과제 4

각종 정렬 구현 및 소요시간 비교하기

20232907 정현승

목차

1. 문제 1: 사전 파일 정렬하기

C언어와 Python 각각 사전 파일 데이터를 삽입 정렬, 퀵 정렬, 힙 정렬을 이용해 정렬하고 이 소요 시간을 화면에 표시한다.

2. 문제 1 해결 방안

먼저 정렬을 시행하기 전 정렬할 데이터를 읽어 배열에 저장해야 하는데, 이 과제가 과제 3에서 이어지는 과제인 만큼 데이터를 읽는 부분은 과제 3의 코드를 그대로 가져왔다. 이번에는 연결 리스트가 아닌 배열을 사용하기 때문에, C 언어 코드는 다형성 구현 부분을 삭제하였으며, Python 코드는 각 단어를 list에 담도록 코드를 약간 수정했다. 또 Python에서 코드를 쉽게 작성하기 위해, 클래스의 대소 비교를 진행해 주는 매직 메서드도 작성했다[1](매직 메서드는 그전까지 그 존재를 모르고 있었기 때문에, 인터넷에서 메서드 이름이나 형식을 찾아 작성했다).

처음 코드를 작성할 때 강의자료에 나온 코드에서 할 수 있는 여러 최적화를 진행하여 코드를 작성하려 했으나, 사용해 보려 했던 최적화 방법이 대부분 실현 불가능한 방법임을 알게 되어 평범하게 코드를 작성했다. 코드를 전부 작성하고 보고서를 작성하는 현재는 그때 떠올렸던 방법이 별로 생각나지도 않는다. 특히 이전의 과제와 달리 강의자료 코드를 보면서 코드를 작성했으므로 어느 정도는 비슷한 부분이 있을 수 있다. 그래도 약간의 최적화를 진행하긴 하였으므로 강의자료의 코드와 다른 부분도 존재한다.

C언어에서는 정렬 함수를 만들기 전 단어의 대소 비교를 진행하는 부분을 따로 떼어 매크로 함수로 작성했다. 이 매크로 함수는 코드의 맨 윗부분이 아니라 이 코드가 필요한 시작 지점 바로 위에 선언하였다. 다만 함수의 반환값이 음수, 0, 양수로 나누어진 형태가 아니라, 뒤에 들어오는 단어가 더 큰가(뒤로 가야 하는가) 아닌가만 판단하도록 했다. 어차피 같은 단어가 들어오지 않기 때문이다. Python에서는 위에 언급한 대로 매직 메서드를 만들어 주었다. 처음에는 __cmp__ 메서드 하나만 만들었다가, 대소 비교 시 이 메서드를 Python이 인식하지 못하는 것을 발견해, 비교와 관련된 모든 메서드를 만들어 주었다. 따라서 대소 비교는 별도의 메서드 없이 부등호만으로도 가능하다.

또한 이번에는 정렬이 잘 되었는지 확인하는 코드도 작성하였다. 이는 작성한 코드에 이상이 없는지 확인하는 용도로 사용하였으며, 이 코드에서 문제가 있다고 하지 않는 한 정렬은 문제없이 된 것으로 해석할 수 있다(물론 이 코드가 정상 동작하는지도 확인해야 한다).

---

[1] 윤영필, 「[번역] 파이썬 매직 메소드 (Python's Magic Methods)」, 『Github』, 2017.1.30 (https://ziwon.github.io/post/python_magic_methods/)

이후 설명에서 '큰 데이터' 또는 '작은 데이터' 같은 단어가 등장할 수 있는데, 데이터의 크기를 직접 언급한 부분이 아니면 '큰 데이터'는 뒤에 와야 하는 단어, '작은 데이터'는 앞에 와야 하는 단어를 말한다. 이는 각 단어를 앞 글자부터 ASCII 코드로 대소 비교를 한 결과와 같다.

먼저 삽입 정렬은 평범하게 앞에서부터 하나씩 데이터를 집은 후, 배열 앞 정렬된 부분을 뒤에서부터 하나씩 뒤로 옮기며 집은 데이터가 들어갈 위치인지 확인하고, 이번에 뒤로 옮겨야 하는 데이터가 집은 데이터보다 앞에 오는 상황이 발생하면 이 사이에 집은 데이터가 들어가야 하므로 뒤로 옮기는 작업을 중지하고 집은 데이터를 넣는다. 처음에는 데이터가 들어가야 하는 위치 계산을 따로 하고, 뒤로 옮기는 반복 작업을 따로 하는 비효율적인 방식을 사용하였으나 강의자료를 확인한 후 코드를 수정했다.

