**알고리즘 설계와 분석(CSE3081) HW2**

**Quick Sort 방법의 효율적인 구현**

담당 교수: 임인성

이름: 남기동

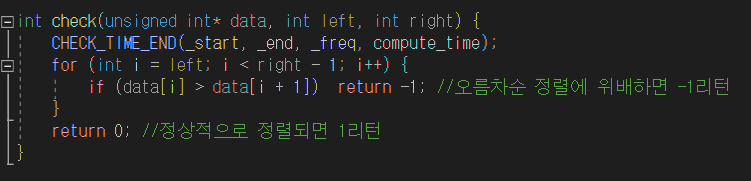
학번: 20180032

**1. 구현**

우선 6개의 함수 각각에 대한 구현을 설명하기 이전에 6개의 함수에서 공통적으로 가지고 있는 형식, 내용에 대해 언급한다. 이 부분 역시 HW2의 요구 사항에 맞게 작업을 수행하기 위한 형식이기 때문에 미리 언급한다.

텍스트이(가) 표시된 사진

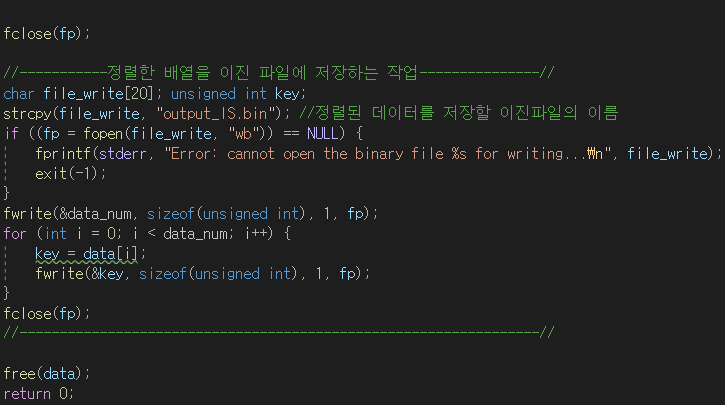
자동 생성된 설명



각각의 함수는 main함수를 갖고 있으며 파일을 이진 파일에 저장되어 있는 데이터를 읽어서 정렬하고 수행 시간을 출력하며 정렬된 값들을 이진 파일을 새로 만들어 저장까지 해야 하기 때문에 위와 같은 구조를 갖는다.

매크로로 설정된 파일명을 갖고 있는 파일이 동일 디렉토리에 위치했다고 가정하며 해당 파일을 이진 데이터를 읽는 모드로 열기 위해 wb 옵션을 사용했다. 그리고 해당 파일들의 첫 번째 4바이트에는 해당 파일에 저장된 데이터의 개수가 저장되어 있기 때문에 fread로 첫 번째 데이터를 data\_num으로 저장한다. 그 이후부터 4바이트 단위로 데이터를 하나씩 읽어서 data 배열에 차례대로 저장한다.

그 이후 각각의 함수에서 설정한 정렬 함수를 사용하는데, 정렬 함수가 시작하기 바로 전에 CHECK\_TIME\_START를 이용하여 시간 측정을 시작한다. 각각의 정렬 함수는 data에 저장되어 있는 데이터를 정렬한 후 함수에서 나오기 이전에 check를 통해 값이 제대로 저장되어 있는 것인지 검사를 해야 한다. 만약 값이 제대로 정렬되지 않았다면 오류 메시지를 출력해야 하기 때문이다. 따라서 시간 측정이 종료되는 시점은 main함수에서 정렬 함수가 끝나고 main함수로 돌아오는 시점이 아니다. 정렬 함수의 정렬이 모두 끝나고 마지막 반환값(status)에 적절히 정렬되었는지를 판단하기 위해 check함수로 넘어간 순간 시간 측정이 종료되어야 한다. 그래야 정렬 함수가 정렬을 수행하는데 걸린 시간을 비교적 정확하게 측정할 수 있기 때문에 위와 같은 방식을 사용했다. 위 코드의 하단에 위치한 주석 처리 부분은 정렬된 데이터를 출력하는 기능으로 데이터의 개수가 많아지면 출력하기 힘들기 때문에 주석 처리한 상태이다.



