정렬 알고리즘 시간 분석 레포트

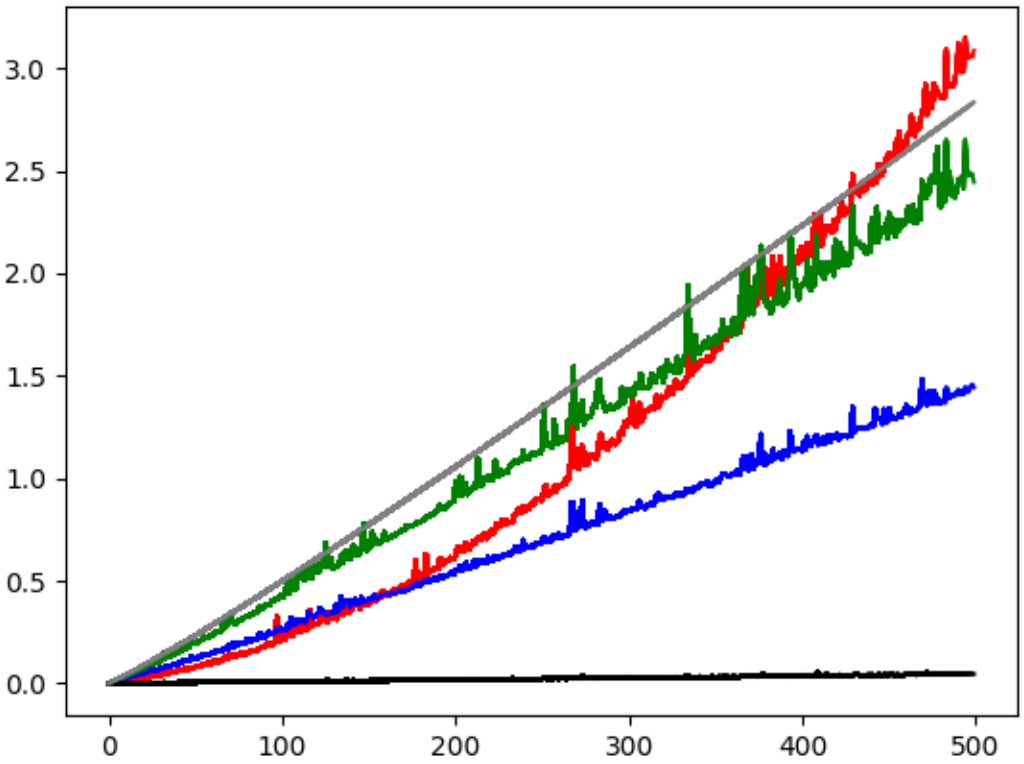
202202976 임준혁

주제

N개의 무작위 수들로 이루어진 배열을 정렬하는 알고리즘은 다양하게 존재하는데 그 알고리즘들은 수행시간에 차이가 존재한다. 알고리즘 별로 얼마만큼의 시간 차이가 벌어지는지 분석하고자 한다. 거기에 더해서 비교연산의 횟수, 교환의 횟수 또한 측정한다.

분석

우선 대표적인 알고리즘 3개(Quick, Merge, Heap)를 파이썬에서 구현한다. 세 정렬 알고리즘 모두 평균 O(n log n)의 시간 복잡도를 가지고 있다. N의 조건은 500,000개 이하로 가정했다.



** 빨간색: quick sort / 파란색: merge sort / 초록색: heap sort / 검은색: tim sort / 회색:* $3n\log_2 n$

그림은 N을 1에서 500,000까지 늘려 나갈 때 각 알고리즘별로 수행시간을 비교해 시각화한 그래프다.
여기서 $3n\log_2 n$ 의 그래프는 파이썬의 1초당 연산의 횟수를 10,000,000로 가정하고 그린 그래프이다.

