**[알고리즘 설계와 분석]**

**2023학년도 1학기 – HW2**



|  |  |
| --- | --- |
| **학번** | 20190785 |
| **이름** | 박수빈 |
| **과목코드** | CSE3081 |
| **분반** | 01 |
| **담당 교수님** | 임인성 교수님 |

**1. 기본 설정**

본격적인 실험 시작에 앞서, 실험에 사용할 컴퓨터의 CPU의 속도 및 메인 메모리의 용량과 같은 기본 정보를 먼저 확인한다. 실험하고자 하는 프로그램은 N\_ELEMENTS라는 이름으로 처리해야 하는 입력 데이터의 사이즈를 정의하고 있다. 본 실험에서 실행하고자 하는 N\_ELEMENTS의 최대 크기는 1,048,576(=2^20)으로, 약 1,048,576개의 unsigned int형 데이터를 컴퓨터가 처리할 수 있어야 한다. 프로그램 수행에 필요한 컴퓨터 메모리는 최소 32MB (=1,048,576 \* 32 bytes + 32bytes) 이상이다. 아래의 정보에 따르면 본 컴퓨터는 7.80GB의 메모리 사용이 가능하므로, 해당 프로그램 수행에 충분한 크기의 메모리를 장착하고 있음을 확인할 수 있다. 참고로, 컴파일러로는 채점 환경과 동일한 조건으로 맞추기 위해 Visual Studio 2022 Release Mode를 선택하였다.

장치 이름 DESKTOP-HUR7U9Q

프로세서 Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz

설치된 RAM 8.00GB(7.80GB 사용 가능)

장치 ID 96F26FC7-5BA8-44F8-BFF1-7DCE5CDD67D1

제품 ID 00325-81573-07053-AAOEM

시스템 종류 64비트 운영 체제, x64 기반 프로세서

컴파일러 Visual Studio 2022 Release Mode

텍스트이(가) 표시된 사진

자동 생성된 설명

또한, Visual Studio 2022에서 ‘스택 예약 크기’를 536,870,912(=2^29)으로 설정도 바꾸어 실험에 적합한 환경으로 조성하였다.

**2. 구현 방법**

본 실험의 주된 목적 중 하나는 여러 종류의 정렬 방법을 구현한 다음, 수업 시간에 배운 이론적인 시간 복잡도와 실제 프로그램 실행 시간과의 관계를 분석하는 것이다. 따라서, 정렬 방법을 구현하는 각각의 함수 코드는 다르지만, 전체적인 틀은 아래와 같다. 아래에서 서술하는 ‘n’은 input data size로, 앞서 언급한 N\_ELEMENTS와 동일하다.

2-1) Skeleton code

1. 바이너리 파일 읽기 (바이너리 파일의 첫 번째 원소는 입력 받을 데이터의 개수 기입)

2. 만일 읽을 바이너리 파일이 없다면 직접 데이터 생성, input binary file 생성

3. 바이너리 파일을 읽어서 버퍼에 원소들을 차례대로 저장

4. 정렬 함수 실행 시간 시작점 측정

5. 정렬 함수 실행

6. 정렬 함수 실행 시간 종점 측정

7. 정렬 함수 실행하는 데 걸린 시간 출력

2-2) Input data

(a) Random: 과제에서 제공한 Test\_data\_generation.cpp에 있는 코드 사용

(b) Ascending: 1부터 n까지 오름차순으로 정리된 데이터로, 아래의 for문을 통해 생성

텍스트이(가) 표시된 사진

자동 생성된 설명

(c) Descending: n부터 1까지 내림차순으로 정리된 데이터로, 아래의 for문을 통해 생성

텍스트이(가) 표시된 사진

자동 생성된 설명

(d) Few swaps: 과제에서 제시한 조건을 만족하는 데이터로, 아래와 같이 구현

텍스트이(가) 표시된 사진

자동 생성된 설명

2-3) Sorting function code

모든 정렬 함수는 어떠한 이유에 의해서든 정렬을 마치지 못한 경우에는 아래 사진과 같이 ‘-1’을 에러 코드로 반환하도록 설계하였다. 그리고 만일 정렬 함수가 -1을 반환하였을 경우에는 ‘The execution of (정렬 방법 이름) sort for (입력 데이터 크기) elements in (데이터 정렬 상태) order is failed.’와 같은 에러 메시지를 출력하도록 설정하였다.