마지막으로 main함수를 종료하기 전에 정렬된 데이터를 output\_IS.bin이라는 이진 파일에 저장하기 위한 코드는 위와 같다. 모든 작업이 끝나고 난 이후에는 동적으로 할당해서 데이터를 저장했던 변수인 data를 해제해주고 종료된다.

뒤에서 설명할 6개의 함수는 모두 위와 같은 동일한 구조, 형식 위에서 작동한다. 다음은 각 정렬 함수에서 변하거나 추가된 부분만을 설명할 것이다.

**(1) Insertion Sort**

텍스트이(가) 표시된 사진

자동 생성된 설명

Insertion sort는 수업 시간에 나온 수업 자료를 참고하였으며 두 번째 자료부터 시작하여 그 앞의 자료들과 비교하여 삽입할 위치를 찾고 지정된 자리에 자료를 삽입하는 방식으로 정렬하는 알고리즘이다.

Insertion Sort의 시간복잡도는 Average Case에 O(n^2), Best Case에 O(n), Worst Case에 O(n^2)이다. 이번 숙제를 수행하는데 사용되는 데이터의 종료는 랜덤, 오름차순, 내림차순, few swap 이렇게 4가지인데, 오름차순으로 정렬된 데이터의 경우 Insertion Sort의 Best Case에 해당하며 내림차순으로 정렬된 데이터는 반대로 Worst Case에 해당한다. 오름차순으로 정렬된 데이터는 IS가 가장 앞에서부터 끝까지 탐색만 하고 삽입이 발생하지 않기 때문이며 반대로 내림차순은 가장 앞에서 끝까지 탐색할 때마다 이번 차례의 기준이 되는 데이터를 항상 맨 앞쪽에 놓아야 하기 때문이다.

**(2) Merge Sort**

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

Merge Sort는 중간 값을 찾아서 중간을 기준으로 왼쪽과 오른쪽으로 분할한 다음 분할한 데이터를 다시 재귀적으로 Merge Sort하는 방법으로 정렬을 하는 알고리즘이다. 분할한 것이 끝나고 나면 Merge 함수를 통해 다시 결합(Combine)하는 과정을 거치는데 이를 위해 코드를 보면, L와 R에 메모리를 동적 할당하여 좌측과 우측의 데이터를 가져오고 각각을 비교하면서 작은 값부터 차례대로 data에 다시 저장시키는 과정을 거친다. 그러다가 한쪽의 배열의 원소가 다 소진되면 마지막 while loop을 통해 남아있는 원소들을 data 배열 뒤쪽에 차례대로 집어넣는다. 이후 동적 할당했던 L과 R을 해제해주는 과정을 거친다.

Merge Sort는 Average, Best, Worst Case 모두 O(n \* log n)의 시간 복잡도를 갖는다. 그 이유는 앞서 보았던 Insertion Sort의 경우와 비교해보면 차이가 명확하다. Insertion sort의 경우에는 좌측에서 우측으로 데이터를 탐색하면서 정렬을 시도한다면 Merge Sort는 중간값을 찾아서 쪼개 나가면서 정렬을 시도하기 때문이다. 따라서 오름차순으로 정렬이 되어 있든, 내림차순으로 정렬이 되어있든 큰 차이 없이 모두 O(n \* log n)의 시간 복잡도로 정렬을 수행한다.

**(3) Quick Sort\_NAIVE**

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

Quick Sort는 분할 정복(Divide and Conquer) 방법을 이용하여 정렬하는 알고리즘 중 하나이다. 주어진 배열을 Pivot을 중심으로 작은 값은 Pivot의 왼쪽, 큰 값은 Pivot의 오른쪽으로 분할한 뒤, Pivot을 제외한 왼쪽 배열과 오른쪽 배열을 재귀적으로 정렬한다. 이 과정을 반복하면서 분할된 배열들의 크기가 작아지고, 최종적으로는 모든 배열이 정렬되는 구조이다.

Quick Sort는 Pivot을 어떻게 선택 하느냐에 따라서 성능이 크게 달라질 수 있다. 이번 과제의 핵심적인 목표이기도 한 것으로 이후 설계하는 QS\_P, QS\_PIS, QS\_PISTRO 모두 Quick Sort의 단점을 개선하기 위해 기능을 향상시킨 함수들이다. 따라서 Pivot을 랜덤하게 선택하거나 Pivot을 선택하는 과정에서 최대, 최소, 중간값을 선택하는 Median of Three 알고리즘 등을 이용하여 성능을 향상시킬 수 있다. 하지만 이번에 설계한 QS\_NAÏVE는 배열 데이터의 원소의 개수가 4개 이상일 때만 재귀적 함수를 호출하고 3개 이하이면 직접 정렬하는 구조를 갖는 동시에 Pivot을 가장 앞의 원소로 사용하게 된다. 따라서 Partition 함수에서 Pivot을 left로 설정하도록 했다.