Big-O 표기법의 한계

$n\log_2 n$의 그래프가 아니라 $3n\log_2 n$로한 이유는 Big-O 표기법의 한계 때문이다. $3000n$과 $\frac{1}{2}n$이 Big-O에서는 동일시 된다. 정확한 시간을 계산하는 것이 아니라 상수 계수를 무시하는데 이것이 오차를 만들어 낸다.

모두 O($n\log_2 n$) 임에도 파이썬 내장 정렬인 Tim sort는 거의 바닥에 붙어있다. 직접 구현한 알고리즘은 1.5초, 3.0초가 걸릴 때 Tim sort는 약 0.001초 안에 끝낸 결과다.

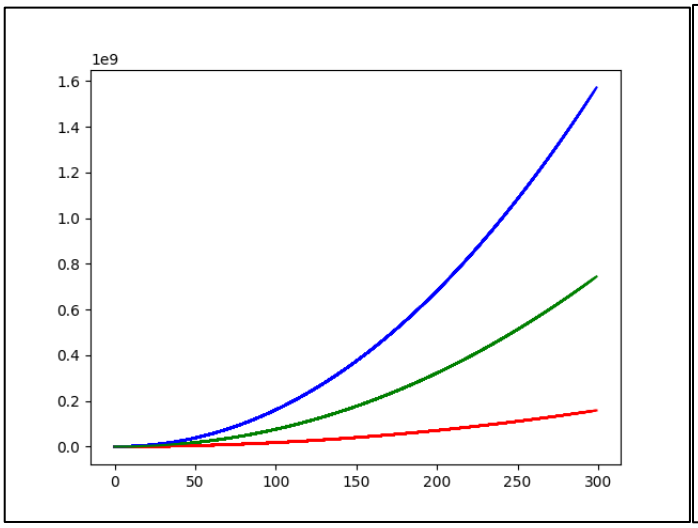
이 차이의 원인은 직접 구현한 정렬들은 Python 인터프리터로 실행하는 반면에 내장 정렬 함수(Tim sort)는 미리 컴파일이 완료되어 극도의 최적화가 적용됐기 때문에 큰 차이가 일어나는 것이라고 생각된다.

이것으로 같은 Big-O 의 O($n\log_2 n$)라고 해도 수천 배 차이가 날 수 있음을 알 수 있다.

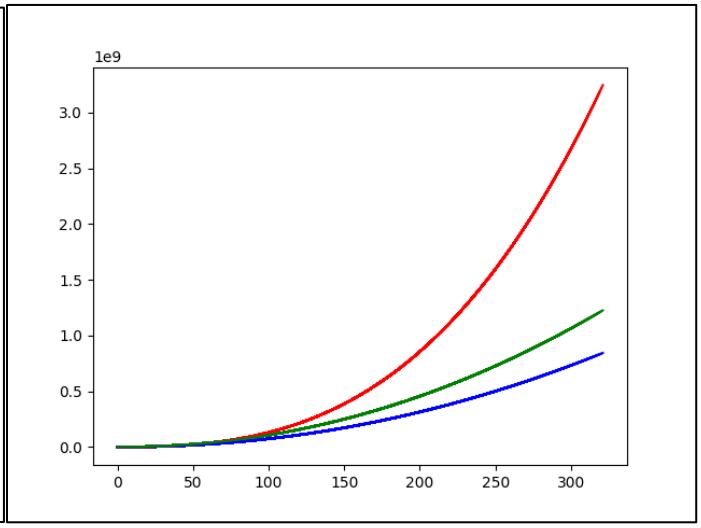
또한 직접 정렬을 구현하지 않고 일반적으로 Python 의 sort()함수를 믿고 사용해도 문제가 없겠다고 느꼈다.

비교-교환 횟수

1. Compare 횟수 (Qc, Mc, Hc)



2. Swap 횟수(Qs, Ms, Hs)



** 빨간색: quick sort / 파란색: merge sort / 초록색: heap sort*

(N 은 1 부터 300,000 까지 1,000 의 간격으로 300 번)

왼쪽 그림은 N 에 따른 비교 횟수를 저장한 Qc, Mc, Hc 의 그래프고 오른쪽은 교환(데이터 이동)의 횟수를 저장한 Qs, Ms, Hs 의 그래프다. 예상대로 Merge sort 의 교환 횟수가 가장 많았으며 비교는 가장 적게 했다.