텍스트이(가) 표시된 사진

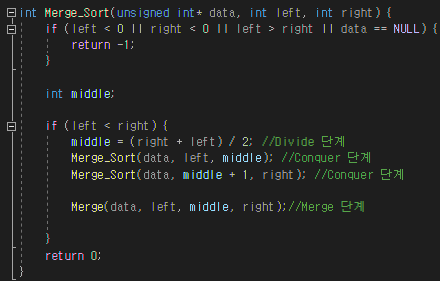
자동 생성된 설명

(a) Insertion sort

텍스트이(가) 표시된 사진

자동 생성된 설명

우선for loop은 검사 대상의 index를 나타낸다. 이 검사 대상과 비교할 대상이 while loop의 data[j-i]에 해당한다. 만일 검사 대상보다 비교 대상이 크면 검사 대상은 비교 대상의 앞에 삽입되어야 하므로, data[j]=data[j-1]이라는 구문을 통해 비교 대상이었던 것을 뒤로 밀어낸다. 이런 식으로 while loop을 돌다가 j가 1에 도달하거나 비교 대상보다 커지는 순간이 오면 while loop을 빠져나와 data[j]=temp라는 구문을 통해 본인이 해당 자리에 들어간다.

(b) Merge sort

합병 정렬은 데이터를 계속 나누다가 다시 합치는divide -and-conquer 방식을 사용한다. 중간에 위치한 원소를 기점으로 left부터 중간까지, 중간 그 다음부터 right까지, 이렇게 두 덩어리로 divide하여 recursion을 통해 conquer 단계를 거친 다음, Merge라는 함수를 통해 다시 합쳐진다.

Merge함수는 아래와 같다. 이 함수에 대한 자세한 설명은 line by line 주석으로 달려있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

간략하게 요약하자면, 입력 받은 데이터를 임시 버퍼에 복사한 다음, 임시 버퍼에서 비교 과정을 거쳐서 그 비교 결과를 원래의 데이터에 반영하는 식으로 작동한다. 비교하는 방법은 중간을 기준으로 왼쪽과 오른쪽을 나누어서, 왼쪽의 맨 왼쪽 원소와 오른쪽의 맨 왼쪽 원소에서 각각 시작하여, 둘 중 작은 것을 먼저 원래 데이터에 반영한 다음 차례대로 넘어가는 식이다.

텍스트이(가) 표시된 사진

자동 생성된 설명(c) Quick sort NAIVE

퀵소트는 피벗 원소를 하나 뽑아, 그 피벗 원소보다 작은 값들은 왼쪽으로, 큰 값들은 오른쪽으로 밀어 넣는 과정을 반복하여 정렬하는 방식이다. 이 정렬의 특이한 점은 Divide 단계와 Conquer 단계는 있지만, Merge 단계는 필요 없다는 점이다. 그리고 과제의 요구 조건을 충족시키기 위해, 입력 데이터의 크기(위의 사진에서는 변수’size’)가 3이하인 경우에는 직접 SWAP을 하여 조정하는 식으로 설계하였다. 이 퀵소트에서는 피벗 원소를 기준으로 원래의 데이터를 나눠주는 partition 함수가 매우 중요하다. 이 partition 함수에서 피벗 원소를 어떤 기준으로 뽑는지에 따라 퀵소트의 효율성이 결정되기 때문이다. 우선, NAIVE 버전에서는, 과제에서 요구하는 조건을 따라 피벗 원소를 단순히 맨 왼쪽 원소로 결정하고 나누는 방식으로 partition 함수를 작성하였다. 이 partition 함수에 대한 자세한 사항은 아래에 사진으로 첨부하였다.

텍스트이(가) 표시된 사진

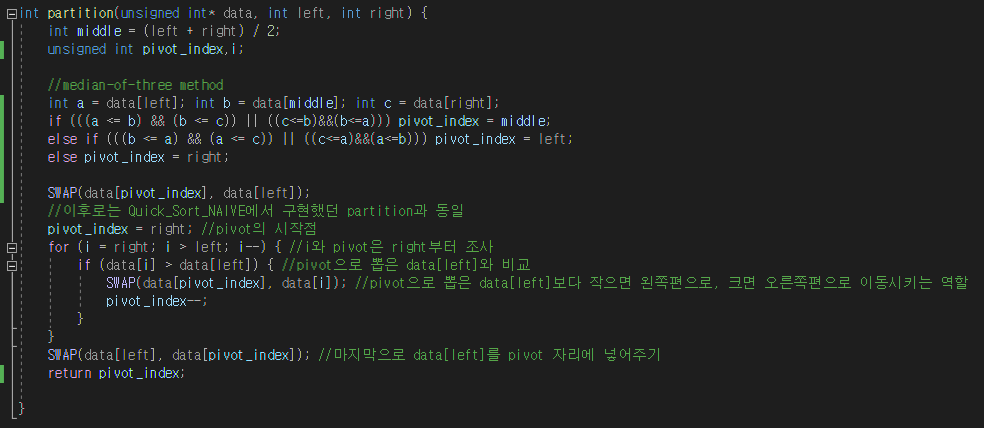
자동 생성된 설명

참고로, 위 사진의 주석들은 코드의 각 줄의 의미를 문법적으로 정확하게 해석하는 데보다는 전반적인 알고리즘을 이해하는 데 초점을 맞추어 작성되었다.