Quick Sort는 일반적으로 O(n log n)의 시간 복잡도를 갖는다. 하지만 일반적으로 Quick Sort는 Pivot을 배열의 첫 번째 값, 마지막 값, 중간 값 등으로 선택하는데, 최악의 경우(정렬된 배열이나 역순으로 정렬된 배열 등)에는 O(n^2)의 시간 복잡도를 갖을 수 있다. 이번 과제에서도 오름차순으로 정렬된 배열, 내림차순으로 정렬된 배열 등이 사용되기 때문에 해당 경우 시간이 추가적으로 소요되는 것을 관찰해보아야 한다.

**(4) Quick Sort\_Pivot**

텍스트이(가) 표시된 사진

자동 생성된 설명

4번째로 구현한 Quick Sort\_Pivot은 Quick Sort에서 중요한 Pivot을 O(1)의 시간 복잡도를 갖고 있는 방법들 중 하나로 좋은 수행을 할 수 있도록 설정하는 것을 목표로 한다. 이번 과제에서는 수업 시간에서도 언급했던 기법으로 Median of Three 기능을 사용했다. Median of Three는 배열 데이터의 제일 좌측, 중간, 제일 우측 세 가지의 값을 조사하여 세 가지 데이터 중 중간 값에 해당하는 값을 Pivot으로 설정하는 것이다. Pivot이 한쪽으로 치우치게 되면 분할 후 정복을 하는 Quick Sort의 특성 상 효과적인 구조가 될 수 없기 때문이다. 따라서 이전에 구현한 Quick Sort\_NAÏVE와 다른 Partition함수 그리고 추가적인 MedianofThree 함수를 추가했다.

코드에 대해서 세부적으로 설명하면 다음과 같다. 먼저 Median-of-Three 기법을 사용하여 pivot을 결정한다. Partition 함수에서는 pivot을 오른쪽 끝의 값으로 설정하고, Partition 과정을 위한 i와 j 인덱스를 설정한다. while문에서는 i 인덱스를 증가시키면서 pivot 값보다 큰 값을 찾고, j 인덱스를 감소시키면서 pivot 값보다 작은 값을 찾는다. 그리고 i와 j가 교차할 때까지 서로의 값을 교환한다. 마지막으로 i 인덱스와 pivot 값을 교환하고, i 값을 반환합니다. MedianOfThree 함수에서는 Pivot 값을 오른쪽 끝에서 두 번째 값으로 설정합니다. 이를 통해 Partition 과정에서 pivot과 교환되는 값을 보호할 수 있다.

**(5) Quick Sort\_Pivot + Insertion Sort**

텍스트이(가) 표시된 사진

자동 생성된 설명

Quick Sort\_PIS는 4번에서 구현했던 QS\_P에서 Insertion Sort를 추가한 정렬 알고리즘이다. 또한 과제 명세서에서 언급되었듯, 배열의 원소의 개수가 M개 이상일 때에는 재귀 함수를 호출하고 이를 통해 불완전하게 퀵 정렬된 전체 데이터를 Insertion Sort 방법을 통해 한 번 더 정렬하는 방법을 취하고 있다. 나머지 부분들은 모두 앞에서 설명한 부분들을 포함하고 있기 때문에 이는 생략한다.

하지만 짚고 넘어가야 하는 것은 M의 값이 무엇인가이다. 이는 이번 과제의 명세서에서도 중요하게 언급되었기 때문에 이번 정렬 함수와 6번에 사용되는 정렬 함수에 적용될 적절한 M 값을 찾기 위해 적절한 크기의 데이터를 정렬하는 과정에서 M의 값을 변경시키면서 소요되는 시간을 비교하여 적절한 M의 값을 유추하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

우선 100을 잡았을 때 소요되는 시간이었으며 이로부터 값을 올리거나 내리면서 비교해보았다.