퀵 정렬은 문제 1에서 스택 오버플로가 발생하지 않았다. 그래서 강의자료에 나온 것처럼 평범하게 코드를 작성했으며, 이 코드를 다른 데이터가 사용하지 않는 Python의 코드는 추가 수정을 하지 않고 재귀 호출로 코드를 작성하였다. 시작과 끝 포인터를 두고 각각 중앙으로 이동하면서 (각각 포인터 또는 변수의 이름은 low와 high이다), 앞부분에는 pivot보다 작은 데이터, 뒷부분에는 pivot보다 큰 데이터가 오도록, 잘못 들어간 데이터 한 쌍씩 서로 변경해 가며 두 포인터가 만나도록 하는 방식이다. 이때 강의자료에 있던, 정렬 시 잘못 들어간 두 데이터 한 쌍을 서로 바꿀 때 포인터가 중앙으로 이동했는지(즉 매 swap 직전에는 서로 바꾸어야 하는 데이터의 포인터가 들어오지만, 이번에는 데이터를 바꿀 한 쌍을 찾은 게 아니라 작업이 끝난 상황이다) 확인하는 if 문을 없애기 위해, low와 high 포인터(값) 이동 작업을 while 문 밖에도 똑같이 만들어 두었다. 이후 맨 앞 pivot 데이터와 작은 값 중 마지막 위치에 있는 값을 서로 맞바꾸어 데이터를 pivot보다 작은 데이터, pivot, pivot보다 큰 데이터 3개로 나누어 pivot의 정렬을 완료하고, 작은 데이터, 큰 데이터 각각을 모아둔 부분 리스트의 단어가 2개 이상이면 함수 재귀 호출로 이를 반복했다. 물론 Python에서는 포인터 개념이 거의 없기 때문에 배열과 index를 함수에 넘기는 방식으로 포인터 정보를 대신하였다. 이때 Python 퀵 정렬은 처음에 리스트만 들어오고 index 정보는 넣어주기 어려울 것으로 판단하여, 퀵 정렬 과정을 list 하나를 인자로 받는 메서드와 리스트에 시작과 끝 index를 받는 두 메서드로 나누어 index를 받는 메서드를 사용하고, list 하나를 인자로 받는 메서드는 단순히 시작과 끝 index를 구해 다른 메서드를 호출하는 작업만 수행하도록 했다.

그러나 문제 2의 재정렬 과정에서 스택 오버플로가 발생했다. 따라서 C 언어 코드는 직접 스택을 구현하여 사용했다. 시작 지점과 끝 지점을 묶어 스택에 저장하고, 스택에서 시작 지점과 끝 지점을 꺼내 pivot보다 작은 데이터는 앞, 큰 데이터는 뒤로 가도록, Python에서 한 것처럼 진행했다. 이후 데이터의 중앙까지 그 작업을 완료하고 작은 값 중 마지막 위치에 있는 값과 맨 앞에 있는 pivot 값을 바꾸어 pivot의 정렬을 완료하는 것까지는 같으나, 작은 데이터, 큰 데이터 각각을 모아둔 부분 리스트의 단어가 2개 이상이라 함수 재귀 호출을 진행해야 할 때 함수 호출 대신 각 리스트의 시작 포인터와 끝 포인터를 스택에 push 하도록 했다. 스택에 있는 시작 지점과 끝 지점 묶음을 하나 꺼내 이 과정을 반복하며, 이 반복은 스택이 비게 될 때까지 한다. 스택이 비면

정렬이 완료된 것이다. 이때 pivot보다 작은 데이터와 큰 데이터를 나눈 결과 한쪽의 데이터 수가 1이면 스택에 push 작업이 일어나지 않는데, 이 때문에 스택에 가장 많은 데이터가 쌓이게 되는 사례는 2개 데이터/pivot 데이터/2개 데이터/pivot 데이터/2개 데이터/.../pivot 데이터/2개 데이터 이와 같은 방식으로 데이터가 쪼개지는 경우이다. 따라서 스택의 최대 길이를 사전 데이터의 길이에 1 더한 후 3으로 나눈 수만큼 잡았는데, 이는 위와 같이 스택에 가장 많은 데이터(qsortThread)가 쌓일 때 쌓이는 데이터(qsortThread)의 수이다. (pivot을 포함하면 3개 데이터(Word)당 1개의 데이터(qsortThread)가 쌓일 수 있음, 또 양 끝에 pivot 데이터(Word)가 없어도 상관없으므로 3으로 나누기 전에 +1을 진행함) 어떻게 되더라도 스택이 부족하지 않게 스택의 크기를 여유롭게 잡았다. 처음에는 스택이 아닌 큐로 구현하려 했으나, 코드만 복잡해지고 결국 결과는 같으니, 스택을 사용하게 되었다. 스택 배열의 이름에 큐가 들어가는 건 큐를 구현하려 했던 흔적이다.