그런데 여기서 의문이 생긴다. 일반적으로 비교보다는 교환이 더 시간이 많이 소요되는 작업이다. 예를 들면 파이썬 코드에서 A > B 보다 A, B = B, A 가 더 많은 작업을 필요로 하는 것과 같다. 그렇다면 페이지 1 의 분석한 파란색의 그래프, Merge sort 가 가장 높은 수치를(가장 많은 시간이 소요했음을) 보여야 하는게 올바르지 않냐는 생각이 들었다. 하지만 반대로 직접 구현한 알고리즘 중에서 Merge sort 가 가장 효율이 잘 나왔다. 그 이유가 오버헤드라고 생각된다.

오버헤드

프로그램을 실행할 때 작업을 수행하는 것 이외의 추가적으로 소비하는 비용을 오버헤드라고 한다. 재귀함수를 호출할 때 오버헤드는 함수를 호출하는 과정 및 복귀하는 과정에서 발생한다. 그렇기 때문에 예상치 못한 부가적인 비용(시간, 메모리 등)이 생겨 프로그램에 성능에 저하가 생긴 것으로 보인다. 재귀 함수의 호출이 깊어질수록 오버헤드가 증가하므로 이론적으로 예상했던 시간보다 더 오래 걸리는 현상이 나타난 것이다. **Merge sort 가 교환 횟수가 가장 많으므로** 시간도 가장 많이 소요될 거라고 예상했던 것이 틀린 이유가 예상치 못한 많은 오버헤드가 일어난 것으로 설명이 가능하다.

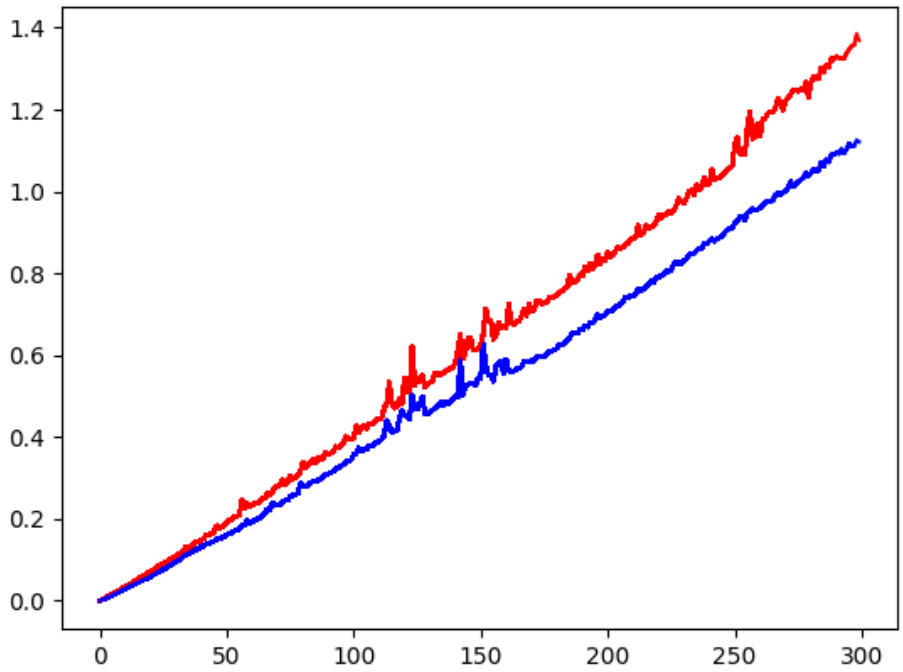
Merge sort 와 Quick sort 모두 재귀 호출을 이용해 구현했기 때문에 Big-O 표기법으로 표현은 가능해도 둘 중 어느 정렬이 더 빠르게 동작할지 알아내는 것은 측정해보기 전까지는 쉬운 일이 아닐 것이다.