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

그러나 M의 값을 100보다 높이면 눈에 띄이게 수행 속도가 느려짐을 확인할 수 있다. 그리하여 M의 100 이하의 값을 대상으로 비교해보았다.

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

M의 값이 감소함에도 큰 차이는 보이지 않았지만 M의 값이 10 이하로 떨어지면 또 다시 수행 속도가 느려짐을 확인할 수 있었다. 그 외에 20, 30, 40, 50 등 약간의 값이 변화함에 따른 수행 속도 차이는 크지 않았기 때문에 이번 과제에서 사용할 M의 값은 50으로 설정하였다.



**(6) Quick Sort\_Pivot + Insertion Sort + Tail Recursion Optimization**

텍스트이(가) 표시된 사진

자동 생성된 설명

QS\_PISTRO는 이전 5번 정렬 함수에서 Tail Recursion Optimization 기능을 추가한 것이다. 이 역시 적절한 M 이상의 경우에만 재귀적 호출을 통해 분할 정복하는 Quick Sort의 방식을 가져가고, 불완전한 퀵 정렬 이후 전체 데이터에 대하여 Insertion Sort를 한 번 더 적용하여 배열을 정렬하였다. 코드에서 수정된 부분을 본다면, 여전히 Partition함수를 통해 Pivot을 정하고 이 Pivot을 정하는 방식 역시 Median of Three 방식을 채택하여 정렬해야 하는 범위의 좌측, 중간, 우측 값 셋 중 중간의 값을 갖고 있는 것을 Pivot으로 설정하긴 한다. 하지만 해당 Pivot이 전체 배열을 나누는 덩어리 중 작은 쪽으로만 Quick Sort를 재귀적으로 적용하는 것이 차이가 있다.

**2. 데이터 생성 및 시간 측정**

이번에 만든 6개의 정렬 함수의 시간을 측정하기 위해서 우선 데이터를 생성하는 과정이 필요하다. 데이터를 생성하는 데에는 과제에 첨부되었던 Test data generation함수를 사용하였고 랜덤 데이터, 정렬 데이터, 역순 데이터, 정렬 데이터를 기반으로 few swap을 거친 데이터 이렇게 4가지 종류의 데이터를 만들었다. 다음은 실제로 만든 데이터의 모습이다.

테이블이(가) 표시된 사진

자동 생성된 설명

데이터는 4개의 종류마다 2^10부터 2^20까지 11개씩의 데이터를 만들었다. 그리고 이를 활용하여 각 정렬 함수를 사용하여 시간을 측정하였고 시간 측정은 한 번에 5번씩 수행하여 나온 값들을 평균 내었다. 아래에는 각각의 정렬 함수를 통해 측정한 모든 데이터와 그 데이터로부터 도출된 평균값, 그리고 각 데이터 종류에 따른 수행 시간을 비교한 그래프를 첨부한다. 이에 대해 자세한 분석은 이후 4-4에서 추가적으로 분석할 예정이다.

테이블이(가) 표시된 사진

자동 생성된 설명 테이블이(가) 표시된 사진

자동 생성된 설명

테이블이(가) 표시된 사진

자동 생성된 설명 테이블이(가) 표시된 사진

자동 생성된 설명

테이블이(가) 표시된 사진

자동 생성된 설명 테이블이(가) 표시된 사진

자동 생성된 설명

**3. 환경**

OS: Windows 10 Education

CPU: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz

RAM: 16.0GB

Compiler: Visual Studio 2022 Release Mode

**4. 분석**

**1) 수행시간과 이론적 시간 복잡도 비교**

실제로 함수를 구현하고 데이터를 정렬하는데 시간을 측정 해보았기 때문에 이를 바탕으로 이론적인 시간 복잡도와 어떤 관련이 있는지 생각해볼 수 있다. 우선 이론적인 시간 복잡도를 이번 시간에 구현했던 함수 중 대표적인 정렬 함수인 Insertion Sort, Merge Sort, Quick Sort를 살펴보면 다음과 같다.

|  |  |  |  |
| --- | --- | --- | --- |
| Sorting Method | Best Case | Worst Case | Average Case |
| Insertion Sort | O(n) | O(n^2) | O(n^2) |
| Merge Sort | O(n \* log n) | O(n \* log n) | O(n \* log n) |
| Quick Sort | O(n \* log n) | O(n^2) | O(n \* log n) |