힙 정렬은 강의자료에 있는 것처럼, 먼저 밑에 있는 트리부터 힙이 되도록, 데이터 교환으로 힙이 깨지면 큰 데이터를 위로 한 번씩 밀며 밑에 있는 트리부터 다시 힙이 되도록 작업한 후, 가장 큰 데이터 하나씩을 꺼내며 힙을 만드는 작업을 반복하는 방식을 사용했다. 강의자료의 힙 정렬은 배열의 index 0은 비워놓는다. 그러나 지금 사용하는 배열은 index 0이 비어있지 않으므로, 추가 작업이 필요하다. 따라서 C 언어에서는 배열의 시작 포인터 값을 1칸 뒤로 이동해 index 0이 비워지도록 했다. 즉 배열의 맨 앞에 더미 데이터를 넣는 것이다. 1칸 뒤로 이동하면 그 공간은 사용할 수 있는 공간이 아니기 때문에 그 데이터를 읽으면 에러가 발생하지만, 데이터를 읽지 않고 1 이상의 index 값을 더한 뒤 값을 읽으면 아무런 문제가 발생하지 않는다. 하다못해 데이터 크기가 커서 1칸 뒤로 이동시켰더니 포인터 위치 값이 0보다 작아져 오버플로가 발생했다 하더라도, index 값을 더해 데이터를 읽으면 포인터 위치 값이 8비트로 저장할 수 있는 최대 정수를 넘어 다시 오버플로가 일어나 올바른 값을 가리키게 되므로 문제가 없다. 이 점을 이용해 시작 포인터 값을 1칸 뒤로 이동하는 방법을 선택하였다. 코드를 수정해 index 0을 비워놓지 않고 이를 활용하도록 할 수도 있었으나, 그러면 child 노드의 index를 계산할 때나 각종 반복문 초기조건 계산 시 불필요한 덧셈과 뺄셈 작업이 추가되어 성능을 떨어뜨릴 것을 우려해 여기서는 시행하지 않았다. 그러나 Python에서는 포인터 개념이 없어 이런 행위를 할 수 없다. 그렇다고 데이터의 맨 앞부분에 더미 데이터를 넣어 해결하기에는 Python의 리스트는 연결 리스트도 아니라 48406개의 단어 데이터를 일일이 한 칸 뒤로 밀어야 하는 문제가 있다. 이런 방법은 성능에 큰 영향을 줄 것으로 생각해, 배열의 index를 1이 아니라 0부터 시작하도록 코드를 수정했다. 이 방법은 방금 C 언어에서 불필요한 덧셈과 뺄셈 작업이 추가된다고 사용하지 않은 방법이다. 그러니까 트리의 맨 윗부분부터 1, 2, 3,... index에 데이터가 저장되는 방식이었다면, 이를 0, 1, 2,...로 변경한 것이다. 배열의 index를 1이 아니라 0부터 시작하도록 코드를 수정할 때는 기존 코드에서 index 부분만 1을 빼 주는 방식으로 진행하니 그리 어렵지 않았다. 예를 들어 원래는 len(targetList) // 2부터 1까지 각각의 index를 root로 하여 adjust 작업을 시행하는데, 이를 len(targetList) // 2 -1부터 0까지로 바꾸어 주고(range(len(targetList)//2-1,-1,-1) 부분), root에 대한 child 데이터의 index는

childIndex=(root+1)*2-1=root*2+1, 그리고 이보다 1 큰 root*2+2가 된다(기존은 root*2과 root*2+1) 이런 식으로만 수정해 주면 힙 정렬 시 배열의 0번째 index 데이터를 버리지 않고 정렬을 수행할 수 있다. 그 대가로 child index 계산이나 반복문 조건 등에 +1 또는 -1 계산이 꽤 많이 들어가게 되었지만, 그래도 배열의 맨 앞에 더미 데이터를 넣는 것보다는 훨씬 성능이 좋을 것이기에 크게 신경 쓰지 않아도 된다. 물론 배열의 포인터를 직접 옮길 수 있어 1회 연산으로 배열의 맨 앞에 더미 데이터를 넣은 결과를 뽑아낼 수 있는 C 언어는 0번째 index를 활용하는 방법이 아닌 맨 앞에 더미 데이터를 넣는 것을 선택했다.

