

Ticker sorting 방법 개선

코드의 전설들

김규민
이동훈
이우남
이연지
한정빈

목차

0. 가정

1. Bubble sort

2. Tim sort

3. sorting algorithm별 평균 실행시간 계산

4. 단기 매매에서 **sorting** 활용과 그 중요성

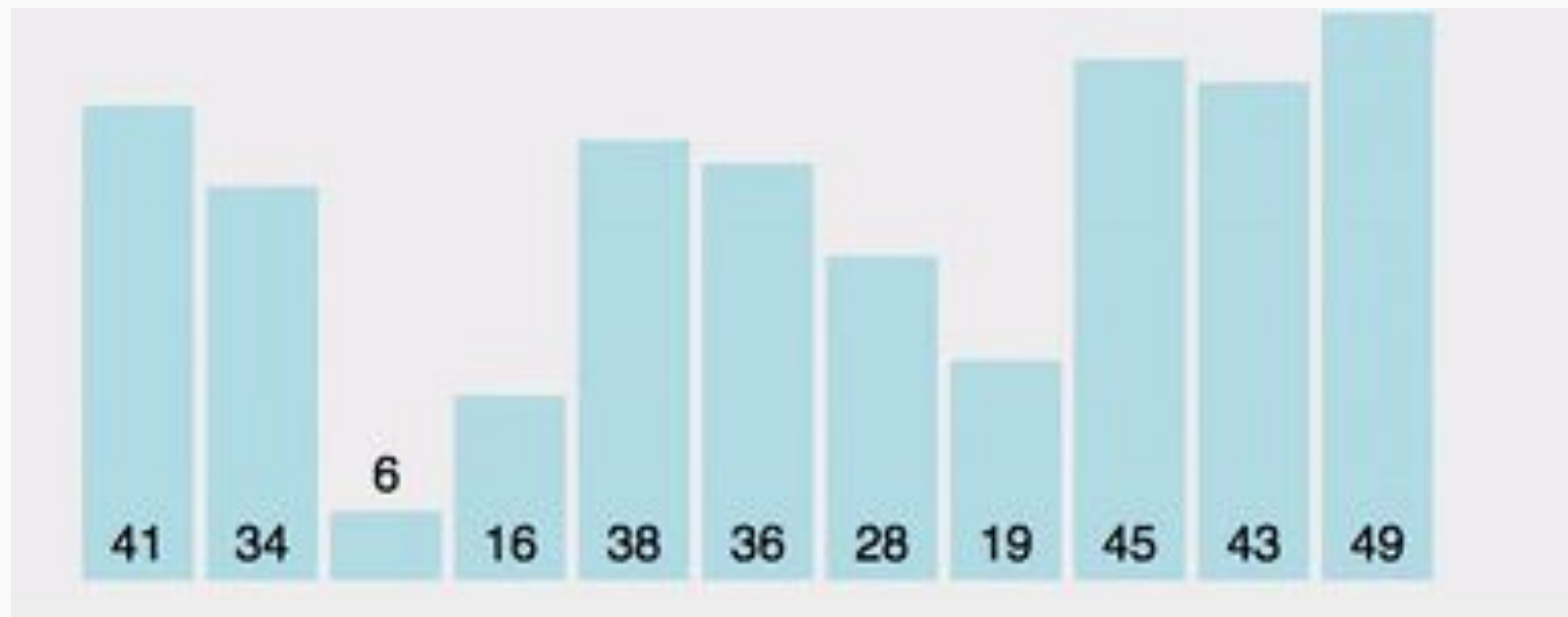
0. 가정

- A증권사는 충분한 메모리 보유해 메모리 관련 큰 제약사항 없음
따라서 필요한 추가 저장공간보다는 시간복잡도를 우선적으로 고려함
- A증권사는 `python` 이용해 알고리즘 개발 및 트레이딩 수행

1. Bubble sort

- 가장 간단하고 직관적인 정렬 방법
- 인접한 두 원소를 비교한 후, 조건에 맞지 않는다면 자리를 교환하여 정렬
- 리스트의 시작지점부터 끝지점까지 수행한 후, 정렬이 완료될때까지 이 과정 반복

1. bubble sort



1. Bubble sort

```
def bubbleSort(x):  
    length = len(x)-1  
    for i in range(length):  
        for j in range(length-i):  
            if x[j] > x[j+1]:  
                x[j], x[j+1] = x[j+1], x[j]  
    return x
```

1. Bubble sort

- 비교 횟수
 - 최상, 평균, 최악일때 $n(n-1)/2$ 로 일정
- 교환횟수
 - 입력 자료가 역순으로 정렬된 최악의 경우 : $3n(n-1)/2$
 - 입력 자료가 이미 정렬된 최상의 경우 : 0

New Sorting Methods



무엇이 있는가?

정렬 알고리즘 시간 복잡도 정리

정렬	최선	평균	최악
선택정렬(Selection Sort)	$O(n^2)$	$O(n^2)$	$O(n^2)$
거품정렬(Bubble Sort)	$O(n^2)$	$O(n^2)$	$O(n^2)$
삽입정렬(Insertion Sort)	$O(n)$	$O(n^2)$	$O(n^2)$
퀵정렬(Quick Sort)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
병합정렬(Merge Sort)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
힙정렬(Heap Sort)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

가장 빠른 것은 무엇인가?