우선 Insertion Sort의 실제 데이터를 가지고 와서 데이터 크기의 변화에 따른 수행 시간의 변화가 연관성이 있는지 살펴보면 다음과 같다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data size | Random | Asecnding | Descending | Few Swaps |
| 2^10 | 0.505 | 0.003 | 1.288 | 0.008 |
| 2^15 | 419.615 | 0.053 | 872.517 | 0.196 |
| 증가분 | 830.920 | 17.666 | 677.420 | 24.5 |

데이터의 크기를 2^20은 RD, DE의 경우 너무 오래 걸리기 때문에 측정하는데 어려움이 있었다. 따라서 2^10부터 2^15 사이의 변화를 통해 시간 복잡도와의 관계성을 고려해본다.

데이터의 크기는 2^10에서 2^15으로 2^5배 증가하였다. 시간 복잡도의 기준으로 Best Case를 제외하고는 O(n^2)이기 때문에 2^5의 제곱인 2^10, 즉 1024배 증가하여야 한다. Insertion Sort의 Best Case는 앞에서도 언급했듯 Ascending이기 때문에 이 경우에는 O(n)에 따라 n배, 즉 32배 증가해야 한다. 표에 나타난 증가분을 보면 실제 생각한 것과는 약간 차이가 있지만 그래도 유사한 상승을 확인할 수 있다.

다음으로, Merge Sort의 실제 데이터를 가지고 와서 데이터 크기의 변화에 따른 수행 시간의 변화가 연관성이 있는지 살펴보면 다음과 같다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data size | Random | Asecnding | Descending | Few Swaps |
| 2^10 | 0.867 | 0.613 | 0.814 | 0.606 |
| 2^20 | 507.670 | 433.999 | 437.540 | 421.664 |
| 증가분 | 585.547 | 707.991 | 537.518 | 695.815 |

이번에는 데이터 크기를 2^10과 2^20 사이에서 발생한 수행 시간을 통해 비교하였다. 앞서 보았듯, Merge Sort는 모든 케이스에서 시간 복잡도 O(n \* log n)을 가지고 있는데 2^10에서 2^20으로 2^10만큼 증가하였고 이를 반영하면 1024(2^10) \* log(2^10)이고 이는 대략 3000쯤이다. 그러나 실제로 증가분을 보면 Random, Ascending, Descending, Few Swaps 사이의 증가분은 비슷하고 수행 속도도 비슷하지만 예상했던 증가분이 나오지는 않은 것을 확인할 수 있다. 이로부터 생각해볼 수 있는 점은 다음과 같다.

우선 첫 번째, Merge Sort는 어떤 상황에서도 시간 복잡도가 동일하기 때문에 이전의 Insertion Sort에서와 같이 데이터 크기의 변화에 다라 큰 수행 속도의 차이를 보이지 않았다는 점을 확인할 수 있다.

그러나 두 번째, 데이터의 크기가 커지면서 Merge Sort를 수행하는 과정에서 데이터를 L과 R에 복사한다거나, 재귀 호출에 의한 오버헤드가 발생하여 증가분이 예상만큼 나오지 않은 것이 아닌가 생각해볼 수 있다.

다음으로, Quick Sort의 실제 데이터를 가지고 와서 데이터 크기의 변화에 따른 수행 시간의 변화가 연관성이 있는지 살펴보면 다음과 같다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data size | Random | Asecnding | Descending | Few Swaps |
| 2^10 | 0.120 | 3.542 | 2.753 | 3.346 |
| 2^15 | 5.462 | 2022.678 | 1414.919 | 2034.047 |
| 증가분 | 45.516 | 571.055 | 513.955 | 607.904 |

이번에는 데이터 크기를 2^10과 2^15 사이에서 발생한 수행 시간을 통해 비교하였다. Quick Sort의 경우에도 Worst Case에는 O(n^2)의 시간 복잡도를 갖게 되는데 앞서 언급했듯, Quick Sort에게 Worst Case는 값이 정렬되거나 역순 정렬되어 있어서 Pivot을 이용한 Partition이 효과적이지 못할 때 발생한다. 이번 과제에서 사용하는 4개의 데이터 종류 중 Random을 제외하고는 모두 Worst Case에 해당한다고 볼 수 있다.