(d) Quick sort P

텍스트이(가) 표시된 사진

자동 생성된 설명피벗 원소를 (c)에서와 같이 무조건 맨 왼쪽 원소로 정하는 식으로 partition 함수를 작성하면, 맨 왼쪽 원소가 해당 입력 데이터의 최소값이거나 최대값인 경우 skewed 형태로 divide되면서 정렬의 비효율성이 극대화된다. 계속 skewed 형태로 나뉘어지는 최악의 경우, 시간 복잡도가 O(n^2)까지 되기 때문에 개선의 필요성이 보인다. 이에 따라, 완벽하지는 않지만 그래도 매번 최악의 상황이 되는 경우를 조금이라도 피하기 위해, partition 함수에서 피벗 원소를 left, middle, right 이 세 개의 값 중 중앙값에 해당하는 값을 피벗 원소로 정하는 median-of-three 방식을 선택할 수 있다. Quick\_Sort\_P 함수는 다른 부분은 모두 (c)에서 구현한 Quick\_Sort\_NAIVE와 동일하지만, divide 단계에 해당하는 partition 함수에서 median-of-three 방식으로 피벗 원소를 선택한다는 점만 다르다. 아래 코드는 이 바뀐 partition함수에 대한 코드이다. 세 값의 중앙값으로 피벗 원소를 정한 다음 data의 맨 왼쪽 원소와 SWAP함으로써, 그 이후에는 partition 함수를 NAIVE버전에서 더 수정할 필요없이 그대로 작성하여 보다 쉽게 구현할 수 있다.



(e) Quick Sort PIS

앞선 (d)번 문항에서 구현한 Quick Sort P는 NAIVE버전에서 partition 함수를 수정하여 시간 복잡도 측면에서 개선시킬 수 있었다. 퀵소트를 개선시킬 수 있는 또다른 방법이 이번 문항에서 구현할 Quick Sort PIS에 해당한다. 이 PIS버전은 사전에 정의한 숫자 M보다 크거나 같은 경우에만 퀵소트를 실행하고, M보다 작은 경우에는 퀵소트를 실행하지 않는다. 이러면 만일 퀵소트를 재귀적으로 실행하면서 심각한 skewed 형태에 도달하는 것을 방지할 수 있다. 이는 앞선 (c)와 (d)에서 구현한 퀵소트의 문제점을 해결해주는 역할을 한다. 그렇다면 M보다 작은 경우에는 정렬이 완료되지 않는다는 것을 의미하기도 하는 것인데, 이에 대한 설명은 아래 문단에 서술하도록 하겠다.

지난 (a)번 문항에서 Insertion sort를 구현했었다. 이 삽입 정렬은 데이터가 어느 정도 정렬되어 있을 때, 특히 검사 대상이 삽입될 위치가 멀지 않는 경우, 비교와 할당 과정 실행 횟수가 적기 때문에 매우 유용하다. 이와 같은 Best case의 경우, 비교 1회 당 할당이 1회 이어진다고 가정하면, 1+1+…+1 (비교 총 n회)= 1\*n 이므로, 삽입 정렬의 시간 복잡도는 O(n)까지도 낮아진다. 이러한 삽입 정렬의 장점을 이용하여 Quick sort PIS 버전을 만든 것이다. M이상이면 재귀적으로 퀵소트를 실행하여 대략적으로 정렬을 해놓고, 마지막에 Insertion\_Sort 함수를 호출하여 삽입 정렬로 정렬하면 정렬이 완료된다. 이로써 우리는 퀵소트의 문제점을 어느 정도 개선시킬 수 있는 것이다.

텍스트이(가) 표시된 사진

자동 생성된 설명이러한 Quick\_Sort\_PIS 함수는 (d)번 문항에서 구현한 Quick\_Sort\_P 함수에서 3이하인 경우에는 직접 SWAP하여 조정하는 구간을 빼고 M이하인 경우는 0을 바로 반환하는 식으로 바꾸고, 이 함수 다음에는 (a)번 문항에서 구현한 Insertion\_Sort 함수를 호출하는 구조로 구현하면 된다.

(f) Quick sort PISTRO

이전 (d)번 문항의 three-of-median 방식으로 피벗 원소를 고르고, (e)번 문항과 같이 M보다 작으면 재귀 호출을 하지 않고 마지막에 삽입 정렬로 정렬하는 방식과 더불어, ‘꼬리 재귀 최적화 (Tail Recursion Optimization)’ 방식도 추가하여 구현한 함수가 바로 이번 문항에서 구현할 퀵소트 PISTRO버전이다. 앞의 두 방식은 이미 이전에 서술하였으므로 생략하고, 꼬리 재귀 최적화 방식에 대해 아래 문단에 서술하도록 하겠다.