정렬 알고리즘 시간 복잡도 정리

Timsort(hybrid)

정렬	최선	평균	최악
선택정렬(Selection Sort)	$O(n^2)$	$O(n^2)$	$O(n^2)$
거품정렬(Bubble Sort)	$O(n^2)$	$O(n^2)$	$O(n^2)$
삽입정렬(Insertion Sort)	$O(n)$	$O(n^2)$	$O(n^2)$
퀵정렬(Quick Sort)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
병합정렬(Merge Sort)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
힙정렬(Heap Sort)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

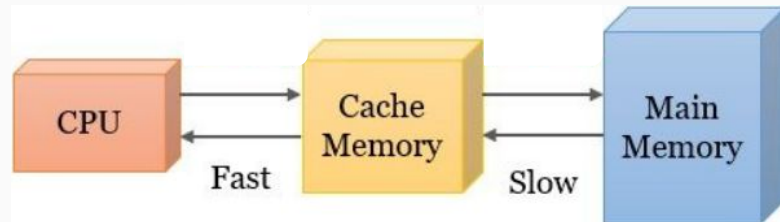
2. Tim sort

- 2002년 소프트웨어 엔지니어 **Tim Peters** 고안한 알고리즘
- Insertion sort와 Merge sort를 결합해 만듦
- 현재 Python, Java SE 7, Android, Google chrome (V8), 그리고 swift까지 많은 프로그래밍 언어에서 표준 정렬 알고리즘으로 채택되어 사용되고 있음



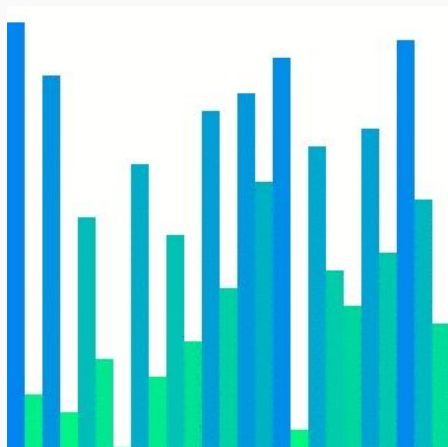
2-0. 참조 지역성 (Locality of Reference)

- CPU는 지연을 줄이기 위해 미래에 원하는 데이터를 예측해 속도가 빠른 장치인 캐시 메모리에 담아 놓는데, 이때의 예측률을 높이기 위해 사용하는 원리
- 캐시에는 **최근에 참조한 메모리(Temporal locality)**나 그 **메모리와 인접한 메모리(Spatial locality)**에 저장된 데이터가 올라가며, 시간복잡도가 같더라도 참조 지역성을 더 잘 만족하는 프로그램이 더 빨리 동작함



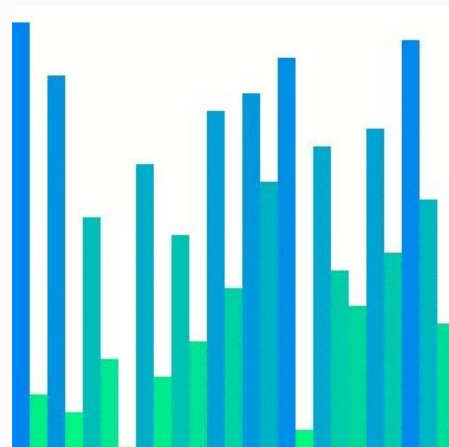
2-0. 참조 지역성 (Locality of Reference)

참조 지역성이 좋지 않은 예
C 값이 높다



Heap sort

참조 지역성이 좋은 예
C 값이 낮다

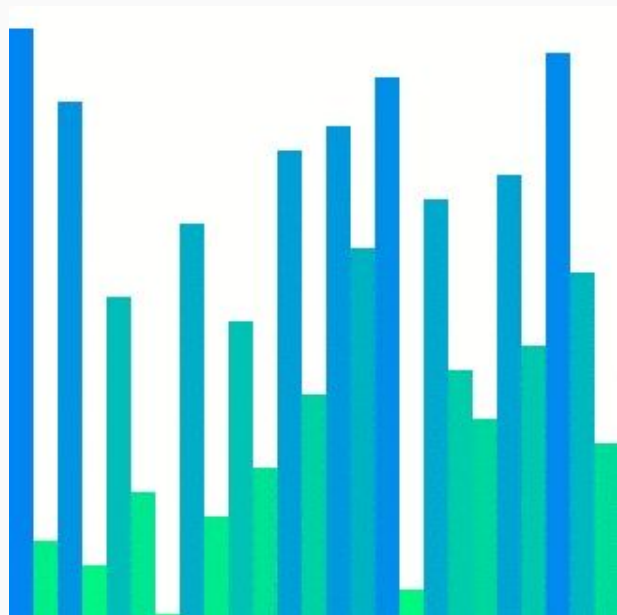


Quick sort

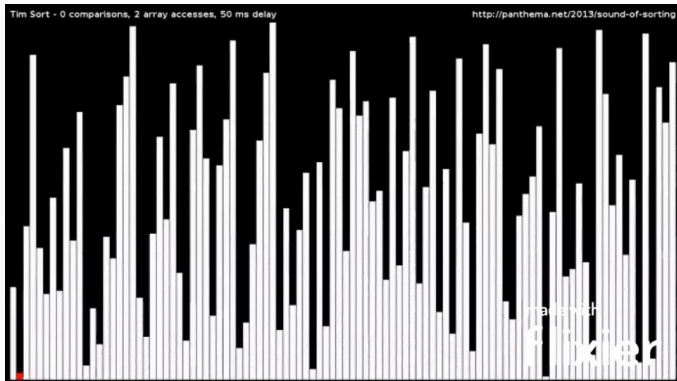
2-0. 참조 지역성 (Locality of Reference)

Insertion sort는 인접한 메모리와의 비교를 반복하기에
참조 지역성의 원리를 **매우 잘 만족**한다.