증가분을 살펴보면, Random 데이터의 경우 2^5 \* log(2^5), 약 48배에 가까운 45배 차이가 나는 것을 확인할 수 있다. 그러나 나머지 경우들에 대해서는 32 \* 32, 약 1000배 차이가 나야 하는데 그것보다 적은 차이가 나는 것을 확인할 수 있다. 이로부터 또 확인할 수 있는 것은 다음과 같다.

우선 첫 번째, worst case에 해당하는 세 가지 경우, 랜덤의 경우보다 역시 높은 시간 복잡도에 따라 수행 속도가 올라갔다는 것을 생각해볼 수 있다. Few Swap에 대해 왜 이 또한 worst case인가 생각할 수 있지만, 데이터의 크기가 높아질수록 정렬된 데이터에서 √n만큼의 교환이 있는 것은 기존의 nlogn이나 n^2의 증가분에 비해 크게 고려되지 않기 때문에 이 역시 worst case와 다르지 않다고 생각해볼 수 있다.

두 번째, 하지만 역시 예상되는 시간 복잡도와 유사한 차이가 실제 측정에서 발생하지 않은 것은 이번 과제에서 구현하고 데이터 측정에 사용된 함수인 Quick sort\_NAÏVE는 온전히 quick sort 방식을 채택하고 있는 것이 아니기 때문이다. 배열의 원소의 크기가 3 이하 일일 때 직접 정렬함으로써 재귀적 호출에 따른 비효율을 줄이는 방향으로 설계되었기 때문에, 예상되는 증가분보다 적게 측정된 것으로 볼 수 있다.

**2) MS와 QS 비교**

데이터의 크기를 변경시키면서 Merge Sort와 Quick Sort를 비교해보았을 때 두드러진 속도 차이를 발견할 수 있었다. 이를 확인하기 위해서는 실제 데이터를 측정하여 평균 값을 낸 테이블을 참고하기로 한다.

테이블이(가) 표시된 사진

자동 생성된 설명테이블이(가) 표시된 사진

자동 생성된 설명

두 테이블은 Merge Sort와 Quick Sort\_Naive에서 측정한 데이터들의 평균값을 정리한 것이다. 우선 두드러지게 차이가 나는 것은 정렬 데이터, 역순 데이터, Few Swap 데이터의 경우 QS\_NAÏVE는 2^18 이상의 데이터는 측정하기에 시간이 너무 많이 소요되어 측정하지 못했지만 MS는 충분히 가용적인 시간 안에 2^20 크기의 데이터까지도 정렬하는데 무리가 없었다는 사실이다. 이는 앞서 보았던 시간 복잡도의 측면에서 Quick Sort는 Worst Case의 경우 O(n^2)의 시간 복잡도를 가지는 반면 Merge Sort는 모든 경우 O(n \* log n)의 시간 복잡도를 갖기 때문이다. Random한 데이터, 즉 두 경우 모두 average case에 대해서는 둘 다 O(n \* log n)의 시간 복잡도를 갖지만 이 역시 앞서 이야기했던 Quick Sort가 적은 개수의 데이터는 직접 정렬을 하여 효율성을 증진했고 Merge Sort는 데이터 할당 등에 사용되는 시간이 추가되는 점 때문에 시간이 차이가 나는 것이라 생각된다.

**3) IS와 QS\_P 비교 (by using random data)**

테이블이(가) 표시된 사진

자동 생성된 설명 테이블이(가) 표시된 사진

자동 생성된 설명

Random으로 배열된 데이터를 활용하여 IS와 QS\_P의 수행 시간을 비교한 테이블은 위와 같다. 이를 계산하여 n / logn과 같은 비율이 측정되는가를 검사할 것이다. 왜 하필 n/logn인가는 IS의 경우 랜덤 데이터일 때 평상시의 케이스로 계산되어 O(n^2)의 시간 복잡도를 가질 것이다 반면 QS\_P는 Quick Sort의 일종이므로 O(n log n)의 시간 복잡도를 가질 것으로 예상되는데 이들 사이의 비율을 구하면, n^2 / n \* log n이고 이것이 곧 n / log n이기 때문이다.

2^13의 경우를 보면 30.253 / 0.815 = 37.12이고 이 경우 n / log n은 8192 / 3.9로 2100이다. 이는 굉장한 괴리가 있는 것인데 이를 다음과 같이 이해해볼 수 있다.