그래프의 노이즈

시간 분석 그래프에 비해 비교-교환 그래프에서는 선이 큰 변동이 없이(노이즈 없이) 아래로 볼록한 곡선이 그려진다. 왜 이런 차이가 발생하는가에 대한 의문의 답은 어떤 값을 측정했는지에 있다. 첫 그래프는 현실 세계의 물리적인 실제 시간을 측정했다. 파이썬이 직접 실행하는 시간은 여러가지 요인의 복합적인 결과가 도출된다. CPU 의 성능을 포함한 개인 컴퓨터의 사양 차이, 위 문단에서 작성한 오버헤드, OS 의 차이 등 등이 시간 측정에 있어서 변수가 된다.

그에 반해 비교-교환 횟수는 논리적인 연산의 결과이기 때문에 입력 데이터가 동일하면 언제나 같은 값을 도출한다. 따라서 곡선이 기울기가 증가하며 늘어나는 꼴을 형성하게 된다.

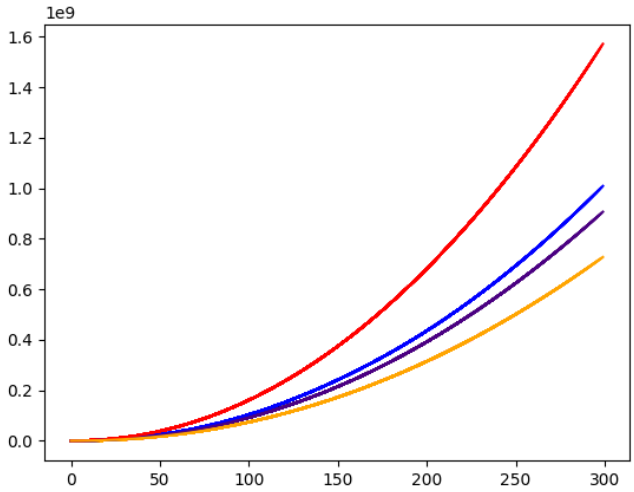
3-way Merge sort



** 빨간색: merge sort / 파란색: 3-way merge sort*

기존 Merge sort 는 중앙을 기준으로 좌우 두 개의 집단으로 계속 나누면서 정렬을 했다. 여기서 3-way, 세 개의 부분으로 나누면 어떨지 분석해 보았다. 그림의 파란색이 3-way merge sort 이고 두 개로 나눈 것보다 빠르게 동작했다. Mid1 = first + (last – first)//3, Mid2 = first+ 2*(last - first)//3 의 식을 통해 삼등분의 점을 두 개 구하고 First ~ Mid1 / Mid1 ~ Mid2 / Mid2 ~ Last 이렇게 세 구간을 재귀적으로 나눈다. 병합을 할 때에는
세개의 부분에 값이 존재하는 경우, 두개의 부분에만 값이 존재하는 경우, 한개의 부분에만 값이 존재하는 경우로 나누어 분할 정복해 구현할 수 있다.
Merge sort 보다 무슨 이유에서 빠르게 동작하는지 이해해 보려면 비교-교환 횟수를 알아야 한다.

3-way 비교-교환 횟수



Ms 와 Mc 는 기존과 동일하고
3Ms 는 3-way Merge sort 의 교환 횟수
3Mc 는 3-way Merge sort 의 비교 횟수를 의미한다.
세 부분으로 나눠서 병합 정렬을 하면 비교 횟수는 늘어나게 되지만 교환 횟수는 눈에 띄게 줄어든 모습을 확인했다. 지금까지의 결론을 통해 **교환 횟수가 확연하게 줄어들었기 때문에** Merge sort 에 비해 3-way Merge sort 가 더 빠르다는 것을 시각적으로 이해가 가능했다.

** 빨간색: Ms / 주황색: Mc / 파란색: 3Ms / 남색: 3Mc*