따라서 작은 n 에 대해선 Insertion sort가 더 빠르다.



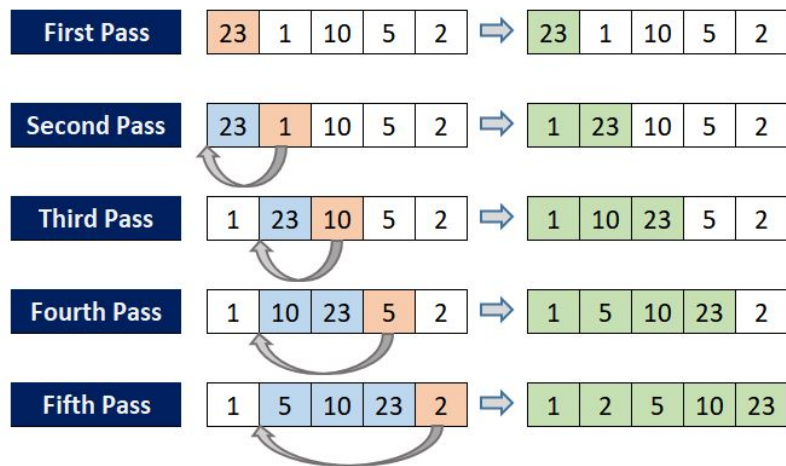
2. Tim sort



1. **Insertion sort**를 통해 정렬을 시키며 덩어리를 만들고,
2. 이를 Merge(병합) 하는 방법이

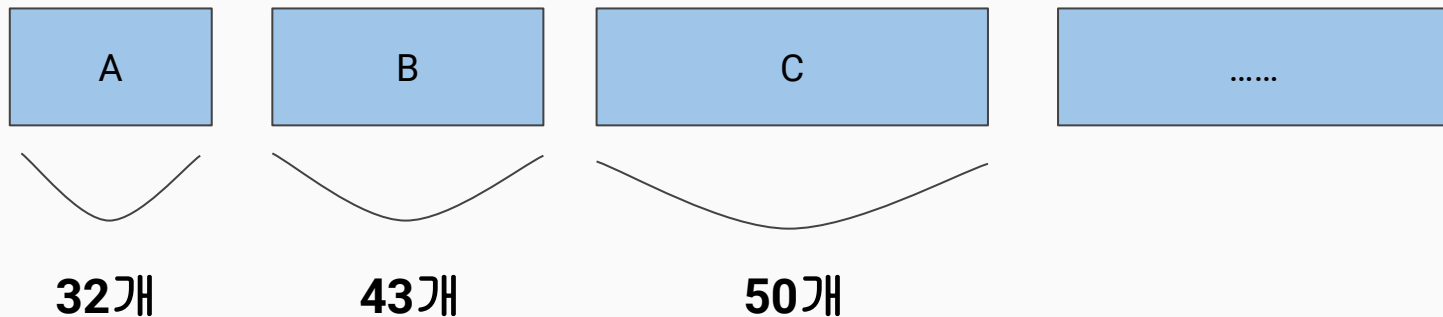
Timsort!