꼬리 재귀 최적화란, 함수의 재귀 호출 완료 이후에 현재 함수 내에서 추가적인 연산을 실행하지 않도록 하는 재귀의 형태를 말한다. 이 방식을 퀵소트에 적용하면, 다음과 같다. 피벗 원소를 뽑은 이후에는 피벗 원소보다 작은 값을 지닌 왼쪽 덩어리, 큰 값을 지닌 오른쪽 덩어리, 이렇게 두 덩어리가 생긴다. 덩어리의 크기가 크면 클수록 재귀 호출을 하는 횟수가 증가하므로, 피벗 원소를 뽑은 이후 덩어리가 두 개 생긴 이후에는 작은 덩어리 쪽으로 재귀 호출을 하면 system stack overflow를 방지할 수 있다. 덩어리 크기가 큰 쪽은 iteration으로 처리하고 덩어리가 작은 쪽은 recursive하게 처리하면 된다. 이와 같은 PISTRO버전은 아래 사진과 같이 코드를 작성하면 구현할 수 있다. 코드에 대한 자세한 설명은 아래 사진에서 주석으로 서술하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**3. 실험방법 및 결과 분석**

이전 2-1)번 문항에서 서술한 것과 같이 각 정렬 함수의 실행 시간을 측정하였다. 이렇게 실제로 실험에서 측정한 함수의 실행 시간과 이론적인 시간 복잡도를 비교하는 방식으로 실험을 진행하였다. 후술할 내용에서 n은 N\_ELEMENTS, 즉 input data size를 의미한다. 실험은 n의 크기를 달리 하여, 각 함수 별로 3번씩 수행하였다. 1번 실험은 n을 32(=2^5)로, 2번 실험은 n을 1024(=2^10)로, 3번 실험은n을 1048578(=2^20)로, 이렇게 3가지 값을 각 실험에서 넣은 다음 컴파일하여 출력된 데이터 생성 시간 및 함수 실행 시간 측정 결과를 토대로 결과 분석을 진행하였다.

(a) Insertion Sort

1) 이론적인 시간 복잡도

삽입 정렬의 이론적으로 구한 시간 복잡도는 아래와 같다.

|  |  |  |
| --- | --- | --- |
|  | Big-O | Example of input data order |
| Worst case | O(n^2) | Descending |
| Average case | O(n^2) | Random, Few swaps |
| Best | O(n) | Ascending |

2) 실험 결과 출력 화면

|  |  |
| --- | --- |
| Input data size | Result |
| 32 (=2^5) | 텍스트이(가) 표시된 사진  자동 생성된 설명 |
| 1024 (=2^10) | 텍스트이(가) 표시된 사진  자동 생성된 설명 |
| 1048576 (=2^20) | 텍스트이(가) 표시된 사진  자동 생성된 설명 |

3) 실험 결과 정리 표

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sorting function | Order of  input data | Execution time according to input data size (N\_ELEMENTS) | | |
| 32 (2^5) | 1024 (2^10) | 1,048,576 (2^20) |
| IS | Random | 0.003 | 0.765 | 33009.000 |
| Ascending | 0.001 | 0.008 | 4.888 |
| Descending | 0.003 | 2.418 | 1307699.375 |
| Few swaps | 0.002 | 0.086 | 145.419 |

합병 정렬은 하나의 검사 대상과 그 검사 대상의 앞에 있는 원소들과 차례로 비교해가며 정렬하므로, 뒤의 원소가 앞의 원소보다 항상 같거나 큰 ascending order에서는 가장 짧은 실행 시간이 나오고 반대로 뒤의 원소가 앞의 원소보다 항상 같거나 작은 descending order에서는 가장 긴 실행 시간이 나오는 것을 위의 실험 결과에서도 확인할 수 있다.

4) 실험 결과 정리 그래프

이론적인 시간 복잡도와 마찬가지로, 입력한 데이터의 개수 증가폭보다 실행 시간의 증가폭이 훨씬 큰 것으로 나타난다. n^2의 그래프와 유사하게 아래로 볼록인 것을 확인할 수 있다.

Ascending이 가장 실행시간이 짧고, Descending이 가장 실행시간이 긴 것을 확인할 수 있다. random과 few swaps은 그 둘 사이에 존재하는 값으로 average case에 속한다는 것도 알 수 있다.

5) 이론과 실험 결과 분석

|  |  |  |
| --- | --- | --- |
|  | Big-O | Example of input data order |
| Worst case | O((n^2)/2) = O(n^2) | Descending |
| Average case | O((n^2)/4) = O(n^2) | Random, Few swaps |
| Best case | O(n) | Ascending |

4가지 input data order 별로 각각 worst case, average case, best case에 속한다는 것을 앞선 그래프에서 확인하였다. 각 4가지 data order 별로 이론적인 시간 복잡도와 실제 실험 결과와 유사한지 수학적으로 계산하였다. 계산한 과정은 아래에 pdf로 첨부하였다. 계산 결과, worst case인 경우 이론적인 시간 복잡도와 거의 유사하게 (n^2)/2인 것을 확인할 수 있었다. best case의 경우 이론보다는 덜 걸리는 것으로, average case는 worst case만큼은 아니지만 대체적으로 이론과 비슷하다는 것을 확인할 수 있었다. 참고로, 이론과 비슷하게 나온 값은 같은 색의 형광펜으로 칠해놓았다.