위와 같이 문제 1을 해결하였다. 그 결과는 다음과 같다.

3. 문제 1의 결과

아래 그림 3.1~그림 3.3은 C 언어의 정렬 확인 함수 작동을 확인한 것이다. 위 3가지 모두 if 문 내부로 들어가지 않았으며(즉 정상적으로 정렬된 것으로 확인함), 실제 단어를 확인해 본 결과도 문제없이 정렬된 것을 확인할 수 있다. 따라서 정렬 확인 함수 작동을 믿을 수 있고, 이에 따라 각 정렬 함수 디버깅 화면은 생략한다. Python도 정렬 확인 함수는 내용이 같기 때문에 정렬 함수 디버깅 화면은 생략한다(물론 보고서에 생략했을 뿐 코드 작성 시에는 디버깅을 전부 진행했다).

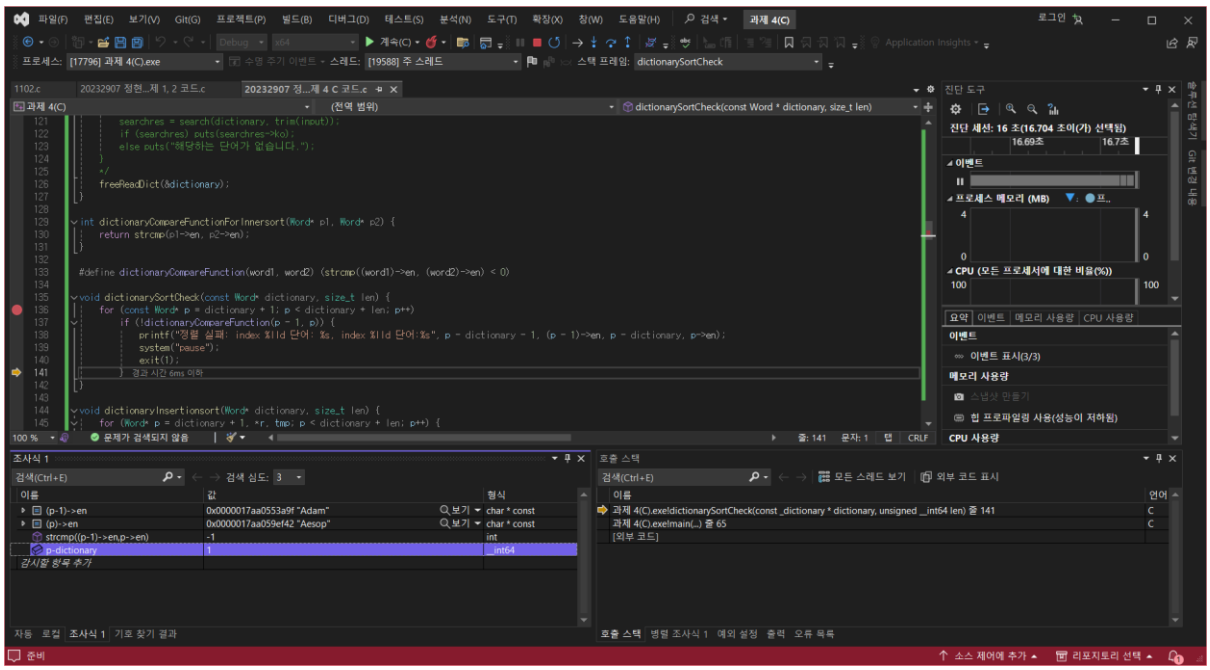

그림 3.1

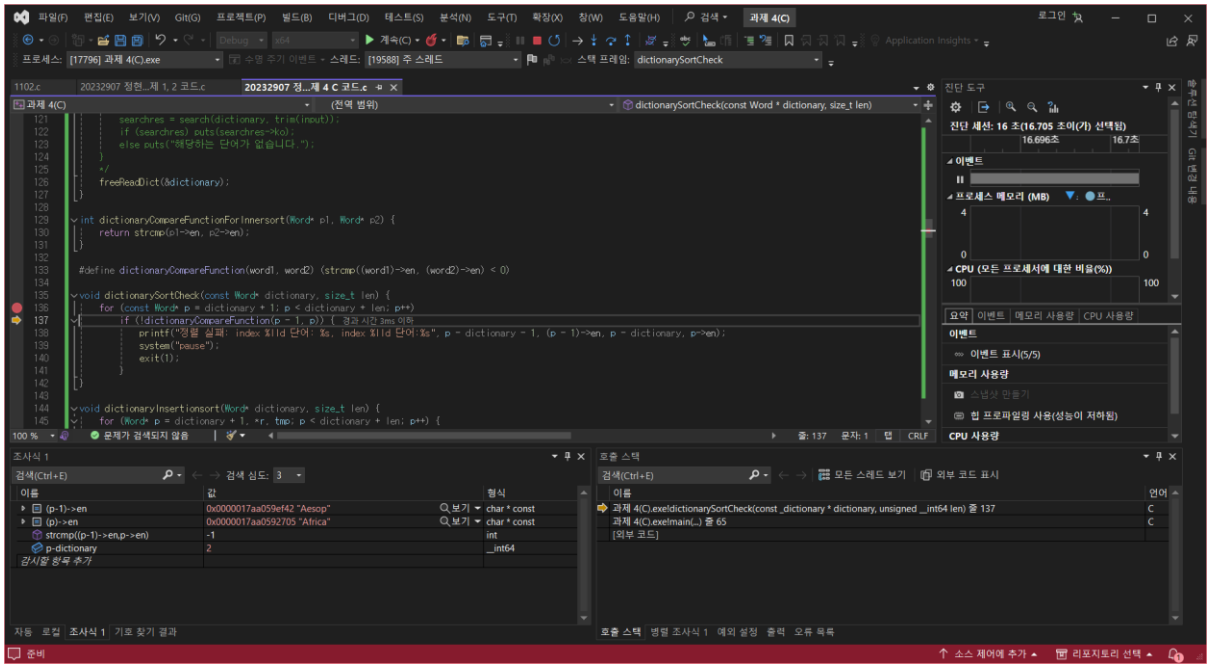

그림 3.2

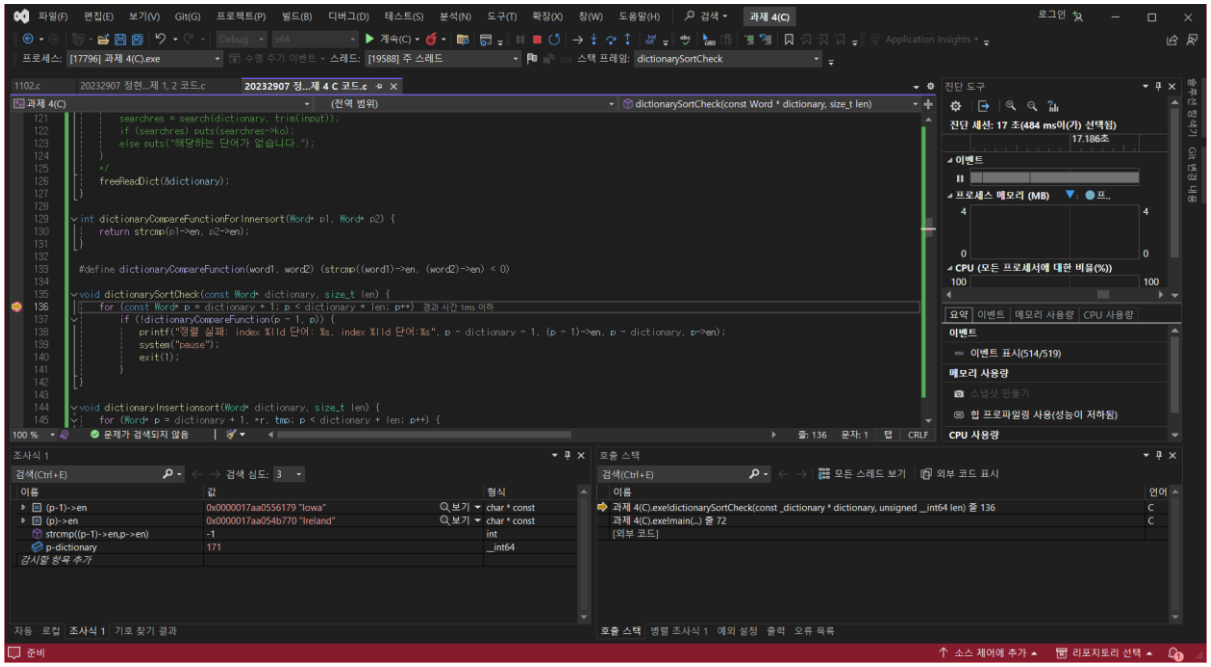

그림 3.3

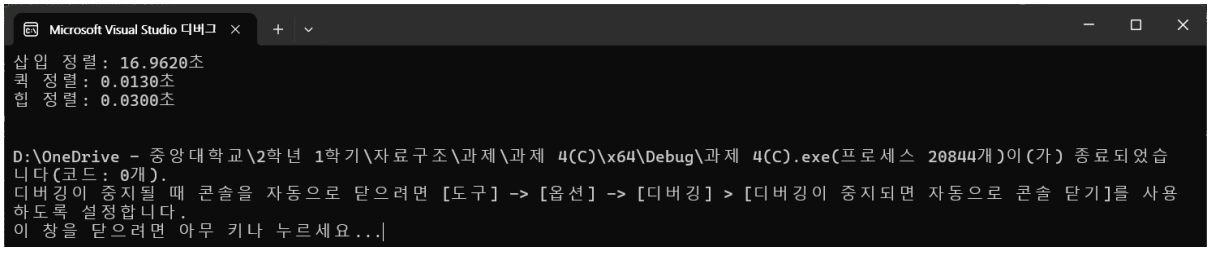

그림 3.4

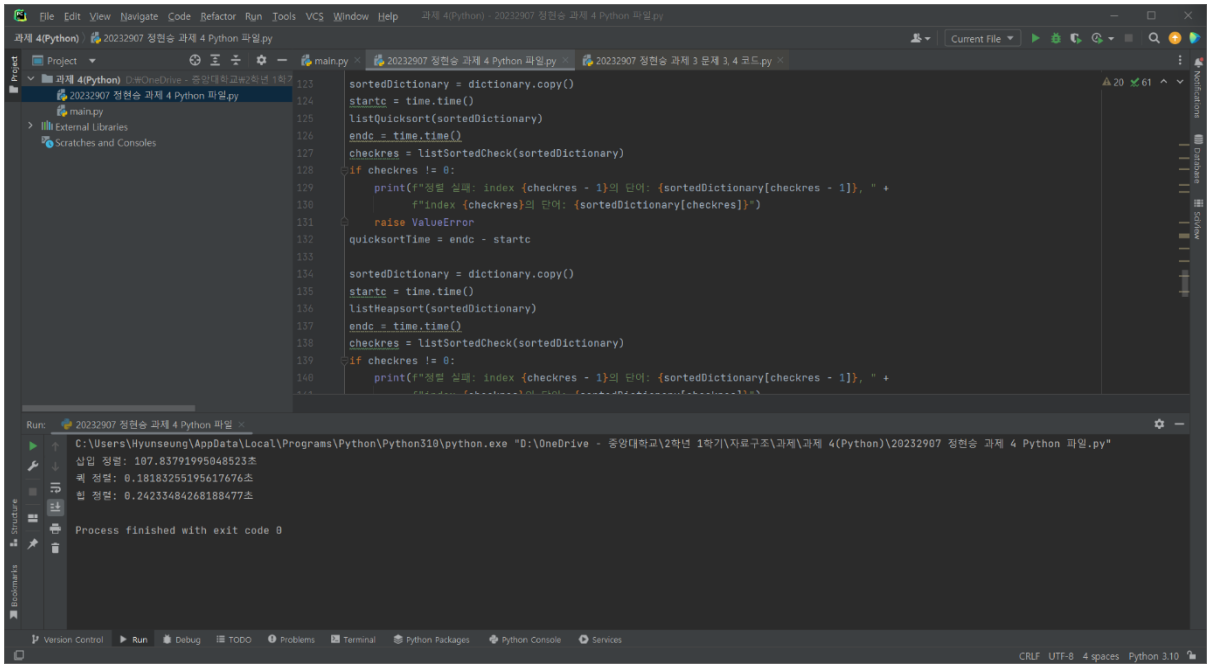

그림 3.5

그림 3.4는 C 언어의 결과, 그림 3.5는 Python의 결과이다. 정렬 확인 부분이 남아있는 상태에서 문제없이 결과가 출력되었으므로 정렬에는 문제가 없다는 점을 알 수 있고, 표시된 소요 시간을 보면 삽입 정렬이 조금 많이 비효율적이라는 점을 알 수 있다.

4. 문제 2: 정렬된 데이터 다시 정렬하기

C언어를 이용하여, 앞에서 이미 정렬된 데이터를 다시 정렬한다.

5. 문제 2 해결 방안

이 문제는 기존에 사용했던 코드를 다시 사용하면 되기에 크게 코드를 작성할 것은 없다.

