Coputer Algorithms
Project - 01

이름 김태연
학번 201511058
이메일 kim77ty@dgist.ac.kr

요 약

본 프로젝트를 통해서 기본적인 정렬 알고리즘들에 대해서 직접 구현을 해보고, 각 알고리즘의 특징을 알아보는 것을 목표로 하며, 각 알고리즘들이 실행시간에서 차이가 나는 이유에 대해서 시간 복잡도와 hidden constant 등을 고려하며 정리를 해본다. 특히, 퀵정렬(quick sort)에 대해서는 pivot 의 질에 대한 의존성을 줄이기 위한 최적화 방법에 대해서 생각해보고, 실제로 이러한 최적화 방법이 성능에 있어서 어느 정도의 효과를 가져오는지에 대해서 알아본다.

1. 서론

컴퓨터 알고리즘 수업에서 배운 여러 정렬 알고리즘들 중에서 비교 정렬 알고리즘들에 대해서 직접 구현을 해본다. 구현하는 알고리즘들로는 insertion sort, merge sort, heap sort, quick sort 가 있다. 정렬 알고리즘들의 특징에 대해서 알아보면, insertion sort 는 하나의 key 를 잡아주고, 그 key 의 앞은 정렬이 되어있다고 가정한 후 앞의 정렬된 array 에서 key 의 위치를 찾아주는 알고리즘으로, n^2 의 시간 복잡도를 가지는 정렬 알고리즘이다. merge sort 는 array 를 가장 작은 단위로 나눠주고 재귀적으로 나누어진 array 들을 merge 시켜주는 알고리즘으로 n*lgn 의 시간 복잡도를 가지며, heap sort 는 heap 을 만들어주며 정렬을 해주는 알고리즘으로, 이 역시 heap 의 길이에 의해서 n*lgn 의 시간 복잡도를 가지게 된다. 마지막으로 quick sort 는 하나의 pivot 을 잡아주고 그 pivot 을 중심으로 left side 는 pivot 보다 작은 수를, right side 는 pivot 보다 큰 수를 모아주어, 각각의 side 에 대해서 quick sort 를 계속 재귀적으로 호출해주는 알고리즘으로, 이 역시 n*lgn 의 시간 복잡도를 가진다. quick sort 는 pivot 의 질에 따라서 시간 복잡도가 최악의 경우에는 n^2 이 될 수도 있다는 문제가 있다. 이에 대해서 hidden constant 가 실행시간에 영향을 끼치지 않을 정도로 pivot 의 임의성을 보안하기 위한 몇가지의 방법을 이용하여 quick sort 를 최적화하고자 한다. 시간 복잡도에 대한 자세한 수식적 증명과 알고리즘에 대한 자세한 설명은 참고문헌을 참고하도록 한다.

2. 본문

각 알고리즘들의 구현은 코드파일의 주석을 통해 알 수 있으므로 생략하도록 한다.

2.1 quick sort 의 최적화

알고리즘에 대한 report 를 쓰고 있기 때문에, 성능 향상을 위해 변경해준 세부적인 내용은 생략하고, quick sort 의 성능을 높이기 위해서 추가해준 알고리즘들에 대해서 알아보면 다음과 같다.

- array 의 크기가 작을 경우에는 quick sort 를 재귀적으로 호출하지 않고 insertion sort 로 정렬을 해준다.

array 의 크기가 작을 때에는 속도의 차이가 크게 나지 않고, 재귀의 깊이를 낮춰주는 역할도 해주게 된다. 구현한 알고리즘에서는 array 의 크기가 50 이하가 되면 insertion sort 로 정렬을 하도록 해주었다.

- pivot 의 질 개선

pivot 을 조금 더 개선시켜 주기 위해서 pivot 을 정할 때 quick sort 에서 시작점과 끝점, 그리고 가운데 지점을 뽑아서 시작점, 가운데 점, 끝점 순서대로 정렬을 해주었다. 그 후, 가운데의 값을 pivot 으로 써주면, 3 개의 값 중 중앙값을 이용하게 되어서 pivot 의 질이 조금 더 좋아지며, partition 을 해줄 때도 시작점과 끝점은 이미 정렬이 되어있다고 볼 수 있으므로, 전체 길이가 count 라고 할 때, (count – 2)의 길이에 대해서만 정렬을 해주도록 구현했다.

2.2 정렬 알고리즘들의 성능 비교

시간 복잡도가 높은 insertion sort 도 같이 비교를 해주기 위해서, 난수를 뽑아 만든 Array 에 대해서, Array 의 크기를 10 부터 시작하여 10 씩 키워주면서 정렬을 10 번 반복한 후, 그 평균값을 구하면서 Array 의 크기가 5000 이 될 때까지 측정을 해줬다.