2-1. 작은 덩어리(run)들로 나누기

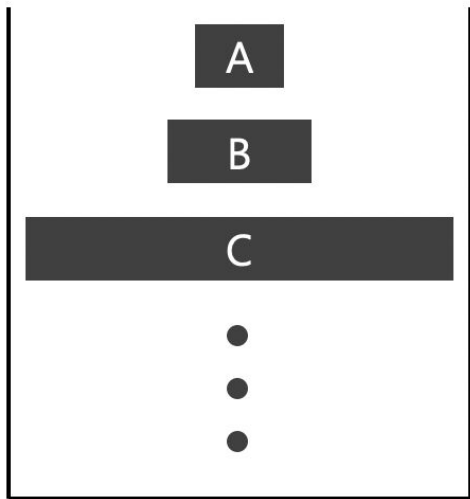


Binary Insertion sort 적용

2-1. 작은 덩어리(run)들로 나누기

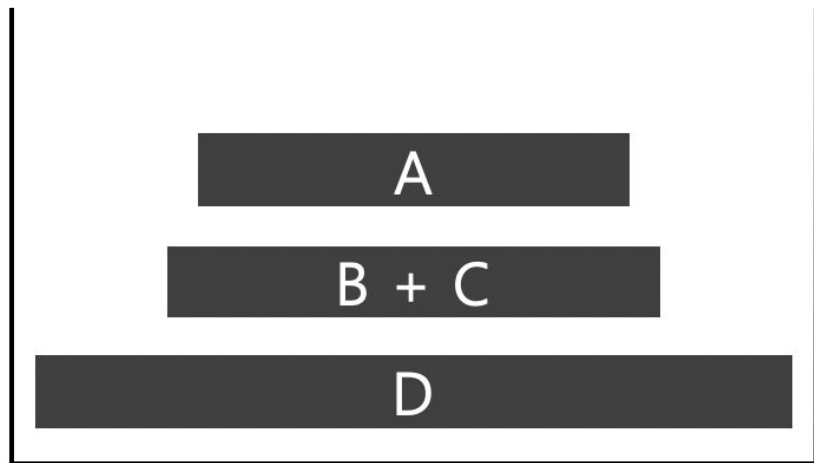
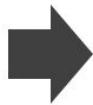
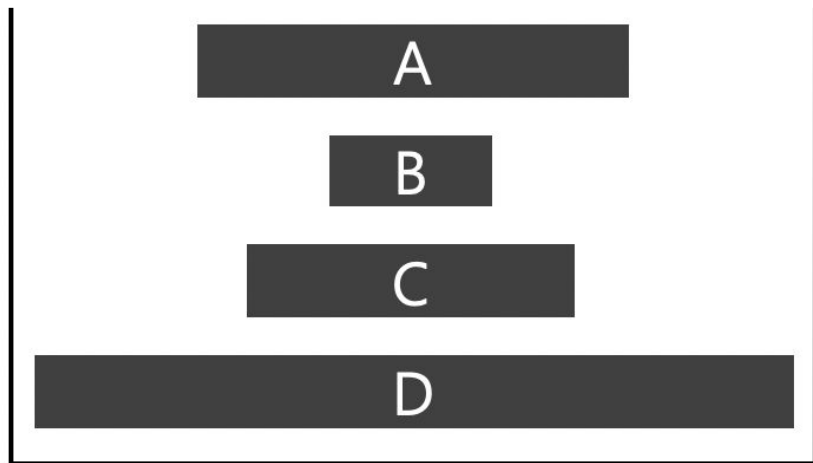


2-2. 병합하는 순서



- 1) $|C| > |A| + |B|$
- 2) $|B| > |A|$

2-2. 병합하는 순서



2-2. 병합하는 순서

A

B

$C > A + B$

$D > C + B$

$E > D + C$

2-2. 병합하는 순서

$$\frac{0}{1} = 0 \quad \frac{13}{8} = 1,625 \quad \frac{233}{144} = 1,618$$

$$\frac{1}{1} = 1 \quad \frac{21}{13} = 1,6153 \quad \frac{377}{233} = 1,618$$

$$\frac{2}{1} = 2 \quad \frac{34}{21} = 1,6190 \quad \frac{610}{377} = 1,618$$

$$\frac{3}{2} = 1,5 \quad \frac{55}{34} = 1,6176 \quad \frac{987}{610} = 1,618$$

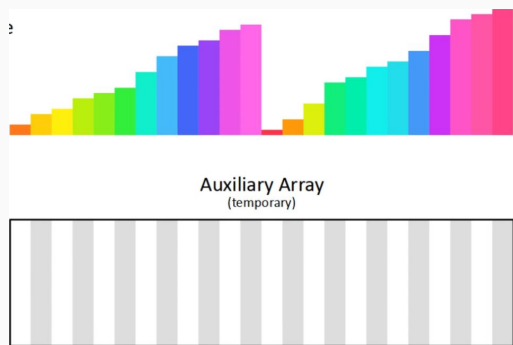
$$\frac{5}{3} = 1,666 \quad \frac{89}{55} = 1,6181$$

$$\frac{8}{5} = 1,60 \quad \frac{144}{89} = 1,6179$$

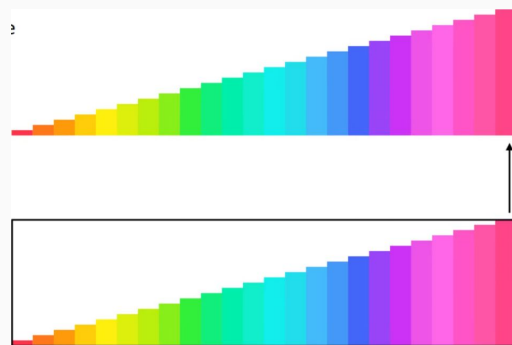
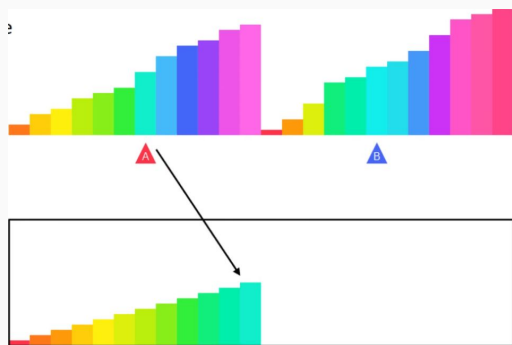
피보나치 수열은 n 이 커짐에 따라 이전
항과의 비가 황금비 ϕ (≈ 1.618)에 수렴한다.