6. 문제 2의 결과

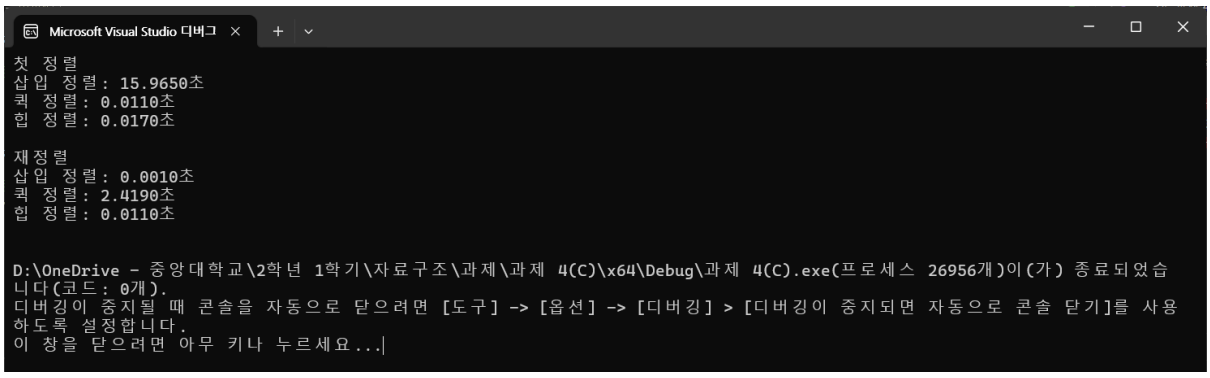

그림 6.1

위 그림 6.1은 첫 정렬을 진행하고, 다시 정렬을 진행했을 때 걸리는 시간을 측정한 결과를 나타낸 것이다. 코드의 정상 작동은 물론이고, 데이터가 정렬된 시점에서는 삽입 정렬이 다른 알고리즘보다 빠르다는 점과 퀵 정렬이 느리다는 점을 알 수 있다.

7. 문제 3: 내장 정렬 방법과 비교

문제 1에서 만든 정렬과 각 언어에 내장된 정렬 방법 각각에 걸리는 시간을 비교한다. 또 Python에서 더 빠르게 정렬하는 방법을 찾아 비교해 본다.

8. 문제 3 해결 방안

C언어는 내장된 qsort를 이용해 이에 걸리는 시간을 측정하는 방법을 사용했다. Python의 내장 정렬 함수는 sorted(list)와 list.sort() 2개가 있는데, 둘 다 사용하고 각각 시간을 측정했다. 이외에 Python은 더 빠르게 정렬하는 외부 라이브러리를 찾아보아야 하는데, Python을 개발한 사람이 직접 만든 정렬 알고리즘보다 더 빠르게 정렬하는 방법은 없을 것으로 생각하기는 했으나 일단 정렬을 지원하는 외부 라이브러리를 찾아보았다. 정렬을 지원하는 외부 라이브러리는 numpy[2][3]와 pandas[4][5]를 찾아보았으나 내장 정렬 방식보다 더 오래 걸렸다. 이외에도 몇 가지 외부 오픈소스를 찾아 테스트했으나 외부 라이브러리보다 내부 정렬 알고리즘이 가장 좋은 방법인 것으로 결론지었다.

9. 문제 3의 결과

---

[2] 조이 생각,「python 리스트 정렬, 정렬된 리스트 인덱스 가져오기(오름/내림차순)」, tistory, 2022.2.23 (https://ziwon.github.io/post/python_magic_methods/)

[3] 스택큐힙리스트,「NumPy 배열을 파이썬 리스트로 변환하세요.」, tistory, 2023.4.12 (https://stack-queue.tistory.com/350)

[4] Muhammad Waiz Khan,「Python의 빠른 정렬」, DelftStack, 2023.1.30 (https://www.delftstack.com/ko/howto/python/quicksort-python/#:~:text= Python의%20빠른%20정렬%201%20numpy.sort%28%29%20메소드를%20사용하여%20Python에서,정렬%20...%203%20Python에서%20빠른%20정렬%20구현%20)

[5] CosmosProject,「Python Pandas : to_list (Series를 list type으로 만들기)」, tistory, 2021.10.27 (https://cosmosproject.tistory.com/396)