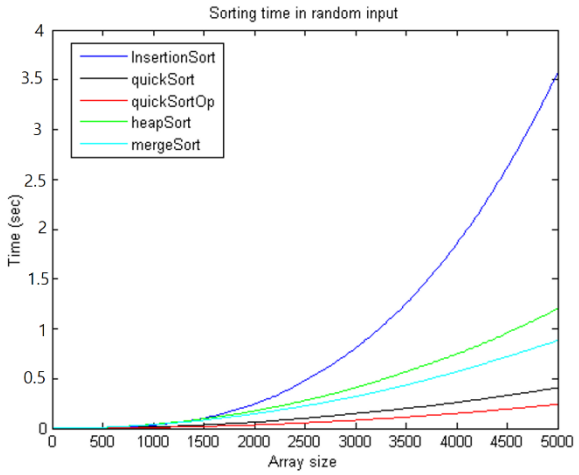

그림 1. Random input 을 받았을 때 sorting 시간

그 결과는 그림 1 의 그래프와 같이 나왔으며, quick sort Op, quick sort, merge sort, heap sort, insertion sort 의 순서대로 성능이 평가된 것을 알 수 있다. 일단 이 측정을 통해서, insertion sort 의 경우는 input data 의 양이 매우 적을 때에는 다른 알고리즘과 비슷하게 잘 작동을 하지만, input data 가 커질수록 그 성능이 크게 저하된다는 것을 알 수 있고, 이론적으로 insertion sort 만 달랐던 시간 복잡도 역시 간접적으로 확인할 수 있었다.
다음으로는, insertion sort 를 제외하고 n*lgn 의 시간 복잡도를 가지는 나머지의 알고리즘들에 대해서 성능을 더 비교해보기 위해서 4 개의 알고리즘들만을 이용해서 난수로 뽑아준 array 에 대해 array 의 크기를 그 크기가 커짐에 따라 증가폭을 늘려주면서 10000000 까지 늘리면서 5 번 측정을 한 후 평균을 구하여 측정을 해 주었다.

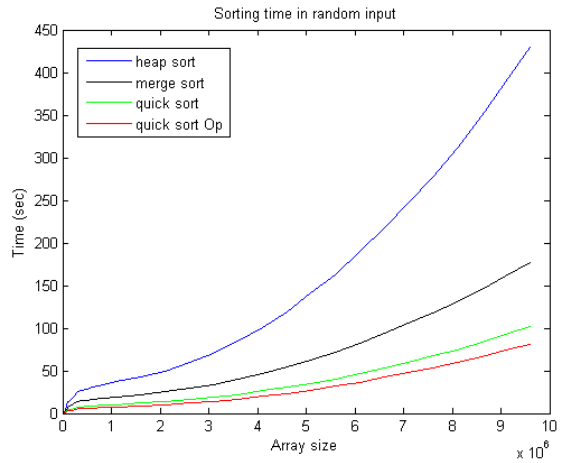

그림 2. Array 의 크기를 10000000 까지 한 측정

그 결과는 그림 2 의 그래프와 같이 나왔다. 성능을 비교해보면 heap sort 가 array 의 크기가 커질수록 성능이 많이 저하되는 것을 알 수 있었고, 그 다음으로는 merge sort, quick sort 그리고 quick sort Op 의 성능이 가장 좋은 것으로 평가되었다.
일단 heap sort 와 merge sort 가 큰 차이를 보인 이유에 대해서 생각을 해 보면, 각 알고리즘의 시간 복잡도 자체는 같으므로, 알고리즘 내의 hidden constant 에 대해서 생각을 해봐야 할 것이다. 가장 큰 부분은, 두 알고리즘 모두 재귀적 호출을 이용하지만, heap sort 의 경우는 heap sort 함수에서 build_max_heap 이라는 함수와 max_heapify 라는 함수를 재귀적으로 호출하게 되고, build_max_heap 의 함수 안에서도 max_heapify 를 재귀적으로 호출하며, max_heapify 함수 역시 max_heapify 함수를 재귀적으로 호출하는 구조가 되기 때문에 다른 알고리즘들보다 재귀호출로 인해 쌓이게 되는 stack 의 깊이가 상당히 깊기 때문이라고 생각 할 수 있을 것이다. 그리고 heap sort 의 알고리즘 자체도 역시 만들어진 heap 자체의 높이는 높지 않을 수 있지만, 이를 정렬된 결과로 뽑아줄 때 가장 큰 index 하나만을 뽑아주고 가장 작은 index 를 heap 의 맨 위로 올린 후 다시 heap 을 만들어주는 과정을 반복하기 때문에 이 부분에서 hidden constant 가 커졌다고 생각할 수도 있을 것 이다.
다음으로 merge sort 와 quick sort 에 대해서 비교를 해보면, 일단 merge sort 는 merge 를 할 때 (끝점 – 시작점) 크기의 array 를 동적할당해주어 구현을 하였기 때문에, 이 부분에서 공간 복잡도가 늘어나면서 정렬 알고리즘의 성능까지도 영향을 주었다고 생각할 수 있다. 또한, merge sort 가 quick sort 에 비해 조금 더 많은 조건문과 while 문이 알고리즘 안에 포함되어 있기 때문에, 이 역시 두 알고리즘의 성능 차이에 영향을 미쳤을 것이다.
마지막으로, quick sort 와 최적화된 quick sort(quick sort Op)를 비교해보면, 최적화된 quick sort 가 더 좋은 성능을 나타냈으므로, 2.1 에서 언급한 최적화의 방법으로 인한 효과가 그로 인해서 추가된 hidden constant 에 의한 효과보다 크다는 것을 알 수 있다.

