알고리즘 설계와 분석(CSE3081-02)

HW2 Quick sort

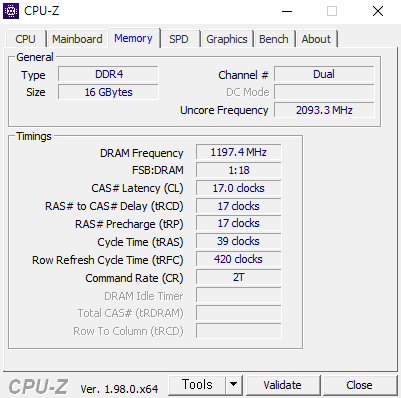
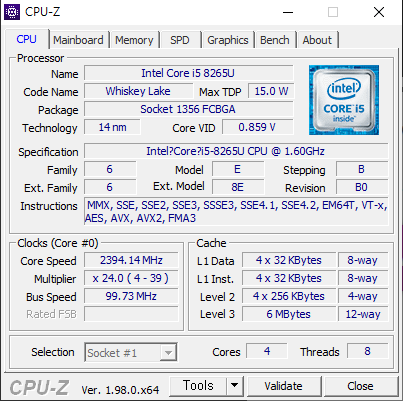
20191286 김나현

본 과제는 Divide and conquer를 사용한 sorting 방법인 quick sort을 변형하여 qsort\_orig, qsort\_median\_insert, qsort\_median\_insert\_iter 함수를 스스로 구현해보고 여러 샘플을 통해 해당 함수들이 얼마나 빠르고 효과적으로 정렬하는지를 정량적으로 비교해보는 것입니다.

우선, qsort\_orig 함수를 구현하기 위해 배열에서 pivot이 위치하는 index를 찾기 위한 partition 함수를 추가적으로 구현하였습니다. 이 함수는 pivot을 배열의 가장 왼쪽, 즉 처음 원소로 가정하고 pivot 값을 기준으로 pivot보다 큰 값은 pivot의 오른쪽으로, pivot보다 작거나 같은 값들은 pivot의 왼쪽으로 정렬한 후, pivot이 위치하는 index를 반환하는 함수입니다. qsort\_orig 함수에서는 partition 함수를 통해 구한 pivot의 index를 가지고 pivot의 index보다 작은 배열과 pivot의 index보다 큰 배열을 두 개로 나눠 재귀적으로 qsort\_orig 함수를 호출하여 정렬해야 할 배열의 크기가 1보다 작거나 같으면 더 이상 함수를 재귀적으로 호출하지 않고 return을 하게 됩니다.

qsort\_median\_insert 함수는 위의 qsort\_orig에서 pivot의 index를 찾기 위해 사용했던 partition 함수와 유사한 median\_partition 함수를 추가적으로 구현하였습니다. median\_partition 함수는 partition 함수와 달리, 가장 왼쪽이나 오른쪽 원소를 pivot이라고 가정하는 것이 아니라 배열의 가장 왼쪽 원소 n개를 사용하여 그 중 중앙값을 pivot이라고 가정하는 방법을 사용합니다. 이때, 배열의 가장 왼쪽 원소 10개와 5개를 사용하여 구현해본 결과, 원소의 개수가 1024개인 unsorted 배열을 정렬하는 데에 10개의 값을 비교하여 중앙값을 찾는 방법은 1.189 ms가 걸렸고, 5개의 값을 비교하여 중앙값을 찾는 방법은 0.696 ms가 걸렸습니다. 이러한 실험 결과를 통해, 배열의 원소 10개를 비교하는 것이 5개의 원소를 비교하는 것보다 더욱 중앙값에 가까운 결과를 얻을 수 있을지는 몰라도 더 오랜 시간이 걸리기 때문에 비효율적인 방법이라고 결론 지을 수 있었습니다. 마찬가지로 이 방법과 위의 qsort\_orig 함수를 이용한 정렬 방법을 비교하여 insertion 정렬을 사용한 것이 얼마나 효과적인지를 확인해볼 수 있었습니다. 우선, qsort\_median\_insert 함수는 배열의 크기가 10보다 작거나 같으면 더 이상 함수를 재귀적으로 호출하지 않고 모든 과정이 끝나면 insertion sort 방법으로 정렬하는 방식으로 구현했습니다. 이때, 초기 배열의 크기가 10이라면 qsort\_median\_insert 함수는 아예 아무런 역할도 하지 않고 insertion sort 함수, 즉 isort 함수만으로 배열을 정렬하게 되고 실제로 시간을 측정해본 결과, 0.003 ms가 걸린 qsort\_orig 함수에 비해 느린 0.005 ms가 걸리지만 배열의 크기가 1024으로 커지면 1.120 ms가 걸린 qsort\_orig 함수에 비해 빠른 1.077 ms가 걸린다는 것을 확인할 수 있었습니다. 이를 통해 배열의 크기가 작을 때는 qsort\_median\_insert 정렬이 qsort\_orig 정렬에 비해 느리지만 배열의 크기가 커질 때 insertion sort 함수가 O(n)과 유사한 시간 복잡도를 가지게 되어 배열의 크기가 1이 될 때까지 재귀적으로 quick sort를 호출하는 qsort\_orig 함수보다 더욱 효과적이라는 것을 알 수 있었습니다. 이때, 본 과제에서 insertion sort을 수행하는 함수는 isort라는 이름으로 구현하였고 for loop를 이용해 index가 1인 원소부터 elementNum-1인 원소까지 자기 자신보다 앞선, 즉 이미 순서대로 정렬된 배열을 지나가며 올바르게 정렬되도록 자기 자리를 찾는 방법으로 구현하였습니다.