우선 QS\_P는 Quick Sort\_NAIVE보다 굉장히 우월한 수행 시간을 보여주고 있는 알고리즘이다. 이는 위의 테이블을 통해 이미 확인할 수 있는데, 그것은 앞서 말했듯 quick sort에서 굉장히 중요한 pivot을 효율적으로 선택하고 있기 때문이다. 그렇다면 IS의 수행시간을 동일한 사이즈의 데이터에서 QS\_NAÏVE를 비교해보겠다.

30.258 / 1.153이 되고 이를 계산하면 26.242가 된다. 이로써 기존에 이론적으로 비교해보았을 때 생각되었던 37배에 유사한 값이 도출되는 것을 확인할 수 있다. 이를 통해 QS\_P이 기존의 quick sort를 수행 시간 측면에서 많이 개선하고 있음을 확인할 수 있다.

**4) 데이터의 정렬된 방식에 따른 수행 시간의 차이**

데이터 정렬 방법에 따라 각각의 정렬 함수에서도 수행 시간의 차이를 확인할 수 있다. 이는 위에서도 보았던 평균값을 정리한 테이블을 참고할 수 있지만 그보다 더 시각적으로, 직관적으로 확인할 수 있도록 테이블로부터 산출된 그래프를 비교함으로써 확인해볼 수 있다.

Insertion Sort의 경우에는 오름차순으로 정렬된 경우와, 이를 바탕으로 few swap이 있는 데이터를 제외하고 모두 굉장히 높은 증가율로 수행 시간이 상승하는 것을 볼 수 있다. 이는 O(n^2)의 시간 복잡도를 갖는 경우로 볼 수 있으며 반면 오름차순과 few swap은 best case로 여겨지는데 O(n)의 시간 복잡도를 갖고 있어서 매우 적은 수행 시간을 소요하는 것으로 확인된다.

Merge Sort는 앞서 말한대로 모든 경우에 대해 시간 복잡도가 비슷하기에 위와 같이 4가지 종류의 데이터의 수행 시간을 나타내는 그래프가 겹치는 모습을 보여준다.

Quick Sort\_NAÏVE의 경우에는 Random한 경우에만 효율적으로 작동하며 나머지는 Worst Case에 해당하여 n^2의 시간복잡도를 갖기 때문에 위와 같은 개형이 그려지는 것으로 이해해볼 수 있다.

Quick Sort에서 변형된 위의 3가지 함수의 경우 모두 유사한 형태의 개형을 가지고 있다. 랜덤 데이터가 가장 많은 시간을 소요하며 내림차순이 그 다음, 그리고 오름차순과 few swap이 비슷한 형태이다. 앞서 말했듯, 데이터의 크기가 커질수록 오름차순과 few swap이 차이를 보이지 않는다는 특성이 여기서 한 번 더 확인되며, random 데이터가 가장 느린 수행 시간을 갖는 것은 다음과 같이 해석할 수 있다.

Quick Sort에서 발전된 함수들을 구현하는 과정에서 median of three를 통해 pivot을 결정하고 있다. 이는 함수의 좌측, 중간, 우측 중 중간에 해당하는 값을 pivot으로 가져올 수 있게 하는 것인데 random 데이터의 경우 그 값이 데이터 전체의 중간값과는 차이가 있을 수 있다. 그 데이터가 중간값과 차이가 클수록 quick sort의 효율적인 구현에 중요한 유사한 크기의 덩어리로 분할하는 것이 잘 이루어질 수 없어서 random 데이터의 경우 소요 시간이 비교적 높은 것으로 이해된다.

그러면 오름차순, 역순, few swap은 어떻든 중간에 위치한 값이 전체 데이터의 중간값이기 때문에 좋은 분할을 가져갈 수 있어서 수행 속도 면에서 큰 이점이 생기는데, 그럼에도 불구하고 내림차순이 수행 속도가 느린 것은 partition함수에서 데이터를 좌측부터 검사하는데 역순은 높은 크기가 큰 데이터가 좌측에 높여 있기 때문에 swap이 비교적으로 많이 발생하여 수행 시간이 느려지는 것으로 이해할 수 있다.