텍스트이(가) 표시된 사진

자동 생성된 설명

(b) MS

1) 이론적인 시간 복잡도

|  |  |  |
| --- | --- | --- |
|  | Big-O | Example of input data order |
| Worst case | O(n\*logn) | Random, Ascending, Descending, Few swaps |
| Average case | O(n\*logn) | Random, Ascending, Descending, Few swaps |
| Best | O(n\*logn) | Random, Ascending, Descending, Few swaps |

2) 실험 결과 출력 화면

|  |  |
| --- | --- |
| Input data size | Result |
| 32 (=2^5) | 텍스트이(가) 표시된 사진  자동 생성된 설명 |
| 1024 (=2^10) | 텍스트이(가) 표시된 사진  자동 생성된 설명 |
| 1048576 (=2^20) |  |

3) 실험 결과 정리 표

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sorting function | Order of  input data | Execution time according to input data size (N\_ELEMENTS) | | |
| 32 (2^5) | 1024 (2^10) | 1,048,576 (2^20) |
| MS | Random | 0.010 | 3.882 | 3495509.500 |
| Ascending | 0.011 | 3.659 | 3406883.000 |
| Descending | 0.017 | 3.946 | 3078886.250 |
| Few swaps | 0.013 | 4.622 | 2996757.000 |

4) 실험 결과 정리 그래프

위 그래프를 통해 한 눈에 알 수 있듯이, 합병 정렬은 input data order와는 연관이 거의 없다는 것을 알 수 있다.

합병 정렬은 input data를 원소가 1개가 될 때까지 무조건 나누고, 다시 combine하는 과정에서 정렬하는 것이므로 실행 시간이 input data order의 영향을 거의 받지 않는다. 위 그래프에서 확인할 수 있듯이, 합병 정렬은 input data size의 영향도 거의 안 받는다는 것을 확인할 수 있다.

5) 이론과 실험 결과 분석

|  |  |  |
| --- | --- | --- |
|  | Big-O | Example of input data order |
| Worst case | O(n\*log2n) = O(n\*logn) | Random, Ascending, Descending, Few swaps |
| Average case | O(n\*log2n) = O(n\*logn) | Random, Ascending, Descending, Few swaps |
| Best | O(n\*log2n) = O(n\*logn) | Random, Ascending, Descending, Few swaps |

앞서 살펴보았듯이, 합병 정렬은 input data order의 영향은 거의 안 받으므로, 임의로 random order인 경우를 골라 이론적인 시간 복잡도와 실험 결과 값을 비교하였다. 그 계산 과정은 아래 사진에 나와 있다. 이로써, 이론적인 시간 복잡도 O(n\*logn)은 사실인 것으로 드러났다.

텍스트, 화이트보드이(가) 표시된 사진

자동 생성된 설명

(c) Quick\_NAIVE

1) 이론적인 시간 복잡도

|  |  |  |
| --- | --- | --- |
|  | Big-O | Example of input data order |
| Worst case | O(n^2) | Descending, Ascending |
| Average case | O(n\*logn) | Random, Few swaps |
| Best case | O(n\*logn) | Selected pivot is near to the median of the data |

퀵소트에서 worst case인 경우는 이전 항목에서 서술하였듯이, 피벗 원소를 뽑은 이후 skewed 형태로 덩어리가 나뉠 때이다. 이는 맨 왼쪽 원소를 피벗 원소로 뽑는 Quick\_NAIVE 정렬에서는 input data order가 ascending과 descending이 이에 해당하므로 worst case의 시간 복잡도는 이론적으로 O(n^2)까지 증가하는 것이다.

2) 실험 결과 출력 화면

|  |  |
| --- | --- |
| Input data size | Result |
| 32 (=2^5) | 텍스트이(가) 표시된 사진  자동 생성된 설명 |
| 1024 (=2^10) | 텍스트이(가) 표시된 사진  자동 생성된 설명 |
| 1048576 (=2^20) | 텍스트이(가) 표시된 사진  자동 생성된 설명 |

3) 실험 결과 정리 표

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sorting function | Order of  input data | Execution time according to input data size (N\_ELEMENTS) | | |
| 32 (2^5) | 1024 (2^10) | 1,048,576 (2^20) |
| Quick\_NAIVE | Random | 0.003 | 0.284 | 117.171 |
| Ascending | 0.004 | 2.990 | 631558.750 |
| Descending | 0.004 | 4.662 | 909972.688 |
| Few swaps | 0.003 | 0.276 | 822311.125 |

4) 실험 결과 정리 그래프

입력 데이터의 order가 Random인 경우, 즉 평균적인 경우일 때는 퀵소트가 이전에 구현한 합병 정렬과 삽입 정렬보다는 훨씬 빠르지만, worst case에 해당하는 ascending이나 descending의 경우에는 실행시간이 급격히 늘어나는 것을 확인할 수 있었다. 바로 이 점이 퀵소트의 최대 단점에 해당한다.