qsort\_median\_insert\_iter 함수는 qsort\_median\_insert 함수를 사용하면서 pivot을 기준으로 두 개로 분할된 배열 중 작은 배열에 대해서만 재귀적으로 qsort\_median\_insert\_iter 함수를 호출하고, 큰 배열은 해당 함수 안에서 iteration, 즉 반복을 통해 정렬해주는 함수입니다. 이때, qsort\_median\_insert\_iter 함수도 정렬해야 할 배열의 크기가 일정 크기 이하이면 정렬을 하지 않아야 하므로 iter\_elementNum이라는 변수를 이용하여 iter\_elementNum가 10보다 작으면 while문을 더 이상 돌지 않고 빠져나와 나중에 insertion sort를 통해 정렬되도록 하게 하였습니다. While문에서는 qsort\_median\_insert 함수에서 사용한 median\_partition 함수를 이용해 pivot의 index를 구하였고, pivot이 iter\_elementNum-pivot-1보다 작으면 배열의 첫 원소부터 pivot보다 하나 작은 index의 원소까지를 정렬하기 위해 재귀적으로 함수를 호출하고 나머지 부분을 정렬하기 위해 배열의 시작 주소를 pivot보다 하나 큰 index의 원소를 가리키도록 바꾸고 iter\_elementNum을 iter\_elementNum-pivot-1로 바꾸었습니다. 반대로, pivot이 iter\_elementNum-pivot-1보다 크거나 같으면 pivot보다 하나 큰 index의 원소부터 배열의 끝 원소까지를 정렬하기 위해 재귀적으로 함수를 호출하고 나머지 부분을 정렬하기 위해 iter\_elementNum을 pivot으로 바꾸었습니다.

다음은 입력 데이터에 따른 각 정렬 방법의 수행 시간을 측정한 것입니다. Sorting해야 할 data의 개수가 1024, 4096(1024의 4배), 16384(1024의 16배), 262144(1024의 256배), 1048576(1024의 1024배)일 때를 기준으로 각각의 quick sort 방법의 수행 시간을 측정해보고 표로 비교해보았습니다. 이때, 본 실험은 다음과 같은 개발 환경에서 이루어졌고, Visual studio의 Release-x64 모드에서 진행되었습니다.

<표 1> 데이터 수가 1024개일 때

<표 2> 데이터 수가 4096개일 때



<표 3> 데이터 수가 16384개일 때

<표 4> 데이터 수가 262144개일 때

<표 5> 데이터 수가 1048576개일 때



위와 같은 실험 결과를 통해, Standard C library에서 제공하는 qsort 함수보다 직접 구현한 qsort\_orig 함수가 더욱 빠르다는 것을 알 수 있었고, 함수 21과 함수 22를 비교하였을 때 qsort\_orig 함수로 데이터를 정렬하는 것과 qsort\_median\_insert 함수와 insertion sort로 데이터를 정렬하는 것의 속도가 비슷하거나 qsort\_orig 함수만을 이용해서 정렬하는 것의 속도가 더욱 빠르다는 사실을 통해 제가 구현한 qsort\_median\_insert 방식이 생각보다 효율적이지 않다는 생각이 들었습니다. 따라서, 이를 보완하기 위해 실제 qsort\_median\_insert 방법에서 정렬해야 할 데이터의 초기 5개의 원소 중 중앙값을 pivot으로 사용하였던 것을 초기 3개의 원소 중 중앙값을 pivot으로 사용하거나, 정렬해야 할 배열의 크기가 10개 이하일 때 재귀적으로 qsort\_median\_insert 함수를 호출하지 않고 insertion sort를 통해 정렬했던 것을 수정하여 정렬해야 할 배열의 크기가 5개 이하나 20개 이하와 같이 10개와 다른 값으로 지정하여 실험을 반복해봄으로써 어떠한 방식이 더욱 효율적인지를 분석해봐야 합니다. 제가 구현한 방법으로는 pivot을 기준으로 둘로 나눈 배열에 대해서 모두 재귀적으로 qsort\_median\_insert 함수를 호출하는 것과 정렬해야 할 배열의 크기가 더 작은 부분에 대해서만 재귀적으로 qsort\_median\_insert\_iter 함수를 호출하는 것 사이의 큰 속도 차이를 발견할 수는 없었습니다. 다만, 데이터의 크기가 커질수록 qsort\_median\_insert\_ iter 함수를 사용하면 qsort\_median\_insert 함수보다 조금 빨라지는 경향성을 확인할 수 있었는데 이는 qsort\_median\_insert 함수를 사용할 때보다 재귀적으로 함수를 호출하는 정도가 확연히 줄어들기 때문에 시스템의 스택을 적게 사용하여 조금 더 빠르게 정렬할 수 있고 제가 수행한 실험의 경우 데이터의 크기가 커져도 두 함수를 이용한 정렬이 모두 성공적이었지만 데이터의 크기가 더욱 커졌을 때 반복적인 재귀 함수 호출에 의한 스택의 오버플로우를 방지할 수 있다는 장점이 있습니다.