2-3. 병합

<오리지날 merge sort가 두 run을 병합하는 방법>



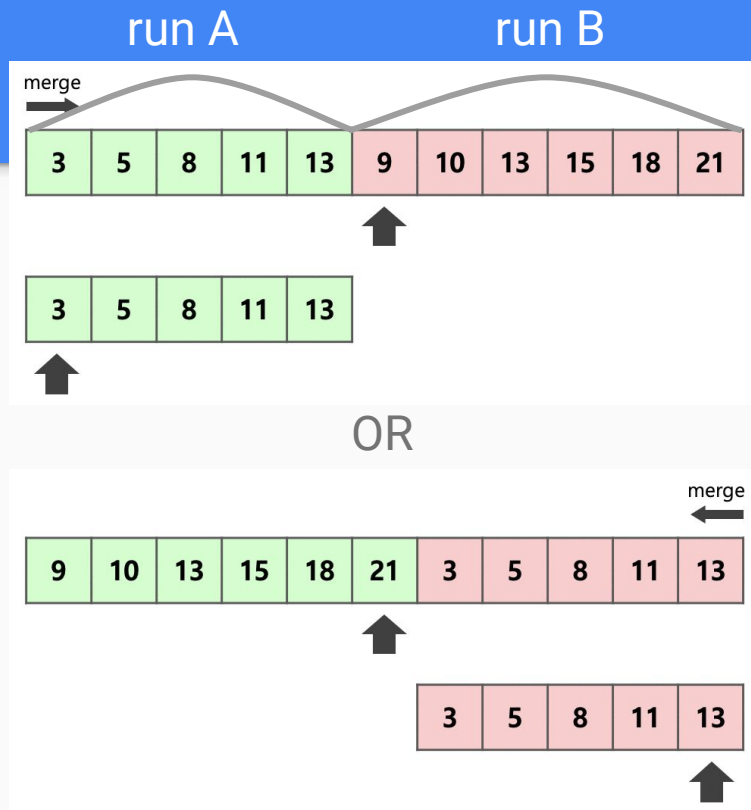
임시 열(Auxiliary Array)
생성



2-3. 병합

최적화 기법 1: run 두 개 중 하나만 복사하기

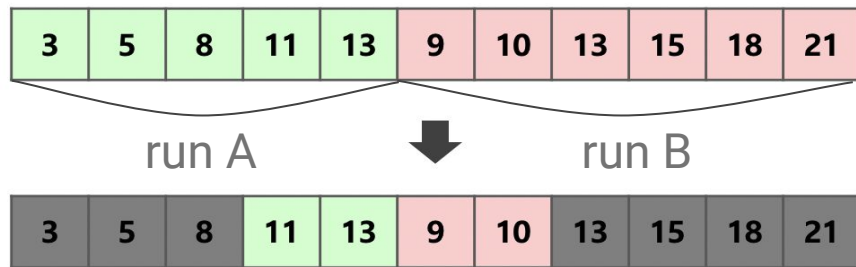
- run A, B 중 더 길이가 짧은 run을 선택
- 증가하는 run이면 시작 부분에,
감소하는 run이면 끝부분에 포인터를 두고
복붙
- 오리지날 mergesort에서 n 의 추가 메모리가
필요했다면, 이렇게 하면 최악의 경우에도
 $[n/2]$ 의 추가 메모리밖에 들지 않음



2-3. 병합

최적화 기법 2: 병합할 필요 없는 부분 생략하기

- run A의 맨 앞 원소 {3,5,8}은 run B의 맨 앞 원소 9보다 작기 때문에 병합하지 않고 현재 위치에 있어도 문제가 되지 않음

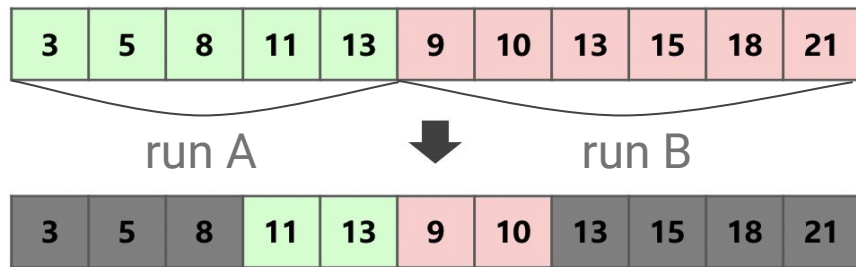


- 마찬가지로, run B의 맨 뒤 원소 {13, 15, 18, 21}은 run A의 맨 뒤 원소 13보다 같거나 크기 때문에 현재 위치에 있어도 문제가 되지 않음
- 즉 [11,13], [9,10] 두 부분만 병합하면 됨

2-3. 병합

최적화 기법 2: 병합할 필요 없는 부분 생략하기

- 이 필요 없는 구간을 찾는 과정은 **binary search**로 진행됨
- 단, 필요 없는 구간이 없을 수도 있기 때문에, 이 최적화 방법은 오히려 속도를 줄일 수도 있음



2-4. tim sort 구현

```
def binary_search(arr, val, start, end):
    if start == end:
        if arr[start] > val:
            return start
        else:
            return start + 1
    if start > end:
        return start

    mid = (start + end) // 2
    if arr[mid] < val:
        return binary_search(arr, val, mid + 1, end)
    elif arr[mid] > val:
        return binary_search(arr, val, start, mid - 1)
    else:
        return mid

def insertion_sort(arr, left, right):
    for i in range(left + 1, right + 1):
        temp = arr[i]
        pos = binary_search(arr, temp, left, i - 1)
        for k in range(i, pos, -1):
            arr[k] = arr[k - 1]
        arr[pos] = temp
```

```
def merge(arr, l, m, r):
    len1, len2 = m - l + 1, r - m
    left, right = [], []
    for i in range(0, len1):
        left.append(arr[l + i])
    for i in range(0, len2):
        right.append(arr[m + 1 + i])

    i, j, k = 0, 0, l

    while i < len1 and j < len2:
        if left[i] <= right[j]:
            arr[k] = left[i]
            i += 1
        else:
            arr[k] = right[j]
            j += 1
        k += 1

    while i < len1:
        arr[k] = left[i]
        k += 1
        i += 1

    while j < len2:
        arr[k] = right[j]
        k += 1
        j += 1
```