데이터 크기에 따른 실행 시간 차이가 이전에 구현한 다른 정렬들보다는 작다는 것을 위의 그래프를 통해 확인할 수 있다.

5) 이론과 실험 결과 분석

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sorting function** | **Order of  input data** | **Execution time according to input data size (N\_ELEMENTS)** | | |
| **32 (2^5)** | **1024 (2^10)** | **1,048,576 (2^20)** |
| IS | Random | 0.003 | 0.765 | 33009.000 |
| Ascending | 0.001 | 0.008 | 4.888 |
| Descending | 0.003 | 2.418 | 1307699.375 |
| Few swaps | 0.002 | 0.086 | 145.419 |
| MS | Random | 0.010 | 3.882 | 3495509.500 |
| Ascending | 0.011 | 3.659 | 3406883.000 |
| Descending | 0.017 | 3.946 | 3078886.250 |
| Few swaps | 0.013 | 4.622 | 2996757.000 |
| Quick \_NAIVE | Random | 0.003 | 0.284 | 117.171 |
| Ascending | 0.004 | 2.990 | 631558.750 |
| Descending | 0.004 | 4.662 | 909972.688 |
| Few swaps | 0.003 | 0.276 | 822311.125 |

위의 표에서도 확인할 수 있듯이, 다른 정렬들보다 퀵소트는 이름 그대로 정렬속도가 빠르다는 특징이 있다. 하지만, 이 특징은 best나 average case에만 해당하고, worst case인 ascending이나 descending의 경우에는 정렬을 진행하는 과정에서 skewed 형태로 데이터들이 나뉘면서 정렬되기 때문에 시간 복잡도가 O(n^2)까지 증가한다는 결정적인 단점이 있다. 특히, input data order가 ascending 인 경우 insertion sort가 quicksort보다 훨씬 빠른 것을 위의 표에서도 확인이 가능하다. 이처럼 input data가 ascending에 가까울수록 삽입 정렬이 퀵소트보다 시간 복잡도의 측면에서 더 우월하다는 것을 실험에서도 확인할 수 있었다. 이러한 퀵소트의 단점을 개선하는 데 보통 크게 3가지 방법이 있는데, 다음 항목부터는 이 퀵소트를 개선하는 데 중점을 두고 실험을 진행할 예정이다.

(d) Quick\_P

이번에 구현할 퀵소트는 이전에 2번 항목에서 설명했던 median-of-three 방식을 이용하여 피벗 원소를 뽑는 식으로 개선되었다. NAIVE버전보다 개선된 것이 맞는지 실험에서 확인하고자 한다.

1) 실험 결과 출력 화면

|  |  |
| --- | --- |
| Input data size | Result |
| 32 (=2^5) |  |
| 1024 (=2^10) |  |
| 1048576 (=2^20) |  |

2) 실험 결과 정리 표

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sorting function | Order of  input data | Execution time according to input data size (N\_ELEMENTS) | | |
| 32 (2^5) | 1024 (2^10) | 1,048,576 (2^20) |
| Quick\_P | Random | 0.004 | 0.231 | 416.791 |
| Ascending | 0.002 | 0.091 | 111.181 |
| Descending | 0.004 | 0.176 | 279.841 |
| Few swaps | 0.003 | 0.183 | 112.574 |

3) 실험 결과 정리 그래프

이전의 NAIVE 버전에서는 ascending과 descending에서 극명하게 시간 복잡도가 증가하는 것을 확인할 수 있었는데, P버전에서는 ascending과 descending도 다른 input data order들과 극명한 차이를 보이지는 않는 것을 확인할 수 있다. 퀵소트의 단점을 어느 정도 해결했다고 볼 수 있는 것이다.

데이터 크기에 따른 실행 시간 차이도 NAIVE버전보다도 많이 줄어든 것을 위 그래프를 통해 육안으로 확인이 가능하다.

4) 이론과 실험 결과 분석

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Quick \_NAIVE | Random | 0.003 | 0.284 | 117.171 |
| Ascending | 0.004 | 2.990 | 631558.750 |
| Descending | 0.004 | 4.662 | 909972.688 |
| Few swaps | 0.003 | 0.276 | 822311.125 |
| Quick\_P | Random | 0.004 | 0.231 | 416.791 |
| Ascending | 0.002 | 0.091 | 111.181 |
| Descending | 0.004 | 0.176 | 279.841 |
| Few swaps | 0.003 | 0.183 | 112.574 |

input data size가 32와 같이 다소 작은 경우에는 NAIVE와 P버전이 큰 차이를 확인하기 어렵다. 하지만, input data size가 커지면 커질수록 두 버전 간의 실행 시간 차이 폭이 점점 증가하는 것을 확인할 수 있다. 시각적인 확인을 위해 그래프로 표현하면 아래와 같다.