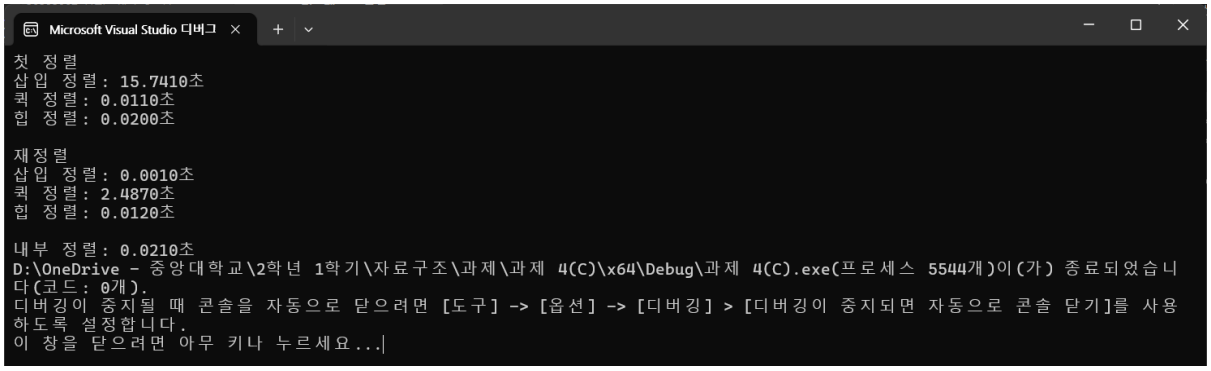

그림 9.1

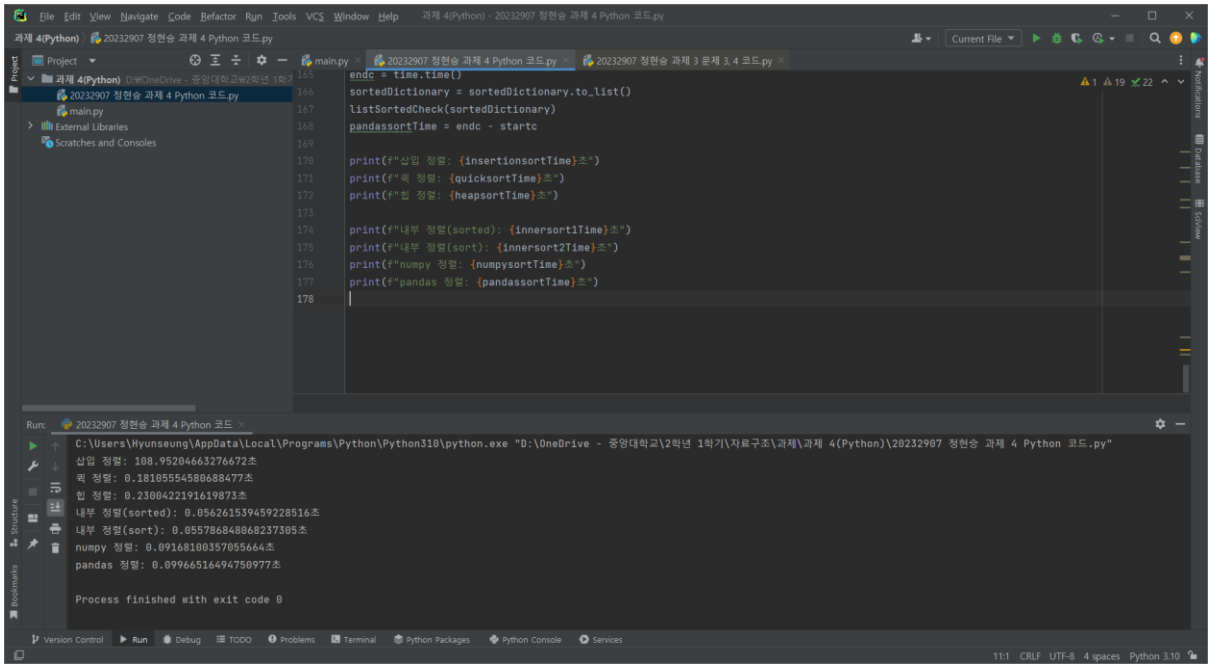

그림 9.2

위 그림 9.1은 C 언어에서 내장 정렬 방법과 문제 1에서 만든 정렬에 각각 걸리는 시간을 측정한 것이고, 그림 9.2는 그림 9.1의 작업을 Python에서 진행한 것이다. 위 결과를 통해 Python은 외부 라이브러리보다 내장 정렬 방법이 더 빠르지만, C언어는 그렇지 않다는 점을 알 수 있다.

10. 소스코드

첨부파일 참조

문제가 계속 이어지지만 3 문제의 코드를 같이 묶어서 하나의 프로그램으로 구성하라는 말씀도, 문제별로 따로 제출하라는 말씀 어느 쪽도 하신 적 없는 것으로 알아, 문제별 파일과 하나의 프로그램 2개 파일을 모두 제출합니다. 단 Python 문제 3 코드는 Python 전체 코드와 같고, 나머지 문제별 코드는 전체 코드와 다릅니다(C 언어 3번 코드도 다시 정렬하는 부분을 삭제하였기에 다시 정렬하는 부분이 남아있는 전체 코드와 다릅니다).