다음으로는 특별한 경우에 대해서 각각의 정렬 알고리즘이 어떻게 작동하는지 알아보기 위해서 input data 로 sorted 된 array 와 reverse sorted 된 array 를 넣어줘서 측정을 해봤다. 측정 방법은, array 의 크기를 10 부터 3500 까지 10 씩 늘려주며 10 번을 반복하여 실행한 후 걸린 시간의 평균을 구해주었다.

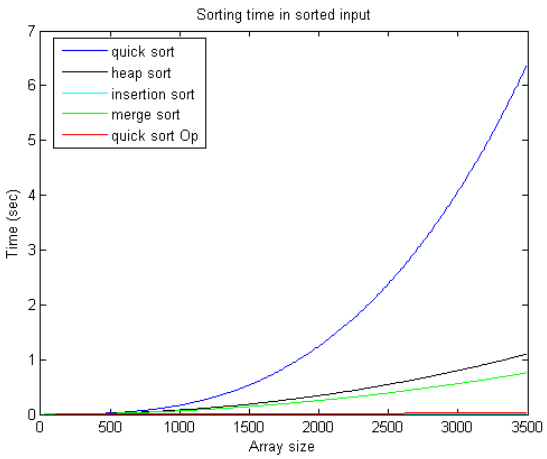

그림 3. Sorted input 에 대한 sorting 시간

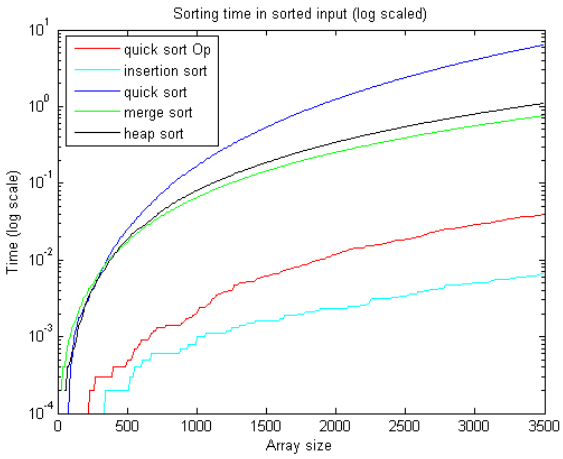

그림 4. Sorted input 에 대한 log scale 의 시간

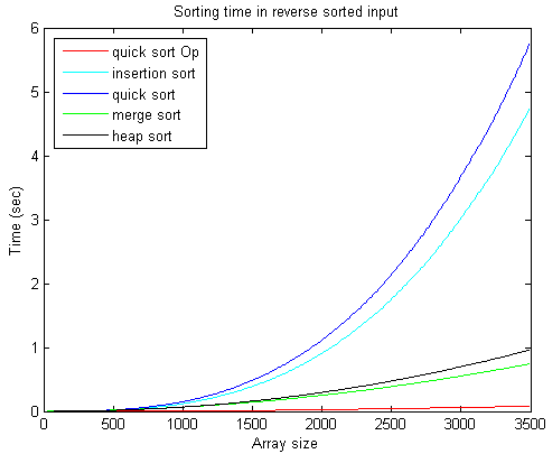

그림 5. Reverse sorted input 에 대한 sorting 시간

먼저, sorted input 에 대한 결과는 그림 3 의 그래프와 같이 나왔으며, 이 때 insertion sort 와 quick sort Op 의 비교가 힘들어서 시간 축을 log scale 로 바꿔준 그래프가 그림 4 이다.