(e) Quick\_PIS

0) 적절한 M 사이즈 결정

실험한 N\_ELEMENTS의 크기는 1024이므로, 그것의 절반인 512부터 시작하였다. 이후로는 한 단계 넘어갈 때마다 M 사이즈를 나누기 4해주다가 실행시간이 줄어드는 지점에서 점점 줄이는 식으로 진행하였다. 그리고 적어도 M이 0인 때보다는 실행시간이 짧아야 한다. 실험 결과는 아래 표와 같이 나왔다. 값이 감소하다가 다시 증가하는 구간을 위주로 탐색한 결과, M=15인 경우가 가장 실행시간이 짧은 것으로 나왔다. 이후 실험에서도 M의 값을 15로 설정하고 진행하였다.

|  |  |
| --- | --- |
| ‘M’ | Result |
| 0 |  |
| 512 |  |
| 128 |  |
| 32 |  |
| 15 |  |
| 10 |  |

1) 실험 결과 출력 화면

|  |  |
| --- | --- |
| Input data size | Result |
| 32 (=2^5) |  |
| 1024 (=2^10) |  |
| 1048576 (=2^20) |  |

2) 실험 결과 정리 표

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sorting function | Order of  input data | Execution time according to input data size (N\_ELEMENTS) | | |
| 32 (2^5) | 1024 (2^10) | 1,048,576 (2^20) |
| Quick\_PIS | Random | 0.002 | 0.094 | 189.072 |
| Ascending | 0.002 | 0.036 | 66.055 |
| Descending | 0.002 | 0.054 | 176.066 |
| Few swaps | 0.002 | 0.073 | 44.446 |

3) 실험 결과 정리 그래프

입력 크기가 32 정도로 작으면 input data order별로 차이가 크지는 않다.

이렇게 PIS 전만 따로 보면 이전의 P버전과 큰 차이를 느끼기 어렵다. 다음 페이지에서 다른 버전의 퀵소트들과 비교할 예정이다.

4) 이전 퀵소트들과의 비교

왼쪽의 그래프는 input data order (n은 2^20으로 고정)의 관점에서 바라본 퀵소트 세 버전의 차이이다. 이 그래프에서도 확인할 수 있듯이, NAIVE 버전이 가장 높은 시간 복잡도를 지니고, P버전과 PIS버전은 그보다는 작은 차이지만 PIS 버전이 더 낮은 실행 시간을 지님을 알 수 있다.

밑의 그래프는 이전 그래프를 x와 y축을 바꾼 관점에서 기록한 퀵소트 세 버전의 차이이다. 확실히 PIS가 거의 모든 input data order에서 가장 작은 시간 복잡도를 지닌다는 점이 확인 가능하다.

(f) Quick\_PISTRO

이번에는 피벗 원소 선정 이후, 덩어리가 큰 쪽은 iteration으로 처리하고, 작은 쪽은 recursion으로 처리함으로써 system stack overflow를 방지할 수 있는 PISTRO 버전을 구현하였다.

1) 실험 결과 출력 화면

|  |  |
| --- | --- |
| Input data size | Result |
| 32 (=2^5) |  |
| 1024 (=2^10) |  |
| 1048576 (=2^20) |  |

2) 실험 결과 정리 표

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sorting function | Order of  input data | Execution time according to input data size (N\_ELEMENTS) | | |
| 32 (2^5) | 1024 (2^10) | 1,048,576 (2^20) |
| Quick\_PISTRO | Random | 0.003 | 0.110 | 145.507 |
| Ascending | 0.002 | 0.033 | 47.344 |
| Descending | 0.004 | 0.073 | 94.028 |
| Few swaps | 0.003 | 0.151 | 44.967 |

3) 실험 결과 정리 그래프

전반적으로 PIS버전과 유사한 형태로 그래프가 그려졌다. 하지만, 실행시간의 정확한 수치는 PIS보다 PISTRO가 더 작았다. 이로써 PIS보다 PISTRO가 시간 복잡도의 측면에서 더 우월하다는 것을 이번 실험으로 직접 확인할 수 있었다.

4) 이전 퀵소트들과의 비교

이제껏 구현한 퀵소트정렬은 총 4가지였다. 확실히 퀵소트의 단점을 보완하는 방향으로 개선하면 기존의 NAIVE버전보다 시간 복잡도의 측면에서 더 나아질 수 있다는 것을 확인할 수 있다.