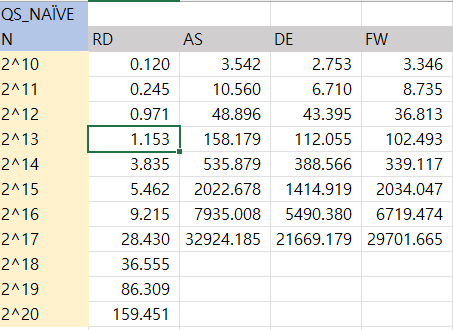
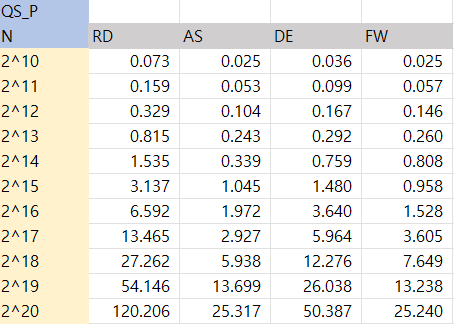
마지막으로 명세서에서 언급했던, Insertion sort 방법의 수행 시간이 quick sort 방법에 비해 그리 느리지 않은 데이터는 오름차순과 Few swap이다. 오히려 그리 느리지 않다기 보다는 오히려 매우 빠르다. 이는 이론적인 시간 복잡도의 비교에서도 드러나는데, best case의 경우 Insertion Sort의 시간 복잡도는 O(n)이기 때문에 O(n log n) 혹은 같은 케이스는 QS\_NAIVE에서 worst case로 O(n^2)이기 때문에 Quick Sort에 비해 매우 빠른 것이 당연하다.

**5) IS와 QS의 차이가 나지 않는 N의 값**

위에서 IS와 QS의 수행 시간을 비교한 대목을 이어서 설명하자면, best case에서는 IS의 속도가 훨씬 빠르다. 반면 다른 경우들에 대해서는 Quick Sort가 매우 빠르다. 그리하여 명세서에서 언급한 이 부분은 맥락상 normal한 케이스에 대한 보편적인 판단을 요하는 것으로 보이기 때문에 random 데이터를 기반으로 비교하였다. N의 값이 올라갈수록 Quick Sort가 빨라지는 것이 확인되었기 때문에 그리 크지 않은 N의 값을 활용하면서 비교하였고 대략 100 이하의 값에서는 유사한 시간이 소요되는 것을 확인하였다.

**6) 효율적 구현을 위한 아이디어**

효율적인 구현을 위해 노력한 부분은 Quick Sort Pivot에서 Median of Three를 도입한 것이다. 앞서 보았듯, QS\_NAÏVE에 비해 QS\_P가 굉장히 우수한 속도를 보이고 있으며 이후 추가된 함수들인 QS\_PIS, QS\_PISTRO에서도 모두 Median of Three를 사용하여 QS\_NAIVE 대비 높은 성능을 보여주고 있음을 확인할 수 있었다.

**7) 느낀 점**

이후에 구현한 기능들, 즉 배열의 원소가 적은 데이터의 경우에는 재귀적 호출로 인한 시간 손해를 보지 않기 위해 Insertion sort를 수행한다던가, Tail Recursion Optimization을 사용하는 등의 개선 사항이 전반적으로 수행 시간 측면에서 이점을 주고 있음을 확인했다. 하지만 이번 과제에서 측정에 사용한 데이터의 최대 크기가 2^20이었는데 이보다 더 큰 사이즈의 데이터들을 생성하고 비교함으로써 QS에서 성능 개선을 위해 추가된 기능들의 효율성을 측정하지 못해 아쉽다. 시험이 얼마 남지 않았고 시간 관계상 이만큼의 비교를 했지만 나중에 위에서 아쉬웠던 점을 탐구해보고 싶다.

확실히 단순히 이론적으로 시간 복잡도를 분석하는 것에서 그치지 않고 실제로 함수를 짜고 높은 크기의 데이터들을 직접 정렬하고 시간을 측정하는 과정에서 많은 공부를 할 수 있었다. 이론적 시간 복잡도의 차이와 다른 실제 수행 시간의 차이를 보고 어떤 부분들이 이러한 차이를 초래하였는지를 고민하는 과정에서 정렬 함수들을 사용하는 상황과 맥락에 따라, 어떤 의도를 가지고 정렬 함수를 사용하는가에 따라 서로 다른 선택들이 있을 수 있겠다는 생각이 들었다.