결과를 보면, quick sort 의 경우는 worst case 가 되어 가장 성능이 저하되고 n^2의 시간 복잡도로, insertion sort 의 난수에서의 정렬보다도 안 좋은 성능을 나타낸다는 것을 알 수 있다. 반대로 insertion sort 의 경우는 best case 가 되어서 가장 좋은 성능을 보이는 것을 알 수 있었다. 그리고 최적화된 quicksort 의 경우는 항상 가운데의 값이 전체의 중앙값이 되기 때문에 best case 가 되어, 난수를 정렬해줄 때 보다 조금 더 빠른 성능을 확인할 수 있었다. 다른 정렬 알고리즘들은 난수를 input 으로 넣어 주었을 때와 비슷한 성능을 나타냈다.

다음으로 reverse sorted input 의 경우는 그림 5 의 그래프와 같이 나왔으며, 이 역시 quick sort 가 worst case 가 되어 가장 성능이 좋지 않게 나왔으며, insertion sort 역시 가장 많이 swap 을 해주게 되어 worst case 로, 난수를 input 으로 넣어 주었을 때와 비교해서 시간 복잡도는 같지만 성능은 조금 저하된 것을 알 수 있었다. 최적화된 quick sort 는 이 경우도 역시 best case 가 되어 난수를 정렬해줄 때 보다 조금 더 빠른 성능을 보여주었다. 다른 알고리즘들의 경우는 sorted input 과 마찬가지로 난수를 input 으로 넣어 주었을 때와 비슷한 성능을 보인다는 것을 알 수 있다.

2.3 결론

이번 프로젝트를 통해서 각 알고리즘들의 특성을 알 수 있게 되었으며, 최적화된 quick sort 가 기존 quick sort 의 문제점들을 잘 개선했는지도 알 수 있었다. 최적화된 quick sort 의 평가와 각 알고리즘들의 특성을 바탕으로 어느 부분에서 유용하게 쓰일지에 대해서 생각해보면 다음과 같다.

먼저 quick sort 에 대해서 보면, 보통의 경우 (난수를 input 으로 넣어줄 때)에 최적화된 정렬 알고리즘의 성능이 더 좋은 것을 알 수 있었고, quick sort 의 가장 큰 문제인 worst case 에 대한 문제도 기존의 worst case 에 대해서는 해결된 것을 알 수 있었다. 또한, 최적화된 정렬 알고리즘의 worst case 에 대해서 생각을 해 보면, 함수를 시행할 때 항상 시작점과 가운데점과 끝점 3 개의 값이 가장 작은 3 개의 값이 되어야 하는데, 이는 기존의 정렬이 항상 pivot 이 가장 작은 값이 되어야 worst case 가 된다는 것을 생각해보면, 기존의 quick sort 보다는 worst case 가 될 확률이 훨씬 낮아졌다는 것을 알 수 있고, worst case 가 된다고 해도, array 의 크기가 작을 때는 insertion sort 를 사용하고, sorting 해야 하는 array 의 크기가 기존의 worst case 보다 2 개씩 더 작아지게 되므로 약간의 성능 향상이 있을 것으로 기대된다. 이러한 quick sort 는 비교 정렬 알고리즘 중에서 가장 빠른 속도를 보이므로, 빠른 응답 속도를 요구하는 환경에서 유용하게 쓰일 수 있을 것이며, 최적화된 알고리즘들로 안정성도 많이 개선되어 가장 많이 쓰이는 알고리즘이 되었다.

다음으로, insertion sort 의 경우는 데이터의 크기가 매우 커지면 그 속도가 매우 느려 지지만, 데이터가 몇 백개 단위 이하로 적을 때, 또는 높은 확률로 정렬이 되어있을 때는 그 성능이 최적화된 quick sort 와 비슷하게 나올 정도로 좋아서 유용하게 사용할 수 있을 것이다.

마지막으로 merge sort 와 heap sort 는 비교적 빠른 속도로 input data 에 크게 영향을 받지 않고 안정적으로 정렬을 하는 것을 알 수 있었다. 최적화된 quick sort 역시 안정적인 동작을 보였지만, quick sort 는 항상 매우 적은 확률이라도 worst case 의 가능성을 가지게 된다. 따라서, 이 알고리즘들은 quick sort 처럼 매우 빠른 속도는 요구하지 않지만, 항상 안정적인 동작을 필요로 할 때에는 유용하게 쓰일 수 있을 것이다. 또한, heap sort 가 merge sort 보다 성능면에서는 조금 떨어지지만, 공간 복잡도까지 고려해야 하는 생황이라면 heap sort 가 더욱 유용할 수도 있다.

참고문헌

[1] 본 레포트 양식은 국내 학술대회 논문을 참고하였습니다.
[2] Introduction to Algorithms (Thomas H. Cormen)
[3] 컴퓨터 알고리즘 강의노트