위의 그래프와 같은 그래프지만, x축과y축만 바꿔 표현한 것이다. PISTRO에서 NAIVE로 갈수록 그래프가 우상향하는 것으로 보아, 퀵소트 정렬이 개선되면 개선될수록 시간 복잡도가 줄어든다는 것을 확인할 수 있다.

**4. 심화 분석**

차트이(가) 표시된 사진

자동 생성된 설명이론적으로 접근해보면, 일반적으로 insertion sort는 O(n^2)의 시간 복잡도를 지니고, quick sort는 O(n\*logn)의 시간 복잡도를 지닌다. f(x) = x\*logx, g(x)=x^2인 두 그래프를 ‘지오지브라’ 라는 그래프 프로그램을 통해 그린 다음 비교해보면 위의 사진과 같다. 위의 그래프에서 확인할 수 있듯이, x의 값이 증가할수록 두 그래프 간의 차이도 커진다. 이에 의하면, x의 값이 작을수록 두 그래프 간의 차이가 줄어든다는 것인데, 시간 복잡도의 측면에서 과연 어느 정도의 x까지 두 그래프 간의 차이가 없다고 봐도 괜찮은 것인지를 중점으로 실험을 설계하였다.

해당 질문을 해결하기 위해 실험에 사용할 input data order를 삽입 정렬과 퀵정렬에서 모두 worst case에 속하는 descending order로 고정시켜 놓고 실험을 진행하였다. 앞에서 실행한 실험들은 input data size가 3가지였다. 그 결과를 아래 표로 정리하였다. 32까지는 비슷한데 1024부터는 확실히 퀵정렬이 빠르다. 이에 따라 32에서 차례로 input data size를 줄여가며 두 정렬의 실행 시간이 비슷해지는 시점을 실험해보았다.

[이전 실험 결과 정리 표]

|  |  |  |  |
| --- | --- | --- | --- |
| **Sorting function** | **Execution time according to input data size (N\_ELEMENTS)** | | |
| **32 (2^5)** | **1024 (2^10)** | **1,048,576 (2^20)** |
| IS | 0.003 | 2.418 | 1307699.375 |
| Quick\_P | 0.004 | 0.176 | 279.841 |

[실험 정리 표]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sorting function** | **Execution time according to input data size (N\_ELEMENTS)** | | | |
| **8** | **15** | **16** | **32** |
| IS | 0.001 | 0.001 | 0.001 | 0.003 |
| Quick\_P | 0.001 | 0.001 | 0.002 | 0.004 |

실험 결과를 위의 표로 정리하였다. 16보다 작으면 IS와 Quick\_P는 실행 시간에 차이가 거의 없는 것으로 나왔다. 따라서 input data size가 15까지는 굳이 퀵정렬을 사용할 필요없이 삽입정렬을 사용해도 크게 문제가 되지 않는다는 것을 유추해낼 수 있었다. 이 값은 이전 문항에서 Quick\_PIS를 구현할 때 구한 M의 값과 동일하다는 것도 확인할 수 있다.

[실험 최종 결과]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sorting function** | **Order of  input data** | **Execution time according to input data size (N\_ELEMENTS)** | | |
| **32 (2^5)** | **1024 (2^10)** | **1,048,576 (2^20)** |
| IS | Random | 0.003 | 0.765 | 33009.000 |
| Ascending | 0.001 | 0.008 | 4.888 |
| Descending | 0.003 | 2.418 | 1307699.375 |
| Few swaps | 0.002 | 0.086 | 145.419 |
| MS | Random | 0.010 | 3.882 | 3495509.500 |
| Ascending | 0.011 | 3.659 | 3406883.000 |
| Descending | 0.017 | 3.946 | 3078886.250 |
| Few swaps | 0.013 | 4.622 | 2996757.000 |
| Quick \_NAIVE | Random | 0.003 | 0.284 | 117.171 |
| Ascending | 0.004 | 2.990 | 631558.750 |
| Descending | 0.004 | 4.662 | 909972.688 |
| Few swaps | 0.003 | 0.276 | 822311.125 |
| Quick\_P | Random | 0.004 | 0.231 | 416.791 |
| Ascending | 0.002 | 0.091 | 111.181 |
| Descending | 0.004 | 0.176 | 279.841 |
| Few swaps | 0.003 | 0.183 | 112.574 |
| Quick\_PIS | Random | 0.002 | 0.094 | 189.072 |
| Ascending | 0.002 | 0.036 | 66.055 |
| Descending | 0.002 | 0.054 | 176.066 |
| Few swaps | 0.002 | 0.073 | 44.446 |
| Quick \_PISTRO | Random | 0.003 | 0.110 | 145.507 |
| Ascending | 0.002 | 0.033 | 47.344 |
| Descending | 0.004 | 0.073 | 94.028 |
| Few swaps | 0.003 | 0.151 | 44.967 |