2-4. tim sort 구현

```
def tim_sort(arr):  
    n = len(arr)  
    min_run = 32  
    for start in range(0, n, min_run):  
        end = min(start + min_run - 1, n - 1)  
        insertion_sort(arr, start, end)  
  
    size = min_run  
    while size < n:  
        for left in range(0, n, 2 * size):  
            mid = min(n - 1, left + size - 1)  
            right = min((left + 2 * size - 1), (n - 1))  
  
            if mid < right:  
                merge(arr, left, mid, right)  
  
        size = 2 * size
```

sorting algorithm별 평균 실행시간 계산

sorting algorithm별 평균 실행시간 계산 (데이터 전처리)

```
# 티커 리스트를 가져오기
import pandas as pd
import random
import time
import FinanceDataReader as fdr

table = pd.read_html('https://en.wikipedia.org/wiki/List_of_S%26P_500_companies')
sp500_tickers = table[0]['Symbol'].tolist()
random.shuffle(sp500_tickers)
tickers = sp500_tickers[:10]

df_kospi = fdr.StockListing('KOSPI')
kospi_tickers = df_kospi['Code'].tolist()
random.shuffle(kospi_tickers)
tickers1 = kospi_tickers[:10]

tickers2 = (kospi_tickers+sp500_tickers)
random.shuffle(tickers2)
```

s&p500 및 코스피 상장 종목 ticker 가져오기

sorting algorithm별 평균 실행시간 계산 (bubble, tim, quick)

bubble_sort

```
def bubble_sort(x):
    length = len(x)-1
    for i in range(length):
        for j in range(length-i):
            if x[j] > x[j+1]:
                x[j], x[j+1] = x[j+1], x[j]

num_trials = 100000
total_time = 0
for _ in range(num_trials):
    start_time = time.time()
    bubble_sort(tickers2.copy())
    end_time = time.time()
    total_time += end_time - start_time

# 평균 시간 계산
average_time = total_time / num_trials
print(f"Bubble Sort Average Time: {average_time}")
```

tim_sort

```
num_trials = 100000
total_time = 0
for _ in range(num_trials):
    start_time = time.time()
    sorted(tickers2.copy())
    end_time = time.time()
    total_time += end_time - start_time

# 평균 시간 계산
tim1_average_time = total_time / num_trials
print(f"Tim Sort Average Time: {tim1_average_time}")
```

quick_sort

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2] # 피벗 설정
    lesser_arr, equal_arr, greater_arr = [], [], [] #배열 분할
    for num in arr:
        if num < pivot:
            lesser_arr.append(num)
        elif num > pivot:
            greater_arr.append(num)
        else:
            equal_arr.append(num)
    return quick_sort(lesser_arr) + equal_arr + quick_sort(greater_arr)

num_trials = 100000
total_time = 0
for _ in range(num_trials):
    start_time = time.time()
    quick_sort(tickers2.copy())
    end_time = time.time()
    total_time += end_time - start_time

# 평균 시간 계산
quick_average_time = total_time / num_trials
print(f"Quick Sort Average Time: {quick_average_time}")
```

sorting algorithm별 평균 실행시간 계산 (merge, radix, heap)

merge_sort

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

num_trials = 100000
total_time = 0
for _ in range(num_trials):
    start_time = time.time()
    merge_sort(tickers2.copy())
    end_time = time.time()
    total_time += end_time - start_time

# 평균 시간 계산
merge_average_time = total_time / num_trials
print(f"Merge Sort Average Time: {merge_average_time}")
```

radix_sort

```
def counting_sort(arr, place):
    n = len(arr)
    output = [0] * n
    count = [0] * 256

    for i in range(n):
        index = ord(arr[i][place]) if place < len(arr[i]) else 0
        count[index] += 1

    for i in range(1, 256):
        count[i] += count[i - 1]

    i = n - 1
    while i >= 0:
        index = ord(arr[i][place]) if place < len(arr[i]) else 0
        output[count[index] - 1] = arr[i]
        count[index] -= 1
        i -= 1

    for i in range(n):
        arr[i] = output[i]

def radix_sort(arr):
    max_len = max(len(s) for s in arr)
    for place in range(max_len - 1, -1, -1):
        counting_sort(arr, place)

num_trials = 100000
total_time = 0
for _ in range(num_trials):
    start_time = time.time()
    radix_sort(tickers2.copy())
    end_time = time.time()
    total_time += end_time - start_time

# 평균 시간 계산
radix_average_time = total_time / num_trials
print(f"Radix Sort Average Time: {radix_average_time}")
```

heap_sort

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[i] < arr[left]:
        largest = left

    if right < n and arr[largest] < arr[right]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapsort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

num_trials = 100000
total_time = 0
for _ in range(num_trials):
    start_time = time.time()
    heapsort(tickers2.copy())
    end_time = time.time()
    total_time += end_time - start_time

