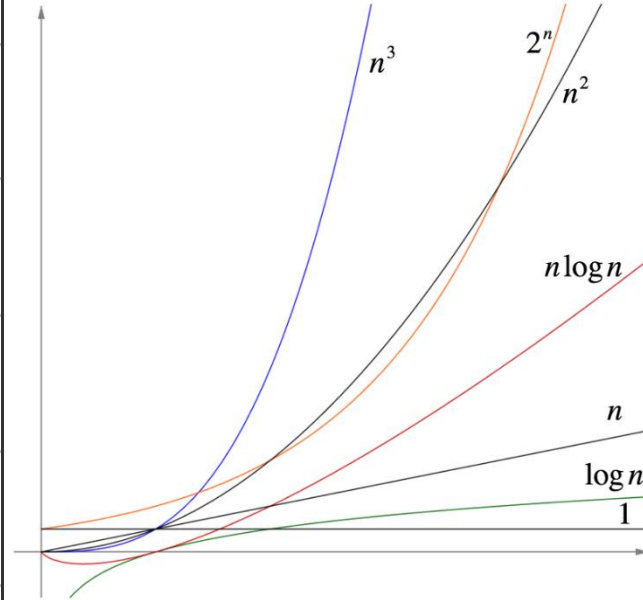
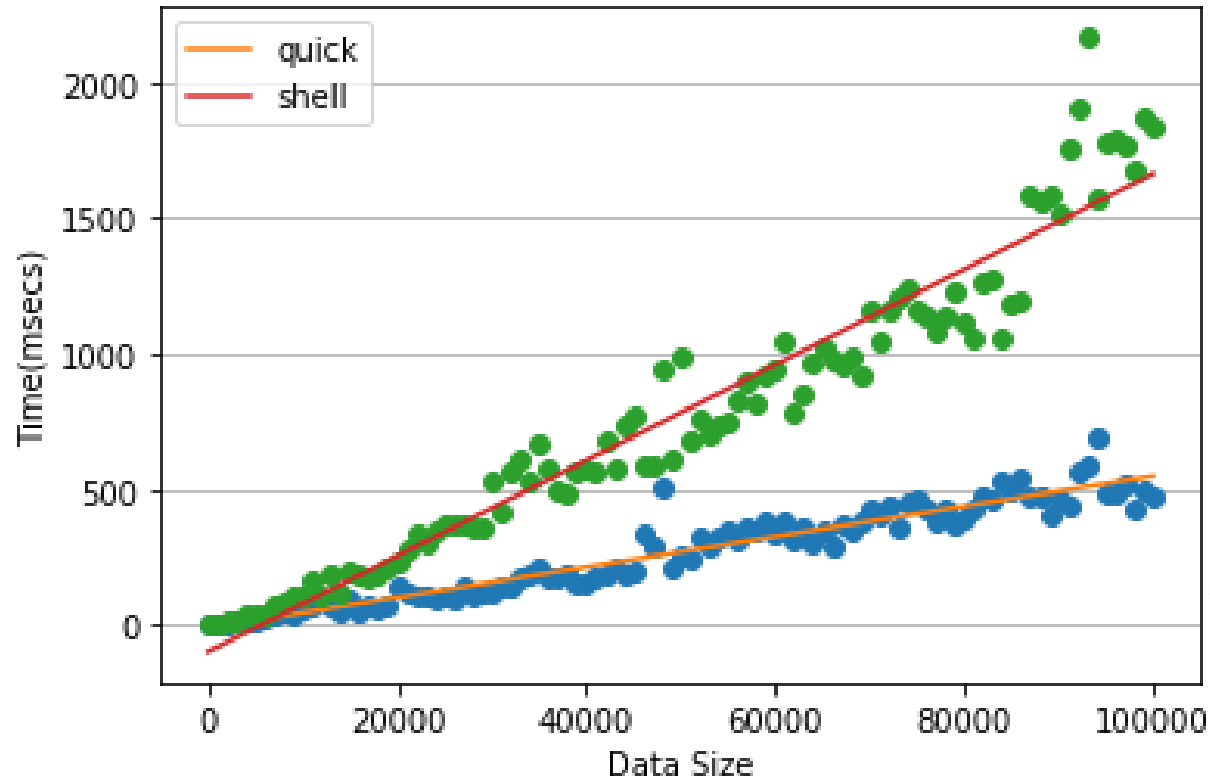
# 04. Execution Time Analysis



# 04. Execution Time Analysis



Linear Regression Model



# 04. Execution Time Analysis



|            | Worst    | Best          | Avg           |
|------------|----------|---------------|---------------|
| Quick Sort | $O(n^2)$ | $O(n \log n)$ | $O(n \log n)$ |

|            | Worst    | Best   | Avg          |
|------------|----------|--------|--------------|
| Shell Sort | $O(n^2)$ | $O(n)$ | $O(n^{1.5})$ |



# THANK YOU

A vibrant, cartoon-style illustration of a desk cluttered with various office and creative items. The scene includes hands interacting with documents, a calculator, a smartphone, a globe, a book, a camera, a coffee cup, and various papers with charts and diagrams. The background is a solid blue color.