# 평균 시간 계산
heap_average_time = total_time / num_trials
print(f"Heap Sort Average Time: {heap_average_time}")
```

sorting algorithm별 평균 실행시간 계산

- S&P500 503개, 코스피 951개 상장종목 ticker를 무작위로 섞은 후 평균 정렬시간 계산

tim sort	quick sort	merge sort	heap sort	radix sort	bubble sort
0.32 ms	2.7 ms	4.6 ms	5.3 ms	6 ms	178 ms

- tim sort를 사용하는 것이 가장 효과적이라 판단 가능.

단기 매매에서의 **sorting** 활용 사례

kospi 주식들의 Ticker sorting

```
1 # Tim sort로 Ticker 정렬
2 import FinanceDataReader as fdr
3 import pandas as pd
4
5 df_krx = fdr.StockListing('krx')
6 print(len(df_krx))
7 kospi_vol=df_krx.loc[:,['Code','Name','Volume']]
8
9
10 top_alphabet_name = kospi_vol.sort_values('Name', ascending = True)
11
12 top_alphabet_name
```

2772

	Code	Name	Volume
1358	060310	3S	101862
950	095570	AJ네트웍스	75176
759	006840	AK홀딩스	3045
1156	054620	APS	33826
696	265520	AP시스템	22884
...
907	000540	흥국화재	49213
2751	000545	흥국화재우	1959
480	003280	흥아해운	694663
1329	037440	희림	349591
1809	238490	힘스	123704

2772 rows × 3 columns

```
1 # bubblesort 로 Ticker 정렬
2 import FinanceDataReader as fdr
3 import pandas as pd
4
5 df_krx = fdr.StockListing('krx')
6 print(len(df_krx))
7 kospi_vol=df_krx.loc[:,['Code','Volume','Name']]
8
9 c = bubblesort(list(kospi_vol.Name))
10 d=pd.DataFrame({'Name':c})
11 e = pd.merge(left = d, right = kospi_vol, on='Name', how = 'left')
12 e
13
```

2772

	Name	Code	Volume
0	3S	060310	101862
1	AJ네트웍스	095570	75176
2	AK홀딩스	006840	3045
3	APS	054620	33826
4	AP시스템	265520	22884
...
2767	흥국화재	000540	49213
2768	흥국화재우	000545	1959
2769	흥아해운	003280	694663
2770	희림	037440	349591
2771	힘스	238490	123704

2772 rows × 3 columns

단기매매에서의 sorting 활용과 그 중요성 (데이터 전처리)

```
import yfinance as yf
import pandas as pd
import datetime
from datetime import timedelta

# S&P 500 티커 목록을 가져옵니다.
table = pd.read_html('https://en.wikipedia.org/wiki/List_of_S%26P_500_companies')[0]
sp500_tickers = table['Symbol'].tolist()
sp500_tickers.remove('BF.B')
sp500_tickers.remove('BAK.B')

# 60일간의 이동평균 거래량
def get_average_volume(ticker):
    # 특정 티커에 대해 지난 60일간의 데이터를 가져옵니다
    data = yf.download(ticker, period="60d", interval="1d")

    # 60일 평균 거래량을 계산합니다
    average_volume = data['Volume'].mean()

    return average_volume

# 최근 거래량
def get_volume(ticker):
    data1 = yf.download(ticker, period="1d")

    volume = data1['Volume']
    return volume

average_volumes = {}
```

s&p 500 티커들의 60일간 이동평균 거래량과
최근 거래량의 비교

단기매매에서의 sorting 활용과 그 중요성 (데이터 전처리)

average_volumes

```
V: 0.000000,0.000000,
'REG': 1055583.3333333333,
'REGN': 458200.0,
'RF': 9471818.333333334,
'RSQ': 1109590.0,
'ROD': 1713173.3333333333,
'RVTY': 806030.0,
'RHI': 764630.0,
'ROK': 834960.0,
'ROL': 3085256.6666666665,
'ROP': 398425.0,
'ROST': 2262105.0,
'RCL': 3117536.6666666665,
'SPGI': 1125656.6666666667,
'CRM': 4644468.333333333,
'SBAC': 824813.3333333334,
'SLB': 8074660.0,
'STX': 2874408.3333333335,
'SEE': 1982848.3333333333,
'SRE': 2717795.0,
'NOW': 1120996.6666666667,
'SHW': 1407091.6666666667,
'SPG': 1556638.3333333333,
'SWKS': 1657616.6666666667,
'SJM': 1406913.3333333333,
'SNA': 226400.0,
'SFSP': 3684822.3333333335
```

volumes

```
UPS: Date
2023-11-10 5683900
Name: Volume, dtype: int64,
'URI': Date
2023-11-10 765600
Name: Volume, dtype: int64,
'UNH': Date
2023-11-10 2480800
Name: Volume, dtype: int64,
'UHS': Date
2023-11-10 528900
Name: Volume, dtype: int64,
'VLO': Date
2023-11-10 2965800
Name: Volume, dtype: int64,
'VTR': Date
2023-11-10 1795300
Name: Volume, dtype: int64,
'VLTO': Date
2023-11-10 1645300
Name: Volume, dtype: int64,
'VRSN': Date
2023-11-10 294200
Name: Volume, dtype: int64,
'VRSK': Date
2023-11-10 777900
Name: Volume, dtype: int64,
'VZ': Date
2023-11-10 12791200
Name: Volume, dtype: int64,
'VRTX': Date
2023-11-10 1315800
```

```
from collections import defaultdict
```

```
def merge_dictionaries(+dicts):
    result_dict = defaultdict(list)

    for d in dicts:
        for key, value in d.items():
            result_dict[key].append(value)

    return dict(result_dict)
```

```
dict1 = volumes
dict2 = average_volumes
```

```
merged_dict = merge_dictionaries(dict1, dict2)
```

```
print(merged_dict)
```

```
2023-11-10 7282900
Name: Volume, dtype: int64, 8799471.666666666], 'VICI': [Date
2023-11-10 9573700
Name: Volume, dtype: int64, 5447155.0], 'V': [Date
2023-11-10 4094400
Name: Volume, dtype: int64, 5428016.666666667], 'VMC': [Date
2023-11-10 839600
Name: Volume, dtype: int64, 860281.6666666666], 'WAB': [Date
2023-11-10 832300
Name: Volume, dtype: int64, 801620.0], 'WBA': [Date
2023-11-10 8793700
Name: Volume, dtype: int64, 12740750.0], 'WMT': [Date
2023-11-10 4773300
Name: Volume, dtype: int64, 5295555.0], 'WBD': [Date
2023-11-10 34526800
Name: Volume, dtype: int64, 21144675.0], 'WM': [Date
2023-11-10 1683880
```

단기매매에서의 sorting 활용과 그 중요성(전처리)

```
# 60일 이동평균 거래량 없는 ticker 제거
```



```
filtered_dict = {key: value for key, value in merged_dict.items() if len(value) == 2}  
print(filtered_dict)
```

```
columns = ['volume', 'average_volume']  
index = list(filtered_dict.keys())  
data = list(filtered_dict.values())
```

```
df = pd.DataFrame(data= data, columns= columns, index= index)
```

```
df['split(%)'] = (df['volume']/df['average_volume'])*100  
df_t = df.astype('float')
```

```
df_t
```

	volume	average_volume	split(%)	
MMM	2395000.0	3.626907e+06	66.034233	
AOS	846600.0	9.930550e+05	85.252076	
ABT	5479100.0	5.427838e+06	100.944421	
ABBV	4586400.0	4.694285e+06	97.701780	
ACN	1433500.0	1.907852e+06	75.136869	
...	
YUM	1317200.0	1.557532e+06	84.569709	
ZBRA	602300.0	4.731067e+05	127.307443	
ZBH	1965900.0	1.807695e+06	108.751753	

단기매매에서의 sorting 활용과 그 중요성(bubblesort 적용)

```
def bubblesort(data):  
    for index1 in range(len(data)-1):  
        for index2 in range(len(data)-index1-1):  
            if data[index2] > data[index2+1]:  
                data[index2], data[index2+1]=data[index2+1],data[index2]  
    return data
```

bubble sort 정렬을 이용하여
이동평균 거래량 대비 최근 거래량이 급등한
종목 찾기

```
c = bubblesort(list(df_t['split(%)']))  
print(c)  
d = c[::-1]  
print(d)  
e=pd.DataFrame({'split(%)': d})  
f = pd.merge(left = e, right = df_t, on='split(%)', how = 'left')  
  
f
```

```
[13.026136107800168, 13.350459992205957, 17.07836866047322, 17.6304  
[156.35706661870947, 140.3142516724666, 118.22984957165954, 112.412
```

	split(%)	volume	average_volume
0	156.357067	4444071.0	2.842258e+06
1	140.314252	784559.0	5.591442e+05
2	118.229850	1665709.0	1.408873e+06
3	112.412804	2220278.0	1.975111e+06
4	111.287493	5024337.0	4.514736e+06
...
496	18.123442	200353.0	1.105491e+06
497	17.630472	87602.0	4.968784e+05
498	17.078369	77889.0	4.560682e+05
499	13.350460	4876201.0	3.652459e+07
500	13.026136	370963.0	2.847836e+06

단기매매에서의 sorting 활용과 그 중요성(Tim sort 적용)

```
def tim_sort(arr):  
    n = len(arr)  
    min_run = 32  
    for start in range(0, n, min_run):  
        end = min(start + min_run - 1, n-1)  
        insertion_sort(arr, start, end)  
  
    size = min_run  
    while size < n:  
        for left in range(0, n, 2*size):  
            mid = min(n-1, left+size - 1)  
            right = min((left + 2*size -1), (n-1))  
  
            if mid < right:  
                merge(arr, left, mid, right)  
  
        size = 2 * size
```

Tim sort 정렬을 이용하여
이동평균 거래량 대비 최근 거래량이 급등한
종목 찾기

```
df_tim_sort = df_t.sort_values('split(%)', ascending = False)  
df_tim_sort
```

	volume	average_volume	split(%)
HD	4444071.0	2.842258e+06	156.357067
FRT	784559.0	5.591442e+05	140.314252
TRMB	1665709.0	1.408873e+06	118.229850
FTV	2220278.0	1.975111e+06	112.412804
ENPH	5024337.0	4.514736e+06	111.287493
...
DRI	200353.0	1.105491e+06	18.123442
IEX	87602.0	4.968784e+05	17.630472
AVV	77889.0	4.560682e+05	17.078369
KVUE	4876201.0	3.652459e+07	13.350460
PXD	370963.0	2.847836e+06	13.026136

501 rows × 3 columns

빠른 sorting 의
필요성

데이터 접근 속도 향상

알고리즘 트레이딩

시장 분석의 효율성